Online Network Intrusion Detection System Using Temporal Logic and Stream Data Processing

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy by

Abdulbasit M. Ahmed

June 2013

Contents

Preface ix

Abstract xi

Acknowledgements xiii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Thesis Contribution 2
- 1.3 Thesis Scope 3
- 1.4 Thesis Organization 3
- 1.5 Summary 5

2 Intrusion Detection Systems 7
- 2.1 IDS Types 8
- 2.2 Intruder Detection Methods 9
  - 2.2.1 Techniques of Anomaly Based Intrusion Detection 9
  - 2.2.2 Techniques of Misuse based Intrusion Detection 10
- 2.3 NIDS Deployment 11
- 2.4 Popular Network Intrusion Detection Systems 13
  - 2.4.1 Open Sources NIDS 13
    - 2.4.1.1 Snort IDS 14
    - 2.4.1.2 Bro NIDS 15
    - 2.4.1.3 Suricata IDS 16
  - 2.4.2 Commercial NIDS 16
- 2.5 Summary 17

3 Temporal Logic and Intrusion Detection System 19
- 3.1 Why Temporal Logic? 19
- 3.2 Related Work 19
  - 3.2.1 MONID 20
  - 3.2.2 ORCHIDS 22
- 3.3 Summary 23

4 Stream Data Processing (SDP) 25
- 4.1 SDP Overview 25

4.2 *StreamBase* Stream SQL . . . 29
4.3 *StreamBase* High Performance, scalability, and high availability Features . 32
4.3.1 High Performance Features . . . 33
4.3.2 Scalability and High Availability . . . 36
4.4 Summary . . . 37

5 Temporal Stream Intrusion Detection System (TeStID) 39
5.1 Formal Specification . . . 39
5.1.1 Abstract View of Network Communications . . . 39
5.1.2 MSFOMTL Syntax . . . 43
5.1.2.1 Terms . . . 43
5.1.2.2 Formulae . . . 43
5.1.3 MSFOMTL semantics . . . 44
5.2 Attack Classification . . . 45
5.3 The Proposed System . . . 50
5.3.1 *TeStID* System Architecture . . . 50
5.3.2 Tools And Software Used . . . 51
5.3.3 The Benefits of the Proposed System . . . 52
5.4 Summary . . . 53

6 Temporal Logic to Stream Queries 55
6.1 Background . . . 55
6.2 The view of Time . . . 57
6.3 Mapping MSFOMTL into *SSQL* . . . 57
6.4 Correctness . . . 69
6.5 The Translator Development . . . 75
6.6 Summary . . . 85

7 Experiments and Results 87
7.1 Experiments Overview . . . 87
7.1.1 Experiments Aims . . . 87
7.1.2 The Experiment Setup and Approach . . . 88
7.2 Single Packet Attacks With Payload Experiment . . . 89
7.2.1 The Experiment Signatures Preparation . . . 90
7.2.2 Results and Analysis . . . 93
7.2.3 Scalability and Performance . . . 96
7.2.3.1 Implementations of Single Packet Attacks With No Concurrency and Multiplicity . . . 98
7.2.3.2 Implementations of Single Packet Attacks With Concurrency and Multiplicity . . . 101
7.3 Multiple Packet Attacks Case Studies . . . 103
7.3.1 Results and Analysis of Multiple Packet Attacks Experiments . . . 105
7.4 Summary . . . 108

8 Potential Use of The New System in Anomaly Based IDS 111
8.1 Anomaly Based Network Intrusion Detection Overview . . . 111
8.2 Protocol Anomaly Specifications . . . 113
8.2.1 Single Step Anomalies . . . 114

8.2.2 Multiple Step Anomalies 116
8.2.2.1 Multiple Step Anomalies for Weak Normal Behaviour Requirements 117
8.2.2.2 Multiple Step Anomalies for Strong Normal Behaviour Requirements 118
8.3 Protocol Anomaly Formulae Mapping 124
8.4 Correctness 125
8.5 Summary 126

9 Conclusion **127**
9.1 Summary 127
9.2 Contribution 128
9.3 Future Work and Research 128

Bibliography **131**

A ANTLR Grammar and String Template Group Files **141**
A.1 Description of The Grammar File Structure 141
A.2 *TeStID* Grammar File 143
A.3 *TeStID* String Template Group File 160

B Single Packet Attacks With Payload Signatures Files **177**
B.1 *SNORT* Single Packet Signatures File 177
B.2 *BRO* Single Packet Signatures File 183
B.3 *TeStID* Single Packet Signatures File 194

Illustrations

List of Figures

2.1 The NIDS Deployment Options 12
2.2 An Example of snort Rule 15

4.1 A Simple Application Running In A Container Inside The *StreamBase*Server 34
4.2 Applying Concurrency only on 4.1 35
4.3 Applying Multiplicity only on 4.1 36
4.4 A main module with a candidate component for using concurrency and multiplicity of 2. 37

5.1 Packet Encapsulation During Sending 40
5.2 Packet Deencapsulation During Receiving 41
5.3 *TeStID* System Architecture 52

6.1 Reset Scan in *StreamBase* Studio 65
6.2 Steps of Specification Translation into Stream Queries 69
6.3 *TeStID* The Translation Process 76
6.4 Parser Rules (1 of 4) For The Misuse Based Detection of *TeStID*. 79
6.5 Parser Rules (2 of 4) For The Misuse Based Detection of *TeStID*. 80
6.6 Parser Rules (3 of 4) For The Misuse Based Detection of *TeStID*. 81
6.7 Parser Rules (4 of 4) For The Misuse Based Detection of *TeStID*. 82
6.8 Calling The String Template Group File During The Parsing 84

7.1 The Testing Environment 89
7.2 bits/sec at 24 X 94
7.3 packets/sec at 24 X 95
7.4 packets/sec at 48 x 96
7.5 bits/sec at 48 x 96
7.6 Graphical representation of the program to capture one single packet attack in the *StreamBase* studio 97
7.7 Equivalent *SSQL* code for the *StreamBase* application 98
7.8 Three possible implementations to run single packet attack detection on the *StreamBase* server without using the parallelism features 99
7.9 Single Packet Attack Program Using Module 101
7.10 All Possible Implementations For The Single Packet Attack With Payload . . 102
7.11 CC Input Code and NCC Attack module With/Without Multiplicity 104
7.12 Maximum packets/seconds for each data test file. Both of the in/out data files of 06/04/1999 were used and they are illustrated in the same graph. . . 106
7.13 Maximum bits/seconds for each data test file 107
7.14 Packets/Sec and Bits/Sec Overlap graph for 05/04/1999 Data File 109

8.1 Network Anomalies 113
8.2 Parser Rules (1 of 3) 120
8.3 Parser Rules (2 of 3) for The Anomaly Based Detection 121
8.4 Parser Rules (3 of 3) for The Anomaly Based Detection 122
8.5 Simultaneous Connection Synchronization (RFC793, September 1981) 123

List of Tables

7.1 Single Packet Attacks Final Results 93
7.2 *TeStID* results without concurrency and multiplicity 100
7.3 Running *SNORT*, *BRO*, *TeStID* without parallel operations 100
7.4 Results of Single Packet Attacks With Concurrency and Multiplicity 103
7.5 Multiple Packets Attacks Results 105

Preface

This thesis is submitted solely to the University of Liverpool in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy. It is my own work and the sources from which material is drawn are identified within. Some parts of this thesis have appeared in the following publications:

[1] A. Ahmed, A. Lisitsa, and C. Dixon. A network intrusion detection system using temporal logic and stream processing. In Proceedings of the 17th Automated Reasoning Workshop. University of Westminster Harrow, United Kingdom, 2010. URL http://www2.wmin.ac.uk/bolotoa/ARW/arw-2010.html.

[2] A. Ahmed, A. Lisitsa, and C. Dixon. A misuse-based network intrusion detection system using temporal logic and stream processing. In P. Samarati, S. Foresti, J. Hu, and G. Livraga, editors, Proceedings of the 5th International Conference on Network and System Security, pages 18, Milan, Italy, 2011. IEEE. ISBN 978-1-4577-0458-1.

[3] A. Ahmed, A. Lisitsa, and C. Dixon.TeStID: A High Performance Temporal Intrusion Detection System. (Accepted) to appear in Proceedings of The Eighth International Conference on Internet Monitoring and Protection, Rome, Italy, 2013.

Papers relating to the following are in progress:

[4] A. Ahmed, A. Lisitsa, and C. Dixon. Temporal logic and stream data processing for misuse-based network intrusion detection. Journal paper (in preparation) relating to Chapters 3-7.

[5] A. Ahmed, A. Lisitsa, and C. Dixon. Temporal logic and stream data processing for anomaly-based network intrusion detection. Conference paper relating to Chapters 3, 4, 5, and 8.

Abstract

These days, the world are becoming more interconnected, and the Internet has dominated the ways to communicate or to do business. Network security measures must be taken to protect the organization environment. Among these security measures are the intrusion detection systems. These systems aim to detect the actions that attempt to compromise the confidentiality, availability, and integrity of a resource by monitoring the events occurring in computer systems and/or networks. The increasing amounts of data that are transmitted at higher and higher speed networks created a challenging problem for the current intrusion detection systems. Once the traffic exceeds the operational boundaries of these systems, packets are dropped. This means that some attacks will not be detected.

In this thesis, we propose developing an online network based intrusion detection system by the combined use of temporal logic and stream data processing. Temporal Logic formalisms allow us to represent attack patterns or normal behaviour. Stream data processing is a recent database technology applied to flows of data. It is designed with high performance features for data intensive applications processing. In this work we develop a system where temporal logic specifications are automatically translated into stream queries that run on the stream database server and are continuously evaluated against the traffic to detect intrusions.

The experimental results show that this combination was efficient in using the resources of the running machines and was able to detect all the attacks in the test data. Additionally, the proposed solution provides a concise and unambiguous way to formally represent attack signatures and it is extensible allowing attacks to be added. Also, it is scalable as the system can benefit from using more CPUs and additional memory on the same machine, or using distributed servers.

Acknowledgements

First and foremost I thank Almighty GOD, the compassionate, the almighty Merciful, who kindly gave me the strength and His showers of blessing to complete the work on this thesis.

I would like to express my deep and sincere gratitude to my primary supervisor Dr. Alexei Lisitsa. Thank you for your guided directions and your highly constructive criticisms and your commitment to making this thesis excellent. I would also like to thank my second supervisor Dr. Clare Dixon. Thank you for your serious commitment to the research, constructive criticisms, and for providing me with positive feedback as well as invaluable advices. I know one thing for sure, my knowledge is not the same as four years ago.

I am proud and feel honoured that I have completed this thesis in the University of Liverpool. I have enjoyed working in the Computer Science Department during my research. I would like to thanks all staff members and colleagues for their prompt help when needed.

I would like to thank the Saudi Ministry of Higher Education for sponsoring me and providing me with all the means to complete my research and for taking care of all my family needs.

I am also sincerely thankful to StreamBase Systems Inc. for giving us the license and permission to use their products in this research.

Last, and by no means least, I am extremely grateful to my mother for her love, prayers, and caring; but I feel sorry for making her worry about me all the time. I am very much thankful to my wife and my daughters for their love, understanding, prayers and relentless support to complete this research work. Actually, their support started when they asked me to step back to education and pursue my PhD degree; it is amazing how things proceeded after that. Also, I would like to thanks all my brothers, sisters, and all my other family members for their love, encouragement, and continuous support.

Chapter 1

Introduction

Intrusion detection is the process of monitoring the events occurring in computer systems and networks and detecting the actions that attempts to compromise confidentiality, availability, and integrity of a resource [90]. Intrusion detection system (IDS) is software that automates this process. When the software takes preventive measures automatically it becomes an intrusion prevention system (IPS). Combining both the functionality of intrusion and prevention gives us what we call the Intrusion Detection and Prevention System or *IDPS* [82].

This thesis proposes developing an online network based intrusion detection system for high speed networks using temporal logic and stream data processing technology. Temporal logic is a logic with operators that represent change over time. We can use temporal logic as a notation to specify attack patterns or normal behaviour of protocols. The temporal formalisms are translated into stream queries which are executed by a stream data processing (SDP) engine to detect and monitor intrusion attempts. SDP is a recent database technology designed with high performance features for processing intensive flows of data. The results of the case studies and experimental work shows that this approach is a promising solution.

The rest of this chapter is organised as follows. The motivation for the work is presented in Section 1.1. The research contribution is presented in Section 1.2, and the scope of work is presented in Section 1.3. Section 1.4 outlines the thesis structure. Finally, a summary of this chapter is given in 1.5.

1.1 Motivation

The increased speed of today's networks (typically Gbits/sec) introduces new problems in the area of computer security. One of these problems appears in the intrusion detection capabilities of existing systems running on customary hardware (i.e., generic PCs or servers). Recently, IDSs have been unable to provide an effective security mechanism for defending high speed networks [25, 48]. Existing Networks Intrusion Detection Systems (NIDSs) can barely keep up with bandwidths of few hundred Mbps, whereas nowadays, the network speed on Ethernet can reach 10 Gbps [48]. The performance of the two

predominant network based open source NIDSs, Snort [87] and Bro [50] in operational Gbps environments was studied in [25]: Snort quickly consumed all the available CPU, while Bro used all the available memory. The performance tests were conducted on both stateless NIDS (i.e., inspecting a single packet at a time for attacks) and stateful NIDS (i.e., inspecting packets of the same session for attacks). Snort is a stateless NIDS, whereas Bro is a stateful NIDS. For stateless NIDS, the CPU load was the limiting factor which is correlated with the types of analysis (e.g., payload inspection) as well as the traffic characteristics (e.g., bursty traffic). For stateful NIDS, the available memory is consumed because stateful NIDS maintains an in-memory representation of the current state of the network at all times. This state provides the context necessary to evaluate network events. The size of the states increases as the traffic volume increases, and is constrained by the system's available memory. The results of these studies indicate that it is no longer possible to have a stateful or even stateless analysis of all the packets that are monitored by a NIDS. To solve this problem the following solutions have been considered:

- data reduction techniques (i.e., data based approach) [32, 49];
- load balancing, splitting, or parallel processing of traffic (i.e., distributed/parallel execution based approach) [48];
- efficient algorithms for pattern matching (i.e., algorithm based) [23];
- hardware based approach such as using graphics processing units [105] or field-programmable gate array (FPGA) devices [45].

Some work use combinations of the above [23, 45, 106]. The research area is still active and there is no solution that can keep up with the increase in bandwidth. In our research, we are trying to address the problem of IDS performance in high speed networks. How we can build a more efficient system which is capable of handling the increasing amount of traffic? What are the recent advances and new technologies that we can utilize to address this problem? Also, we address the difficulties and non transparent ways for specification of multiple packet attacks (or normal behaviour).

1.2 Thesis Contribution

This thesis contributes to the area of the performance of IDSs in high volume networks addressed in the previous section. In the last decade, a new database technology has emerged that deals with flows of data and high volume or data intensive applications. This technology is called the stream data processing [1, 6, 7, 28, 92]. In addition, we will use temporal logic for the formal specification of attack patterns or normal behaviour. Using temporal logic provides high level declarative, concise, and clear semantics formalisms. The main research question is: "Can stream data processing technology be utilised in conjunction with temporal logic to develop a system that works efficiently

and accurately in high volume networks?" By "efficiently", the following sub-question is addressed: "To what extent the proposed system uses its available resources to detect all attacks in high volume networks?" By "accurately", the following sub-question is addressed: "To what extent is the system successful in detecting all the attacks in the traffic?" Also, this research addresses if the performance and scalability of the proposed system can be extended. The main contributions of this thesis are as follows:

- The combined use of TL and SDP as proposed in this thesis is a novel approach and solution towards the issue of performance of IDS. There are two methods that can be used in devising IDS: misuse based and anomaly based. The misuse based approach is for detecting known attacks and the anomaly based approach is for detecting deviations from normal behaviour as possible attacks. Both of these detection methods are addressed in this thesis.
- Many Sorted First Order Metric Temporal Logic (MSFOMTL) using temporal operators annotated with timing constraints is defined allowing us to unambiguously and concisely represent complex, temporal attacks.
- Extensive experiments are conducted for testing and studying the parallel operations effect on the performance.

1.3 Thesis Scope

The scope of this thesis is summarized as follows:

- The proposed system is a network based IDS that will use the IP packets (headers and payloads) as the source of inputs.
- Correlated distributed events attacks are beyond the scope of this thesis. In this type of attacks, attacks are distributed in space as well as time. The events are from many distributed resources such as logs and alerts from firewalls, intrusion detection systems, operating systems, or other software. This category of attacks we see as an issue by itself and as possible further extension to this research.
- TCP/IP protocol is selected in our case studies because it is the internet communication protocol and commonly used for wide area network. The developed system can be viewed as proof of concept and could be extended to other protocols.

1.4 Thesis Organization

The thesis is organised as follows:

- **Chapter 1 Introduction**: This chapter contains an introduction to the field of intrusion detection and the thesis. It contains information about the research motivation, contributions, the scope of the research, and the thesis structure.

- **Chapter 2 Intrusion Detection Systems**: This chapter provides an overview of intrusion detection systems. It includes information about existing IDSs, explanations of different types of IDSs, and existing methods for intrusion detection. A brief overview of the techniques used in each method are provided. Also, the basic deployment options in switched and non switched environment are presented and explained. Finally, an overview about popular network IDSs is given.

- **Chapter 3 Temporal Logic**: This chapter is about temporal logic and its use in formal specification. It presents the basic theoretical background of using temporal logic in intrusion detection systems. Also, the chapter presents the related work.

- **Chapter 4 Stream Data Processing**: This chapter presents an overview of the stream processing technology. In this thesis, *StreamBase* is used as the stream data processing engine. The available features and capabilities of *StreamBase*is presented and explained. *StreamBase* uses stream SQL (SSQL) as the query language, the semantics of *SSQL* are given.

- **Chapter 5 Temporal Stream Intrusion Detection System (TeStID)**: This chapter presents the proposed system *TeStID*. The formal specifications of attack patterns used in *TeStID* are presented. The abstract view of network communications, the syntax/semantics of MSFOMTL, and the attack specification and their syntactical forms are given. Also, *TeStID* proposed system architecture, descriptions of the tools and utility used for the development, and benefits are provided.

- **Chapter 6 Temporal Logic to Stream Queries**: This chapter provides some background of using temporal logic to query databases. The view of time in the proposed system is presented. Also, the mappings of formulae written in MSFOMTL into *SSQL* and the correctness of the approach are presented. Finally, the developed translator that parses and maps the syntax of MSFOMTL into *SSQL* is presented.

- **Chapter 7 Experiments and Results**: This chapter provides case studies for single and multiple packets misuse attacks. Experiments are described and the results are presented. A discussion of the results is provided along with a discussion of the scalability and performance aspects of the proposed system.

- **Chapter 8 The Potential Use of The New System In Anomaly Based IDS**: This chapter presents a basic overview of anomaly based NIDS. Also, it illustrates how temporal logic is used to formally represent the specifications of parts of protocols. The mapping of the formal specifications and the correctness of the approach are also explained in this chapter.

- **Chapter 9 Conclusion**: In this chapter a conclusion and a discussion of the work done are provided. Finally, future work and further research opportunities are discussed.

1.5 Summary

This chapter has provided an introduction to thesis and described the context of its research. Briefly the research field, intrusion detections, is introduced and the motivations for the research is provided. Further more, the contributions and the scope of the research work which are related to the issue of performance of IDSs in high speed network is declared. Finally, an overview of the thesis structure is provided.

The following chapter presents the background of intrusion detection systems. It provide basic information about the IDS: types of IDSs, the existing methods and techniques for detecting intruders, and the deployment options for IDSs. Also, information about popular NIDSs are given.

Chapter 2

Intrusion Detection Systems

Nowadays, information has become an organization's most precious asset and everything an organization does involves using information in some way or another [74]. The way of conducting business has changed as the world has become more interconnected, and the increase in this connectivity provides access to varied resources of data instantly; moreover, it provides an access path to the data from virtually anywhere in the network-based environment. The internet connectivity is no longer an option for most organizations [90], consequently, securing the business running environment is one of the primary concerns of an IT department that exist in any organization.

In 1980, Anderson's seminal paper [5] introduced the notion of intrusion detection. He proposed the idea that audit trails contain vital information that could be valuable if geared to the function of system security, that is, tracking user unauthorized use of the resources. This work has the following impact:

- It focused attention on the deficit of security in the computer system environments.
- It introduced new notions and definitions (e.g., notion of intrusion, intruder definition and classification).
- It formed the basis for the development of IDS.
- It suggested the need for more research in this field.

During the following 30 years, more research were conducted and intrusion detection systems were developed. Intrusion Detection Systems aim to detect the actions that attempt to compromise the confidentiality, availability, and integrity of a resource by monitoring the events occurring in computer systems and/or networks. These attempts could be from intruders either from inside or outside the organization [76]. According to Price [76] FBI studies have revealed that 80% of intrusions and attacks come from within organizations. Network security approaches include building firewalls to protect the internal systems and networks, using an intrusion detection system to detect an intrusion, having strong authentication approaches, and using encryption to protect sensitive data as they transit the network [56]. Authentication systems or firewalls

can not prevent legitimate users from carrying out harmful operations on computers or networks. This functionality is provided by the IDS. Some of the current IDSs can operate in either passive or active mode. In passive mode, the IDS monitors and detects intrusion attempts. In active mode, the IDSs monitors, detects, and takes specified preventive measures automatically. In the latter case, the system is called an intrusion prevention system (IPS). Normally, the function of IPS could be enabled or disabled by security administrators [82].

Section 2.1 presents information about types of IDS. Intruder detection methods are presented in Section 2.2. In Section 2.3, the deployment options for network based intrusion detection system are provided. Section 2.4 presents information about some of the popular open source or commercial Network IDSs. Finally, a summary is given in Section 2.5.

2.1 IDS Types

There are two main types of IDS: *Host-based Intrusion Detection System* (HIDS) and *Network-based Intrusion Detection System* (NIDS). HIDS reside on a single host and monitor all events for suspicious activity. Examples of events a host-based IDS might monitor are network traffic (only for that host), event logs, system logs, running processes, file access and modification, and system and application configuration changes. The role of HIDS is to flag any tampering with a specific host and it can automatically replace altered files when changed to ensure data integrity. These systems vary from vendor to vendor, but they are usually system centric in their analysis. Most host-based systems have agent detection software [82] and can be deployed as standalone or distributed in which each agent monitors the activity on a single host and transmits the data to management servers, which may optionally use database servers for storage. Each agent is typically designed to protect one of the following:

- A server: agents monitor a server's operating system (OS) and some common applications.
- A client host (desktop or laptop): agents monitor the OS and common client applications such as e-mail clients and Web browsers.
- An application service: agents perform monitoring for a specific application service only, such as a Web server program or a database server program.

The other type is *Network-based Intrusion Detection System* which resides on the network, and it is designed to monitor network traffic for particular network segments or devices and analyses the network and application protocol activity to identify suspicious activity. The NIDS examines the traffic packet by packet in real time, or close to real time, to attempt to detect intrusion patterns.

The NIDS is deployed with sensors. Sensors are agents that monitor networks on a real time basis. They are available in two formats: appliance based and software based.

An appliance-based sensor is comprised of specialized hardware and sensor software. The hardware is typically optimized for sensor use, including specialized network interface cards and drivers for efficient capture of packets, and specialized processors for the analysis of traffic. Additionally, parts or all of the IDS software might reside in firmware for increased efficiency. The second sensor format is the software only sensors. This software can be installed onto hosts that meet certain specifications. It might include a customized OS, or it might be installed onto a standard OS just as any other application would [82]. If more than one sensor is used, one or more servers and consoles for management functions are used also in the deployment.

2.2 Intruder Detection Methods

There are two main approaches to devise an intrusion system to detect intruders: anomaly based and misuse based detection systems. In anomaly based detection, intrusions are identified as unusual behaviour that differs from the normal behaviour of the monitored system. In misuse based detection methods, intrusions are detected by matching the events that occur in the monitored system with a predefined pattern of known attacks.

Each of these two approaches has advantages and disadvantages. The main advantage of anomaly based IDSs is their ability of detecting new attacks. In these systems, the intrusive activity generates an alarm because it deviates from normal activity, not because the system is looking for specific traffic. Thus, these deviations could be unknown or new attacks. The main disadvantage of anomaly IDSs is that they tend to be computationally expensive because several metrics are often maintained that need to be updated against every system activity. Another disadvantage is that the anomaly IDSs may be gradually trained incorrectly to recognize an intrusive behaviour as normal in the future due to insufficient data. Furthermore, there is no guarantee that an attack will generate an alarm if the intrusive activity is too close to normal activity. Consequently, these factors affect their detection rate to be low and their false alarm rate to be high. On the other hand, misuse based IDSs have the limitation that they look only for known attacks, and may not be of much use in detecting unknown intrusions. On the advantage side the misused IDSs have a very high detection accuracy and very low false alarm rate.

Different methods have been proposed for detecting intrusions based on the anomaly based or the misuse based approach. In the following subsection we will summarize each method and its classification based on the employed techniques.

2.2.1 Techniques of Anomaly Based Intrusion Detection

There are many anomaly detection algorithms proposed in the literature that differ according to the information used for analysis and according to the techniques used to

detect deviations from normal behaviour. The techniques found in the literature can be classified as follows:

- **Statistical methods** in which the user, system, or network behaviour is monitored by measuring certain variables over time (e.g., login and logout time of each session). The basic models keep averages of these variables and detect whether thresholds are exceeded based on the standard deviation of the variable [33, 65].
- **Rule based systems** in which the normal behaviour of users, networks and/or computer systems are summarized by a set of rules and anomalous behaviour are detected as deviations from them. Examples of rule based IDSs include ComputerWatch [24] and Wisdom & Sense [51].
- **Distance based** approaches attempt to overcome limitations of statistical outlier detection approaches in higher dimensional spaces where it becomes increasingly difficult and inaccurate to estimate the multidimensional distributions of the data points and they detect outliers by computing distances among points [2].
- **Model based** in which anomalies are detected as deviations for a model that represents the normal behaviour. Very often, researchers have used data mining techniques such as Neural Networks to model the normal behaviour of individual users, to build profiles of software behaviour or to profile network packets and queue statistics [30, 80].
- **Profiling methods** in which profiles of normal behaviour are built for different types of network traffic, users, programs, etc., and deviations from them are considered as intrusions. Profiling methods vary greatly ranging from different data mining techniques to various heuristic-based approaches. *ADAM* (Audit Data and Mining) is a hybrid anomaly detector trained on both attack-free traffic and traffic with labelled attacks. The system uses a combination of association rule mining and classification to discover attacks in TCPDUMP data [8].

2.2.2 Techniques of Misuse based Intrusion Detection

The concept behind misuse detection schemes is that there are ways to represent attacks in the form of a pattern or a signature so that even variations of the same attack can be detected [41]. Attacks are encoded as a set of footprints, which are patterns that occur every time an attack takes place. The implementation of such systems usually involves an expert system that performs the matching against the stored rule-base. The misuse detection approaches can be classified into the following four main categories:

- **Signature-based methods** in which monitored events are matched against a database of attack signatures to detect intrusions. Signature-based IDSs are unable to detect unknown and emerging attacks since the signature database has to be manually revised for each new type of intrusion that is discovered [85].

- **Rule-based systems** use a set of "If-Then" implication rules to characterize computer attacks [54].
- **Transition based** in which state transition analysis requires the construction of a finite state machine, in which states correspond to different *IDS* states, and transitions characterize certain events that cause IDS states to change. IDS states correspond to different states of the network protocol stacks or to the integrity and validity of current running processes or certain files. Every time the automation reaches a state that is flagged as a security threat, the intrusion is reported as a sign of malicious attacker activity [107].
- **Data mining methods** in which each instance in a data set is labelled as "normal" or "intrusive" and a learning algorithm is trained over the labelled data. These techniques are able to automatically retrain intrusion detection models on different input data that include new types of attacks, as long as they have been labelled appropriately [22, 61].

In this thesis we propose developing an online NIDS for both misuse and anomaly based intrusion detections. In misuse based intrusion detection, we use signature based detection method. We use temporal logic for formal representation of attack signatures that will be checked against the incoming events which form the temporal models. For anomaly based intrusion detection, we use model based detection method. We use temporal logic to model the normal protocol behaviour and detect any deviation. *MONID* [62] and *ORCHIDS* [69] are two NIDSs that use temporal logic for signature based intrusion detection. Also, *MONID* uses statistical based method for anomaly intrusion detection. Both of these systems are discussed in Sections 3.2.1 and 3.2.2.

2.3 NIDS Deployment

Network based IDS monitors network traffic destined for all the systems in a network segment. A sensor can be deployed for each network segment. Basic deployment options are explained here along with the commonly used terminology.

In non switched networks or hub networks, the traffic coming to one port is copied to all other ports. Thus, the network sensor device can be connected to any port on the hub (see Figure 2.1(a)). The sensor device is either a commodity computer or server that runs the intrusion detection software or a high performance device specially manufactured with highly integrated number of CPUs, large amount of memory, many network interfaces, and usually is designed to be high tolerant device and extensible.

In switched network environment where connection exists between communicating points NIDS are usually implemented using a Test Access Point (TAP) device or Switch Port ANalyser (SPAN ports). SPAN is a mirror port on the switch that has a copy of the data that goes through other ports. The NIDS sensor can be connected to this port to monitor intrusions as illustrated in Figure 2.1(b). TAP is a passive traffic

splitting mechanism installed between the sensor device and the network. Any data pass through the TAP is passed to the sensor as well. Figure 2.1(c) shows this type of deployment where the sensor is monitoring inter-switch traffic or connection to the internet or intranet. As shown in Figure 2.1(d) we can use both TAP and SPAN to deploy NIDS. In this setup, we use one sensor but we could use two: one is connected to the TAP and one is connected to the SPAN port. If we use one sensor to monitor multi network segments then we would use a hub to connect these segments with the sensor.

It is worth mentioning that the TAP implementation can not monitor intra-switch traffic. Hence, using the SPAN port for leaf nodes and the TAP for inter-switch trunks is a common practice. Regardless of how the sensor is implemented, it needs to passively listen to all the traffic. This means that the network interface needs to work in promiscuous mode. In promiscuous mode, all the traffic that passes through the network interface are passed to the core machine. In non-promiscuous mode only the traffic addressed to the network interface is passed.

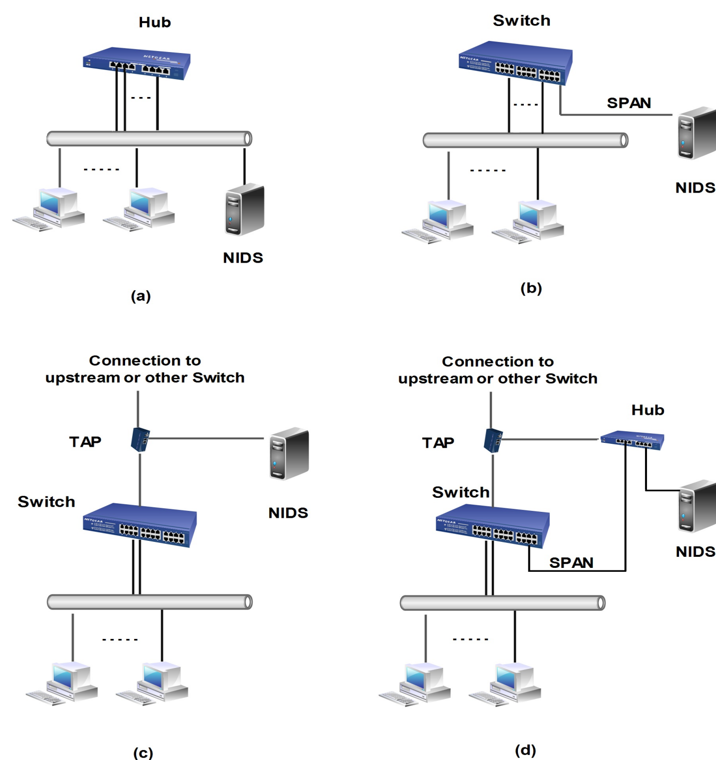


Figure 2.1: The NIDS Deployment Options. Figure 2.1(a) shows NIDS in non-switched networks, Figure 2.1(b) shows NIDS in switched networks using SPAN port, Figure 2.1(c) shows NIDS deployment using TAP, and Figure 2.1(d) shows NIDS deployed using both SPAN and TAP.

There is a difference in deploying intrusion systems for detection only or for detection and prevention. In intrusion detection and prevention system the sensor must be able to block network traffic to stop attacks. Hence, the sensors are deployed inline so that the network traffic it is monitoring must pass through it. In the next section, some of

the popular NIDSs can operate in this mode and prevent attacks. Later in Chapter 8, we will see how using inline mode is effective as virtual software patch.

In intrusion detection only systems the sensor is deployed in passive mode where no traffic is passed through the sensor, that is, only a copy of the real traffic is monitored. Later in Chapter 7, we will explain the experiment setup and we will see that it is actually equivalent to Figure 2.1(d), that is listening in passive mode to internal and external replayed traffics.

2.4 Popular Network Intrusion Detection Systems

In recent years, network based intrusion detection systems have attracted more attention due to the increasing use of internet. The data centric world needs security protection against cyber attacks. NIDS can provide this security measure as it can monitor network packets exchanged between computer systems. Popular NIDS that exist today can be classified into two categories: open source NIDS and commercial NIDS. The following subsections present detailed information about some of these NIDSs.

2.4.1 Open Sources NIDS

Open source NIDSs have been available for more than a decade ago. Snort [17] and Bro[50] are the oldest and most popular NIDSs known today [103]. In 1998, Snort was developed and released by Martin Roesch [87]. According to the Snort organization site, over 4 million downloads and more than 400,000 registered users show the wide popularity of Snort as a deployed IDS solution. Bro, was developed by Vern Pxson in 1999 [73] primarily as a research platform for intrusion detection and traffic analysis. It is not intended to be an “out of the box” solution. This means that Bro is not readily available to be deployed. It has no subscription service where you can download new attack signatures like what the Snort community provides. Research into Bro is ongoing and Vern Paxson continues to lead the project jointly with a core team of researchers and developers at the International Computer Science Institute in Berkeley, CA.

Another interesting open source NIDS is Suricata developed by the Open Information Security Foundation (OISF) [68]. OISF is a non-profit foundation part of and funded by the Department of U.S Homeland Security’s Directorate for Science and Technology, by the the Navy’s Space and Naval Warfare Systems Command, as well as by members from the industry. The Suricata Engine, as the authors describe, is:

> Next Generation Intrusion Detection and Prevention Engine. This engine is not intended to just replace or emulate the existing tools in the industry, but will bring new ideas and technologies to the field.

A beta version was released in December 2009, with the first standard release following in July 2010 [68].

Even though all of these systems are network based intrusion detection, they vary in their approach in monitoring and detecting intrusions, features, and capabilities. In the following sections, we will look into these NIDSs in more detail.

2.4.1.1 Snort IDS

Snort is an open source network intrusion detection and prevention system. It performs a packet level traffic monitoring and analysis. These packets are examined for matching attack signatures and for protocol anomaly. Snort has a powerful signature matching ability and an easy to code and expand signature system. It has a broad community which contributes by adding new rules (Snort refers to signatures as rules) and suggests improvements.

Snort can run in one of three operations mode available:

- Sniffer mode, which simply reads the packets and displays them in a continuous stream on the console.
- Packet Logger mode, which logs the packets to disk.
- Network Intrusion Detection System mode, which allows Snort to analyse network traffic for matching attack signatures or to detect protocol anomalies. If Snort deployed inline with the IDS mode, then prevention actions can be configured to be launched when an attack or an anomaly is detected.

The Snort architecture is modular and consists of a packet decoder, preprocessor, detection engine, and output plug-ins. Snort does not have native sniffer and uses LIBPCAP[1] to capture packets from the network interface device. The decoder upon receiving the raw data through LIBPCAP decodes the protocol packet elements at the data link, network, and then at the transport layers. These data packets are then stored in a data structure which is ready to be processed by the preprocessor and detection engine. The preprocessor consists of functions that can be called to make sure that packets are normal or as it should be. When Snort is deployed in inline mode the malformed packets can be configured to be dropped. The detection engine is the primary component of Snort. It builds attack signatures by parsing Snort rules. The detection engine matches the attack signatures with the packets it receives from the processor and the outcome or matched signatures are sent to the output plug-ins. There are many plug-ins that can be used for the output such as XML, CSV, or Database.

Another interesting feature of Snort is the Snort rules. Snort uses a simple description language that is easy to learn and use. The rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rule's action (e.g., alert, log, and drop), protocol, source and destination IP addresses and net masks, and the source and destination ports information. The rule option section contains

[1]LIBPCAP is free software library developed by Lawrence Berkeley Laboratory. It consists of an application programming interface (API) which can be used by other programs such as IDS to capture network traffic.

alert messages and information on which parts of the packet should be inspected (e.g., payload, flags, and headers) to determine if the rule action should be taken. Figure 2.2 shows an example with the two logical structures. In the example, the rule is:

```
alert tcp any any -> any 25 (msg:"SMTP expn root", flags:A+;
      content: "expn root"; nocase; classtype:attempted-recon;)
```

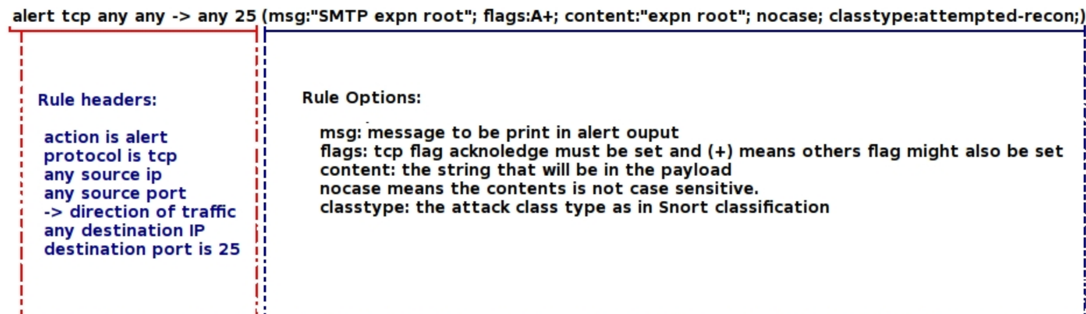


Figure 2.2: An Example of snort Rule

In this example, the action is *alert* which means an alert will be generated when there is a match, the protocol where this rule applies to is *tcp*, the source ip is *any*, the source port is *any*, "->" is the direction of communication (i.e., from to), the destination ip is *any*, the destination port is *25*, the *msg* is what to be printed when the alert is generated, *A+* means the Acknowledgement tcp flag is set and optionally other flags, *content* specifies the packet payload data to match, and *classtype* is the internal classification used for this attack in Snort. More about the Snort rule syntax with some examples are given in Section 7.2. A full list of the rule options is in [17, 87].

2.4.1.2 Bro NIDS

Vern Paxson [73] described Bro as a standalone system for detecting network intruders in real time by passively monitoring a network link over which the intruder's traffic transit. Bro has many interesting features as a NIDS. Like Snort, Bro can work on packet per packet base when detecting attacks. In fact, it has Snort compatibility support that can convert Snort signature into Bro signature. In addition, Bro maintains state information of the communication protocols and store them as a related records of events and can detect attacks over multiple packets.

The system architecture of Bro has three major components: a packet capturing and filtering, an event engine, and policy script interpreter. Similar to Snort, Bro uses the LIBPCAP packet library to capture and filter the traffic. The filtered packets are then passed along to Bro's event engine, which converts the filtered stream into higher level events. Finally, Bro's policy script interpreter executes custom scripts written in the Bro scripting language. These scripts express a site's security policy, i.e., what actions to take when the monitor detects certain types of activity. The scripting language allows Bro to track detailed information (or events) about the network's activity and generate real time alerts. The following code is an example of the policy script:

```
global attempts: table[addr] of int &default=0;
global threshold = 50;

event connection_rejected(c: connection)  # Whenever a connection is rejected
{
local source = c$orig_h;           # Get source address of the connection
local n = ++attempts[source];      # Increase counter.

if ( n == threshold )          # Check for threshold.
NOTICE(Scanner, source);       # If so, report.
}
```

Two global variables are defined: (attempts) is a table of IP addresses and (threshold) is assigned the value 50. The script increments the counter (attempts) whenever a connection_rejected event occur. The connection_rejected event occurs whenever a server rejects a TCP connection request from a client. When a connection is rejected, the (attempts) counter for that source address (orig_h in the connection record (c)) is incremented by one and assigned to the local variable (n). If (n) is equal to 50 (the threshold) then the source will be reported through the NOTICE output facility of Bro. This example is simple and the policy script becomes more complicated when the number of processed events increases. For each multiple packet attack a policy file must be written specifically.

2.4.1.3 Suricata IDS

Suricata is an open source NIDS that was officially released in July, 2010. The primary goal of OISF as they state is to remain at the leading edge of open source IDS/IPS development, community needs, and objectives.

The Suricata NIDS goal is to bring new technology to the IDS. The engine support multithread processing to benefit from the multiple cores and multiple CPUs systems that are becoming ubiquitous these days. Also, to achieve higher performance they use hardware acceleration. NVIDIA, a technology partner of Suricata, developed CUDA which is a parallel computation library that can be called to offload some computation to GPU (Graphic Processing Unit).

In terms of detection Suricata is a misuse based and protocol anomaly based IDS. The Snort rules are supported in Suricata and can be used as they are or they could be modified into Suricata optimized rule sets to take advantages of the Suricata engine. Protocol anomalies are provided with Suricata preprocessors and if it is deployed inline, then prevention functionality could be activated.

2.4.2 Commercial NIDS

Many commercial network based intrusion detection systems are also available. These systems are either appliance (hardware) based or software only based. RealSecure Server

Sensor from IBM Internet Security Systems is one of the popular NIDS software based system [38]. RealSecure monitors, detects intrusions, and can also prevents intrusions by blocking network packets. IBM also has an appliance based solution which is the IBM Security Network Intrusion Prevention System [36]. Another appliance based NIDS is the CISCO IOS Intrusion Prevention System [15]. This product can be installed on any of the CISCO IPS 4200 series which scales up to meet the business performance requirements. These IBM and CISCO products are signature based NIDSs that can detect known attacks and can prevent protocol anomaly based traffic. A more recent NIDS is Endace [26]. In 2007, Endace launched appliance based NIDS. It uses the Linux-based operating system and the open source Snort inspection engine. Beside its use for intrusion detection, Endace can be used for network forensics. It has analytic application that allows IT security professionals to understand and review what really happened on their networks. Endace appliance has 32 terabytes on board traffic buffer that enables back-in-time contextual analysis of events.

Enterasys Intrusion Prevention System (also known as Dragon IPS) is another NIDS from Enterasys Secure Networks [27]. Enterasys IPS is unique in its ability to be deployed as network based or host based IDS. Enterasys' Distributed Intrusion Prevention (US Patent 7581249) and threat containment can block attackers at the source physical port. Effective threat containment requires the removal of the attacker's ability to continue the attack or to mount a new attack. The Enterasys Distributed IPS identifies a threat or security event, locates the exact physical source of the event, and mitigates the threat through the use of enforceable bandwidth rate limiting policies, quarantine policies, or other port level controls.

2.5 Summary

This chapter provided an overview of intrusion detection systems. There are two types of IDS: host based and network based. Host based IDS resides on a server or host and protect that host through using the audit files on that host and/or the traffic it receives from other systems. Network based IDS resides on the network and monitor packets for intrusion attempts.

Two main detection methods exist for intrusion detection. The first method is misuse based in which the attack signatures or patterns are known and the IDS system tries to match it. The second method is anomaly based in which the normal behaviour of the system is modelled and any deviation from this normal behaviour is raised as an alarm of possible attack. Both of these detection methods have techniques that have been used to devise IDS. A brief overview of these techniques was provided with some examples.

The basic deployment options in switched and non switched environments were presented and explained. Also, passive deployment or active (inline) deployment were discussed. Inline deployment is a must if we need to perform intrusion prevention.

Finally, an overview is given about commonly used NIDS. An overview was given for the popular open source and free NIDSs (Snort, Bro, and Suricata) and some commercial NIDSs.

The next chapter is about temporal logic and its use in formal specification and system verification. The syntax and semantics are defined for the formal representations of attacks in misuse based IDS or the representations of normal specifications in anomaly based IDS.

Chapter 3

Temporal Logic and Intrusion Detection System

3.1 Why Temporal Logic?

Developing formal methods for specification and automatic verification of concurrent and real-time systems has been an active computer science research area. Verification techniques are concerned with showing that a system satisfies its specification. Some of those verification techniques allow checking whether an execution of a system under scrutiny satisfies or violates a given property.

Logical-based formal methods are commonly used in runtime verification techniques. The reasons are mainly because they provide unambiguous and concise ways to formally represent system specifications and provide the necessary mechanisms to reason about given properties. Temporal logic is the extension of classical logic with operators that deal with time which allows us to formally specify temporal events. In dealing with real-time systems quantitative temporal properties play a dominant role. Koymans [47] introduced the Metric Temporal Logic (MTL). In MTL, the qualitative temporal operators are turned into quantitative or metric temporal operators. This transformation from qualitative into quantitative is done in MTL by constraining the temporal operator with bounded or unbounded intervals. For instance, $\Diamond_{[0,5]}\phi$ means eventually ϕ will be true within 0 to 5 seconds from now. This metric extension to temporal logic is very useful in relating event occurrences in real-time systems.

The next section shows how temporal logic can be used in intrusion detection systems and presents previous work using temporal logic for intrusion detection.

3.2 Related Work

In Chapter 2, two common approaches toward devising IDSs were explained: misuse based and anomaly based. In misuse based systems the attack patterns are monitored in the incoming traffic and an alarm is raised if an attack pattern detected. In anomaly based deviations from normal behaviour is monitored in the incoming traffic and an

alarm is raised as a possible attack if a deviation from a specified normal behaviour is detected.

In intrusion detection systems, temporal logic is used to formally represent attack patterns or normal behaviour. The theoretical base of using *TL* in misuse based detection can be viewed as checking whether a formula representing the signature of the attack is satisfied in a model M. These temporal models are created from a linear sequence of events (packets). So, typically, the idea in misuse based detection is to check that ϕ (an attack specification) holds in M ($M \models \phi$). These signatures are the attack patterns that we need to match against network traffic and raise an alarm when detected. For anomaly based detection, normal protocol procedures or network behaviour are formally represented using temporal logic. A property ϕ formally representing a normal procedure or a network behaviour is checked against the temporal models M that were created from the incoming network traffic. An alarm is raised if ϕ does not hold in M ($M \not\models \phi$).

The following subsections present two related work *MONID* [62] and *ORCHIDS* [69] that use temporal logic in network intrusion detection.

3.2.1 MONID

MONID [62] is a prototype tool based on *Eagle* [9]. *Eagle* is a runtime verification or runtime monitoring system that uses finite traces. The implemented finite trace monitoring logics in *Eagle* include future and past time temporal logic, extended regular expressions, real-time logics, interval logics, forms of quantified temporal logics, and so on. Simply, it is an observer which monitors the execution of a program and checks its conformity with a requirements specification, often written in a temporal logic or as a state machine.

The safe behaviour specification of a system is continuously monitored by *Eagle* in a state-by-state basis, without storing the execution trace. An execution trace σ is a finite sequence of program states $\sigma = s_1 s_2 ... s_n$, where $|\sigma| = n$ is the length of the trace. A trace s_i, where $i > 1$, satisfies a specification if each monitored formula for that specification is satisfied from state s_1 up to state s_i.

In *Eagle*, the specifications are written as rules. These rules are recursively parameterized by both logical formulae and data values, over a set of three primitive modalities "next-time", "previous-time", and "concatenation". The "next-time" and "previous-time" operators can be used for defining future time and past time temporal logic, respectively. The "concatenation" operator can be used to define interval logics and an extended regular expression language.

Naldurg et al. [62] propose the use of *Eagle* in building a prototype online intrusion detection system, which they called *MONID*. In *MONID*, linear temporal logic (LTL) with real time constraints and statistical properties are used to represent a safety formula ϕ (specification of the absence of an attack) and the system continuously evaluates ϕ against a model M representing a finite sequence of events. Whenever ϕ is violated an intrusion alarm is raised (i.e., $M \not\models \phi$). The work in [62] showed how *Eagle* can

be used to express security attacks with complex temporal event patterns, as well as attacks whose signatures are inherently statistical in nature. The authors also see that *MONID* can be used for simple type of anomaly based detection, that is, by specifying the normal behaviour of systems or networks as temporal formulae involving statistical predicates, and monitor the system execution to check if it violates these formulae. If the observed execution violates any formula then an alarm is raised. This alarm means that an intrusion has likely occurred.

The framework of *MONID* consists of three functions: information preprocessing, event monitoring, and reporting. The preprocessing function starts by gathering the information from various sources into a server. Then these gathered information are merged by timestamp to form a single event trace. Subsequently, this timestamped stream of events is monitored by the event monitoring function against a given specification and an intrusion alarm is raised by the reporting function when the specification is violated.

The safety formula or monitored formula (M) uses data-values and parameterized recursive equations and it is typically specified in terms of maximal and minimal fixed points of interpretation as follows:

$$\text{max: } \Box F = F \wedge \bigcirc(\Box F)$$

$$\text{min: } \Diamond F = F \vee \bigcirc(\Diamond F)$$

$$\text{mon: M } = \Box(\text{max} \rightarrow min)$$

($\Box$ = always, $\Diamond$ = eventually, $\bigcirc$ = next)

One of the given example in [62] is the “cookie-stealing” attack. A cookie is a session-tracking mechanism issued by a web server to a client and store client session information. An attack occurs when a malicious user hijacks a session by reusing an old cookie issued to a different IP address. The formula below asserts that a particular cookie must always be used by the same IP address.

$$\text{min: SafeUse(string c, int i)} = ((\text{name} = \text{c}) \rightarrow (\text{ip} = \text{i})) \wedge \odot\, \text{SafeUse(c, i)}$$

$$\text{mon: CookieSafe} = \text{Always(SafeUse(name, ip))}$$

A Cookie-stealing attack is detected whenever the monitored formula CookieSafe is violated. SafeUse is a parametrized rule that corresponds to an event with name (cookie name) and IP address of the client. Notice here also the previous operator $\odot$ is used.

This example along with the other examples in [62] show how *MONID* is expressive in representing complex temporal events with time constraints and statistical properties. On the other hand, specifying safety formulae is not an easy task. There is no systematic method in defining events, this is very clear by looking into the examples in [62] as events have different schemas that need to be defined specifically per attack. Additionally,

defining the named rules and their interpretations as maximal or minimal and defining the monitoring named rule is not straight forward as these rules contains recursive equations. This would be clear if we try to define an attack signature with many steps. The more steps we have, the more complicated these rules become to code and follow.

MONID, as the authors explain, was mostly tested offline to detect attacks and to calculate the overhead of detecting attacks. They found the overhead is low and the authors concluded that this suggests *MONID* could be used as an online NIDS. No information about the volume of traffic during the experiments was given, and as far as we know the purpose of *MONID* was not to detect attacks in high volume networks. Thus, it is hard to compare the experimental results to ours as we use online and multiple speed replays of the data (in Chapter 7).

3.2.2 ORCHIDS

ORCHIDS [69] is another intrusion detection system that uses temporal logic. This research project goal was to design and develop a prototype of an online IDS which is capable of analyzing and correlating events over time. *ORCHIDS* is specialized in the intrusion detection by scenario or sequence recognition. It provides a new correlation methods based on efficient model checking of temporal logic. Correlation is the process of recognizing event positions in time and the relations among them. *ORCHIDS* as described in [69] uses misuse based detection approach. Bad behaviour or attacks is specified in temporal logic and alerts are notified whenever a bad behaviour is detected. This is essentially a model checking problem against linear Kripke models which are formed from linear sequences of events.

In *ORCHIDS*, first order linear time temporal logic is extended with Wolper style LTL [79]. This is done to increase the expressiveness of the logic to describe properties of sequences that cannot be expressed in temporal logic. One example of properties that can not be expressed in classical LTL is event count. Another example is when expressing a given event that has to happen exactly every n steps. In general, Wolper style LTL allows the representation of properties that are expressible in language based on regular expressions.

The architecture of *ORCHIDS* [67] is composed of five main parts: a set of rule defined using the logic we mentioned earlier, a compiler which translates rule definitions into an internal automata representation, a set of compiled rules which is the knowledge base of the whole system, a massively parallel virtual machine which simulates non deterministic finite automata, and a set of input modules responsible for decoding incoming data from external sources. There are two kinds of input sources : real time and polled inputs. Real time input sources notify their events autonomously (e.g., receiving messages in real time via the UDP network protocol). The second kind of input sources are polled input sources which are events data that need to be checked and retrieved periodically (e.g., system and network logs files).

Temporal formulae are typically of the form:

$$F_1 \wedge \Diamond(F_2 \wedge \Diamond(F_3 \ldots) \vee F_2^{'} \wedge \Diamond(F_3^{'} \ldots))$$

where the temporal operator $\Diamond$ is "there exist in the future". The attack signatures are represented with formulae which are described internally in the system as automata. The *ORCHIDS* online algorithm matches these formulae against the logs and returns enumerated matches. *ORCHIDS* deals with finite traces and are constrained to only specify eventuality properties (a transition denotes either no time passes at all or "eventually in the future" $\Diamond$, and no next $\bigcirc$). This is because the model-checker needs to work on-line on some finite (and expanding over time) prefix of infinite sequence of events. The model checker is not allowed to make multiple passes over the flow of events. This means a product automaton cannot be rebuild each time a new event is added.

Similar to *MONID*, *ORCHIDS* testing was for the proof of concept and not for detecting attacks in high volume networks. The sequence recognition in *ORCHIDS* is equivalent to the pattern matching using the third syntactical form of the proposed method of specification in this thesis which is presented in Chapter 5. In that chapter, we explain the proposed Many Sorted First Order Metric Temporal Logic (MSFOMTL) syntax and semantics for representing temporal patterns of attacks or normal behaviour.

The main differences between the system we described in this thesis as compared with *MONID* and *ORCHIDS* is that the model checking problem ($M \models \phi$) is reduced to the stream query evaluation, which is subsequently executed by high-performance *SDP* engine. Also, in the proposed system, the way of using temporal logic takes advantage of its expressiveness and conciseness to allow the user to express attack signatures or normal specifications transparently and independently from the underlying technical implementations. In addition, using predefined syntactical forms for attacks, give the benefit of producing *SSQL* code that does not require retesting beyond the testing of the system that were carried following the development of the translator from these syntactical forms.

3.3 Summary

This chapter was about temporal logic and its use in representing attack patterns in IDS. The basic theoretical background of using temporal logic in intrusion detection systems was explained. Also, the use of temporal logic in two related work were presented and explained: *ORCHIDS* and *MONID*.

The next chapter presents information about stream data processing technology. *StreamBase* is used for implementing *TeStID*. The high performance features and scalability of *StreamBase* are presented and explained. Also, the semantics of stream SQL, the query language of *StreamBase* is given.

Chapter 4

Stream Data Processing (SDP)

The previous chapter presented information about the background of using temporal logic in intrusion detection system and the related work. In this chapter we present the stream processing technology which we make use of as the execution engine in our proposed solution. Section 4.1 presents an overview of the background, the main features of these systems, and the current systems. *StreamBase* is one such system which is used to develop the proposed NIDS in this thesis. *StreamBase* uses Stream SQL as the stream query language. An overview of Stream SQL language and its semantics are provided in Section 4.2. Section 4.3 presents the high performance features, scalability, and high availability of *StreamBase*. Finally, a summary for this chapter is provided in Section 4.4.

4.1 SDP Overview

Stream Data Processing is referred to in the literature with different names, such as, stream query processing [44], stream database manager [95], data flow database [10], data stream management system [7], complex event processing [92], etc. Regardless of the different names used, all these methods are concerned with handling and processing flows of data. The data stream can be defined as a sequence of events that arrive continuously in real-time. These sequence of events are ordered by their arrival time either implicitly or explicitly by a time stamp [31].

The fundamental difference between traditional database systems and Data Stream Management System (DSMS) is that the data takes the form of continuous data streams rather than finite stored data sets. This difference makes DSMSs suitable for data-intensive applications where the data model is transient data streams and not persistent relations. These applications could be the capital market, network traffic monitoring, telecommunication data management, and others. The difference between the DBMS and the DSMS with respect to time processing is that the DSMS are suitable for applications that require real time processing. This is due to the reason that DSMS can work totally in main memory.

DSMS are designed to handle large volumes of data arriving in rapid, time-varying and continuous streams. They can handle queries that are issued once and then continuously evaluated over the data (continuous queries). For example, "write an alert whenever price of X is less than 200". Another useful feature is sliding window query processing. This sliding window can be based on an ordered field (e.g., time) or tuple count. With this feature, recently arrived data is maintained, meaning that old data must be removed as time goes on. It is an approximation technique for bounded memory that we can use to extract a finite relation from an infinite stream and to compute online statistics. These features are well suited for applications like network monitoring, network traffic analysis, and intrusion detection. Multiplexing and demultiplexing are features that are provided by DSMSs and it is very useful for stream processing. demultiplexing is the process of partitioning the stream into substreams based on the data in each record (tuple). Multiplexing is used to recombine substreams with the same data structure. Other useful features are aggregation, pattern matching, merging, and mapping the stream based on a value(s) in the data.

Using stream data processing become a programming paradigm in its own because of the natural requirements of processing ordered sets, that is data streams [44]. According to Parker et al. [44], it is advantageous to generalize the set foundation of relational data model to an ordered set model because ordered data can be processed more efficiently than unordered data. Other reasons related to high bandwidth traffic analysis are mentioned by Sullivan and Heybey [96] in their work of building *Tribeca* which is a database stream management system that deals with network traffic are as follows:

- Fast sequential access to network traffic data is crucial but unlike conventional relational database systems transactional updates, fast access to random records, and concurrency control access are not.
- Traffic data is used few (or once) and the load time is significantly costly.
- The traffic traces contain many small records with few bits size and this noticeably increase the database size.
- As network traffic is a sequence of time stamped protocol headers, processing streams require sequence and temporal DBMSs.
- It is necessary to run batches of queries over single pass over the data and these queries might use each other's partial results when they run concurrently.

Stream processing systems can be classified into research, open source, private proprietary, and commercial systems. Initially, stream processing started as a research area. SEQ [84] and Tangram [44] were important early works in this field. In 1995, Seshadri et al. [84] presented the SEQ which is a model for dealing with sequence data with an expressive range of sequence queries. They viewed sequenced data over ordered domains such as time or linear position. The model was introduced as the basis for a system to manage temporal data. The model was not for specific stream oriented applications but

a general purpose model with algebraic operators to query sequence databases. In 1989, Parker et al. [44] developed a system that they named the Tangram. The Tangram was a stream processing system implemented as an extension to Prolog that integrates with the UNIX operating system and database managers. The Tangram Stream Processor (TSP) system is founded as abstraction to stream transducers. A transducer is the basic building blocks in TSP that map one or more input streams into one or more output streams. The transducers are maintained in an extensible library. For example, the selection transducer expression in TSP is defined as:

```
SELECT(S,T,C)
Where:
S: is a substream of terms.
T: is a template the S must unify (or match) with.
C: is the condition that terms in S must satisfy.
```

The sliding windows, reasoning about time and pattern matching were among the important functionalities of this system. Between 1996 and 1998 Sullivan and Heybey [95, 96] introduced Tribeca. Tribeca was a special system for analysing network traffic. It only works for sequence data in contrast to SEQ that works for sequence and relational data. The stream processing and data management implemented in the same engine to have a tightly integrated running environment. This is something the authors see as advantageous over the Tangram system that has Prolog as a back end and DBMS as a front end. More recent work is STREAM [7]. STREAM is a prototype general-purpose DSMS supporting a large class of declarative continuous queries over streams and traditional stored data set. The project started in 2002 and it was wound down in January 2006. To handle queries over streams, they developed the Continuous Query Language. Another interesting project was conducted by the Brandeis University, Brown University, and Massachusetts Institute of Technology. They developed AURORA which is a general-purpose DSMS in 2003. In 2005, the AURORA project was superseded by the BOREALIS project. BOREALIS is a distributed multi-processor version of AURORA [1].

A recent research presented Reflex which is a programming model facilitating the construction of highly responsive Java applications [89]. Reflex is designed to make it easy to write and integrate simple periodic tasks or complex stream processors. Reflex tasks run in a part of memory free from garbage collection interference. These tasks are organized in a graph and communicate through uni-directional, non-blocking channels. To demonstrate the power and applicability of Reflexes on real world applications, Spring [89] implemented a prototype intrusion detection system.

Open source stream processing systems are also available. Two examples are ESPER [28] and IEPSE [70]. ESPER is an open source Event Stream Processing (ESP) and Complex Event Processing engine (CEP) written in JAVA. It provides rich Event Processing Language (EPL) to express filtering, aggregation, and joins, possibly over a sliding windows of multiple event streams. It also includes pattern semantics to express complex temporal causality among events. ESPER can be fully embeddable in existing

Java based architectures such as Java Application Servers or Enterprise Service Bus. It can also be used as a standalone container in any existing standalone applications [28]. IEPSE (Intelligent Event Processor Service Engine) is another open source CEP and ESP engine [70] analogous to ESPER.

GIGASCOPE [18] is a proprietary DSMS and was developed by AT&T and it is currently used in many AT&T network sites. All commercial or non open source DSMS are proprietary, that is owned and controlled by a proprietor, but here we mean it is owned and controlled and used privately by the proprietor. *GIGASCOPE* is special purpose DSMS engine for detailed network applications including traffic analysis, intrusion detection, router configuration analysis, network research, network monitoring, and performance monitoring and debugging. Johnson et al. [43] from AT&T research lab argued that *GIGASCOPE* can serve as the foundation of the next NIDSs because of the functionality and performance. They presented some written examples to detect Denial of Service attacks.

Even though SDP early research started about 30 years ago, several commercial systems only started to appear in the market a decade ago. Some of these systems started as academic research that have progressed to commercial products. At Cambridge university*APAMA* was developed and then commercialized and was acquired by Progress Software [77]. *CORAL8* is another commercial product that based on Stanford research. It has been recently acquired by Sybase Inc. [97]. It has been superseded by *Sybase CEP* which is a re-branded and updated version of the *CORAL8* product. *StreamBase* was originally research conducted jointly by Brandeis University, Brown University, and Massachusetts Institute of Technology during the early 2000s [92]. Finally, there are also some other SDP products that are offered by large vendors. IBM acquired CEP pioneer AptSoft during 2008, renamed it *WebSphere Business Events* [37]. Oracle, thanks to its BEA acquisition, offers a product called *ORACLE CEP*. *TIBCO product* is a leader in the middleware CEP from TIBCO Software Incorporation. In 2008 they had a market share of 40% [99].

All of these systems provide similar SDP functionalities. Some of them might offer more than others and some might offer less than the others. The naming of functions or operators might differ from a system to another system because there is no standard for the stream SQL language. For instance in [96] the authors refer to partitioning the stream into substreams based on the data *demultiplexing*, whereas in [92] it is called *filtering*.

In this research we used *StreamBase* for the development of the proposed system. *StreamBase* is a complex event processing software from STREAM BASE Inc. In the next sections, an overview of *StreamBase* and its query language Stream SQL (*SSQL*). Also, *StreamBase* high performance and scalability features are presented.

4.2 *StreamBase* Stream SQL

StreamBase [91] Complex Event Processing (CEP) platform allows us to build a system that can analyse and act on real time data. *StreamBase* is extensible, that is, it allows the developers to write adapters to connect to their input or output resources. Also, developers can write their own functions, stored procedures, and operators and integrate them with the *StreamBase* engine. *StreamBase* has a rich SDP functionalities that enable us to translate from MSFOMTL to its stream SQL. It has a fine-grained high performance and scalability features.

StreamBase provides the developers with two development approaches. The first approach is for rapid development which is the graphical event flow tools (*StreamBase* Studio). The second approach is text based which is the StreamSQL language (*SSQL*). *SSQL* [94] is a query language that extends SQL with the ability to process continuous data streams. Dealing with continuous streams means the stream query is evaluated continuously. *SSQL* language has the following processing capabilities:

- Non temporal operators: These operators does not have time window and act continuously on the stream (as the event arrives). These operators enable us to filter streams, merge, create sequence, create timers, create table (in memory or external storage), map values to the stream (e.g., adding time stamp), correlate multiple streams, etc.
- Temporal operators: These operators have windowing constructs. These operators allow us to query temporal events over a specified time window. The query is evaluated continuously within the sliding window. The pattern match and aggregate operators both have time window. The pattern match operator accepts inputs and matches specified temporal patterns in the query with these inputs. The aggregate operator performs aggregations and computations on real time streams or stored tables. The time window in this aggregate operator specifies the time window for the aggregation and how to advance following the time window expiration.
- Extensibility: The StreamSQL operator set is extensible, developers can add functions, operators, adapters (input or output operators).

SQL is primarily intended for manipulating relations (or tables), which are set of tuples (or rows). A tuple is an ordered set of elements (columns). Each element has a domain where its value is mapped from. *SSQL* manipulates streams, which are infinite sequences of tuples that are not all available at the same time.

The *SSQL* language [93] has many operators. Some of these operators are data definition language (DDL) operators, and some of them are data manipulation language (DML) operators. To query data streams, the select statement is used (from DML). The semantic of query depends on the clauses that are used in the select statement. We have the filter, the pattern and the aggregate operators that are formed from the select statement. The syntax and semantics for these operators are as follows:

- The syntax of the filter operator is:

```
SELECT target_list_entry [, target_list_entry...]
  FROM event_source [...] |
  [WHERE predicate]
  [INTO stream_identifier]
```

  where:

  - target_list_entry: fields identifiers;
  - event_source: the source of input such as stream or table;
  - predicate: conditions on select fields that limit the returned set by the select statement;
  - stream_identifier: unique stream identifier. The stream can be either final output stream or just stream that can be used by other *SSQL* components.

  In stream processing a query is evaluated continuously. So, at any moment of time a running query might be answered. Query is answered if it returns some results and this means the query valuation is true, otherwise it is false. The filter operator is used to query the incoming events and if there is a tuple that satisfy the restrictions specified on the columns at a moment of time τ, then that tuple is returned (output). This means the query valuation at τ is true. If no tuple returned, then the query valuation at τ is false.

- The syntax of the pattern operator is:

```
SELECT target_list_entry [, target_list_entry...]
 FROM event_source [...] |
 FROM PATTERN template [pattern_operator template ...]
 WITHIN (interval TIME)
 [WHERE predicate]
 [INTO stream_identifier]
```

  where:

  - target_list_entry: fields identifiers;
  - event_source: the source of input such as stream or table;
  - predicate: conditions on select fields that limit the returned set by the select statement;
  - interval time: timeout in seconds;
  - template: evaluates to stream identifier (one of the event source);
  - pattern operator: logical operator that relates two pair of template (NOT streamA, streamA AND streamB , streamA THEN streamB, streamA OR streamB);

  – stream_identifier: unique stream identifier. The stream can final output stream or just stream that would be used by other *SSQL* components.

The pattern operator query the event sources and only return true valuation if a query on the first template return a tuple at τ that satisfy the conditions in the where clause. Then within the specified time interval (within $\tau +$ interval TIME) the second query on the second template return a tuple that satisfy the conditions on the where clause and the logical pattern operator between the two templates. No tuples are returned if the pattern operator fails to match any pattern and this means the valuation of the pattern operator is false.

- The syntax of the aggregate operator is:

```
SELECT field_identifier_grouping [, ...]]
 FROM stream_identifier '['window_specification | window_identifier']'
 [WHERE predicate]
 [HAVING predicate]
 [GROUP BY field_identifier_grouping [, ...]]
 [ORDER BY field_identifier_ordering [, ...] [DESC] [LIMIT number]]
 [INTO stream_identifier]
```

where:

  – field_identifier_grouping: an output field used to group the entries in the result set returned by the select statement. The "GROUP BY" clause must specify the same set of this field_identifier_group;
  – stream_identifier: the source of input;
  – window_specification: specify how the aggregation is conducted by time, number of tuples, or specific values in the input fields. Also, size for the window and an advance value can be specified;
  – window_identifier: a named window specification previously declared with a "CREATE WINDOW window_identifier" statement;
  – predicate: conditions on select fields that limit the returned set by the select statement. In a HAVING clause, the predicate can contain logical operators, mathematical operators, and/or a "BETWEEN-AND" clause;
  – field_identifier_ordering: An output field used to order the entries in the result set returned by the statement. The ordering can be descending (DESC) and can be limited to n number (LIMIT);
  – stream_identifier: unique stream identifier. The stream can final output stream or just stream that would be used by other *SSQL* components.

The Aggregate operator computes aggregations over moving windows of tuple values. Each window is a view on a part of the input data. It accepts a single input stream and start counting from zero. The type of aggregation to perform is based on one of the following:

- the number of tuples in the window. A new window is established and evaluated based on the number of arriving tuples. A tuple containing the results of the aggregation is emitted and the window closed when it contains a specified number of tuples;
- the time tuples arrive. A new window is managed and evaluated based on the time a tuple arrives. For example, a tuple containing the aggregate results may be emitted and the window closed when a tuple arrives outside the period allowed for the currently open window. The new tuple begins the next window;
- a field in the input tuples. A new window is established and evaluated based on the value of a numeric field in the incoming tuple. A tuple containing the results of the aggregation is emitted and the window closed when a tuple arrives whose field value exceeds the specified range of the open window;
- a predicate expression. A new windows are opened, emitted from, and closed based on the evaluation of a predicate expression;

We can see from the above list that the evaluation of the aggregate operators depends on the aggregate window opening and closing. A tuple is emitted with the field identifier grouping field(s) as heading and the computed aggregate value(s).

More information about the *SSQL* query operators mentioned here and all other constructors can be found at [93].

4.3 *StreamBase* High Performance, scalability, and high availability Features

The fact that stream processing deals with high volume data motivates the researchers and vendors to implement a fast, robust, and scalable system to satisfy the processing requirements. Many vendors have implemented high performance mechanisms and techniques to meet the requirements of processing high volume data [37, 93, 99]. Another aspect that the users like to have in SDP systems is the ability of the system to grow as the data usage increases in the future, that is, the scalability of the system. Finally, for critical environments that can not afford to have down time, the users like to have zero down time robust system.

Looking and evaluating all existing systems are impossible due to the factors of time, license, expertise, etc. Hence, in this project we selected the *StreamBase* for the featured mentioned in the previous section. By using the high performance features of *StreamBase* , we obtained a very promising results as can be seen in Chapter 7. In this section these features are explained in some detail.

4.3.1 High Performance Features

High performance features in *StreamBase* can be achieved by processing the data in parallel. In fact, *StreamBase* provides a very fine-grained control mechanism for parallel execution. The option of running part of the code (can be a module or a single operator) in parallel or multithreaded or both can be specified. There are some exceptions and some operators where the parallel options can not be used, but these are few. For example, the sequence operator which generates a new sequence value each time a tuple pass through. Obviously, the parallel option is not suitable for this operator.

This fine-grained mechanism means that in an application there are many choices for running different components in parallel. The best way to implement the parallelism features require two important tasks need to be carried out by the developers. The first task is to thoroughly analyse the application to find out the candidate components for parallelism. The second task is to find out the suitable values for the degree of parallelism that gives better performance. A component is a candidate for separate threading if the component is long-running or compute-intensive, can run without data dependencies on other components, and would not cause the containing module to block while waiting for the component to return. The developer must have a good understanding of the parallelism features and the execution order of *StreamBase* (i.e., the program execution sequence before applying any parallelism). To help the developer, a profiling tool is provided that helps in analysing the application execution and the developer can profile a *StreamBase* application while it is running to extract statistics about the operators, queues, and threads in that application, and about system resources consumed by that application. The information can be viewed interactively or stored in a file for later review and analysis.

The parallelism features in *StreamBase* are controlled with the concurrency and multiplicity options. Here, we give explanations of these options along with the related concepts and terms with some simple examples. The *StreamBase* manuals [93] contain much more detailed and rich information.

First we explain how a simple module runs in the *StreamBase* server. A simple running module, running in a container in StreamBase Server, with default concurrency settings (no concurrency), operates in a single parallel region. The StreamBase container is the most elementary parallel region. In Figure 4.1, a simple application that will be used as a running example is shown. The application is a simple application with four operators as follows:

- Input Adapter (IP_Sniffer): this is an input adapter that reads the network packet and passes it to the mapping operator. The schema of the packet is simply defined as source address, source port, destination address, destination port, fragmentation offset, and packet total length.

- Mapping operator (TotPacketSize): for each tuple (packet) it maps the value of the total packet size up to this packet if it is a fragmented packet (if fragment offset > 0). This can be calculated with the formula:

  $\text{fragmentation offset} * 8 + \text{packet total length}$

  In the above formula, the offset is in bytes, so to convert it into bits we multiply by 8. This calculated value is assigned to a variable say SumPacket.

- Filter operator (SizeAlert): Selects all tuples that has SumPacket > 5000.

- Output Operator (Output) : output the tuples found.

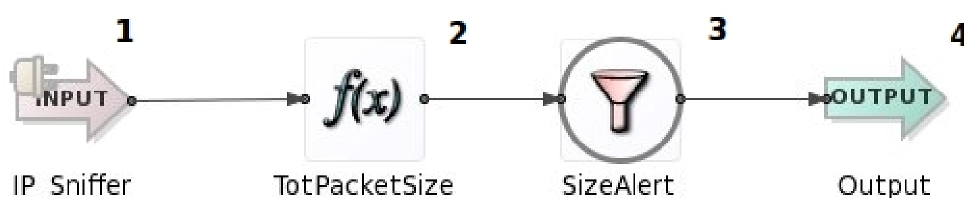


FIGURE 4.1: A Simple Application Running In A Container Inside The *Stream-Base*Server

In this application, the sequence of data is numbered 1 to 4. When a tuple arrives it has to go through the sequence in order 1 to 4 and subsequent tuples will be queued at the input adapter operator. The developer can speed up the processing and use the concurrency and multiplicity options. The concurrency option means the instance(s) of the component runs in its (their) own thread(s). Multiplicity refers to the number of the instances of the components. The developer can set the concurrency option or the multiplicity or both. The following scenarios are possible for a candidate component for concurrency and multiplicity inside an application:

1. If the concurrency option is set without the multiplicity, this means a single instance is to run in its own thread parallel to the main parallel region (main container).

2. If only multiplicity is set, the number of instances designated by the multiplicity in their own threads inside the main parallel region.

3. If both concurrency and multiplicity are set, the number of instances designated by the multiplicity run in their own threads (parallel regions).

Back to our example in 4.1, the concurrency and multiplicity are applicable as the process of each tuple is completely isolated from the processing of other tuples. To apply the concurrency option, there are two options. The first is to use it for the filter operator

only. Second option is to rewrite the mapping and filtering operators as a referenced module and use the concurrency and multiplicity operator on this module. A referenced module is a module that can be called by another module (or main). For the explanation purposes and not necessarily for best performance, we use the first option to explain the concurrency and multiplicity.

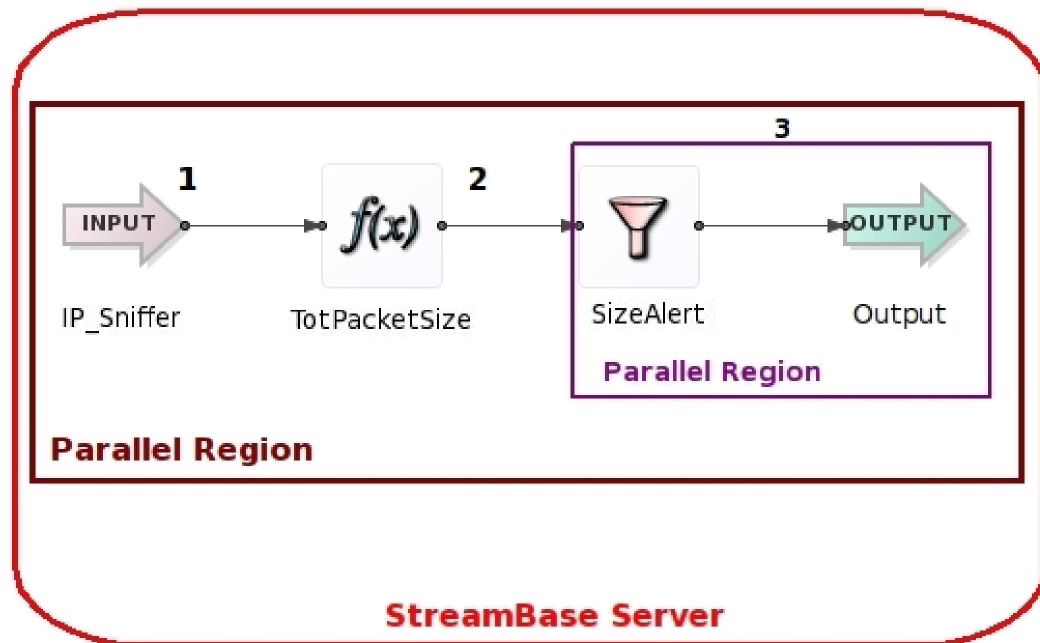


FIGURE 4.2: Applying Concurrency only on 4.1

Figure 4.2 shows the use of concurrency only, that is, the filtering will run in a separate thread (parallel region). The difference here from the sequential execution in 4.1 is that the stream out from the mapping (step 2) is going to the parallel region of the filtering (step 3) which means the processing in this region is parallel to the main parallel region, and if tuples arrive rapidly and need to be queued, it will be queued here.

Figure 4.3 shows the use of multiplicity only (two are used) as applied to 4.1. With the multiplicity, the dispatch style needs to be specified. The dispatch style designates how each instance will receive its input. Three options are there: broadcast, round robin, and according to a specified data value. In broadcast each instance will receive a copy of every tuple. In round robin, the first tuple goes to the first instance and the second goes to the second and so on. Based on value involves checking the value against a test condition and then dispatching to the designated instance for that value. The option that make a common sense for this example is to use round robin in Fig 4.3. The execution sequence is labeled 1 to 6. No parallel execution as the concurrency option is not used, so the first instance is processed up to completion and the second will be processed by the second instance.

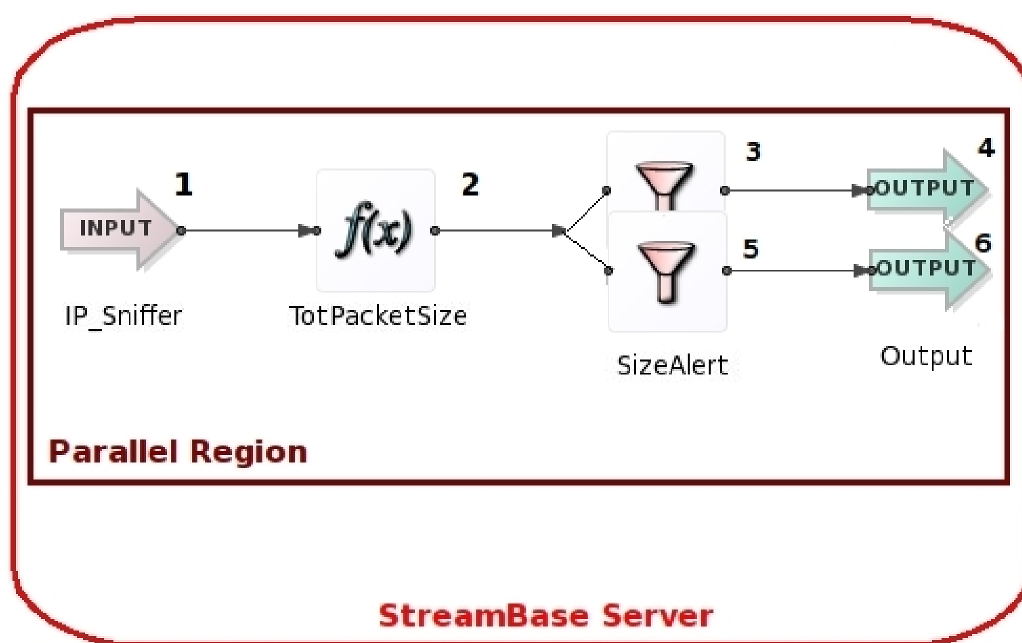


FIGURE 4.3: Applying Multiplicity only on 4.1

Figure 4.4 shows the implementation using both the concurrency and multiplicity (of two) as applied to 4.1. Here, two instances of the filtering operator are simultaneously running each in its own thread and parallel region. They receive input in round robin fashion. Each instance operates independently from the other instance and has its own queue if the tuples are dispatched rapidly and the instance could not process it fast enough. We could expect better performance using this implementation, but really this depends on how many cores exist on the machine. Please notice that, arbitrarily, we used two instances but the optimal choice depends on how many cores exist on the machine and if other applications are running. In general the rule of thumb is to use a number of instances less than or equal to the number of the existing cores on the running machine.

4.3.2 Scalability and High Availability

Scalability refers to the capability of the systems to expand to meet an increase in the traffic volume. *StreamBase* is scalable in two ways. First more CPUs can be added and memory can be upgraded. The second way of scalability is through the support of distributing processing over multiple servers. This is true for most of the commercial SDP systems, if not all, that exist today.

High availability refers to having a production system running with minimal or zero down time. *StreamBase* uses a high-availability (HA) solution. The solution is based on a standard process-pairs approach in which two servers, a primary and a secondary, operate as a processing pair. While the production system runs on the primary, the secondary keeps a backup process that accumulates enough information through check pointing and synchronization approach, to be able to pick up execution without gaps

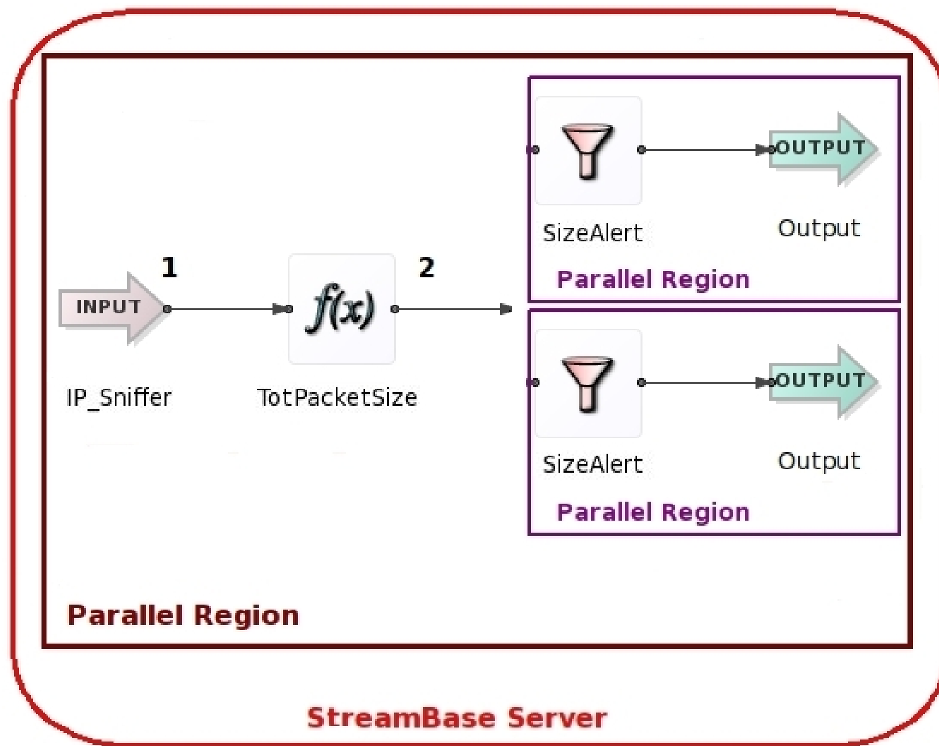


FIGURE 4.4: A main module with a candidate component for using concurrency and multiplicity of 2.

when the primary fail. In this way, fail over to the secondary server is transparent to the client applications.

4.4 Summary

In this chapter we presented an overview of the stream data processing technology. The differences between conventional RDBMSs and SDPs were highlighted. *StreamBase* is the stream data processing system we make use of as the execution engine in the proposed solution. We introduced it in this chapter and gave an overview of its query language (*SSQL*) syntax and semantics. In addition, we described the high performance features of *StreamBase* which is used in designing the proposed system in this thesis (*TeStID*). The next section presents the proposed system.

Chapter 5

Temporal Stream Intrusion Detection System (TeStID)

The last three chapters presented literature review of intrusion detection systems, temporal logic, and stream data processing. This chapter presents the proposed system Temporal Stream Intrusion Detection (TeStID). Section 5.1 is about the formal specification used in *TeStID*. It presents abstract view of network communications and the syntax and semantics of the proposed logic. In Section 5.2, the attack classification and the syntactical forms corresponding to this classification are given. Section 5.3 is about the proposed system *TeStID*. The system description, architecture, and benefits are given. Finally, a summary for this chapter is provided in Section 5.4.

5.1 Formal Specification

To formally model a real-time system and conduct runtime verification an abstract view of the system events or behaviour is needed. The process models of sequence of events can be timed or untimed. In the untimed process models, the sequence of events are modelled but not the time at which the events occur. In the timed processing model both the sequence of events and their occurrence times are modelled. The abstraction choice depends on the system verification requirements.

The following subsections explain the abstraction view of the network communications that will be used in *TeStID*, the time model, and the syntax and semantics of MSFOMTL.

5.1.1 Abstract View of Network Communications

Nodes (any devices connected to a computer network) in networked environment communicate by exchanging data over the network physical media. In Ethernet networks, this physical media can be coaxial cable, twisted pair cable, or fibre cable. In order for nodes to communicate, there are certain protocols that need to be followed and executed by every communicating node for the communication to be successful.

In the OSI reference model[1], when a node is ready to send data across the network, this data is encapsulated with headers. Each succeeding network layer wraps a header around the transferred data and thus the data can be handled properly by the layer below (see Figure 5.1). The following steps summarizes the sending process:

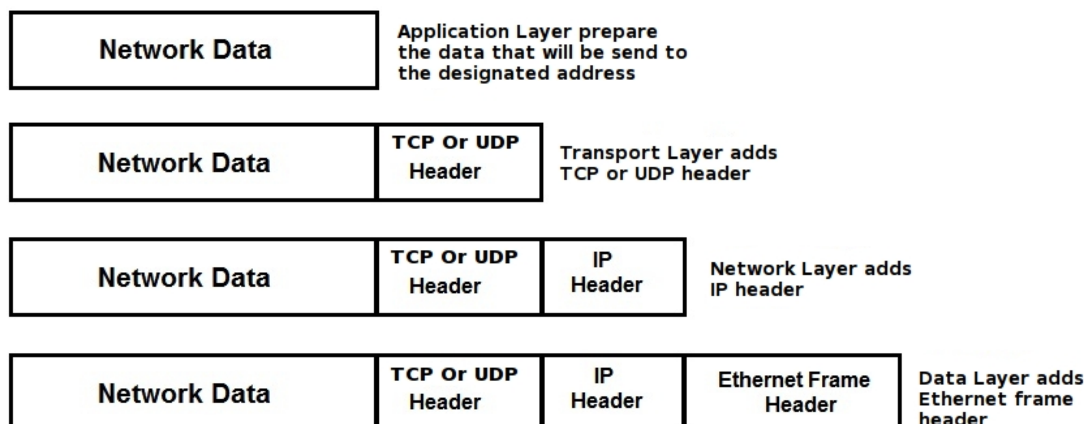


FIGURE 5.1: Packet Encapsulation During Sending

- At the application layer, the data (payload) is created and this is actually just simple data with a designated destination.
- At the transport layer, headers are added to these chunks of data to create TCP or UDP segments. These headers include linking information to specific processes at the destination.
- At the network layer, the Internet Protocol (IP) headers are added to create IP data grams. The IP header includes information about the source and destination address and thus at this point the data is being directed to a specific process running on a specific computer on a specific network.
- At the data link layer, the Ethernet frame header is added which includes information such as the physical MAC addresses of the source and destination and checksums.

During the receiving process, the operation is reversed in direction (i.e., from the physical layer to the application layer) and the headers are stripped away as the data is moved and finally delivered to the destination process (see Figure 5.2). The destination node monitors the Ethernet for frames addressed to its Ethernet network interface MAC address. If one exists, then the following steps take place:

[1]Open System Interconnection Reference Model, a standard for network architecture developed by International Organization for Standardization (ISO). It consists of a set of seven layers that define the different stages that data must go through to travel from one device to another over a network.

- The data link layer strips the Ethernet header and the IP datagram is delivered to the next top layer.
- The network layer strips the IP header and delivers the TCP (UDP) datagram to the transport layer.
- The transport layer strips the TCP (UDP) header and delivers the data or payload part to the destination process.

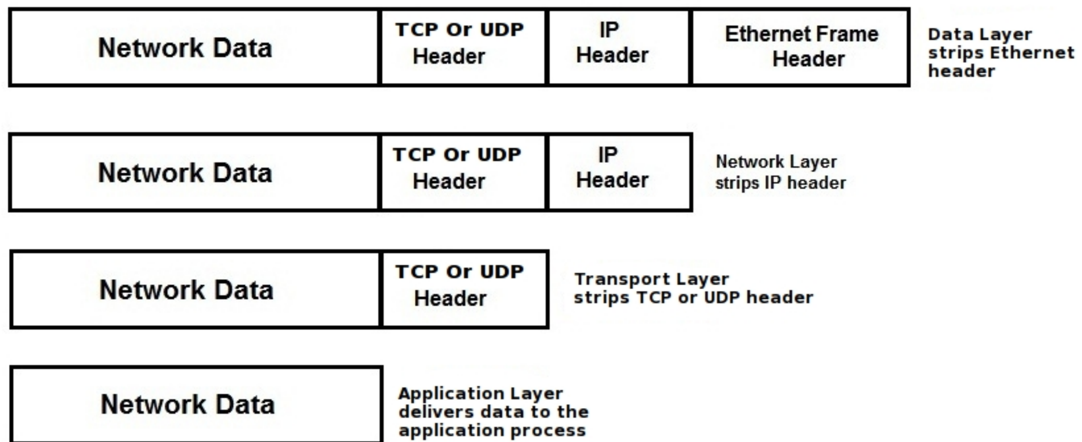


FIGURE 5.2: Packet Deencapsulation During Receiving

Logically, the incoming network stream packets form the temporal model $\mathcal{M}$. Packets are captured in order by arrival time τ. Each captured packet belongs to a certain network communication protocol (TCP, UDP, ICMP, etc). Formally, we represent each type of packet as a predicate P which we denote as P_{TCP}, P_{UDP}, P_{ICMP}, etc. where the subscript reflects the type of the protocol. The set of all packets of all possible types is denoted by $[\![\mathcal{P}]\!]$, that is:

$$[\![\mathcal{P}]\!] = [\![P_{TCP}, P_{UDP}, P_{ICMP}, \ldots]\!]$$

In this thesis as we use only TCP as a case study and as there is no ambiguity we will use P to represent a predicate of the TCP protocol type with 12-arity $s_1 \times ... \times s_{12}$ where s_n $(1 \leq n \leq 12)$ is a sort of a particular predicate argument (i.e., sender address, receiver address, sender ports, etc.). These fields are selected based on our need to represent attacks. In misuse IDS, we have the advantage of knowing in advance the attack signatures and it is easy to identify the features or fields required. This is simply done by cross referencing the attacks and the features needed by them. These features are then used in the system. Of course, if new attacks are discovered requiring new features, these would need to be included. The specification of a TCP predicate is:

$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12})$$

where:

- x_1 : is a string variable representing the sender IP address;
- x_2: is an integer variable representing the sender port;
- x_3: is a string variable representing the receiver IP address;
- x_4: is an integer variable representing the receiver port;
- x_5: is an integer variable representing the sequence number;
- x_6: is an integer variable representing the acknowledgment number;
- x_7: is a Boolean variable representing the *ack* flag;
- x_8: is a Boolean variable representing the *syn* flag;
- x_9: is a Boolean variable representing the *rst* flag;
- x_{10}: is a Boolean variable representing the *push* flag;
- x_{11}: is a Boolean variable representing the *urg* flag.
- x_{12}: is a string representing the payload or data.

Packets arrive at some point in time. So, we can consider the arrival of these packets as instantaneous arbitrary occurring events. Two packets can not arrive at the same time, one must be before the other. The model of time consists of a set of arrival points $\mathcal{T} \subset \mathbb{R}^+$ (where $\mathbb{R}^+$ is the set of non-negative real numbers) and we require $\mathcal{T}$ to be discrete: for any finite interval $[a, b]$, the set $[a, b] \cap \mathcal{T}$ is finite. The model is represented as $\mathcal{M} = \langle \mathcal{T}, <, \mathcal{I}, I_s \rangle$ where:

- $\mathcal{T} = \{\tau_0, \tau_1, \dots\} \subset \mathbb{R}^+$, where $\mathbb{R}^+$ is a non-empty set of positive real numbers and $\mathcal{T}$ is the set of all arrival moments.
- $<$ is a linear order on $\mathcal{T}$.
- $\mathcal{I}$ is an interpretation which maps $\mathcal{T}$ into $[\![\mathcal{P}]\!]$:

  $$\mathcal{I} : \mathcal{T} \rightarrow [\![\mathcal{P}]\!]$$

  So, $\mathcal{I}(\tau_i)$ represents a packet arriving at a moment $\tau_i \in \mathcal{T}$.
- I_s is an interpretation over a domain $\mathcal{D}_s$ for sort s.

5.1.2 MSFOMTL Syntax

The syntax of *MSFOMTL* (Many Sorted First Order Metric Temporal Logic) is based on the work of Manzano [55], Koymans [47], and Alur [4]. The symbols that are allowed to be used in expressions can be grouped into logical and non-logical symbols. Logical symbols are the quantifiers $\forall$ and $\exists$, the logical connectives $\wedge$, $\vee$ and $\neg$, the logical binary predicate symbols $=, \neq, <, \leq, >$, and $\geq$, the bounded future temporal operators $\Diamond_{[t_1,t_2]}$ ("eventually"), and $\Box_{[t_1,t_2]}$ ("always"), the past bounded temporal operators $\blacklozenge_{[t_1,t_2]}$ ("sometimes in the past"), and $\blacksquare_{[t_1,t_2]}$ ("always in the past"). The subscripts $[t_1, t_2]$ in the operators refer to their scope (between the moments t_1 and t_2 from now).

In many sorted logic, the arguments of predicate and function symbols may have different sorts s and every sort $s \in S$ where S is a finite set of sorts. The non-logical symbols of *MSFOMTL* consist of the finite disjoint sets of predicate symbols, function symbols, constants, and variables. The alphabet of the language of *MSFOMTL* $\mathcal{L}$ consists of the union of all the non-logical symbols.

5.1.2.1 Terms

The set of terms in $\mathcal{L}$ of sort s is the smallest set of expressions with the following properties:

- Each constant symbol c of sort s is a term where $s \in S$.
- Each variable v of sort s is a term where $s \in S$.
- If f is a function symbol of n-arity $s_1 \times ... \times s_n \rightarrow s$ and te_i is a term of sort s_i, then $f(te_1, ..., te_n)$ is a term of sort s.
- If te_1 and te_2 are numeric terms of sort s, then $te_1 + te_2$, $te_1 - te_2$, $te_1 \times te_2$ and $te_1 \div te_2$ are terms of sort s, where $s \in S$.

5.1.2.2 Formulae

The formulae of *MSFOMTL* are defined as follows:

- If $te_1 \times ... \times te_n$, where each te_i is a term of sort s_i, and P is a predicate symbol with n-arity $s_1 \times ... \times s_n$, then $P(te_1, ..., te_n)$ is an atomic formula.
- If te_1 and te_2 are terms of the same sort s then $te_1 = te_2$, $te_1 \neq te_2$, $te_1 > te_2$, $te_1 < te_2$, $te_1 \leq te_2$ and $te_1 \geq te_2$, are formulae.
- Every atomic formula is a formula.
- If φ is a formula then $\neg\varphi$ is a formula.
- If φ is a formula then $\Diamond_{[t_1,t_2]}\varphi$, $\Box_{[t_1,t_2]}\varphi$, $\blacklozenge_{[t_1,t_2]}\varphi$ and $\blacksquare_{[t_1,t_2]}\varphi$ are formulae.
- If φ and ψ are formulae then $\varphi \wedge \psi$ and $\varphi \vee \psi$ are formulae.
- If φ is a formula and x is a variable of sort s then $(\forall x)\varphi$ and $(\exists x)\varphi$ are formulae.

5.1.3 MSFOMTL semantics

In *MSFOMTL* the arguments of functions and predicates have different sorts and each sort s ranges over a domain $\mathcal{D}_s$. We denote I_s as the interpretation over the domain $\mathcal{D}_s$ of constant, variable, and symbol. The following mapping can be defined:

- Each variable v in $\mathcal{L}$ of sort s is evaluated as v^{I_s} which is an element in $\mathcal{D}_s$.
- Each constant symbol c in $\mathcal{L}$ of sort s is evaluated as c^{I_s} which is an element in $\mathcal{D}_s$. The constants are viewed to be rigid which means they do not change overtime.
- Each functional symbol f of arity n and of sort $s_1 \times ... \times s_n \rightarrow s$ is mapped to a function $\mathcal{D}_{s_1} \times ... \times \mathcal{D}_{s_n} \rightarrow \mathcal{D}_s$. The evaluation of a function $f(te_1, ..., te_n)^{I_s} = f^{I_s}(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}})$, where te_i is a term of sort s_i.
- Each predicate symbol P with n-arity $s_1 \times ... \times s_n$ is mapped to a predicate $\mathrm{P}^{\mathcal{I}}$ which is a subset of $\mathcal{D}_{s_1} \times ... \times \mathcal{D}_{s_n}$. A predicate is evaluated to be true when $(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}}) \in \mathrm{P}^{\mathcal{I}}$, where te_i is a term of sort s_i.

A temporal formula φ holds at $\mathcal{M} = \langle \mathcal{T}, <, \mathcal{I}, I_s \rangle$ at an arrival time $\tau_i \in \mathcal{T}$, that is, $\mathcal{M}, \tau_i \models \varphi$ is defined recursively as follows:

$$
\begin{array}{ll}
\mathcal{M}, \tau_i \models P(te_1, ..., te_n) & \text{iff } P^{\mathcal{I}}(te_1^{I_{s_1}}, ..., te_n^{I_{s_n}}) = \mathcal{I}(\tau_i) \\
\mathcal{M}, \tau_i \models \neg\varphi & \text{iff } \mathcal{M}, \tau_i \not\models \varphi \\
\mathcal{M}, \tau_i \models \varphi_1 \wedge \varphi_2 & \text{iff } \mathcal{M}, \tau_i \models \varphi_1 \text{ and } \mathcal{M}, \tau_i \models \varphi_2 \\
\mathcal{M}, \tau_i \models \varphi_1 \vee \varphi_2 & \text{iff } \mathcal{M}, \tau_i \models \varphi_1 \text{ or } \mathcal{M}, \tau_i \models \varphi_2 \\
\mathcal{M}, \tau_i \models \Diamond_{[t_1,t_2]}\varphi & \text{iff } \exists\ \tau^{'}\ (\tau_i + t_1 \leq \tau^{'} \leq \tau_i + t_2) \text{ s.t. } \mathcal{M}, \tau^{'} \models \varphi \\
M, \tau_i \models \Box_{[t_1,t_2]}\varphi & \text{iff } \forall\ \tau^{'}\ (\tau_i + t_1 \leq \tau^{'} \leq \tau_i + t_2)\ \mathcal{M}, \tau^{'} \models \varphi. \\
\mathcal{M}, \tau_i \models \blacklozenge_{[t_1,t_2]}\varphi & \text{iff } \exists\ \tau^{'}\ (\tau_i - t_1 \geq \tau^{'} \geq \tau_i - t_2) \text{ s.t. } \mathcal{M}, \tau^{'} \models \varphi \\
M, \tau_i \models \blacksquare_{[t_1,t_2]}\varphi & \text{iff } \forall\ \tau^{'}\ (\tau_i - t_1 \geq \tau^{'} \geq \tau_i - t_2)\ \mathcal{M}, \tau^{'} \models \varphi. \\
\mathcal{M}, \tau_i \models (\forall x)\varphi & \text{iff } \forall\ I_s\ (x^{I_s} = a)\ \mathcal{M}, \tau_i \models \varphi[x/a] \\
\mathcal{M}, \tau_i \models (\exists x)\varphi\tau & \text{iff } \exists\ I_s\ (x^{I_s} = a) \text{ s.t. } \mathcal{M}, \tau_i \models \varphi[x/a]
\end{array}
$$

In some attacks, we need to find out the number of moments where a formula φ (the attack specification) holds within a period of time. To represent this, we introduce a new operator "Repeated" R. This operator is time bounded and indicates the number of times a formula holds repeatedly. The syntax is $R^n_{[t_1,t_2]}$, where n is a natural number, and in words it means "for all $t_1, t_2 \in \mathcal{T}$ there are at least n many $\tau \in (t_1 \leq \tau \leq t_2)$ such that $\mathcal{I}(\tau) \in [\![\mathcal{P}]\!]$ ". Formally, the definition of this operator is as follows:

$$\mathcal{M}, \tau_i \models R^n_{[t_1,t_2]}\varphi \ \text{ iff } |\ \{\tau^{'}\ |\ (\tau_i + t_1 \leq \tau^{'} \leq \tau_i + t_2) \text{ and}$$

$$\mathcal{M}, \tau^{'} \models \varphi\} \mid \geq \ n$$

If terms te_1 and te_2 are terms of the same sort s then the (in)equality of terms are atomic formulae. These terms are evaluated in the usual way, for example:

$$\mathcal{M}, \tau_i \models (te_1 = te_2) \quad \text{iff} \quad te_1^{I_s} = te_2^{I_s}$$

Finally, the arithmetic functions (e.g., $, +, -, \times, \div$) for numeral terms in $\mathcal{L}$ are the standard binary operations for arithmetic functions.

5.2 Attack Classification

The proposed first order temporal logic *MSFOMTL* has a formal semantics, it can represent temporal patterns, and complex attacks can be represented concisely. The temporal patterns are the attack specifications that we need to match within the incoming events. To represent these attacks formally and efficiently, we need to understand them and to understand the nature of incoming events. The objective is to ease the process of representing attacks using MSFOMTL.

Attacks in a network can be carried using one single packet or using sequences of packets in some order. We identify four main attack patterns. One involves a single packet and the other three are multiple packet attacks. The multiple packet attacks are further divided into three groups based on the temporal properties of the attacks: *forward multiple packet attacks*, *backward multiple packet attacks*, and *repetition attacks*. In the following, we present more detailed information and the syntactical patterns used to express many known attacks. For instance, the first syntactical form covers single packet attacks. In Chapter 7 we selected 50 single packet attacks as they defined in Snort. The latest version of Snort has about three thousand signatures [88]. These attacks can be represented by the first syntactical form. The other syntactical forms can cover all DARPA TCP/IP multiple packet attacks [60]. In Chapter 6, these syntactical patterns are used by the translator to produce stream queries. The syntactical forms are as follows:

1. *Single packet attacks:* These attacks have no temporal aspect and involve only one packet. Technically, the attacker sends a packet with some fields in the header set with values that take advantages of vulnerabilities in the victim's machine. In some attacks, the data or payload part of the packet is used as well. The canonical form of these attacks is:

$$P \wedge Q \tag{5.1}$$

   where:

   - P is first order predicate (representing a packet).
   - Q is conjunction of Boolean formulae built of the terms of P.

An example of this type of attack is the *Land* attack [60]. In this attack, the attacker sends a spoofed *syn* packet in which the source address is the same as the destination address. In some implementations of TCP/IP, this packet causes a Denial of Service (DoS) attack. The attack can be represented formally as below:

$$\begin{aligned}(\exists x_1, x_3, x_8)((\exists y_2, y_4, y_5, y_6, y_7, y_9, y_{10}, y_{11}, y_{12})\\ P(x_1, y_2, x_3, y_4, y_5, y_6, y_7, x_8, y_9, y_{10}, y_{11}, y_{12})\\ \wedge(x_1 = x_3 \wedge x_8 = 1))\end{aligned} \tag{5.2}$$

Another example of single packet attacks using the payload is the *code red v2* worm [13]. This worm takes advantage of a vulnerability in Microsoft Index Server 2.0 described in security bulletin MS01-033 [57]. It causes a DoS attack and replicates itself to other connected vulnerable machines. It sends a crafted Http request to the vulnerable server on port 80. In the payload we can look for the word "root.exe". Formally, we can represent this *code red v2* as:

$$\begin{aligned}(\exists x_3, x_7, x_{12})((\exists y_1, y_2, y_4, y_5, y_6, y_8, y_9, y_{10}, y_{11})\\ P(y_1, y_2, x_3, y_4, y_5, y_6, x_7, y_8, y_9, y_{10}, y_{11}, x_{12})\\ \wedge(x_3 = 80 \wedge x_7 = 1) \wedge x_{12} = f(".*root.exe.*"))\end{aligned} \tag{5.3}$$

The function f represents regular expression function. The function is called with regular expression string syntax parameter ".*root.exe.*" where the ".*" means a single character repeated zero or many times. We check if the TCP flag *ack* is set and the other TCP flags are not significant in this signature.

2. *Forward multiple packet attacks:* In these attacks more than one packet is sent in a specific order as the time progress. Each packet is a step of the protocol communication. This allows us to represent the attacks that take advantages of the vulnerabilities in the communication protocols. The canonical form of these attacks is:

$$\varphi \wedge \Diamond_{[t_1,t_2]}\psi$$

or:

$$\varphi \wedge \neg\Diamond_{[t_1,t_2]}\psi \tag{5.4}$$

where:

- φ is a first order predicate (representing a packet).
- ψ is either a first order predicate or the same formula as 5.4.

A typical example of this type of attack is the WinNuke attack [66]. WinNuke is a DoS attack against the Windows NT that sends Out Of Band data (MSG_OOB)

to port 139 (NetBIOS), crashing the Windows NT machine. To detect this attack, we need to look for a NetBIOS connection setup (handshake) followed by a packet sent with the *urg* flag set. A distinguished sequence of events is as the following:

- The attacker sends a packet with his IP as the source IP (x_1), his port as the source port (x_2), the destination or receiver IP (x_3), the destination port equal to 139, the *syn* flag set, the *ack* flag unset, the acknowledge number equal to 0, and the *urg* flag unset, and the Initial Sequence Number (ISN) (x_5).
- The NetBIOS service (port 139) on the victim's machine (the receiver) will respond by sending a packet with his IP as the source IP (x_3), the source port 139, the *syn* flag set, the *ack* flags set, acknowledgment number $= x_5 + 1$, and its own ISN $= x_6$. This is the second step of the three tcp handshake connection setup, the receiver acknowledges the sender ISN by incrementing it by one ($x_5 + 1$) and sending the sum as an acknowledgment number. This increment is the mechanism to let the sender know the packet sequence he expects to receive next from the sender.
- The attacker sends a packet with his IP as the source IP (x_1), his port as the source port (x_2), the destination IP (x_3), the destination port 139, *ack* flag set, sequence number $= (x_5 + 1)$ (which the receiver expects), acknowledge number $= x_6 + 1$, the *ack* flag set, the *syn* flag unset, and the *urg* flag unset.
- The connection now is considered synchronized. The attacker sends a packet to carry the attack that has his IP (x_1), his port as the source port (x_2), the destination IP (x_3) destination port equal to 139, his sequence number (x_5+1), the *syn* flag unset, the *ack* flag set, the acknowledge number equal to x_6+1, and the *urg* flag set. Notice that the attacker sends his sequence number which is the same as in the previous step, the reason that the acknowledge only packet did not contain data and thus the sequence number was not incremented.

Formally we represent this attack as:

$$\begin{gathered}(\exists x_1, x_2, x_3, x_5, x_6)((\exists y_6, y_9, y_{10}, y_{12})P(x_1, x_2, x_3, 139, x_5, y_6, 0, 1, y_9, y_{10}, 0, y_{12}) \wedge \\ \Diamond_{[0,42]}((\exists z_9, z_{10}, z_{12})P(x_3, 139, x_1, x_2, x_6, x_5 + 1, 1, 1, z_9, z_{10}, 0, z_{12}) \wedge \\ \Diamond_{[0,42]}((\exists w_9, w_{10}, w_{12})P(x_1, x_2, x_3, 139, x_5 + 1, x_6 + 1, 1, 0, w_9, w_{10}, 0, w_{12}) \wedge \\ \Diamond_{[0,1]}(\exists k_9, k_{10}, k_{12})P(x_1, x_2, x_3, 139, x_5 + 1, x_6 + 1, 1, 0, k_9, k_{10}, 1, k_{12})))) \quad (5.5)\end{gathered}$$

In the above formula the time for the second predicate to hold is within 42 seconds from the time that the first predicate holds and the same is true for the third predicate when the second predicate holds. We assume that the timeout for the TCP protocol is 42 seconds on the Windows NT but this can be changed to reflect

the real value used by the administrator. The third predicate is the last step of the connection setup (TCP handshake). The attacker will send the fourth predicate within one second following the session establishment with the *urg* flag set.

3. *Backward multiple packet attacks:* These attacks can be identified by an event observed now and that certain events did not happen in the past. The canonical form of these attacks is:

$$\varphi \wedge \blacksquare_{[t_1,t_2]} \neg\psi \tag{5.6}$$

where:

- φ and ψ are first order predicates (representing packets).

An example of this attack is the *Reset Scan*. The *Reset Scan* is a probe attack in which the attacker tries to learn or discover some services running on the network and it is usually the initial stage to launching other types of attack. In this attack, TCP packets with the *rst* flag set are sent to a list of IP addresses in the network to determine which machines are active. If there is no response to the reset packet, the machine is alive or exists. If a router or a gateway responds with "host unreachable" then the machine does not exist [60].

The *Reset Scan* attack can be identified by looking for reset packet requests for non existing sessions. The following are the sequence of events:

- No communication exists between node A and node B. The node is either receiving packets and then acknowledging (sending acknowledgements for the received packets) or sending packets to the other node and (the other node is acknowledging), so, the absence of communication can be checked by the absence of TCP packets sent or received by either node.
- A reset packet is sent to A (B).

In the first step above we check if either node is receiving or sending because in the TCP/IP protocol the node is either receiving (and acknowledging) or sending to the other node and (the other node is acknowledging).

To represent this attack formally:

$$\begin{gathered}(\exists x_1, x_2, x_3, x_4)((\exists y_5, y_6, y_7, y_8, y_{10}, y_{11}, y_{12})\\ P(x_1, x_2, x_3, x_4, y_5, y_6, y_7, y_8, 1, y_{10}, y_{11}, y_{12})\\ \wedge\blacksquare_{[0,300]}(\exists z_5, z_6, z_7, z_8, z_{10}, z_{11}, z_{12})\\ \neg P(x_1, x_2, x_3, x_4, z_5, z_6, z_7, z_8, 0, z_{10}, z_{11}, z_{12}))\end{gathered} \tag{5.7}$$

In the above formula, notice that the *rst* flag is set in the first predicate and not in the second and the session timeout is set to be 300 seconds.

4. *Repetition attacks:* These attacks have no effect unless they are repeated finitely many times (n) within a specified time window. The canonical form of these attacks is:

$$R^{n}_{[t_1,t_2]}\varphi \tag{5.8}$$

where:

- φ is any formula of the previous categories.
- n is the number of times φ holds repeatedly within $[t_1, t_2]$.

The *Neptune* attack is an example of this type of attack. *Neptune* is a DoS attack that causes a machine to stop accepting incoming TCP/IP connections. An attacker sends many connection requests or *syn* packets with a spoofed IP address to a particular port on a host. The victim responds with *syn-ack* packets, but the sender host is unreachable, thus, this causes many half-open TCP connections. Each half-open TCP connection causes the server to add a record to the data structure that stores information describing all pending connections. This data structure is of finite size, and eventually it will overflow. This means that the server will be unable to accept any new incoming connections until the table is emptied out. Normally, there is a timeout associated with a pending connection, so the half-open connections will eventually expire and the victim server system will recover. However, the attacking system can simply continue sending spoofed IP packets requesting new connections faster than pending connections on the victim system expire [60].

A *Neptune* attack can be identified by looking for a number of *syn* packets destined for a particular machine which are coming from an unreachable host. This means that we need to look for all the initiated connections that have not received the third handshake *ack* packets. The sequence of events are the following:

- The attacker sends a TCP packet with the *syn* flag set with a spoofed IP address of an unreachable host.
- The receiver responds with either:
  - *syn-ack* packet (i.e., *syn* and *ack* flags set) if the connection request was for a service using a valid port; or
  - a *rst* packet (i.e., *rst* flag set) if the connection request was for a non existing service or invalid port.
- This *syn-ack* request does not reach the host that initiated the request, thus no acknowledgment packet will be received.

This will be repeated many times and the sender may use a different port every 20 times to send the packet from, also, he might select a different target port (not necessary for the attack).

For the representation of this attack, there are two out of the three sequences mentioned earlier that are repeated finitely often. These two sequences are the first and the last one. We use these two sequences as a signature of this attack and we write the following formula:

$$\begin{aligned}(\exists x_1, x_3, x_4)R^{20}_{[0,1]}((\exists x_2)((\exists y_5, y_9, y_{10}, y_{11}, y_{12})\\ P(x_1, x_2, x_3, x_4, y_5, 0, 0, 1, y_9, y_{10}, y_{11}, y_{12}) \wedge\\ \neg\Diamond_{[0,42]}\ (\exists z_5, z_6, z_9, z_{10}, z_{11}, z_{12})\\ P(x_1, x_2, x_3, x_4, z_5, z_6, 1, 0, z_9, z_{10}, z_{11}, z_{12})))\end{aligned} \tag{5.9}$$

This formula can be read as follows: a sender with address x_1 sends a packet to receiver with address x_3 and with port x_4 using sender port x_2 and the sender is not sending a packet with *ack* flag set within 42 seconds; this is repeated for at least 20 times within a second. The 42 seconds is the assumed timeout for the third or last TCP handshake arrival. This timeout is operating system dependent and can be changed by the system administrator.

5.3 The Proposed System

The proposed system is a network based intrusion detection that uses both misuse based and anomaly based detection methods. The main idea for the proposed system is:

- to use temporal logic for specifications of attacks in misuse based method and for specifications of the normal behaviour in the anomaly based method.
- to use SDP as the attack detection engine.

We combine the expressiveness, concise representations, and clear semantics of temporal logic with the temporal ordering and high volume data handling capabilities of stream data processing technology to develop NIDS. The proposed system is named Temporal Stream Intrusion Detection (*TeStID*).

5.3.1 *TeStID* System Architecture

The system architecture of *TeStID* is shown in Figure 5.3. It consists of the following components:

- Data preprocessor: This component captures or sniffs the data that traverses the network. When it captures the data, it can perform certain preprocessing or filtering to the data as required by the module that uses it. For example, the module for detecting attacks against TCP/IP will need to process all the TCP packets, the data preprocessing will filter out all the TCP traffic and provide it as a stream to the module. More filtering is possible like filtering by source port or destination port. If the attack specification designates a certain source port as

part of the attack signature then it would be better for the performance of the system if the attack code for this signature checks only the traffic with a source port matching the one in the signature.

- Attack Signature: All the attack signatures are needed for the misuse based detection component. The signatures are written using the syntax and semantics of MSFOMTL (see Section 5.1). These are stored in a file and then it will be read and parsed by the translator.
- The misuse based intrusion detection: After finishing from parsing the attack signatures, the translator will translate each attack signature specified as MSFOMTL formula into the equivalent Stream SQL language (SSQL). The *SSQL* code are ready to run immediately on the *StreamBase* server and any traffic matching the attack signature will be reported.
- Protocol Specification: The protocol specifications are needed by the anomaly based detection component. The normal protocol specifications are written using MSFOMTL syntax/semantics (similar to the attack signature component). These will be stored in a file and will be fully parsed by the translator.
- The anomaly based intrusion detection: The parsed protocol specification will be translated into the equivalent *SSQL* language. These *SSQL* codes can be run on the server and any deviation from the protocol specification are reported.
- Intrusions log and reporting: Provide reporting and logging for the misuse and anomaly based detection components.

The above are the components for *TeStID* to handle misuse based and anomaly based intrusion detection. A detailed explanations of the specification and parsing of attacks and protocol specifications are given in Chapter 5, 6, and 8. In the subsequent sections we will introduce the tools used and the translation process from MSFOMTL syntactical form into *SSQL*.

5.3.2 Tools And Software Used

In this section we will describe the software and tools that were used to build our proposed *NIDS* as described in the previous section. These are as follows:

- **TCPDUMP** is commonly used free software developed by Lawrence Berkeley Laboratory to intercept and display packets being transmitted or received over the network to which a host is attached. All packets (TCP, ICMP, UDP, etc.) can be captured and not only TCP as the name of the software might indicate. It can be used online at the command prompt to capture network traffic and either display or write the captured packets into a file. It can also read previously captured file as source of input [98].

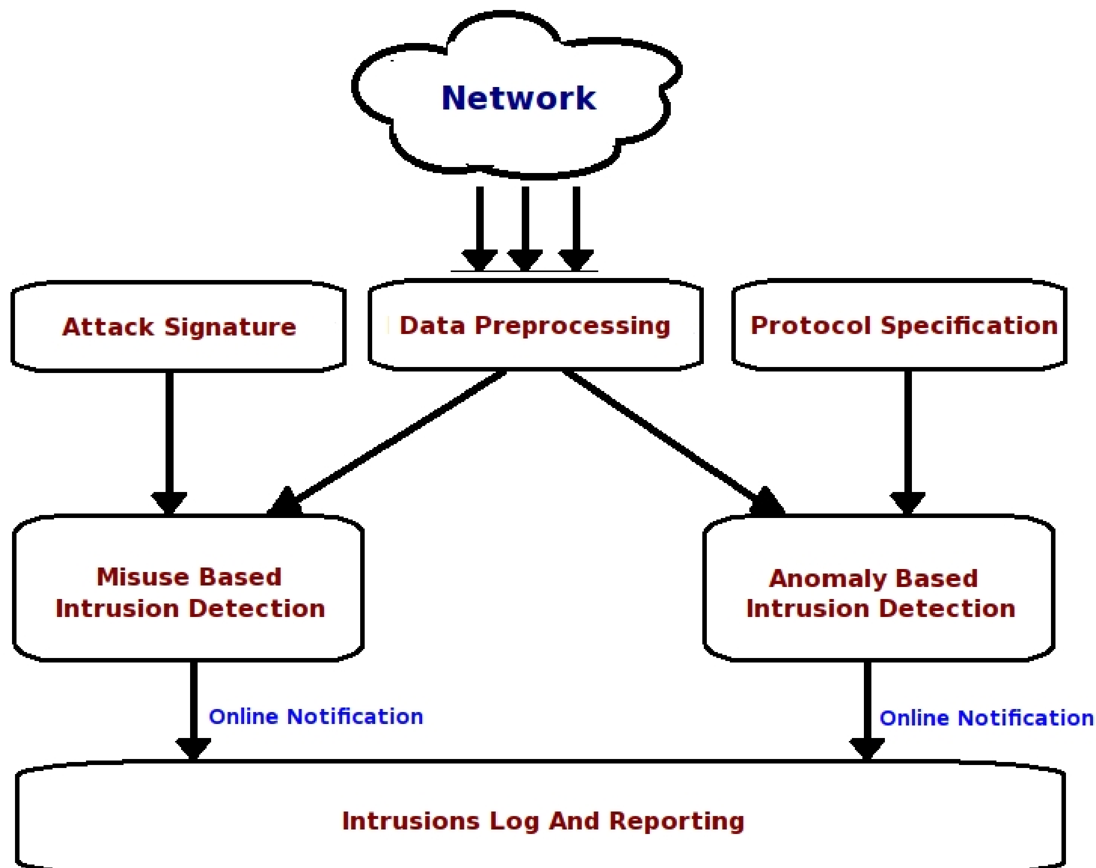


FIGURE 5.3: *TeStID* System Architecture

- **LIBPCAP** Also is a common free software library developed by Lawrence Berkeley Laboratory. It provides a high level user interface to the TCPDUMP capture system. It consists of an application programming interface (API) which can be used by other programs such as IDS to capture network traffic [98].

- **JPCAP** is a Java library based on **LIBPCAP** for capturing and sending network packets [46].

- **STREAMBASE** is the stream data processing software described in Sections 4.1 and 4.2.

- **ANTLR** (ANother Tool for Language Recognition). ANTLR is a tool that provides a framework for constructing recognizers, compilers, and translators from formal or grammatical descriptions [72]. Using ANTLR allow us to parse the MSFOMTL formulae and to translate them into *SSQL* as presented in Chapter 6.

5.3.3 The Benefits of the Proposed System

There are many benefits of the proposed system that we can summarize as follows:

- It provides a concise and unambiguous way to formally write the attack specifications or protocol specifications (details in Chapters 5 and 8).
- It is extensible as in case of new attacks all that is needed is to add the attacks formula in the formulae files and run the translator again.
- It is scalable.
- The experiments results in Chapter 7 shows that it is a promising solution even though that we used a development version of *StreamBase* engine as the server deployment engine provides much powerful execution speed.

5.4 Summary

This chapter presented the proposed system *TeStID*. In building the proposed system in this research we use Many Sorted First Order Metric Temporal Logic (MSFOMTL). The syntax and semantics of this logic was given. *MSFOMTL* is very expressive and allows the specification of complex temporal patterns of events in concise and unambiguous way. We presented here several syntactic attack patterns and showed how to use them to represent some known attacks. This chapter also presented *TeStID* system architecture, descriptions of the tools and utility used for the development, and highlighted the benefits of the proposed system.

The next chapter provides some background of using temporal logic to query databases. Also, it presents the mappings of the MSFOMTL syntactical forms defined for attack patterns into *SSQL*, the correctness of the approach, and the translation process.

Chapter 6

Temporal Logic to Stream Queries

In Chapter 5 we proposed the use of Many Sorted First Order Metric Temporal Logic *MSFOMTL* to specify complex temporal patterns of events. Also, we presented several syntactic attack patterns using a fragment of the proposed logic which are sufficient to express many known attacks. In Chapter 4 we introduced the stream data processing and its features in handling complex stream of events through in memory processing of stream queries. Specific details about *StreamBase* were given as we selected it as the stream processing engine to develop *TeStID*.

In this chapter we will present how to map and translate the temporal logic formulae into *SSQL*. The outcome of the translation is the efficient stream queries that use the practically defined temporal patterns in *MSFOMTL*. The translations here cover the misuse based detection part of *TeStID* and the anomaly based detection will be covered in Chapter 8.

Section 6.1 provides a historical background of using temporal logic to query databases. In Section 6.2, we explain the view of time in the temporal logic and the temporal database. Section 6.3 is about the mapping of MSFOMTL into *SSQL*. The correctness of this approach is given in Section 6.4. The detail of how the translator is built is given in 6.5. Finally, a summary is presented in 6.6.

6.1 Background

Time is an important aspect of the real world. Events occur in specific points of time and time is used to relate their occurrences. Thus, modelling the temporal aspect of the real world is very important in real-time computer systems. These temporal events are stored in temporal databases. Temporal databases hold all the information required by applications where the timing properties are an essential part of the processing. Early research started more than 30 years ago and temporal logic was seen as the natural choice for querying temporal databases [14, 75, 86]. This research mainly took three directions. In the first direction, the research concentrated on extending the existing SQL language

with temporal handling capabilities [63, 81]. So, they believed and tried to show that extending the capabilities of these well known SQL language is sufficient to express the temporal processing requirements. The researchers in the second directions believed in developing a temporal logic-based high level query language for temporal specification and reasoning. This approach attempted to take full advantage of the mathematical and temporal expressiveness properties of the logic to build an optimized system [29, 104]. The third directions is actually between the two earlier mentioned directions. In this approach, a high level temporal queries written by the user is translated into temporal SQL. The work done by Chomicki et al.[14] is the earliest work in this direction in which temporal logic formulae were translated into a subset of ATSQL, a temporal extension to SQL-92. The advantage of the approach is that it provides the user with a high level abstraction to write queries against the temporal database. Practically, these queries are translated into the equivalent ATSQL language and then executed.

In temporal databases there are different time aspects that could be used. These are *valid time*, *transaction time*, and *bitemporal time* (the temporal database contains both the *valid time* and the *transaction time*). These times were initially defined in the work of Snodgrass and Ilsoo [86]. The *valid time* is defined as the time period where the data is considered true with respect to the real world. The *transaction time* is the time in which the database fact is/was stored. For instance, in a student registration database, a student (e.g., Abdul) is registered to study from 24/09/2011 till 24/09/2012. So, this reflects the *valid time* for the student registration data. This student registration data is entered on the 01/09/2011 into the database and represents the start of the transaction time. Both the *valid time* and the *transaction time* have a start time and an end time. If the end time of the valid time is unknown, then it would be filled with infinity (∞) in the database. Also, the end time can be infinite in the transaction time and it means that there is no record supersedes the current record. Obviously, the data manipulation requirements of the application with respect to time, specify the choice of the timing that will be used.

Stream databases involve a move from static to data streams. Unlike conventional and temporal databases they are characterized by the requirements of continuous processing over flows of data (i.e., streams). The data model is transient (stored in memory) and not persistent (stored in the database files). Queries are continuously evaluated as the data flows which are potentially unbounded. When data items arrive, they can be processed with stream SQL queries and are implicitly ordered by their arrival time or explicitly by time stamps. These time stamps are allocated or mapped to each arrived tuple. Chapter 4 of this thesis presented more historical and detailed information about the stream management systems.

As it was presented in in Section 5.3, in the proposed system the MSFOMTL formulae will be mapped into *SSQL* which is the language that can run directly on the *StreamBase* server. This mapping will be used to develop the translator. The rest of this chapter presents the translation process in detail.

6.2 The view of Time

Verification of real-time systems require formal specification of properties with respect to time. Hence, using quantitative time is suitable for modelling and specifying real-time systems [4]. Understanding the real system timing aspect influences the abstract view of time that will be used in modelling with temporal logic.

For intrusion detection systems, the inputs are all the packets that traverse the network. These packets arrive arbitrarily as time goes in increasing order sequence. The fact that these packets arrive arbitrarily influence our choice to select a dense-time model. We can look at these packets as events occurring in linear order. This linear order can be represented by the set of positive real numbers $\mathbb{R}^+$ as we monitor the traffic online. Thus, in MSFOMTL (Chapter 5), we considered the time to be dense and it is point based on the arrival time of packets. In the dense-time model there is a point between any two points, we consider the set of points $\mathcal{T}$ in any bounded interval countable, that is, $\mathcal{T} \subset \mathbb{R}^+$.

This formal model of time must have a corresponding execution model on the target language. *SSQL* syntax and semantics supports processing streams of events by their arrival times, that is, implicitly ordered. In fact it also supports explicit ordering by time stamp (through system function calls) or even a numeric sequence order number (through sequence generator function). Any reference to time in this thesis refers to arrival time. In *SSQL*, the stream is a sequence Σ of events (tuples), each event is a σ indexed by its arrival time τ_i, that is, $\Sigma = (\sigma_{\tau_0}, \sigma_{\tau_1}, \sigma_{\tau_1}, ...)$. An event σ_{τ_i} occurrence at an arrival point of time τ_i means $\sigma_{\tau_i} \in \Sigma$.

Each packet in the network is represented as a predicate in temporal logic and as a tuple in the *SSQL*. In temporal logic the predicate valuation is linked to its arrival point, whereas in *SSQL*, the presence of a tuple is linked with its implicit actual arrival time point. This will be clear in the discussion of the next section.

6.3 Mapping MSFOMTL into *SSQL*

A well formed formula written using MSFOMTL needs to be evaluated over finite but expandable sequence of events as the time advances. *SSQL* provides the capability of continuously evaluating a query against flows of data. So, the mechanism of evaluation is the one needed to conduct run time verification. Basically, these queries are the mappings of the temporal formulae. The mapping is the process of translating the syntax and semantics of MSFOMTL into the equivalent syntax and semantics of *SSQL*.

We consider that the translation is successful if the formula semantic is equivalent (has the same valuation) in the translated code. SQL was originally developed based on the influential paper of Edgar Codd [16] of relational model based on first order predicate logic. In that model, a relation with n degree is a table with n columns, a tuple is a row of a table, and an attribute is a column of a table. In logic the predicate formally represents a packet. This predicate name in *SSQL* is simply a table name (relation in

terms of temporal database). Predicates are mapped to tuples, that is, the terms in the predicate are mapped to the fields of the tuple. These terms are of different sorts where each sort has a predefined range. Stream SQL provides temporal extensions to SQL.

We show here how to translate the syntactical forms that we have defined using a fragment of the MSFOMTL into *SSQL*. Some related definitions to the mapping process are as follows:

- *Schema* in *SSQL* refers to a relation or a table with n degree. In the proposed system *TeStID* (Figure 5.3), a preprocessing of the network data is conducted. So, the traffic is filtered based on the protocol type and used as an input stream. In this thesis, we use the TCP/IP protocol as a case study, thus, a schema is defined for the TCP/IP protocol. The columns of the schema correspond to the sorts defined in Section 5.1.1.
- *A tuple* is a row of a table consists of an ordered list of elements.
- *Input Stream* is the source of input for the stream query. The input stream is structured according to the schema definition. In *TeStID* this is implemented as input Java adapter that uses the TCPDUMP/LIBPCAP API (introduced earlier in Chapter 5) to capture packets from the network interface in promiscuous mode (i.e., listening to not only the traffic addressed to the interface but also to all the traffic seen by the interface).
- *Output Stream* is the result of the query sent to the I/O device.
- *Stream* is an intermediate stream. A query processing an input stream can send its output to an output stream or to an intermediate stream that acts as an input stream to another query.

Four syntactical forms were defined in Section 5.2. In the following we define the mapping functions for each of these syntactical forms. These functions map the formulae of the same syntactical form into the same *SSQL* constructor and following the same mapping process.

1. *Single packet attacks*: the syntactical form of these attacks was represented in Section 5.2 and is reproduced here:

   $$P \wedge Q$$

   where:

   - P is first order predicate (representing a packet).
   - Q is conjunction of Boolean formulae relating to terms in P.

   The mapping process consists of two stages: pre-mapping preparations and the actual mapping process. In the pre-mapping preparations, the data preprocessing

and the supporting *SSQL* components needed for the mapping process are handled. In Figure 5.3 preprocessing of data is one component of *TeStID* in which the data is preprocessed for further processing. *StreamBase* in general, recommends having preprocessing or filtering of data as early as possible in stream processing for more efficient processing (processing only the relevant or filtered tuples by the later components). Also, in this pre-mapping stage, certain components need to be defined to support the mapping process. This depends on the target *SSQL* constructor for the mapping. The target mapping *SSQL* constructor for this syntactical form is the *filter* constructor. This constructor requires an input stream, output stream, and it has a "WHERE" clause for specifying conditions on the data. The input and output stream is created in the pre-mapping stage as explained in the following:

- The input for the system is provided by the Java input adapter. This adapter sniffs packets from the network and all the TCP/IP packets are streamed in to what is called the inputstream. Subsequently, the inputstream can be used by other *SSQL* components. The following is the code generated for this:

```
CREATE STREAM inputstream ;

APPLY JAVA "TCP_W_Payload" AS TCP_W_Payload (
  schema0 = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
  <schema name=\"schema:TCP_W_Payload\">\n
  <field description=\"\" name=\"x1\" type=\"string\"/>\n
  <field description=\"\" name=\"x2\" type=\"int\"/>\n
  <field description=\"\" name=\"x3\" type=\"string\"/>\n
  <field description=\"\" name=\"x4\" type=\"int\"/>\n
  <field description=\"\" name=\"x5\" type=\"long\"/>\n
  <field description=\"\" name=\"x6\" type=\"long\"/>\n
  <field description=\"\" name=\"x7\" type=\"bool\"/>\n
  <field description=\"\" name=\"x8\" type=\"bool\"/>\n
  <field description=\"\" name=\"x9\" type=\"bool\"/>\n
  <field description=\"\" name=\"x10\" type=\"bool\"/>\n
  <field description=\"\" name=\"x11\" type=\"bool\"/>\n
  <field description=\"\" name=\"x12\" type=\"string\"/>
  \n</schema>\n"
)
INTO inputstream;
```

  In the above code there are two *SSQL* commands. One command is used to create the stream inputstream (CREATE STREAM) and one is used to call the Java adapter (APPLY JAVA) that will sniff all the TCP/IP packets and direct the output to the inputstream. Now, the input stream is ready for use the other components.

- Another supporting *SSQL* component is the outputstream. This is the component that can receive the result of queries and direct them to the I/O device. The code to create the outputstream is:

```
CREATE OUTPUT STREAM outputstream;
```

The second stage is the mapping process. For the mapping, we define the mapping function (M1) which maps a subset of MSFOMTL (Δ) into a subset of *SSQL* (Θ):

$$M1 : \Delta \longrightarrow \Theta$$

The basic elements of the mapping function M1 are defined as follows:

$\boxdot \in \{=, <>, >, <, >=, <=\}$ // $\boxdot$ represents a relational operator

$\boxtimes \in \{+, -, *, /\}$ // $\boxtimes$ represents a mathematical operator

$M1 : P(x_1, x_2, ..., x_n) \mapsto$ " SELECT $x1, x2..., xn$ FROM *input_stream*"

$M1 : \wedge \mapsto$ "WHERE " //the conjunction between P and Q

$M1 : \wedge \mapsto$ " and " //the conjunction in Q

$M1 : \vee \mapsto$ " or " //the disjunction in Q

$M1 : c_i \mapsto ci$ // where c is constant

$M1 : x_i \mapsto xi$ // where x is variable

$M1$: "(" $\mapsto$ "(" // parenthesis is in the Q

$M1$: ")" $\mapsto$ ")" // parenthesis is in the Q

$M1 : regexp \mapsto regexp$ //regexp is regular expression in Java syntax

$M1 : x_i \boxdot \langle c_j | x_j \rangle \mapsto xi \boxdot \langle cj | xj \rangle$

$M1 : x_i \boxdot (x_j \boxtimes \langle x_k | c_k \rangle) \mapsto xi \boxdot (xj \boxtimes \langle xk | ck \rangle)$

$M1 : LF \mapsto$ " INTO outputstream; " //line feed

In the above notice that the mapping of "$\wedge$" is translated based on its location in the syntactical form.

Example 6.1: the following formula ϕ is a typical example of this syntactical form:

$$(\exists x_1, x_3, x_4)(P(x_1, ..., x_{12}) \wedge ((x_1 = x_3) \wedge (x_4 = 80 \vee x_4 = 8080))$$

The mapping to *SSQL* using the mapping function M1 will be :

$$\begin{aligned} Mapping(\phi) = \ & M1(P(x_1, ..., x_{12}) + M1(\text{`` } \wedge \text{ ''}) + M1(\text{``(''}) + M1(\text{``(''}) + M1(x_1 = x_3) \\ & + M1(\text{``)''}) + M1(\text{`` } \wedge \text{ ''}) + M1(\text{``(''}) + M1(x_4 = 80) \\ & + M1(\text{`` } \vee \text{ ''}) + M1(x_4 = 8080) + M1(\text{``)''}) + M1(\text{``)''}) \end{aligned}$$

The mapped *SSQL* code is the following:

```
SELECT * FROM inputstream
WHERE ((x1 = x3) and (x4 = 80 or x4 = 8080))
INTO outputstream;
```

2. *Forward multiple packet attacks*: these attacks cover attacks carried in multiple packets with temporal distances in between. The descriptions for this classification and syntax were given in Section 5.2 and the syntax is reproduced here:

$$\varphi \wedge \Diamond_{[t_1,t_2]}\psi$$

or:

$$\varphi \wedge \neg\Diamond_{[t_1,t_2]}\psi \tag{6.1}$$

where:

- φ is a first order predicate (representing a packet).
- ψ is either a first order predicate or the same formula as 6.1.

Like in the previous syntactical form, the mapping is done in two stages: pre-mapping preparations and mapping to the *SSQL pattern* matching operator. The pre-mapping stage consists of all the steps mentioned for the pre-mapping of the first syntactical form in (1) plus another preprocessing step. This step will create two additional streams by filtering the main stream by the contents of the constant values of each predicate. The code template for this looks like:

```
CREATE STREAM Filter1;     // creates input stream for first predicate
CREATE STREAM Filter2;     // creates input stream for second predicate
SELECT * FROM Inputstream
WHERE xi = cj [and ...] // condition(s) on constant value(s)
INTO Filter1           // in predicate 1
WHERE xi = ck [and ...] // condition(s) on constant value in the predicate
INTO Filter2;          // in predicate 2
```

For the mapping, we define the mapping function (M2) that maps a subset of MSFOMTL (Δ) into a subset of *SSQL* (Θ):

$$M2 : \Delta \longrightarrow \Theta$$

Notice here that we use different function name (M2) from the previous syntactical form (M1). The reason is that M1 maps to a different *SSQL* constructor from M2. M1 maps to the *filter* constructor that has no temporal properties (the first syntactical form does not have metric operator). M2 maps to the *pattern* constructor that has temporal properties.

The mapping process to the pattern constructor deals with at least two predicates. We can define the basic elements of the mapping function M2 as follows:

$\boxdot \in \{=, <>, >, <, >=, <=\}$ // $\boxdot$ represents a relational operator

$\boxtimes \in \{+, -, *, /\}$ // $\boxtimes$ represents a mathematical operator

$M2 : constant \mapsto constant$

$M2 : x_i \mapsto xi$

$M2$: "(" $\mapsto$ "(" // parenthesis is in the Q

$M2$: ")" $\mapsto$ ")" // parenthesis is in the Q

// The first predicate is the first input to the pattern constructor and it is

// aliased as input1 and mapped as:

$M2 : P(x_1, ..., x_n) \mapsto$ " SELECT input1.x1 as input1_x1, ..., input1.xn as input1_xn,"

// The second predicate is the second input to the pattern constructor and is

// aliased as input2 and mapped as:

$M2 : P(x_1, ..., x_n) \mapsto$ " input2.x1 as input2_x1, ..., input2.xn as input2_xn,"

// The pattern template is constructed from the temporal operator of the

// formula as follows:

$M2$: " $\wedge \Diamond$" $\mapsto$ " FROM PATTERN (Filter1 as input1 THEN Filter2 as input2)"

$M2$: " $\wedge \neg\Diamond$" $\mapsto$ " FROM PATTERN (Filter1 as input1 THEN NOT Filter2 as input2)"

$M2 : [t_1, t_2] \mapsto$ " WITHIN $t_2 - t_1$ TIME "

// The WHERE clause is constructed from the bound variables which must have

// the same values in all the predicates:

$M2$: (bound variables) $\mapsto$ "$WHERE''$

for example:

$(\exists x_1, x_3)P(x_1, x_2, x_3, ..., x_n) \wedge \Diamond_{[t_1,t_2]}P(x_3, x_4, x_1, ..., x_n) \mapsto$

" WHERE input2.x1 = input1.x3 and input2.x3 = input1.x1 ... ;"

$M2 : LF \mapsto$ " INTO outputstream; " //line feed

Example 6.2: We can use the *TCP Reset* attack which is used as case study in Chapter 7 and represented with formula 7.1. The formula is reproduced below:

$$(\exists x_1, x_2, x_3, x_4, x_5)((\exists y_{10}, y_{11}, y_{12})$$
$$P(x_1, x_2, x_3, x_4, x_5, 0, 0, 1, 0, y_{10}, y_{11}, y_{12}) \wedge$$

$$\Diamond_{[0,1]}(\exists z_{10}, z_{11,z_{12}})$$
$$P(x_1, x_2, x_3, x_4, (x_5+1), 0, 0, 0, 1, z_{10}, z_{11}, z_{12}))$$

First we apply the pre-mapping rules for this syntactical forms and this gives the following *SSQL* code:

```
CREATE STREAM inputstream ;

APPLY JAVA "TCP_W_Payload" AS TCP_W_Payload (
  schema0 = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
  <schema name=\"schema:TCP_W_Payload\">\n
  <field description=\"\" name=\"x1\" type=\"string\"/>\n
  <field description=\"\" name=\"x2\" type=\"int\"/>\n
  <field description=\"\" name=\"x3\" type=\"string\"/>\n
  <field description=\"\" name=\"x4\" type=\"int\"/>\n
  <field description=\"\" name=\"x5\" type=\"long\"/>\n
  <field description=\"\" name=\"x6\" type=\"long\"/>\n
  <field description=\"\" name=\"x7\" type=\"bool\"/>\n
  <field description=\"\" name=\"x8\" type=\"bool\"/>\n
  <field description=\"\" name=\"x9\" type=\"bool\"/>\n
  <field description=\"\" name=\"x10\" type=\"bool\"/>\n
  <field description=\"\" name=\"x11\" type=\"bool\"/>\n
  <field description=\"\" name=\"x12\" type=\"string\"/>
  \n</schema>\n"
)
INTO inputstream;

CREATE OUTPUT STREAM outputstream;

CREATE STREAM Filter1;
CREATE STREAM Filter2;

SELECT * FROM InputAdapter
WHERE x9 = 0 and x8 = 1 and x7 = 0 and x6 = 0
INTO Filter1
WHERE x9 = 1 and x8 = 0 and x7 = 0 and x6 = 0
INTO Filter2
;
```

In the second stage the mapping processing is conducted. The *SSQL* code is the concatenation of the results of the following M2 function calls:

$M2(P(x_1, ..., x_n))+$ //P is the first predicate

$M2(P(x_1, ..., x_n))+$ //P is the second predicate

$M2(\wedge\Diamond)+$

$M2 : ([t_0, t_1])+$

$M2$(bound variables)

The mapped *SSQL* code is the following:

```
SELECT input1.x1 AS input1_x1, input1.x2 AS
input1_x2, ...
input2.x1 AS input2_x1, ...
FROM PATTERN (Filter1 as input1 THEN Filter2 as input2)
WITHIN 1 TIME
WHERE input2.x1 = input1.x1 and
input2.x2 = input1.x2
and input2.x3 = input1.x3
and input2.x4 = input1.x4
and input2.x5 = (input1.x5 + 1)
INTO outputstream;
```

If three predicates are involved, then two pattern operators are used. The first pattern operator will be mapped as described above. The outcome of the first pattern is used as the first input to the second pattern operator, and the third predicate will be used as the second input. If four predicates are involved, then three pattern operators are needed, and so on.

3. *Backward multiple packet attacks*: As we describe this classification in 5.2, these attacks occur when observing an event and a certain event was not present in the past. The syntax form of this class of attacks is reproduced as:

$$\varphi \wedge \blacksquare_{[t_1,t_2]} \neg\psi$$

where:

- φ and ψ is a first order predicate (representing packets).

The mapping of this class is more complicated than the previous ones. This is because the match pattern operator work with forward sliding windows. This means it can not be used to look for something happen in the past (for some windows) when certain events occur at the moment. But the requirements in the formula can be translated using other built in *SSQL* operators.

As in the previous syntactical forms there are two stages: the pre-mapping and the mapping process. To explain the the steps of the mapping, a typical example is given and presentations of the steps are explained.

Example 6.3: As an example we take the *Reset Scan* attack that was represented earlier in formula 5.7 and reproduced as follows:

$$(\exists x_1, x_2, x_3, x_4)((\exists y_5, y_6, y_7, y_8, y_{10}, y_{11}, y_{12})$$
$$P(x_1, x_2, x_3, x_4, y_5, y_6, y_7, y_8, 1, y_{10}, y_{11}, y_{12})$$

$$\wedge\blacksquare_{[0,300]}(\exists z_5, z_6, z_7, z_8, z_{10}, z_{11}, z_{12})$$
$$\neg P(x_1, x_2, x_3, x_4, z_5, z_6, z_7, z_8, 0, z_{10}, z_{11}, z_{12}))$$

To help in presenting the steps of the mapping process, we use Figure 6.1 which is a graphical representation of the mapped *SSQL* code. The steps for the mapping is as follows:

(a) The pre-mapping stage is similar to the one done in the first syntactical form. The input stream is created by calling the Java input adapter and the output stream is created with the CREATE STREAM statement. Here, an additional pre-mapping step is performed. Each incoming tuple will be associated with a time stamp value (predicate_time). In Figure 6.1 it is represented by the map operator that maps a time stamp to each tuple comes out of the input adapter. The code is:

```
CREATE STREAM out_Map;
SELECT
    inputstream.* AS *,
    to_milliseconds(now()) AS predicate_time
FROM   inputstream
INTO   out_Map
;
```

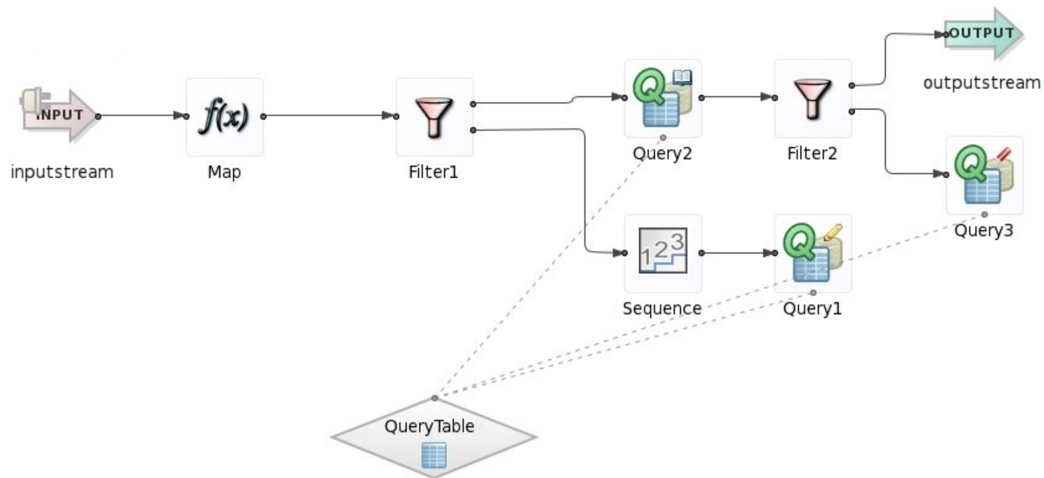


FIGURE 6.1: Reset Scan in *StreamBase* Studio

(b) In the next step preprocessing is performed similar to the preprocessing in the previous example, that is by the constant values in each predicate. The only constant in our example formula is x_9 and is represented by the operator Filter1 in Figure 6.1. The *SSQL* code is as follows:

```
CREATE STREAM out_Filter1_1;
CREATE STREAM out_Filter1_2;
```

```
SELECT * FROM out_Map
where x9 = 1 INTO out_Filter1_1
where x9 = 0 INTO out_Filter1_2
;
```

We have two filtered streams, one with the the reset flag set and the other one with the reset flag unset. Further processing from here is split into two. In one path, all the tuples with flag unset are stored in the memory table with a sequence number for each tuple that will be the primary key because the sequence number is unique. In Figure 6.1 this is represented by the Sequence operator (generate a sequence for each tuple), the Query1 operator inserts the tuples in the QueryTable (memory table). On the other path, whenever a tuple arrives with the reset flag set, a query is issued against the memory table to see if there is any tuple exist with the values equal to the bounded variables specified for all the predicates in the formula, that is x_1, x_2, x_3, x_4. Also, the tuple arrival time is checked if it is within the last 300 seconds which is specified in the formula. If there is no tuple that satisfies these conditions then this means that there was no communication for 300 seconds when the reset request is received. The operations in this path are represented in Figure 6.1 by Query2 (to read all the records in the table outer joined with the incoming stream), Filter2 (to check if no tuple exist within 300), and the OutputStream (emit the result). Also, there is a maintenance operations that are performed against the memory table which is represented by the second path of Filter2 in the diagram, these are the Query3 and QueryTable. Query3 is a query to read and delete all the records found (this means there were some communications) or if the arrival time (predicate_time) is more than 300 seconds as we are only concerned with the last 300 seconds. The following steps go through the mapping for the first path and then for the second path.

(c) To generate the sequence the following code does this and the output stream from this step is assigned to out_Sequence.

```
DECLARE sequence1id long DEFAULT 0;
CREATE STREAM gen_seqIdSetter1 (
    packetno long
);

CREATE STREAM out_Sequence ;
SELECT sequence1id + 1 AS packetno FROM out_Filter1_2
INTO gen_seqIdSetter1;
SELECT *, sequence1id AS packetno FROM out_Filter1_2
INTO out_Sequence;
UPDATE sequence1id FROM (SELECT * FROM gen_seqIdSetter1)
;
```

(d) The sequenced stream in the previous step is writing into the memory table. The memory table needs to be created first and the code for that is:

```
CREATE MEMORY TABLE QueryTable1
(
packetno long,
predicate_time double,
x1 string,
x2 int,
x3 string,
x4 int
)
PRIMARY KEY (packetno) USING BTREE
SECONDARY KEY (predicate_time,x1,x2,x3,x4) USING HASH
;
```

Notice here we used the created sequence number and the global bounded variables in the formula. As the primary key is sorted, the btree index type is selected (also is the default in *StreamBase*). The secondary key is optional, but as we access the memory table later with these keys, using the index makes it faster. Next, the tuples are inserted. This can be inserted with the insert or replace *SSQL* commands. Here we use the replace command as there is no need to worry if a duplicate record is inserted as it would be replaced.

```
REPLACE INTO QueryTable (packetno, predicate_time, x1,x2,x3,x4 )
    SELECT packetno, predicate_time, x1,x2,x3,x4
    FROM out_Sequence
;
```

(e) The second path mentioned in (b) starts when a tuple arrives with the reset flag set. As we described in (b) the following code is for the Query2 operator:

```
CREATE STREAM out_Query2 ;
SELECT out_Filter1_1.x1, out_Filter1_1.x2, out_Filter1_1.x3,
out_Filter1_1.x4, out_Filter1_1.x5, out_Filter1_1.x6,
out_Filter1_1.x7, out_Filter1_1.x8, out_Filter1_1.x9,
out_Filter1_1.x10, out_Filter1_1.x11, out_Filter1_1.x12,
out_Filter1_1.predicate_time,
QueryTable.packetno AS tablepacketno,
QueryTable.predicate_time AS tablepredicate_time,
QueryTable.x1 AS tablex1,QueryTable.x2 AS tablex2,
QueryTable.x3 AS tablex3,QueryTable.x4 AS tablex4
FROM out_Filter1_1 OUTER JOIN QueryTable

WHERE QueryTable.predicate_time >=
                       (out_Filter1_1.predicate_time - 300)
and QueryTable1.predicate_time <=
                        out_Filter1_1.predicate_time
and QueryTable.x1 = out_Filter1_1.x1
```

```
and QueryTable.x2 = out_Filter1_1.x2
and QueryTable.x3 = out_Filter1_1.x3
and QueryTable.x4 = out_Filter1_1.x4
LIMIT 1
INTO out_Query2
;
```

The following code is for the Filter2 where we check the outcome of the query if some tuple does not exist within the last 300 seconds. If there are some records, this means there is no attack and the stream is directed to do the maintenance step mentioned in (b).

```
CREATE STREAM out_Filter2 ;
SELECT * FROM out_Query2
WHERE isnull(tablex1) INTO OutputStream
WHERE true INTO out_Filter2
;
```

(f) In the maintenance step mentioned in (b), Query3 is the query for delete operation and the code for this is as follows:

```
DELETE FROM QueryTable
USING out_Filter2_2
WHERE (QueryTable.predicate_time < out_Filter2.predicate_time - 300)
or (QueryTable.x1 = out_Filter2.x1 and
QueryTable.x2 = out_Filter2.x2 and
QueryTable.x3 = out_Filter2.x3 and
QueryTable.x4 = out_Filter2.x4)
;
```

4. *Repetition attacks*: As described in Section 5.2, these attacks have no effect unless they are repeated finitely many times (n) within a specified time window. The syntactical form of these attacks is reproduced as:

$$R^{n}_{[t_1,t_2]}\varphi \tag{6.2}$$

where:

- φ is any formula of the preceding classifications.

φ is translated according to its class. We just need the output of the matched patterns that will be used as input to the repetition operator. The repetition operator is mapped using the available aggregate constructors of the *SB* language using the repetition number and window time as parameters. As example, suppose φ was the formula in Example 6.3 and suppose the repetition number was "(20)" and the time window is [0,5]. After we translate the formula as we presented in Example 6.3, we will use the aggregate select statement to count the number of packets arriving in a window of size of 20 tuples and window advance by 1. Then we check to see if the last arrived matched pattern minus the first arrived matched

pattern is less than 5 seconds. The added code to Example 6.3 will be to create the aggregate pattern window (the size and advancement) and then read the final output stream (outputStream) in this pattern context:

```
CREATE STREAM patternSum ;
CREATE WINDOW sumOfPattern(SIZE 20 ADVANCE 1 TUPLES);

SELECT
    count() AS Numberpackets,
    firstval(predicate_time) AS FirstPatternT,
    lastval(predicate_time) AS LastPatternT,
    firstval(*) AS input_*
FROM outputstream[sumOfPattern]
INTO patternSum;

CREATE OUTPUT STREAM outputstream2;
SELECT * FROM patternSum
WHERE ((LastPatternT - FirstPatternT) <= 5000) INTO outputstream2
;
```

The `firstval(p_time)` and `lastval(p_time)` represent the time of the first and the last tuple matched pattern in a current window respectively. We need the the time between the first pattern and the last to be less or equal to 5 seconds.

6.4 Correctness

In this research we propose the use of Many Sorted First Order Metric Temporal logic (MSFOMTL) to formally represent attack signatures or normal behaviour. These are the temporal patterns that we need to match against network traffic. Then, we translate the *MSFOMTL* formulae that represent attacks into the *SSQL* code. Consequently, this *SSQL* code is run to detect temporal patterns specified in the original formula in the incoming events.

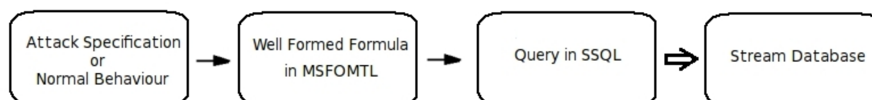


Figure 6.2: Steps of Specification Translation into Stream Queries

Initially, the relational database model was introduced by Codd [16]. Later research studied the relationship between logic and relational databases [78]. In these research the database can be viewed as a set of first order formula. This view is called the model-theoretic view and can be describe as:

- A database (DB) is a model (i.e., DB is an interpretation $\mathcal{I}$ of a first-order logical language $\mathcal{L}$).
- A query is a formula α of $\mathcal{L}$.
- A query evaluation is a formula α evaluation with respect to the model (DB).

A query is an expression $\mathcal{Q}_\alpha$ of the following form:

$$\mathcal{Q}_\alpha = [\bar{x} \mid \bar{Q}\bar{y}\ \alpha(\bar{x}, \bar{y})]$$

where:

- $\alpha(\bar{x}, \bar{y})$ is quantifier free first order formula with free variables $(\bar{x}, \bar{y})$.
- $\bar{Q} = Q_1, Q_2, ...$ where Q_i is either existential quantifier $\exists$ or universal quantifier $\forall$.

In logical perspective, $\mathcal{Q}_\alpha$ is evaluated in model $\mathcal{M} = \langle \mathcal{T}, <, \mathrm{I} \rangle$ at arrival time τ as:

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = \{\bar{a} \mid \mathcal{M}, \tau \models \bar{Q}\bar{y}\ \alpha(\bar{a}, \bar{y}) \ \text{ where } \bar{a} = \bar{x}^I\}$$

In stream query perspective, a stream query (SQ) is evaluated continuously against stream of incoming events. These incoming events form the temporal models. Each event is a packet arrived at some moment of τ. The following are some basic mappings used during the translation from α into *SSQL* query:

- predicate names are mapped to schema names,each of which is a relation or a table with n-degree. This schema name refers to packets of type TCP/IP, UDP, ICMP, etc.;
- interpretations of predicates ($P(\bar{x}, \bar{y})$) are mapped to set of tuples $(\bar{x}^I, \bar{y}^I)$;
- sorts are mapped to tuple elements (or columns);
- constraints on sorts are mapped to constraints on columns.

At each moment (τ) a running query may return a set of tuple or no tuple at all. Query valuation (v) is considered true in a given database if it is answered, that is, a set of tuples is returned. If no tuple returned then the valuation of the query considered false. So, if α holds in $\mathcal{M}$, then the translated stream query SQ of α in $\mathcal{Q}$ must have true valuation. Thus the evaluation of $SQ_{\mathcal{Q}_\alpha}$ in a given database M (DB) is as follows:

$$Eval(SQ_{\mathcal{Q}_\alpha}, M(DB), \tau) = \{\text{set of tuples } (\bar{x}) \text{ for a given } \bar{y}\}$$

As can be seen from Figure 6.2, we write the attack signatures (or normal behaviour specification) in well formed formula α using MSFOMTL. This formula is translated

into *SSQL* query. So, the formula evaluation in a model $\mathcal{M}$ will be transformed into query evaluation in M (DB). The translation is correct if the following is true:

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = Eval(SQ_{\mathcal{Q}_\alpha}, M(DB), \tau)$$

In the proposed system, there are four syntactical forms. For each syntactical form, the translation is correct if the formula interpretation at a moment τ is equivalent to the stream query valuation at τ of the translated formula with respect to their models. For the first syntactical form, and using its syntax as defined in formula 5.1, we can write the following typical syntax formula:

$$(\exists x_1 \ldots \exists x_n)((\exists y_1 \ldots \exists y_m)P(x_1, ..., x_n, y_1, ..., y_m) \wedge [\text{conditions on } x_1...x_n]) \quad (6.3)$$

This formula corresponds to query $\mathcal{Q}_\alpha = [\bar{x}, \bar{y} \mid P(\bar{x}, \bar{y}) \wedge [\text{conditions on } \bar{x}]]$ and it will be translated to the following stream query ($SQ_{\mathcal{Q}_\alpha}$):

```
SELECT x1,..xn,y1,..ym FROM schemaname
WHERE [Conditions on x1...xn]
INTO output;
```

To show the correctness of the translation, we compare the interpretation of the formula and the valuation of the query. We need to show that they produce the same set of tuple when they evaluated in their respective models. Using the semantics defined in Section 5.1.3, formula 6.3 is evaluated to true at τ if all the sorts including the ones which have constraints on them have mapping values from the domains of the sorts, that is:

$$P^I(x_1^{I_{s1}}, ..., x_n^{I_{sn}}, y_1^{I_{s1}}, ..., y_m^{I_{sm}}) = \mathrm{I}(\tau) \quad (6.4)$$

In *SSQL* according to the semantics discussed in 4.2 the query $SQ_{\mathcal{Q}_\alpha}$ will return tuples with columns that have values satisfying the constraints. Notice that these constraints within $SQ_{\mathcal{Q}_\alpha}$ specify the same constraints on the tuples which make original formula true. It follows that for the first syntactical form (formula 6.3):

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = Eval(SQ_{\mathcal{Q}_\alpha})$$

The second syntactical form is defined in formula 5.4. This type of formulae has temporal operator(s). It will be translated to the pattern operator in *SSQL*. The pattern operator accepts inputs and allows us to specify the pattern required between the inputs. The following is typical syntax for this form:

$$(\exists x_1 \ldots \exists x_n)((\exists y_1 \ldots \exists y_m)P(x_1, ..., x_n, y_1, ..., y_m) \wedge$$
$$[\neg]\Diamond[t_1, t_2](\exists w_1 \ldots \exists w_m)P(x_1, ..., x_n, w_1, ..., w_m) \quad (6.5)$$

The above formula corresponds to the following query:

$$\mathcal{Q}_\alpha = [(\bar{x}, \bar{y}, \bar{x}, \bar{w}) \mid P(x_1, ..., x_n, y_1, ..., y_m) \wedge [\neg]\Diamond_{[t1,t2]} P(x_1, ..., x_n, w_1, ..., w_m)$$

It will be translated into the following typical *SSQL* query ($SQ_{\mathcal{Q}_\alpha}$):

```
SELECT input1.x1,...,input1.xn,input1.y1,...,input1.ym,  // input1
       input2.x1,...,input2.xn,input2.w1,..., input2.wm  // input2
FROM PATTERN (input1 THEN [NOT] input2)
WITHIN [t2 - t1] TIME
WHERE input1.x1 = input2.x1 and        // the common bound variables
       ...input1.xn = input2.xn        // for both inputs
INTO output;
```

Now, we need to examine the evaluations of $\mathcal{Q}_\alpha$ and $SQ_{\mathcal{Q}_\alpha}$ w.r.t their models. The formula 6.5 is a WFF of MSFOMTL and it will be evaluated to true if $P(x_1, ..., x_n, y_1, ..., y_m)$ holds now at a moment τ and the second predicate $P(x_1, ..., x_n, w_1, ..., w_m)$ holds at a moment of time between $[\tau + t_1, \tau + t_2]$ (or does not hold at all if there is a negation before this predicate). This means $P(x_1, ..., x_n, y_1, ..., y_m)$ holds at τ, that is:

$$P^I(x_1^{I_{s_1}}, ..., x_n^{I_{s_n}}, y_1^{I_{s_1}}, ..., y_m^{I_{s_m}}) = \mathrm{I}(\tau) \tag{6.6}$$

Then, $P(x_1, ..., x_n, w_1, ..., w_m)$ must hold (or must not hold if there is a negation) at $\tau^{'}(\tau + t_1 \leq \tau^{'} \leq \tau + t_2)$ and all the common bound variables $(x_1, ..., x_n)$ must have the same values as in the predicate in formula 6.6, such that:

$$P^I(x_1^{I_{s_1}}, ..., x_n^{I_{s_n}}, w_1^{I_{s_1}}, ..., w_m^{I_{s_m}}) = \mathrm{I}(\tau^{'}) \tag{6.7}$$

In the translated *SSQL* query $SQ_{\mathcal{Q}_\alpha}$, the query is evaluated to true if a tuple $(\bar{x}, \bar{y})$ is selected first with columns that have values satisfying the constraints. Notice that these constraints on tuples are the same constraints specified original formula for this part 6.6 that make the formula true. Then, within the time window mapped from original formula 6.5, some tuples $(\bar{x}, \bar{w})$ are selected with columns that have values satisfying the constraints (or no tuple is selected if there is negation). These tuple(s) is (are) selected within the specified time window and have the same constraints on sorts that are specified in original formula 6.7 that make the formula true (or no tuple in case of negation). It follows that :

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = Eval(SQ_{\mathcal{Q}_\alpha})$$

The second syntactical form can have two or more predicates. If there are three predicates, then the pattern operator will be used twice. Once with the first two predicates as inputs and again using the outcome tuple from this pattern as first input and the third predicates as second input. The second pattern structure and semantics is like the first pattern. This also true if more patterns are in the translated formula. So, we can

see that, inductively, the correctness are preserved.

The third syntactical form is defined in formula 5.6. A Typical syntax for this formula is as follows:

$$(\exists x_1, \ldots, x_n)((\exists y_1, \ldots, y_m)P(x_1, \ldots, x_n, y_1, \ldots, y_m) \\ \wedge \blacksquare_{[t_1,t_2]}(\exists w_1, \ldots, w_m)\neg P(x_1, \ldots, x_n, w_1, \ldots, w_m)) \quad (6.8)$$

The above formula corresponds to the following query:

$$\mathcal{Q}_\alpha = [(\bar{x}, \bar{y}, \bar{x}, \bar{w}) \mid P(x_1, ..., x_n, y_1, ..., y_m) \wedge \blacksquare_{[t1,t2]} P(x_1, ..., x_n, w_1, ..., w_m)$$

In stream SQL the pattern operator does not have past windows. Thus, to implement this type of formula we used other stream SQL operators. Figure 6.1 is typical example of this category. You can see that we need ten different operators. The main idea of the implementation is as follows:

- Create a memory table to store packets.
- Use a memory table to store packets represented by the second predicate bound variables in the formula.
- When a packet arrives that represents the first predicate, the table is checked for the the occurrences of the second predicate within the past time window. If there are no records retrieved then an alarm is raised.

For the proof of correctness we examine the queries that correspond to the predicates in formula 6.8. The formula will be evaluated to true if $P(x_1, ..., x_n, y_1, ..., y_m)$ holds now at a moment τ and $P(x_1, ..., x_n, w_1, ..., w_m)$ did not hold at any moment $\tau^{'}$ in the past $(\tau - t_1 \geq \tau^{'} \geq \tau - t_2)$. This means $P(x_1, ..., x_n, w_1, ..., w_m)$ holds at τ, that is:

$$P^I(x_1^{I_{s1}}, ..., x_n^{I_{sn}}, y_1^{I_{s1}}, ..., y_m^{I_{sm}}) = \mathrm{I}(\tau) \quad (6.9)$$

and, $P(x_1, ..., x_n, w_1, ..., w_m)$ did not hold at $(\tau - t_1 \geq \tau^{'} \geq \tau - t_2)$ that is:

$$P^I(x_1^{I_{s1}}, ..., x_n^{I_{sn}}, w_1^{I_{s1}}, ..., w_m^{I_{sm}}) = \mathrm{I}(\tau^{'}) \quad (6.10)$$

In *SSQL* all the tuples of the past event (representing by the second predicate in formula 6.8) are selected from the input stream and inserted into memory table, the code for that is as follows:

```
INSERT INTO QueryTable (predicate_time, x1,...)
    SELECT predicate_time, x1,...
    FROM stream
;
```

Another query runs to check if a tuple arrives with the specified constraints values at τ. These constraints values are the same values for sorts in formula 6.9 that make it true. If it arrives, then the following query is issued to check if there is at least one tuple matching the arrived packet and the query table tuple in the same time window that reflect the time of window in 6.8. Another query is run to examine the result of the previous query if it is null (no match) then the final query result is true, otherwise it will be false. The stream queries (simplified) are:

```
SELECT stream.x1,...
stream.predicate_time,
QueryTable.predicate_time AS tablepredicate_time,
QueryTable.x1 AS tablex1,...,
FROM out_Filter1_1 OUTER JOIN QueryTable

WHERE QueryTable.predicate_time >=
                       (stream.predicate_time - t2)
and QueryTable.predicate_time <=
                        styream.predicate_time
and QueryTable.x1 = stream.x1
..
and QueryTable.x4 = stream.x4
LIMIT 1
INTO output1
;
SELECT * FROM output1
WHERE isnull(tablex1) INTO OutputStream
;
```

These two queries are the translated *SSQL* for the second predicate in formula 6.8. Checking the table tuples with constraints originally mapped from formula 6.10 means if there is no tuple found then only tuple $(\bar{x}, \bar{y})$ of the first predicate (formula 6.9) is returned which makes the valuation of $SQ_{\mathcal{Q}_\alpha}$ true.
We can conclude from the above, that:

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = Eval(SQ_{\mathcal{Q}_\alpha})$$

The fourth syntactical form defined in formula 5.8 and we represent it in the following typical syntactical form:

$$R^n_{[t_1,t_2]}\varphi$$

Where φ can be one of the other syntactical forms (i.e., formulae 6.3, 6.5, and 6.8). The output of the matched patterns (φ) that will be used as input to the repetition operator. Thus, here, we are concerned with the correctness of mapping this output which will be the input to the aggregate *SSQL*. First let us examine the semantics of this according to MSFOMTL, taking the repetition number as “n” and the time window is $[t_1, t_2]$, R

holds in a model M at arrival moment τ iff:

$$\mathcal{M}, \tau \models R^{n}_{[t_1,t_2]}\varphi \text{ iff } |\{\tau' \mid (\tau_i + t_1 \leq \tau' \leq \tau_i + t_2) \text{ and } \mathcal{M}, \tau' \models \varphi\}| \geq n$$

This means there are at least n arrival points τ' between t_1 and t_2 where φ is true. In *SSQL* having the repetition number as "n" and the time window is $[t_1, t_2]$, means the aggregate select statement that the repetition formula above will be mapped to has to count the number of tuples arriving in a window of size of n tuples and window advance by 1 (i.e., we count n arrived tuples). Then we check to see if the last arrived matched pattern minus the first arrived matched pattern is within the time window. The mapped code will be the following:

```
CREATE STREAM patternSum ;
CREATE WINDOW sumOfPattern(SIZE n ADVANCE 1 TUPLES);

SELECT
    count() AS Numberpackets,
    firstval(predicate_time) AS FirstPatternT,
    lastval(predicate_time) AS LastPatternT,
    firstval(*) AS input_*
FROM outputstream[sumOfPattern]
INTO patternSum;

CREATE OUTPUT STREAM outputstream;
SELECT * FROM patternSum
WHERE ((LastPatternT - FirstPatternT) <= t2 -t1) INTO outputstream
;
```

The first select or query counts the arrival of n tuples as defined by the window clause. In addition, it records the time of the first arrival and the last arrival of these n tuples. The second query checks if the time for these tuples to arrive is between the time window specified. Tuples is in the logical model are the matched predicates. Comparing the two semantics we conclude that the predicates holds n times within a specified time window is equivalent to the query counting n tuples in the same specified time window and vice versa. We conclude that:

$$Eval(\mathcal{Q}_\alpha, \mathcal{M}, \tau) = Eval(SQ_{\mathcal{Q}_\alpha})$$

6.5 The Translator Development

In Figure 5.3, the system specifications (attacks or protocol) is specified in MSFOMTL. This need to be translated into the equivalent *SSQL* code that can be run on the *StreamBase* server to detect intrusions. The overall view of the process of the translation is shown on Figure 6.3. There are three main labeled process(es) as shown in the figure.

The process of the translation starts with the process labeled as (1) that is providing the input text that needs to be translated. The input text is the specifications written in a flat file using the MSFOMTL syntax. The group of processes labeled as (2) are the translator processes. The code for these processes was built using ANTLR to recognize and translate the specifications. This translator first reads the specifications as stream of characters. This is done using the lexical analyzer which breaks up the input stream into tokens. Next, the parser analyzer feeds off these tokens to recognize the formula structure. if there is (are) no error(s) the formula will be translated into *SSQL* in the final phase (the emitting phase). The output from the translation can be emitted directly by executing actions that are positionally triggered during the parse process or through the use of string templates. String templates are text documents with holes in it. These holes are filled by the emitter with incoming data values or expressions that operate on these values. The final process labeled as (3) of the translator is the *SSQL* deployment files. These files can be deployed and run on the *StreamBase* server by the administrator.

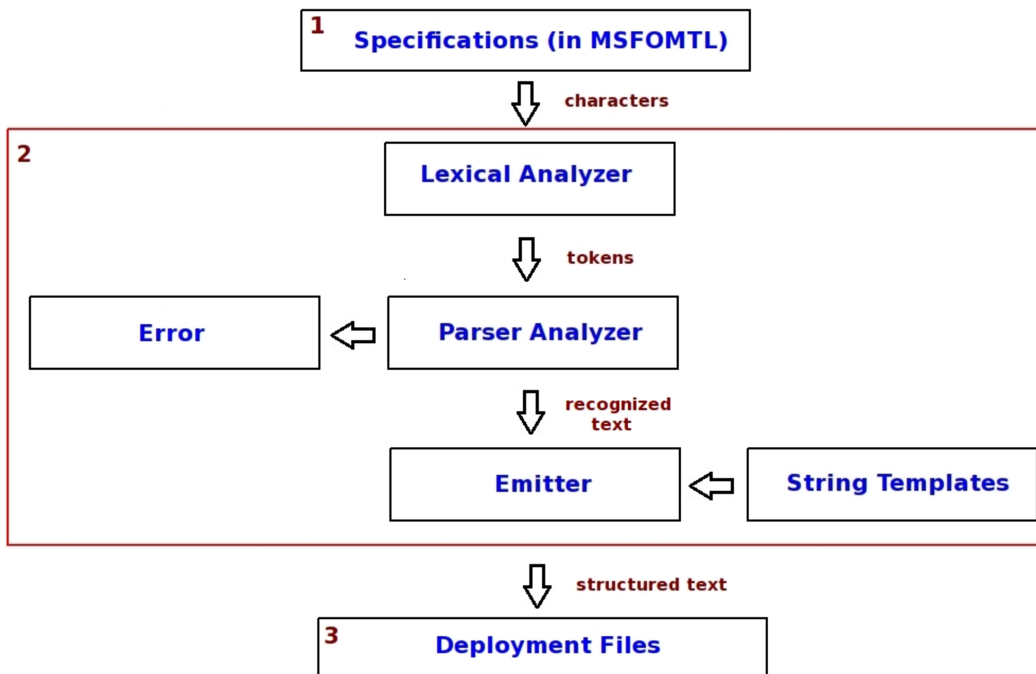


Figure 6.3: *TeStID* The Translation Process

Technically, the translation process is a mapping from the syntax/semantics of MSFOMTL into the syntax/semantics of *SSQL*. ANTLR allows us to define the desired syntax of the formulae in a grammar file. This grammar file contains the lexical and parser rules from which ANTLR will generate the lexical and parser analyzers. The translation is performed by executing embedded actions within the grammar and emit output directly or by using string templates. These actions are executed according to its positions in the grammar file. The description of the general structure of the grammar file is given in Appendix A.1. Before using ANTLR to generate the parser and translator,

it is important to know precisely how to translate each MSFOMTL formula. In Section 5.2 four syntactical forms are used for the misuse based attacks. During the translation, each of these forms has its own mapping structure and rules. More syntactical forms can be defined in the same way if needed in the future. The grammar file for the misuse is in Appendix A.2. The grammar file contains all the parser and lexical rules. It has many variables (of scalar types or list arrays) which are assigned values during the actual parsing of formulae and sent to string template for constructing structured texts. Also these variables could be used by embedded Java classes for constructing text. These Java classes are part of the grammar file and it will be included in the generated target file (i.e., the translator). The translation also is embedded or specified in the grammar file and a string templates are used to format the output to emit the output. These string templates are grouped in string group file (see Appendix A.3).

It is very difficult to follow the grammar file and it is much easier to follow the syntax diagrams provided by ANTLR. These parsing rules are shown in Figures (6.4, 6.5, 6.6, and 6.7) and is explained as follows:

- tokens or constants: there are tokens defined at the beginning of the grammar file for later use by the parse rules these are:

```
tokens {
 PLUS = '+' ;
 MINUS = '-' ;
 MULT = '*' ;
 DIV = '/' ;
 PREDSYM = 'P';
 AND = '&';
 OR = '|';
 ALWAYS = 'G';
 ALWAYSP = 'H';
 EVENTUALLY = 'F';
 NEG = '';
 EQUAL = '=';
 NOTEQUAL = '<>';
 GT = '>';
 GE = '>=';
 LT = '<';
 LE = '<=';
 EXIST = 'E';
 REOCCUR = 'R';
}
```

- prog: is the first rule that will be triggered during the parsing process. It just says that there is one or more formulae in the input file need to be parsed. Each formula correspond to one attack classification (Section 5.2).

- formulaseq: When reading the formulae from the file each formula is preceded by "# sid - number" + new line where the white space is ignored and number is digit(s) (0-9) as in Figure 6.6. The "sid" designate the system identifier for this attack. Later during the translation it will be attached to the output of the matched pattern when reporting attacks. Next, the formula should be available for parsing. Also, it is possible to allow extra new lines in the input file to make reading easier (for human).

- formula: formula can be in one of four forms: formula1, formula2, formula3, or formula4. These are the syntactical forms as presented in Sections 5.2.

- formula1: this is for the single packet attack. It has a first order predicate and conjunction of disjunction Boolean formulae of terms in the predicate.

- formula2: this is the forward multiple packet attack.

- formula3: this the backward multiple packet attack.

- formula4: this is the repetition attacks.

- atomic formula: is first order predicate.

- terms: can be either var,constant, or functions. One function is defined which is for searching for text in the packet payload. It is a term and it is represented as function call with regular expression syntax as argument. The regular expression syntax is defined in the regex rule in Figure 6.6. The regex rule specifies all the characters that are possible to use in the syntax. Also, a match rule can be part of the regex rule and it corresponds to the operator of regular expression "match at the beginning of the line". The regex syntax here is based on the *StreamBase* regular expression function syntax.

- other rules are simple and used by the above rules, like exq for the existential quantifier, number (one or more digits), digit for a single digit from 0-9, bigchar for capital letter, smallchar for small letters, newline for the new line or carriage return, whitespace for space, tab and form feed, and unicode for defining unicode (used in the regexp rule).

We can give an overall view of how the actual translator work using the grammar file in A.2 and the templates group file in Appendix A.3. Also, Figure 6.8 is used to explain how the string templates group file is instructed or called to emit the output (the translation) from within the grammar file.

When a formula is read from the input file, the parser starts the processing by the start rule which is prog. Prog is considered the start rule in Antlr terminology and it just points to the formulaSeq rule. The formulaSeq rule reads the formula and then the formula rule is invoked. If the formula is of the first syntactical form then rule formula1 is invoked, if it is of the second form, then the second rule formula2 is invoked, and so

FIGURE 6.4: Parser Rules (1 of 4) For The Misuse Based Detection of *TeStID*.

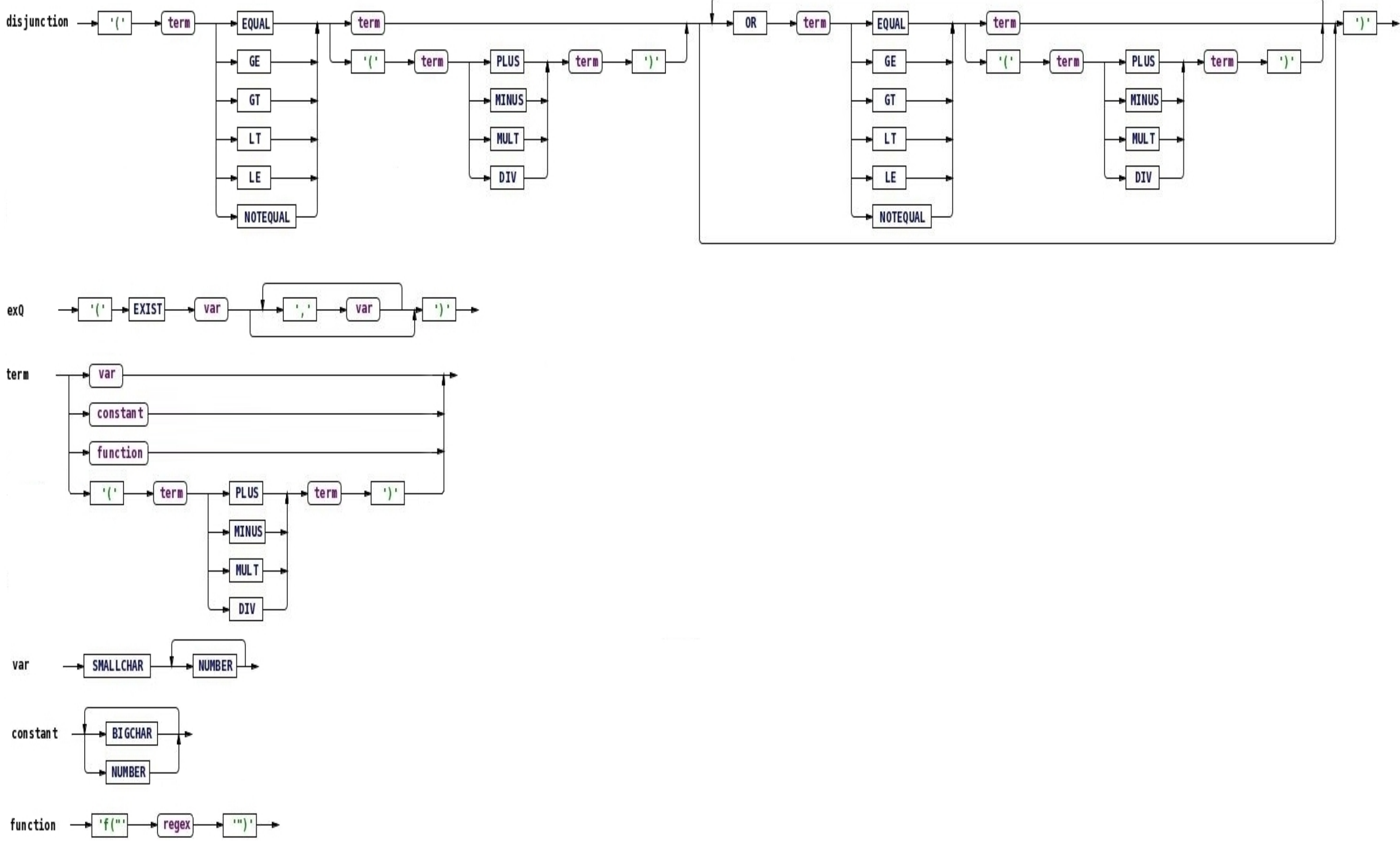


FIGURE 6.5: Parser Rules (2 of 4) For The Misuse Based Detection of *TeStID*.

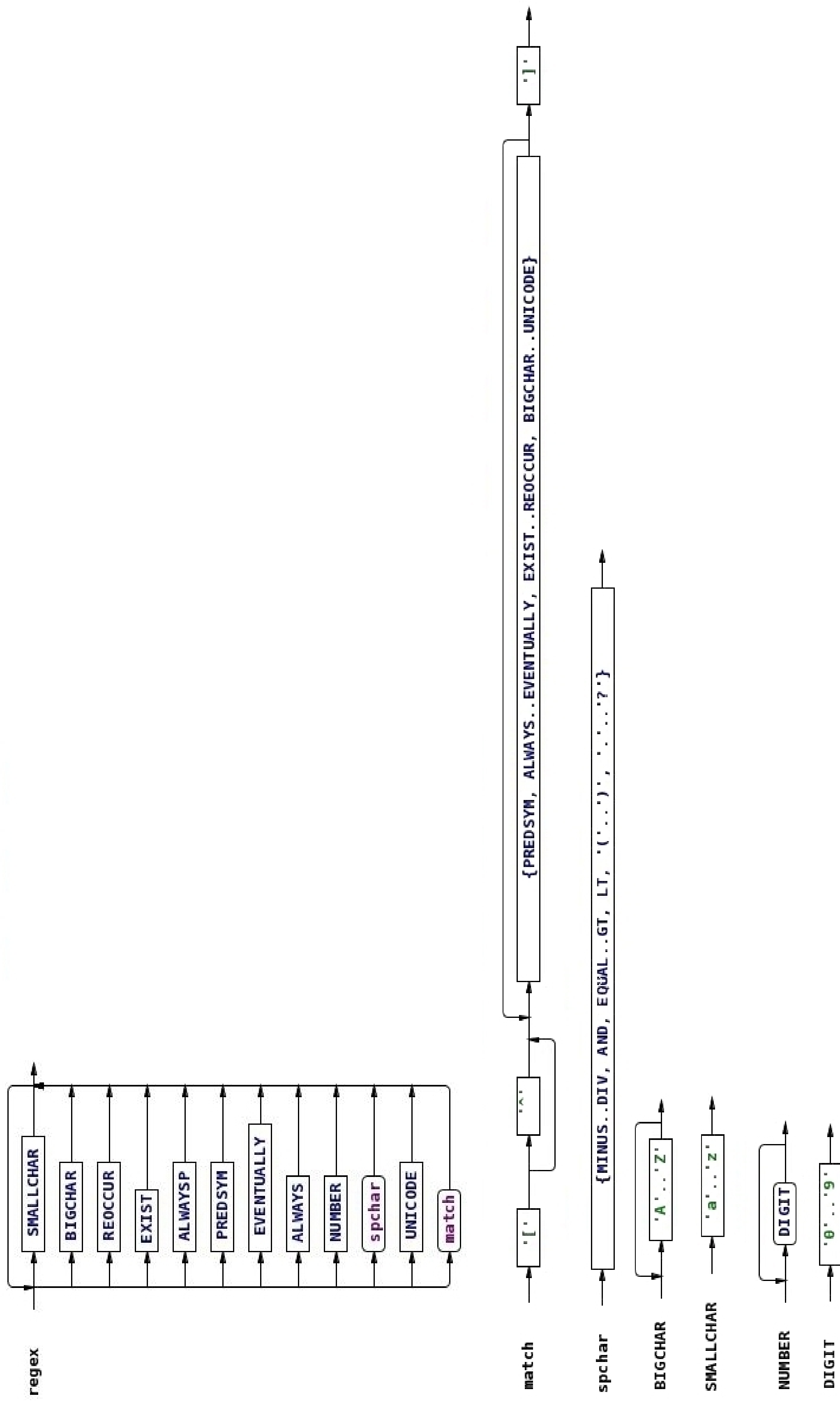


Figure 6.6: Parser Rules (3 of 4) For The Misuse Based Detection of *TeStID*.

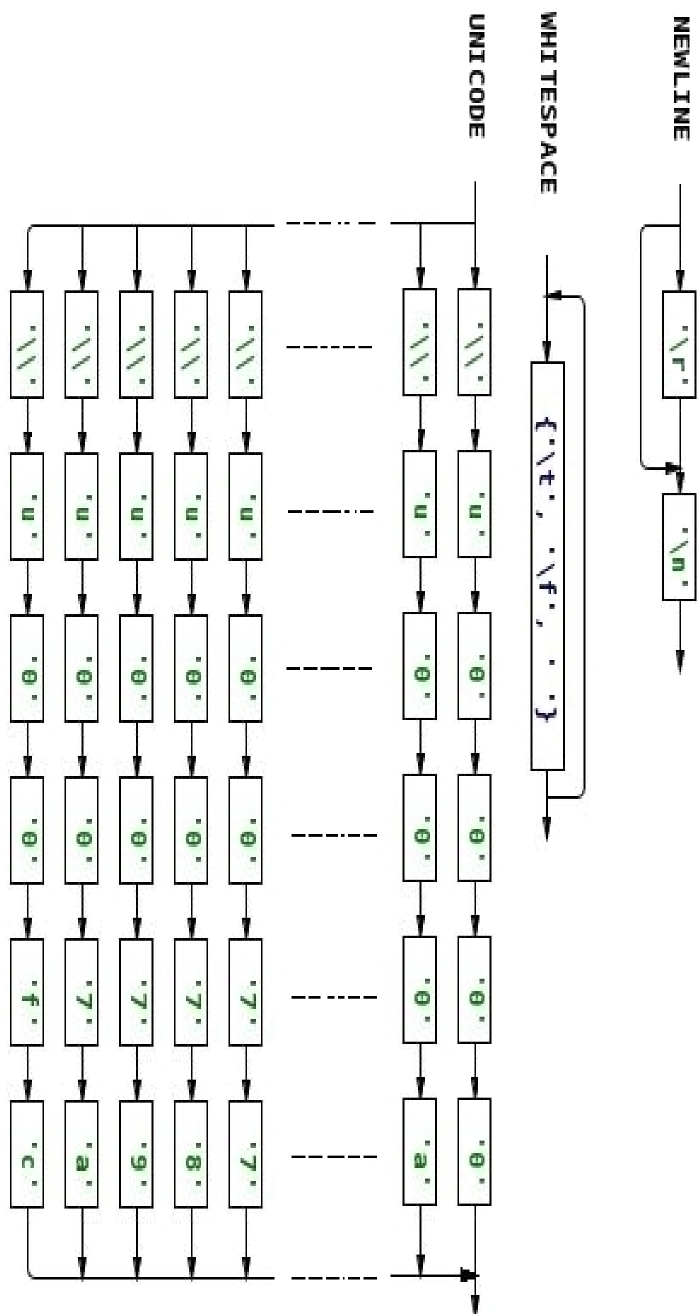


FIGURE 6.7: Parser Rules (4 of 4) For The Misuse Based Detection of *TeStID*.

forth for the third and the fourth syntactical forms. From within formula1, formula2, formula3, or formula4 all the other parsing rules that deal with each formula syntax are invoked and are not shown in Figure 6.8.

During the parsing of rules, there are some actions or string template defined. An action is a code written in the target language and is included inside curly brackets. The target language is the language that will be used to write the parser and analyzer by Antlr. The target language is defined in the grammar file (A.2) in the options section (language=Java;). String template is defined by a name and argument(s) as: $template_name(arg_1 = value_1, \ldots, arg_n = value_n)$. An argument can be text from the input, calculated value with a Java call to methods written as members inside the grammar files, or a variable defined in a rule and manipulated with action (e.g., formula counter, Boolean variable, etc.).

In Figure 6.8, When formulaSeq is parsed, the operator "`->`" directs the translator to use the string template prog. Prog has two arguments formulae and isFormula1. Prog plays an important rule as it is the entry to the string template group. In the template group file Appendix (A.3), it is the first string template defined after the group name (SB):

```
group SB;
prog(formulae,isFormula1) ::= <<
<if (isFormula1)>
.
.
<formulae; separator="\n">
>>
```

So, this template is called with two arguments: isFormula1, which is set to true in formula1 rule as can be seen from the grammar file and if it is true then some deployment files will be created for this category of formulae (will be explained later in this section). The other argument (formulae) is Java list array variable defined at the prog rule level. When the formula rule is parsed an action is defined to add the computed string template of formula rule to the formulae list array (prog::formulae.add(formula.st);) (Figure 6.8). In Antlr, each rule has an associated string template. So, the formula rule has four possible values: formula1.st, formula2.st, formula3.st, and formula4.st. In the prog template "`<formulae; separator="\n">`" is a template string language expression. It means process each value of the attribute (formulae) where these values are separated by newline. So, for instance, if the value is formula1.st, then when formula1 rule is processed, it will invoke the the string template category1. As can be seen in the grammar and template files category1 has many arguments that are used to write the translation of the first syntactical form. The details of expressions and statements used in the template can be found at [108].

Producing Deployment files: As shown in Figure 5.3 the translation process produces ready for deployment files. It does not just translate the MSFOMTL into the *SSQL* code only. It creates all the necessary files needed ultimately to run on the server.

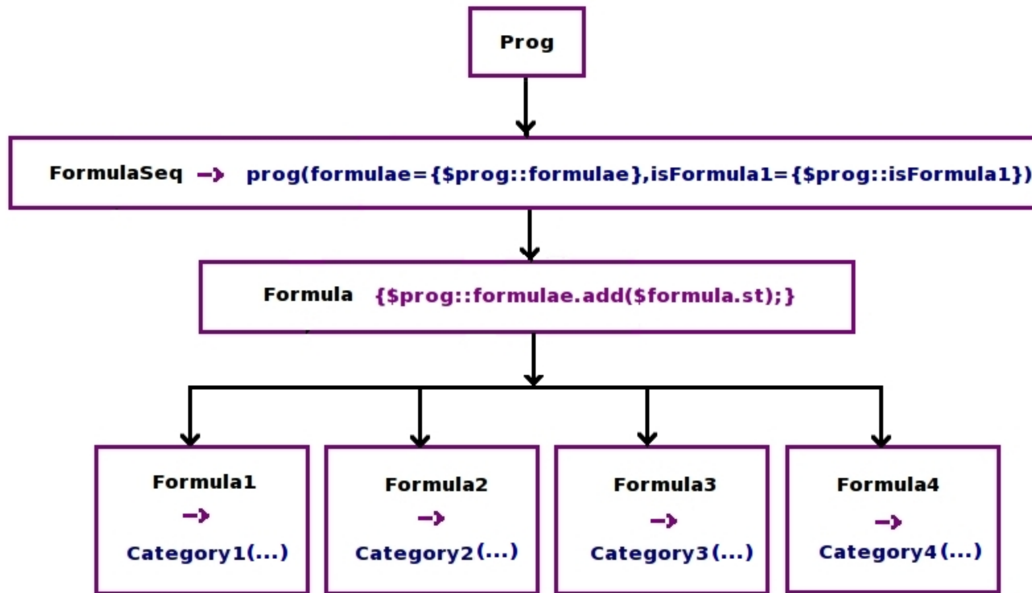


FIGURE 6.8: Calling The String Template Group File During The Parsing

This means nothing needs to be done after the translation other than copying these files in the *StreamBase* server and run them. The number of deployment files depends on the number of attacks specified in the input file. Also, this number depends on the syntactical form of the attack. In multiple packet attacks, the *SSQL* code for each attack is in one file. For single packet attacks it is much more complicated as to deploy this type with the ability to use the parallelism features (Section 4.3.1 and 7.2.3), we have to write the attacks as referenced modules and in *StreamBase* referenced modules must be in a file on its own (explained in Section 4.3.1). Also, there will be data preprocessing or filtering (Figure 5.3). The filtering is done based on the source port or destination port if specified in the attack (these are commonly used TCP services). This means three files are generated for filtering: one based on the source ports, one based on the destination ports, and one for all other streams. So, for single step formulae the deployment files are:

- main.ssql which is the main file that calls all the other module files and links them.
- allp.ssql, allsrc.ssql, and alldst.ssql are the filtered stream files. These are generated once for the input file if it contains at least one single step attack formula. They read all the incoming traffic and output filtered traffic.
- m<number>.ssql where number is the number of formula in the input file. Each attack module will be written with a unique name as a referenced module. The referenced modules obtain their input streams from one of the three filtered streams. The translator will assign the filtered stream to the referenced module automatically based on whether common source or destination port is specified.

The emitting of texts into different files is not possible in the current version of ANTLR. To overcome this, the output is generated (using the template) with embedded Unix shell commands. Simply, in this script we echo each piece of the translation into the designated output file. When executing this file all the deployment files will be generated.

When the string template named (prog) is called and the Boolean variable (isFormula1) is set to true, then three filtering files (allp.ssql, src.ssql, and dst.ssql) will be be generated that read from the Java adapter which captures the network traffic. These will be used as input to the first syntactical formula as the requirement of the formula in terms of input preprocessing filtering needs. The code for this starts from (<if (isFormula1)>) and ends at (¡endif¿) in the string template group file (Appendix A.3).

In this section, we give the overall view of how Antlr process the grammar and how it is linked to the string templates group file to emit the output. The lexer and parser files that are generated from the grammar by Antlr are about 5300 lines of Java code.

6.6 Summary

This chapter presented the concept and the work related to the process of using temporal logic to query temporal and stream databases. Many research exist and the temporal logic seems like the monotonic choice. The view of time, the mapping, and the correctness of the approach were presented and explained. Finally, an overview of the development process of building the translator was given.

Using the translator produced, we conducted many experiments for single packet, single packet with payload, and multiple packets attacks. These attacks were written using MSFOMTL and translated with the translator built as described in this chapter. The next chapter has the experiments and results details.

Chapter 7

Experiments and Results

7.1 Experiments Overview

The previous chapters proposed the new methodology and the new system *TeStID* for intrusion detections. This method is based on specifying patterns of attack (or normal behaviour) with temporal logic which will be translated it into *SSQL* code. This code can be executed to identify the existence of an attack in the case of misuse based intrusion detection or the deviation from normal behaviour in the case of anomaly based intrusion detection.

This chapter presents the experimental work done to demonstrate the capabilities of the system to detect misuse based attacks. Section 7.1.1 states the experiment aims. Section 7.1.2 describes the approach and setup used during the experimental work.

Section 7.2 deals with single packet attacks with payload. We run the experiments on *TeStID* and two well known open source intrusion detection systems, *SNORT*[87] and *BRO*[50], and the results were compared. Section 7.2.3 provides discussion on the scalability and performance aspects of *TeStID*. Also, the experimental work and results of using the parallel features of *StreamBase* is given. In Section 7.3, we provide case studies on multiple packet misuse attacks and illustrate the results of these experiments. A summary of the work included in this Section was published in [3]. Finally, a summary is given for this chapter in Section 7.4.

7.1.1 Experiments Aims

This research project introduced a new system for network intrusion detection in a high volume network environment. The experiments were conducted to address the following issues:

- Coverage: One of the main performance requirement of any *NIDS* is the ability of the system to detect all the attacks that exist in the traffic. This coverage rate of the tested systems could be measured quantitatively against specified attacks in a data set.

- Efficiency: As the traffic volume increases to what extent is the system able to detect the attacks successfully?
- Performance: The performance of the systems in quantitative and qualitative terms. Quantitatively, the use of system resources and maximum bandwidth achieved are the criteria used to compare the tested systems. Qualitatively, the scalability and performance of the systems are discussed for the tested systems.

Misuse based intrusion detection systems are well known for their superior capabilities to detect known attacks with very low false alarms. This has been proven by the analysis of the systems that participated in the DARPA *IDS* evaluation in 1999 [52]. Moreover, the rates of false positives (false alarms) in the misuse based attacks are usually very low compared to anomaly based NIDS [52, 82]. In our experiments, we are interested in the false negatives rate (the inability of the system to detect real security events) in high volume traffic. It was mentioned in the introduction that this rate drops as the traffic volume increases (see Section 1.1).

7.1.2 The Experiment Setup and Approach

An obvious choice for testing our system is on a real high volume network. However, whilst this is possible such an unrestricted environment does not allow us to control the data (e.g., the load) being sent or to know precisely how many attacks are present in the traffic. For testing NIDS you need to have data that includes some attacks. These specified attacks must be known to measure the detection rate of the tested system. Moreover, if we need to do the test repeatedly then working with trace files is much better choice. Other reasons for not using a real environment are the security of the environment and the privacy of the users.

The setup of our environment must closely resemble the actual deployment of NIDS. In all the deployment options mentioned in Section 2.3, the network sensor device receives a copy of all the traffic that traverse the network. In fact, the way it gets a copy of the data is irrelevant. Thus, the testing environment is setup as follows:

- The *TeStID* is installed on an INTEL® Core™ i5 2.26 GHz machine with 4 GB of memory and a Gigabit Network interface that is capable of running in promiscuous mode (i.e., listening to all network traffics).
- Another computer (INTEL® Core™2 Quad Processor Q6600 and 2 GB memory) with a Gigabit Network interface is used to replay the data.
- TCPREPLAY was used to replay the trace files. TCPREPLAY [102] is a tool that replays libpcap format files at specified speeds onto the network. Libpcap is a portable C/C++ library for low-level network monitoring used by TCPDUMP and many other network capturing tools. TCPDUMP/Libpcap are open source software originally developed at Lawrence Berkeley Laboratory by Jacobson et al. [42] and it is now maintained by the the TCPDUMP organization or group [98].

- A switch to connect the two PCs or simply crossover network cable (see Figure 7.1).
- *WIRESHARK* the open source network protocol analyzer tool kit was used to analyze the results and produce graphs [71].

In Section 7.3 the experiments are concerned with multiple packet misuse attacks against TCP/IP. We used the US Defence Advanced Research Projects Agency (DARPA) publicly available IDS evaluation data sets [19, 59]. For each multiple packet attack we implemented we used the corresponding DARPA data test files that contain the attack.

In Section 7.2 the experiments are concerned with the single packet attacks with payload against the TCP/IP. The DARPA data set does not have an adequate number of attacks to test these kind of misuse attacks and as far as we know there is no publicly available data set we knew about exist. So, we customized the data which we prepared by using a DARPA dump test file and injected some known attacks from the free license version of Traffic IQ Professional™[39].

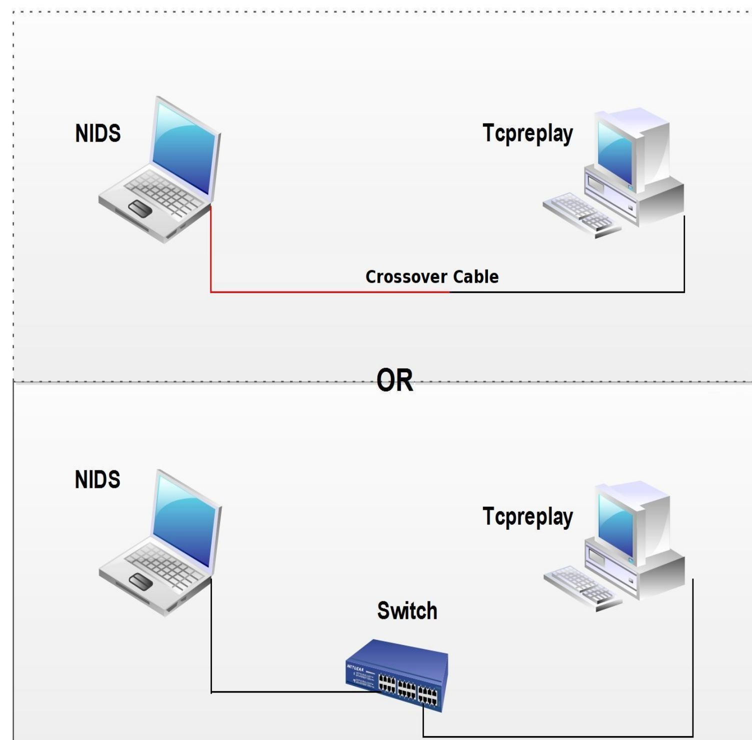


Figure 7.1: The Testing Environment

7.2 Single Packet Attacks With Payload Experiment

This section is concerned with single packet attacks. Single packet attacks are attacks that are launched against a victim machine by sending a single packet. In Section 5.2 we explained this category of attacks and how the intruder can by taking an advantage

of a vulnerability in an operating system, an application, or a service in the targeted machine launch a misuse attack. The main aim is to show *TeStID*'s capabilities in deep packet inspection and pattern matching and comparing the results with *SNORT* and *BRO*.

Previously, in Section 7.1.2 we explained how the test data is prepared for the Single packet attack experiments. Section 7.2.1 presents explanations of the signatures used and how it is obtained. In Section 7.2.2, the results and analysis of experimenting with *TeStID* are presented. Additionally, we run the same test using *SNORT* and *BRO* on the same testing environment for the purpose of comparison and evaluation. The results achieved with *TeStID* using different scalability and performance features, these are presented and discussed in Section 7.2.3. Finally, a summary is given in Section 7.4.

7.2.1 The Experiment Signatures Preparation

The single packet attacks can be classified into two types. In the first type, only the packet headers are used (i.e., IP and TCP headers) in the attack. In the second type, the payload or the data field of the packet is also used. Intuitively, the second type needs more processing time per packet. This processing time varies depending on how far from the beginning of the payload field the NIDS needs to match a specific string, word, or pattern of characters (the payload size in each TCP/IP packet is between 46 and 1500 bytes). For instance, if a packet of payload size 1500 needs to be processed and if the signature of the attack specifies that the string "CD root" must be matched at the beginning of the payload , then it will be processed faster than if the signature specifies that it needs to be checked at the end. Of course, more processing time will be needed for matching if the signature specifies that "CD root" should be matched anywhere in the payload.

To achieve the best processing speed possible when handling text patterns matching, *SNORT* and *BRO* use regular expressions (regexp). *SNORT* uses Perl Compatible Regular Expression (PCRE[1]) library and *BRO* follows FLEX's[2] regular expression syntax [50]. *TeStID* uses Java regular expression available from the *StreamBase* Language standard functions library [93]. Using these variants of regular expression syntax to represent an attack results in different regular expression search strings. This needs to be coded carefully for testing the attacks with payload for each variant. A single missing/extra wild card character results in missing an attack.

For this experiment, a selected set of attacks from *SNORT* official signatures are used which were developed and tested by the Sourcefire Vulnerability Research Team® (VRT) [88]. This set consists of 50 attacks specified by *SNORT* syntax signatures (see Appendix B.1). Using *BRO* signature language, we can rewrite the same *SNORT*

[1]PCRE is an open source library written in C. The library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5. More information at http://www.pcre.org.

[2]Fast LExical analyzer is free software for scanning and recognizing lexical patterns in text. More information at http://flex.sourceforge.net

signatures. *BRO* has a Python script (snort2bro) that converts *SNORT*'s signatures into *BRO* signatures. We make use of this to convert from *SNORT* to *BRO*. It works well most of the time but manual intervention is sometimes needed [11]. Appendix B.2 presents the equivalent *BRO* signature file used in the experiment. For *TeStID* these selected *SNORT* signatures are written using the proposed MSFOMTL syntax and semantic in Section 5.1. Each attack is represented by a MSFOMTL formula; the file that contains all the selected attacks is presented in Appendix B.3.

To illustrate the differences between these three systems in terms of writing or specifying a signature, let us take an example which is *SNORT* signature id 255 (sid-255). In *SNORT* signature syntax it is specified as follows:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"sid-255 DNS zone transfer
TCP"; flow:to_server,established; content:"|00 00 FC|"; offset:15;
metadata:policy security-ips drop; reference:arachnids,212; reference:cve,
1999-0532; reference:nessus,10595; classtype:attempted-recon; sid:255;
rev:17;)
```

Explanations for the keywords used in this signature are:

- "alert tcp" means that alert should be generated and it is a signature for TCP.
- "$EXTERNAL_NET any" refers to the source IP address which is defined in the system globally by the variable $EXTERNAL_NET and the source port (any port).
- "$HOME_NET 53" refers to the destination IP address which is defined globally in the system by the variable $HOME_NET and the port 53.
- "msg:"sid-255 DNS zone transfer TCP"" is the message written into the log file when an attack occurs.
- the "flow" refers to the direction of the flow.
- "content" is the content in hexadecimal that needs to be checked in the payload. In *SNORT* you can specify the contents using hexadecimal or strings or combination of both.
- "offset" means how many bytes to skip from the beginning of the payload.
- "sid" is the signature id as identified in *SNORT*.
- the rest keywords are information keywords. A full description of all the available keywords are available at [17].

The equivalent for this *SNORT* signature using *BRO* signature language is:

```
signature sid-255 {
  ip-proto == tcp
  src-ip != local_nets
```

```
    dst-ip == local_nets
    dst-port == 53
    event "DNS zone transfer TCP"
    tcp-state established,originator
    payload /.{14}.*\x00\x00\xFC/
  }
```

Explanations for the keywords used in this signature are:

- “signature” refers to the signature id for identifying and logging.
- “ip-proto” refers to the protocol type (TCP, UDP, etc.)
- “src-ip” refers to the source IP address. This address could be specific or globally defined in a variable (local_net).
- “dst-ip” refers to the destination IP address.
- the “dst-port” refers to the destination port.
- “event” is part of the logged message when this attack occur.
- “tcp-state” information about the tcp-state at the time of receiving this packet.
- “payload” search string in the payload.

Notice here that not specifying the source port means any port. Also, the content that need to be search in the payload is typed directly with regexp syntax.

In MSFOMTL, formally the representation of the attack is:

$$\begin{gathered}(\exists x_4, x_{12})((\exists y_1, y_2, y_3, y_5, y_6, y_7, y_8, y_{10}, y_{11}) \\ P(y_1, y_2, y_3, x_4, y_5, y_6, y_7, y_8, 1, y_{10}, y_{11}, x_{12}) \\ \wedge(x_4 = 53) \wedge (x_{12} = f(\texttt{"}.\{14\}.*\backslash\texttt{u0000}\backslash\texttt{u0000}\backslash\texttt{u00fc}.*\texttt{"})))\end{gathered}$$

In the above formula, the regular expression is expressed as Java regexp syntax. x_{12} is the payload and x_4 is the destination port. All other arguments are exactly as explained in Section 5.1.1.

Beside the signature preparation overhead for these three systems, there is the overhead of the interpretation of the logged events. *BRO* reports by session and this means it would report less than *SNORT*. This means the attacks for the same session is reported per session not per each packet that trigger the attack in the session. *TeStID* reports each packet that carries an attack. Fortunately, *BRO* and *SNORT* have the capability to run in offline mode (reading the dump file directly) and this allows us to establish how all the existing attacks in the trace are reported by each system. For *TeStID* this was done manually by validating the results of running against the test file with the result obtained by running *SNORT* and *BRO* in the offline mode. In addition, *WIRESHARK* helped analyzing and verifying the packets inside the trace file.

7.2.2 Results and Analysis

The three systems were tested independently using a custom data file. The data files have in total 3,014,600 packets. The 50 different testing attacks are distributed over 792 packets. This means 792 total instances of the 50 attacks. The time it takes to replay the file at normal speed (recorded speed) is about 13 hours.

Tested IDS	x 2	x 4	x 8	x 16	x 24	x48
BRO v1.5.1	0	0	-1	-28	-78	-85
SNORT v2.9.1	-119	-125	-134			
TeStId v1.0	0	0	0	0	0	-1

Table 7.1: Single Packet Attacks Final Results

The final results of testing these three systems are given in Table 7.1. The first column shows the tested system name. Columns two through seven contain the results of running each system while using multiple replay speeds of 2, 4, 8, 16, 24, and 48 respectively, to send the packets (blank if no test is done).

The packets size and not the packets number affects the amount of system processing and resources. Here, each packet header and the payload is checked per signature. To minimize the number of packets that need to be checked per signature, both *SNORT* and *BRO* (*TeStID* also) have a prefiltering for the traffic or partitioning the traffic based on the tcp port or service. Even though this prefiltering helps to minimize the processing number of packets per signature, the intensity of the traffic to a certain port or service results in an increase in the demand of the system processing power and the increase of the rate of missing attacks due to packets dropping. According to Cabrera et al. [12] the time spent in processing a signature accounts for 75% of the processing time with mean payload checking time 4.5 times larger than mean header checking time. Figures 7.2 and 7.5 show the intensity of the test data in terms of bits/sec during the experiment replay speed of 24 and 48, respectively. The intensity in terms of packets can be seen in Figures 7.3 and 7.4, respectively.

We can see from Table 7.1 that *BRO* performed well and did not miss any attacks until the replay speed increased to multiple of 8. The missing number of attacks increased from 1 at speed of 8 until it reaches 85 at speed of 48. From Figure 7.5 we can see that there were many spikes at 30 Mbps or close to 30 Mbps at replay speed of 48. At speed 8 most of the spikes are close to 8 Mbps (not possible to show the graph).

SNORT on the other hand missed 119 attacks when the replay speed was 2 and 125 when the replay speed was 4; this is why further replay speed tests were not carried out for *SNORT*. At speed 2 the volume was hitting 4.5 Mbps. This was surprising at first but looking at the work in [21] in which *SNORT* was missing attacks when hitting 3.1 Mbps with dropped packets rate of 2%, our results for *SNORT* sound reasonable. Day and Burns [21] concluded in their experiments that *SNORT* does not utilize multi-cores machine.

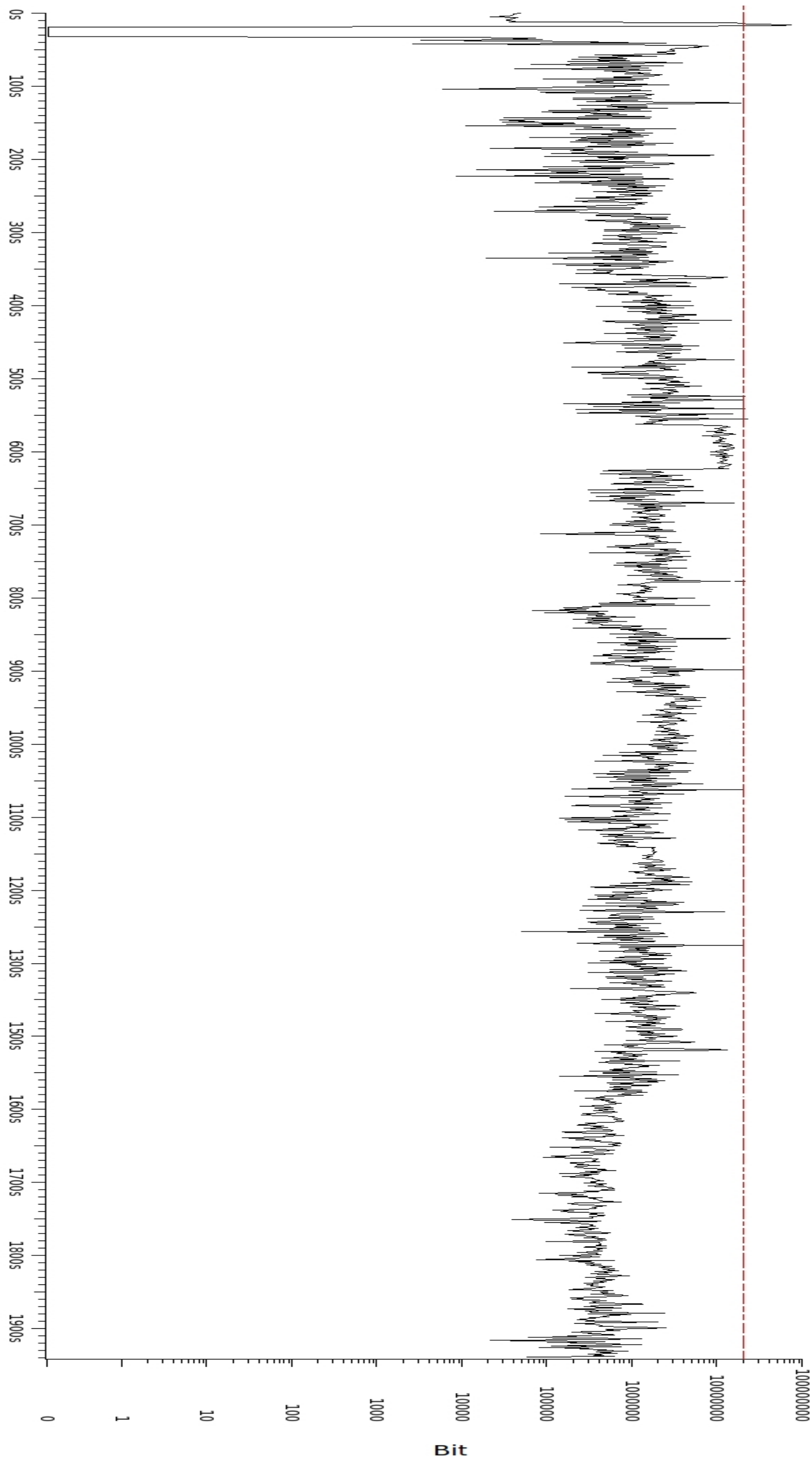


Figure 7.2: bits/sec at 24 X

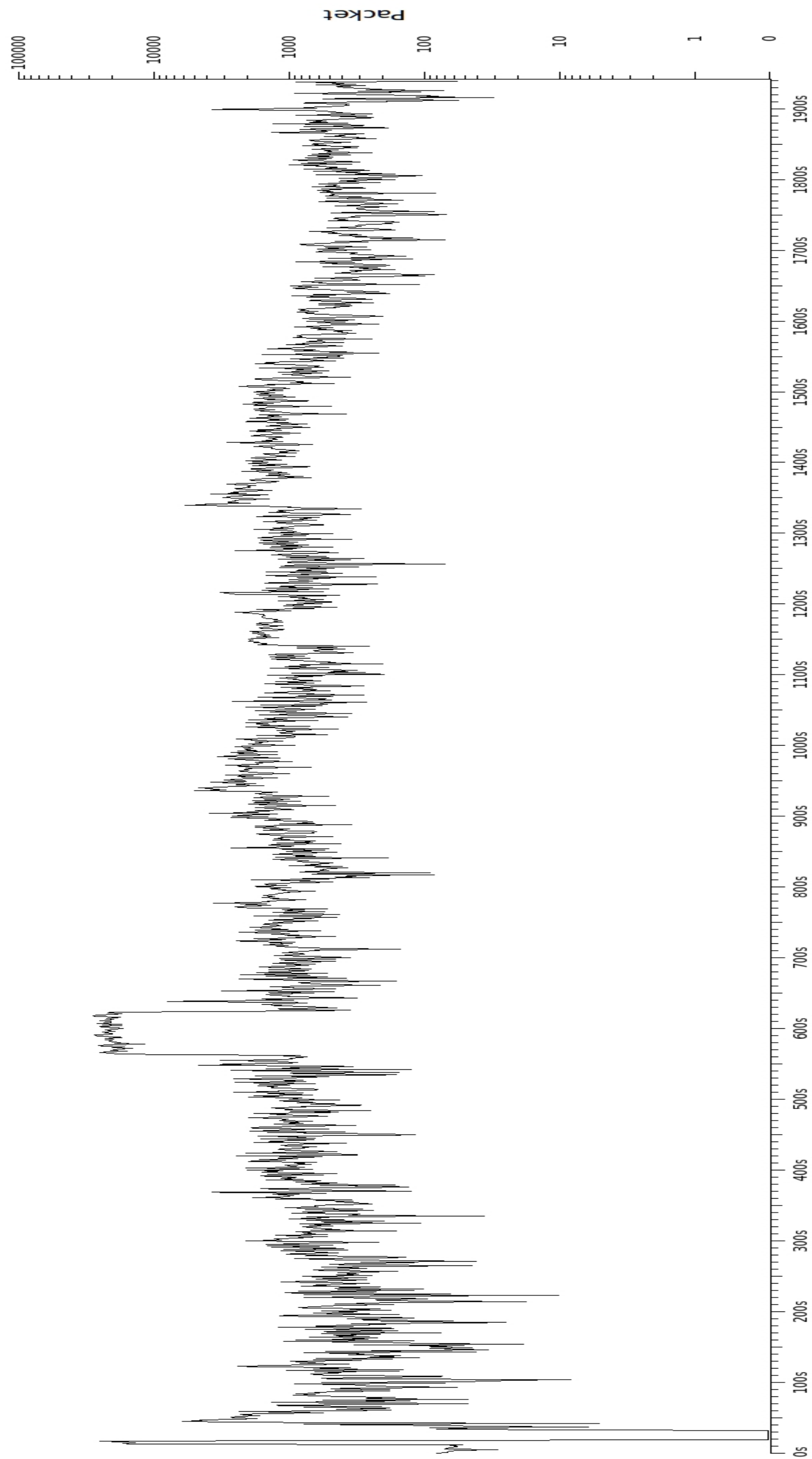


FIGURE 7.3: packets/sec at 24 X

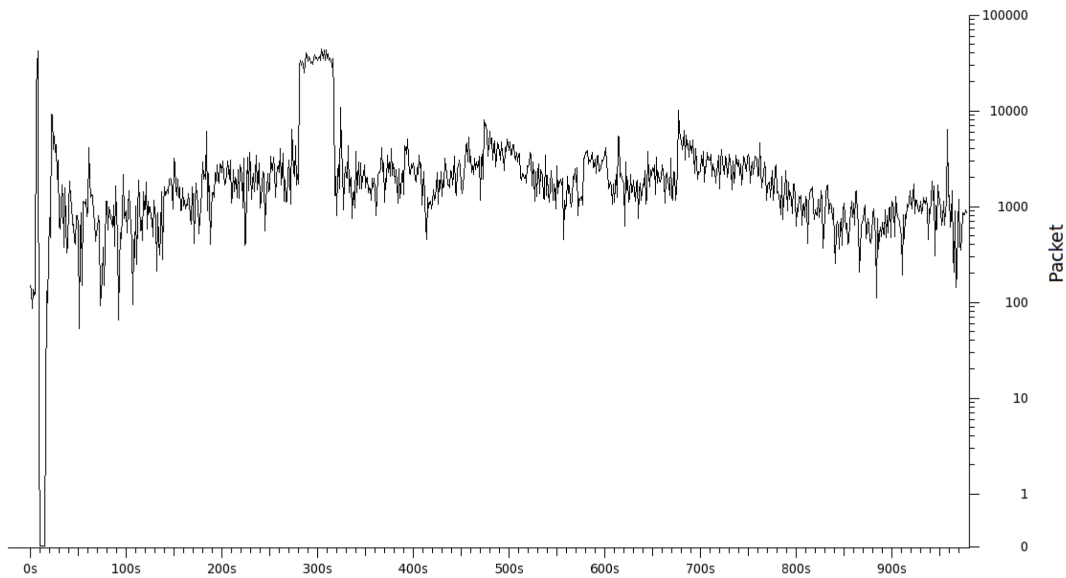


FIGURE 7.4: packets/sec at 48 x

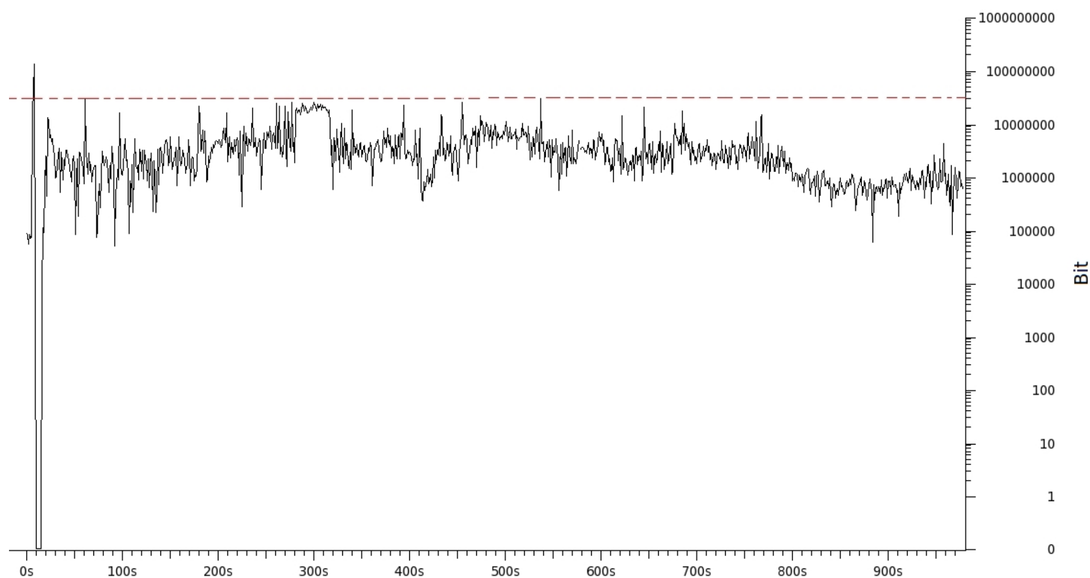


FIGURE 7.5: bits/sec at 48 x

TeStID did very well up to the multiple speed of 48. Actually, many implementations using the fine-grained parallelism features of *StreamBase* (see Section 4.3.1) were tested. In the next section, a detailed view of these implementations and the results obtained are provided.

7.2.3 Scalability and Performance

The volume of traffic in high speed links increases due to many reasons. As organizations move to higher speed links, the nature of use, the increase in the number of users, and the increase in the dependencies on network applications may contribute to the increase of the network traffic as time goes. Obviously, having an intrusion detection system that

is scalable, is what the system administrators are looking for. By scalable we mean the system capabilities to handle an increased volume of traffic without a significant decline in performance. Here, we consider the different features *StreamBase* provides to deal with parallelism.

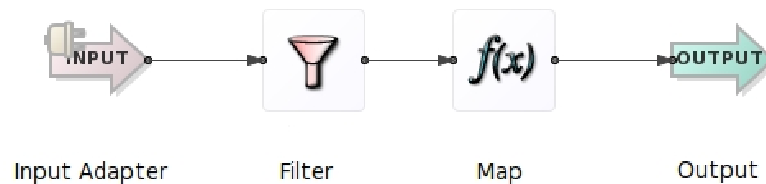


FIGURE 7.6: Graphical representation of the program to capture one single packet attack in the *StreamBase* studio

In Section 4.3.1 we presented the *StreamBase* scalability and high performance features. *StreamBase* has a very fine-grained parallel control mechanism for application scalability. This fine-grained control can be specified at the *SSQL* command level. In fact, the final results obtained in Table 7.1 for *TeStID* obtained by using these features. Different parallel control mechanisms were implemented to obtain an optimal performance. To explain these implementations, we start by showing in Figure 7.6 the operators needed to capture one single packet attack. It is a simple application with only four operators, these are:

- Input adapter: This is an input adapter written in JAVA which captures the packets from the network interface.
- Filter: This filter operator is used to match the required signature.
- Map: This operator maps or appends the attack identifier to the already matched packet by the filter operator.
- Output: The output stream is dispatched to the standard output device.

The mapping of these operators into the equivalent *SSQL* code can be seen in Figure 7.7. The reason for showing the graphical representation is that it is much easier to explain the implementations using the diagram later in the text.

According to the *StreamBase* manual, the default when processing data in an application is that all operations are executed in a predictable order and input tuples are each processed individually to completion (i.e., no parallel execution). If portions of the *StreamBase* application can run without dependencies on the other streaming data in the application, the overall throughput of the application can be improved by specifying that some portions of the application run in their own processing threads. Running multiple threads in parallel, can result in faster performance on multiprocessor machines (for more detail see Section 4.3.1).

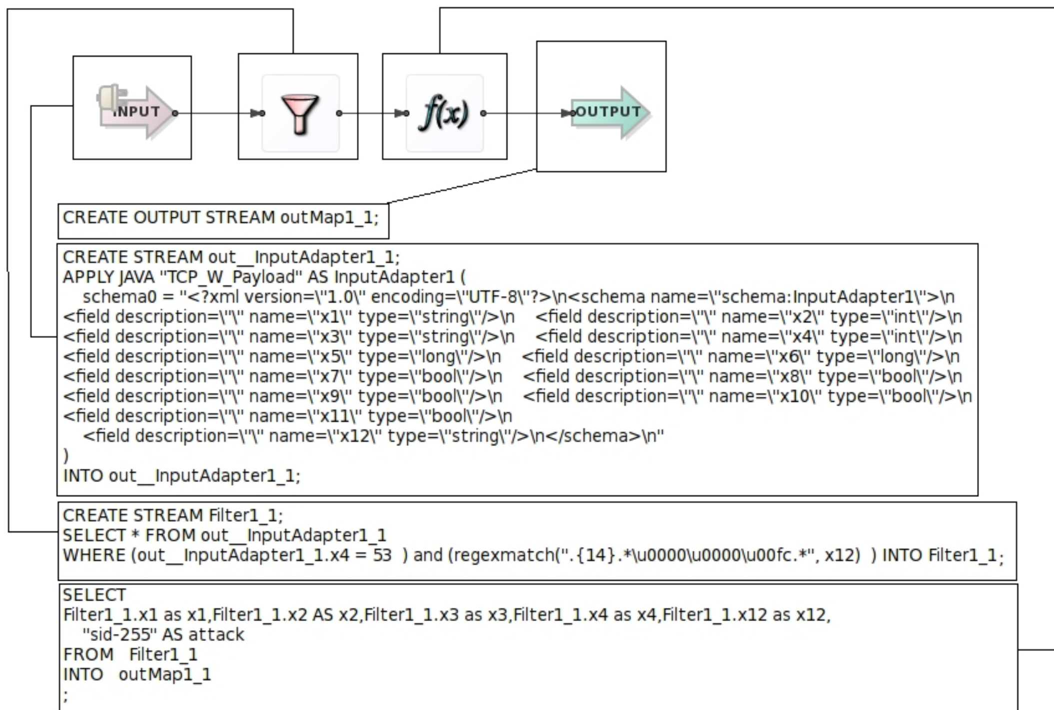


FIGURE 7.7: Equivalent *SSQL* code for the *StreamBase* application

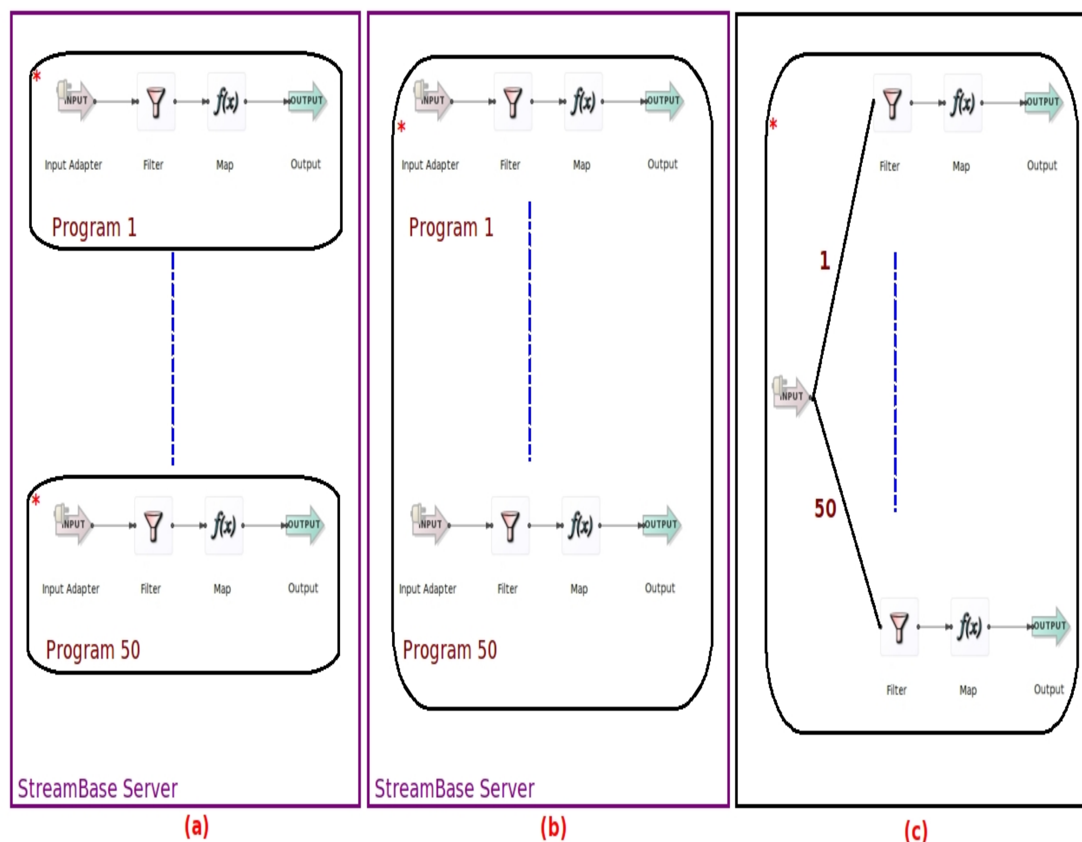
Two categories of experiments were conducted. The first category is without using the concurrency and multiplicity options. Setting the concurrency option causes the code to run in its own thread. Using the multiplicity option we specify how many instances of the same code will be running. The second category of experiments use these options. In all these experiments, the customized data (Section 7.1.2) was used.

7.2.3.1 Implementations of Single Packet Attacks With No Concurrency and Multiplicity

This set of experiments were conducted without using the parallel features of *StreamBase*. Figure 7.8 shows three possible ways of structuring the system to run on the *StreamBase* server without using the parallelism features. Table 7.2 shows the results of running these three different ways of structuring the system. The first column of the table contains description of the implementation used. The second column contains the result obtained while replaying the customized data file at multiple speed of 2. The third and fourth columns contain the results obtained while replaying the customized data file at multiple speed of 4 and 8, respectively.

In the first row, 50 different program files run each with the *SSQL* code in Figure 7.7. The difference between these files were in the filter and mapping codes as they are coded to detect different attacks. This means each attack has its filter and mapping codes. The filter contains the stream *SQL* code for one of the 50 attacks and the mapping code appends the correct attack identifier to the matched signature. This implementation

gives the worst coverage rate as it misses six attacks in the multiple speed of two. The result makes sense as each program needs to capture the packets with the input adapter code which in turns calls a JAVA code that interfaces with the network interface and then the rest of the program processes the packets (tuples) in sequential fashion. This scenario is the same for all the 50 programs which means each program will have its own space (parallel region or container in *StreamBase* terminology). A container is the basic execution unit running on the *StreamBase* server. The *StreamBase* engine suffers from the overhead of having to interact with all these regions (see Figure 7.8 (a)).



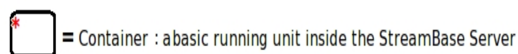


FIGURE 7.8: Three possible implementations to run single packet attack detection on the *StreamBase* server without using the parallelism features. In Figure 7.8(a), each attack is written as an independent program that runs in its own container. Figure 7.8(b) shows how each attack is an independent program but all programs are using the same container. The last Figure 7.8(c), all the attacks are written as one program and run in one container sharing the input adapter.

In the second row, all the 50 attack codes are encapsulated in one program file (i.e., one container) and run (Figure 7.8(b)). This implementation gives slightly better results as it misses 116 attacks in the multiple speed of four. With this implementation the *StreamBase* engine done more efficient processing with the decrease of inter process

handling overhead. Here, still there are 50 JAVA calls to feed the filter and map of each attack.

In the last row, all the attacks codes are in one file but only one input adapter code is used (Figure 7.8(c)). All the filtering codes are reading from the same input adapter. This means less resources from the system are used. In *StreamBase* when a tuple (packet) arrives it must be processed till completion before the next tuple arrives. This means the input adapter feeds the tuple to the first coded filter and then coded map of the first attack. If another tuple (packet) arrived, it will be retained in a buffer until the first processing is finished (more information about execution order in Section 4.3.1). This type of implementation was the best without the use of concurrency and multiplicity options as it misses 6 attacks at the multiple speed of four.

Implementation	X 2	X 4	X 8
50 independent program files	-6	-139	-384
50 independent programs in one file	0	-116	-323
one adapter code + 50 filter/map codes	0	-6	-74

Table 7.2: *TeStID* results without concurrency and multiplicity

The last type of implementation in Table 7.2 is the fairest experiment to compare with *SNORT* and *BRO* (both are single-threaded engines). In this type of implementation, in *TeStID*, all the attack detection codes take the input from a shared input adapter and everything run in single thread (or container). This is similar to *SNORT*, where the attack detection engine share the output of the decoder and preprocessor. In *BRO*, the output of the packet filtering and capturing are used by all the attack scripts file. Table 7.3 contains the results of running *SNORT*, *BRO*, and *TeStID* without parallel operations. *BRO* misses only one attack in the multiple speed of eight, whereas *TeStID* misses 6 at the speed of 4.

Tested IDS	X 2	X 4	X 8
BRO v1.5.1	0	0	-6
SNORT v2.9.1	-119	-125	-134
one adapter code + 50 filter/map codes	0	-6	-74

Table 7.3: Running *SNORT*, *BRO*, *TeStID* without parallel operations

These results gave the motivation to investigate the high performance features of *StreamBase* and thus a second set of experiments were carried out as follows in the next section.

7.2.3.2 Implementations of Single Packet Attacks With Concurrency and Multiplicity

Using concurrency and multiplicity options allow us to achieve higher performance. The understanding of the following definitions will help to understand the set of experiments in this section:

- Concurrency means part of the code (operator or referenced module (group of operators)) run in its own thread.
- Multiplicity refer to the number of instances of the code.
- Dispatch style is related to the multiplicity. The dispatch style specifies how each instance receives tuples in round robin, broadcast, or based on a data value. In broadcast each instance will receive a copy of the incoming tuple. In round robin the first tuple goes to the first instance and the second goes to the second and so on. Based on value is by checking the value against a test condition and then dispatch to the designated instance for that value.

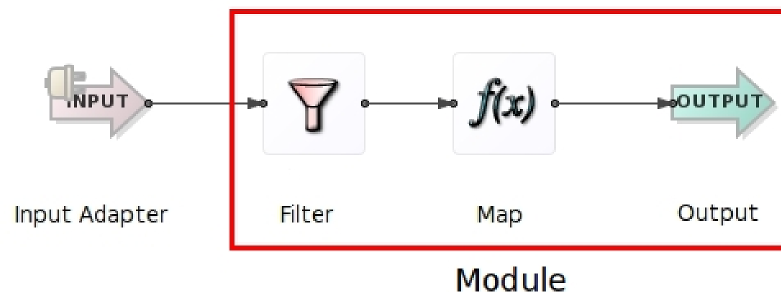


Figure 7.9: Single Packet Attack Program Using Module

In *StreamBase* the concurrency can be set, the multiplicity, or both. According to the *StreamBase* manual, these options can be used for portions of the application that meet certain criteria:

- if the code portion is long-running or compute-intensive;
- can run without data dependencies on the rest of the application;
- it would not cause the containing module to be waiting or blocked.

To use these options in our program in Figure 7.6, we need to apply these guidelines. Also, in *SSQL*, it is allowed to use these options for modules only. In *StreamBase*, a module is a set of operators that include input and output streams, and is written in one SQL file. Figure 7.9 illustrates the change. The application contains two components: the input adapter and a referenced module (from here on we refer to them as input

code and attack module). The attack module contains the filter, map, and output operators for a single attack written as a referenced module. For the input code, we can use the concurrency option but not the multiplicity as it has no input stream. For the attack module we can use both options. This means that there are eight possible implementations as can be seen in Figure 7.10. Table 7.4 shows the results of all the

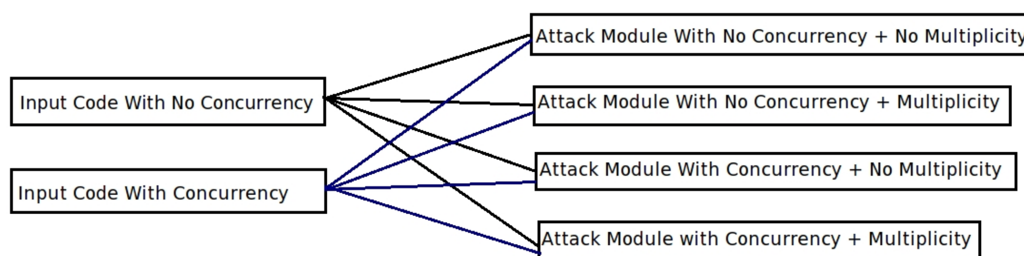


FIGURE 7.10: All Possible Implementations For The Single Packet Attack With Payload

experiments of this category. In this table (CC) denotes "Concurrency". (NCC) denotes "No Concurrency", (1) denotes single instance or no multiplicity, and (n) denotes n multiplicity where n is an integer such that n > 1. For instance, the first row shows the results of running non concurrent input code (NCC) and non concurrent (NCC) 50 attack modules of (1) instance each (i.e., no multiplicity). The following are observations on these results:

- The first row result is almost the same result obtained previously with no concurrency and no multiplicity (the third row in Table 7.2). The difference is that we implement the attack code as a module, but we did not use the concurrency or the multiplicity.

- Using input code with concurrency and no concurrency for the attack module gives better results in general (rows 7-11).

- Row number 9 has the result of best performance where three non concurrent instances of attack modules are used. In rows 7-8 less than three number of instances used and in rows 10-11 more than three instances used. Increasing the number of instances above three cause the performance to degrade. This is consistent with *StreamBase* rule of thumb that is for best performance the number of instances should be equal to the number of cores on the machine or less. So, we have three instances in addition to the input module running on four cores on the testing machine.

- Using input code with concurrency and multiplicity for the attack module gives us the best result that exceeded the results of *SNORT* and *BRO* which are presented in Table 7.1.

TeStID perform well with parallel input code and running multiple threads of the attack modules (rows 7-11). Simple graphical representation for this implementation is in Figure 7.11. When multiplicity option are set, the attack module threads receive input in round robin fashion, so each thread will process a tuple (packet) and the subsequent packet will be processed by another thread. This scenario occurs for all the 50 attack modules. These threads all exist in the main region (no concurrency) which gives the *StreamBase* engine less overhead when it was running in parallel (with concurrency) in the rows 5 and 6.

Finally, the the developer version *StreamBase* Server was used to run all the experiments. The enterprise edition is an ultra low-latency application server optimized for high production level performance [92], but the results achieved by using the developer version was adequate to show the efficiency of using SDP.

	Implementation	X2	X4	X8	X16	X24	X48
1	NCC Input Code + 50 NCC Attack Module (1)	0	-3	-79			
2	NCC Input Code + 50 NCC Attack Module (2)	0	-2	-67			
3	NCC Input Code + 50 CC Attack Module (1)			-293			
4	NCC Input Code + 50 CC Attack Module (2)			-299			
5	CC Input Code + 50 CC Attack Module (1)	-3		-126			
6	CC Input Code + 50 CC Attack Module (2)		-84	-128			
7	CC Input Code + 50 NC Attack Module (1)			0	0	0	-5
8	CC Input Code + 50 NC Attack Module (2)	0	0	0	0	0	-2
9	CC Input Code + 50 NC Attack Module (3)					0	-1
10	CC Input Code + 50 NC Attack Module (5)					0	-2
11	CC Input Code + 50 NC Attack Module (10)					0	-5

Table 7.4: Results of Single Packet Attacks With Concurrency and Multiplicity

7.3 Multiple Packet Attacks Case Studies

As a case study we selected a set of typical multiple packet attacks against the TCP protocol to test *TeStID*. The test data used was selected from the DARPA publicly available IDS Evaluation Data sets [19, 59]. The selected attacks are the DoSNuke (also known as the WinNuke), TCP *syn* flood attack (Neptune), and the Reset Scan attack. With the exception of the TCP Reset attack, all the other attacks were fully explained in detail and their formal specifications were given in Section 5.2. Here, we give the detail and the formal specification for the TCP Reset attack.

The TCP Reset attack is a Denial of Service (DoS) attack in which the attacker sends a forged TCP packet with *rst* flag set to disrupt a TCP connection [19, 60]. When the attacker has access to the network and as soon as he sees a TCP connection request he sends a forged TCP reset packet to disrupt the connection.

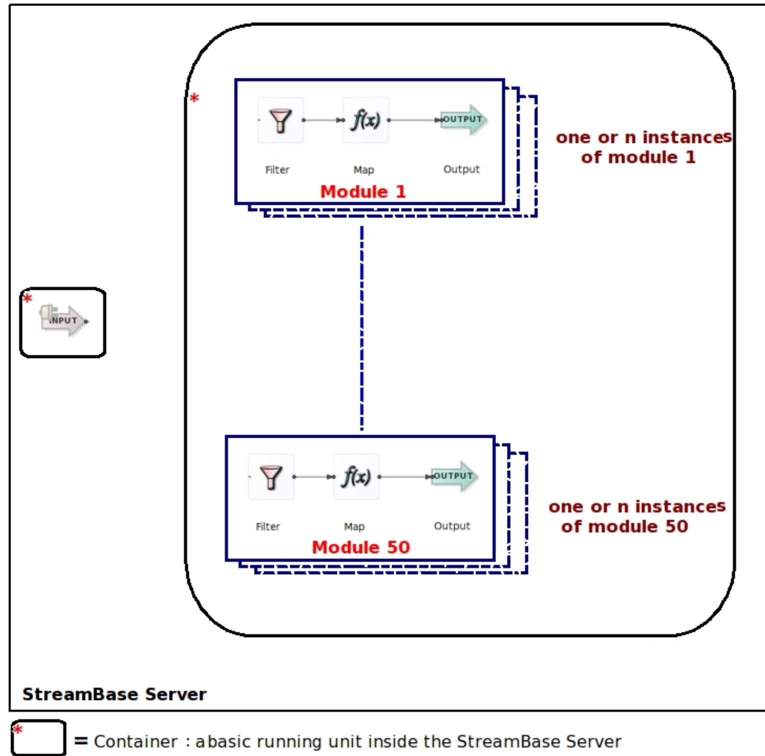


FIGURE 7.11: CC Input Code and NCC Attack module With/Without Multiplicity

One way to identify this attack with minimum false alarms is to look for the session setup and reset request originating from the same machine (the sender). The sequence of the events are as follows:

- The sender sends a TCP packet with the *syn* flag set and Initial Sequence Number (ISN) $= x_5$.
- The attacker will send a forged TCP packet on behalf of the sender with *rst* flag and sequence number equal to $x_5 + 1$. To guarantee the success of the attack, the attacker sends the reset packet as soon as he sees any connection request. In our test we set the time for this second packet to be sent to one second or less.

The representation of the attack using *MSFOMTL* is:

$$\begin{aligned}
&(\exists x_1, x_2, x_3, x_4, x_5)((\exists y_{10}, y_{11}, y_{12}) \\
&P(x_1, x_2, x_3, x_4, x_5, 0, 0, 1, 0, y_{10}, y_{11}, y_{12}) \wedge \\
&\Diamond_{[0,1]}(\exists z_{10}, z_{11,z_{12}}) \\
&P(x_1, x_2, x_3, x_4, x_5 + 1, 0, 0, 0, 1, z_{10}, z_{11}, z_{12}))
\end{aligned} \tag{7.1}$$

Notice that the *syn* (term 8) in the first predicate is set. In the second predicate the *rst* flag (term 9) is set and the ISN (x_5) is incremented by 1. Bear in mind that the sooner the attacker sends the *rst* packet, the better success rate he gets.

7.3.1 Results and Analysis of Multiple Packet Attacks Experiments

The new system was evaluated and analysed using the experimental setup described in Section 7.1.2 and the DARPA evaluation data files. The result of running the experiments is in Table 7.5.

For each attack name in the first column, one or more evaluation data files were used. Originally, DARPA recorded these data files inside and outside the local network everyday for five weeks. The first three weeks contain training data and the last two weeks contain testing data. Each file is labeled by the date and the place where it was recorded inside (in) or outside (out) the LAN. For instance, the first row at Table 7.5 shows the results of running the inside TCP dump file of the 1st of April, 1999. The third column shows the number of packets replayed from the designated day. These replayed packets are equivalent to the actual total number of packets recorded in each designated file. In the fourth column, we state the number of attacks captured at normal replay (as recorded). It takes about 22 hours to replay each file at a rate of 30 to 43 packets per second depending on the number of packets in each file. The fifth column shows the number of attacks captured while using the multiplier option to replay the packets at 1350 times the original recorded speed. We compare the results we got with DARPA's attack detection results (shown in column 6) and we discovered that the system was successful in detecting all of the existing attacks in the used test data files.

Attack Name	Data Set	Packets Replayed	Normal Replay	1350X Normal	Actual No. of Attacks
DoSNuke	01/04/99 in	2,356,503	1	1	1
	05/04/99 in	2,291,319	2	2	2
	06/04/99 in	3,404,824	1	1	1
Neptune	05/04/99 in	2,291,319	1	1	1
	06/04/99 out	2,558,481	3	3	3
	09/04/99 in	3,393,918	1	1	1
TCPReset	06/04/99 in	3,404,824	2	2	2
	07/04/99 in	2,087,942	1	1	1
	09/04/99 in	3,393,918	1	1	1
ResetScan	08/04/99 in	3,201,381	2	2	2

Table 7.5: Multiple Packets Attacks Results

A total of seven different files were used in evaluating the new system. Figure 7.12 shows that the average number of packets successfully replayed is between 29,830 and 43,650 packets/sec. The highest or peak number of packets replayed is between 100,000 and 250,000 packets/sec. The time elapsed to replay these files is between 70 and 85 seconds.

The maximum number of bits/sec replayed is between 160,000,000 and 550,000,000. The average number of bits/sec is between 5,988,352 and 16,467,111 (see Figure 7.13).

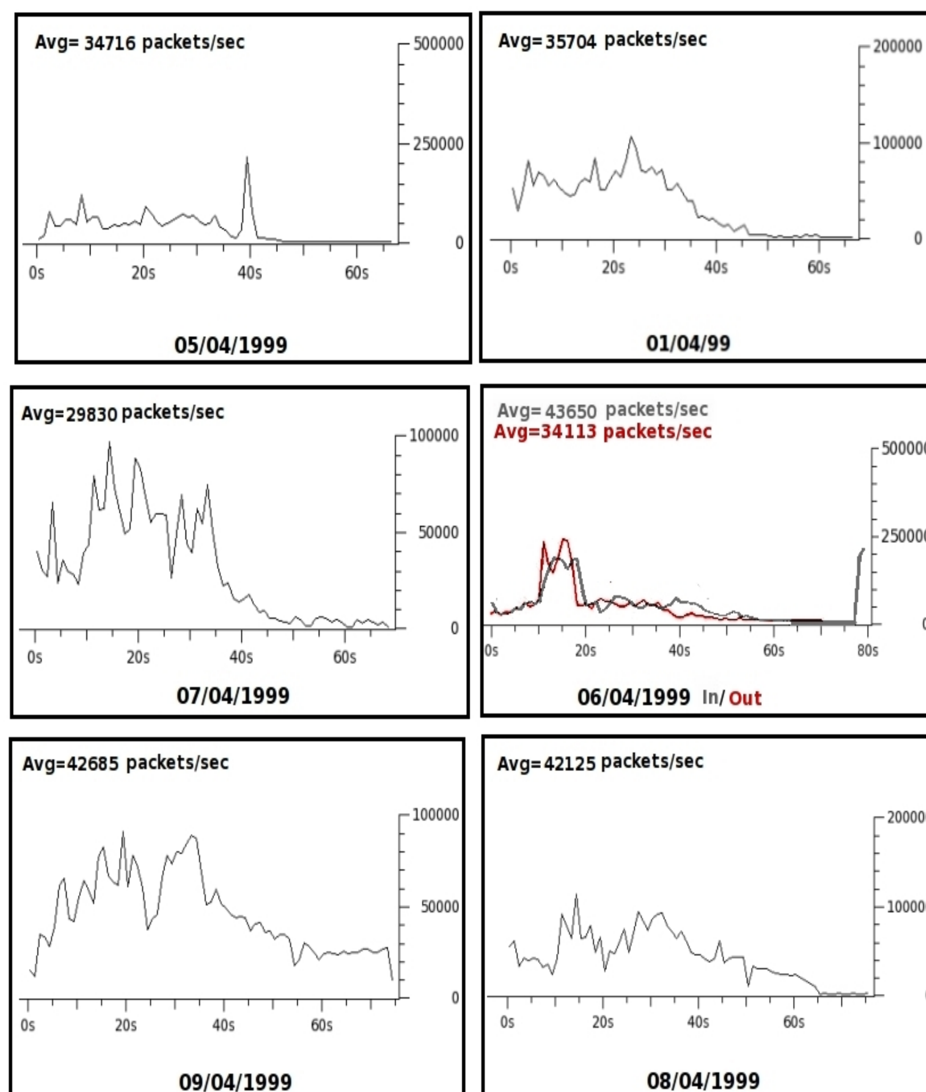


FIGURE 7.12: Maximum packets/seconds for each data test file. Both of the in/out data files of 06/04/1999 were used and they are illustrated in the same graph.

We used *WIRESHARK* the open source network protocol analyzer tool kit to analyze and produce these figures [71]. *WIRESHARK* actually captures and measures the data at the link layer and above for graphical analysis purposes. Thus, the maximum bandwidth rate recorded does not include the overhead of the Ethernet physical layer. Every packet sent over the wire must be preceded by seven preamble bytes and one start frame delimiter byte [53]. These eight bytes allow a device to recognize a new incoming frame. In addition, there is an inter-frame gap which is the idle time between frames. Transmitters are required to wait for a period of transmitting 96 bits (12 bytes) before transmitting subsequent frames. So, a total of 20 bytes (160 bits) Ethernet overhead in sending a packet is not accounted for in the calculation of the maximum bandwidth by *WIRESHARK*. We can calculate the actual bandwidth by multiplying the number of packets per second reached at the maximum bits transmitted per sec by 160. From

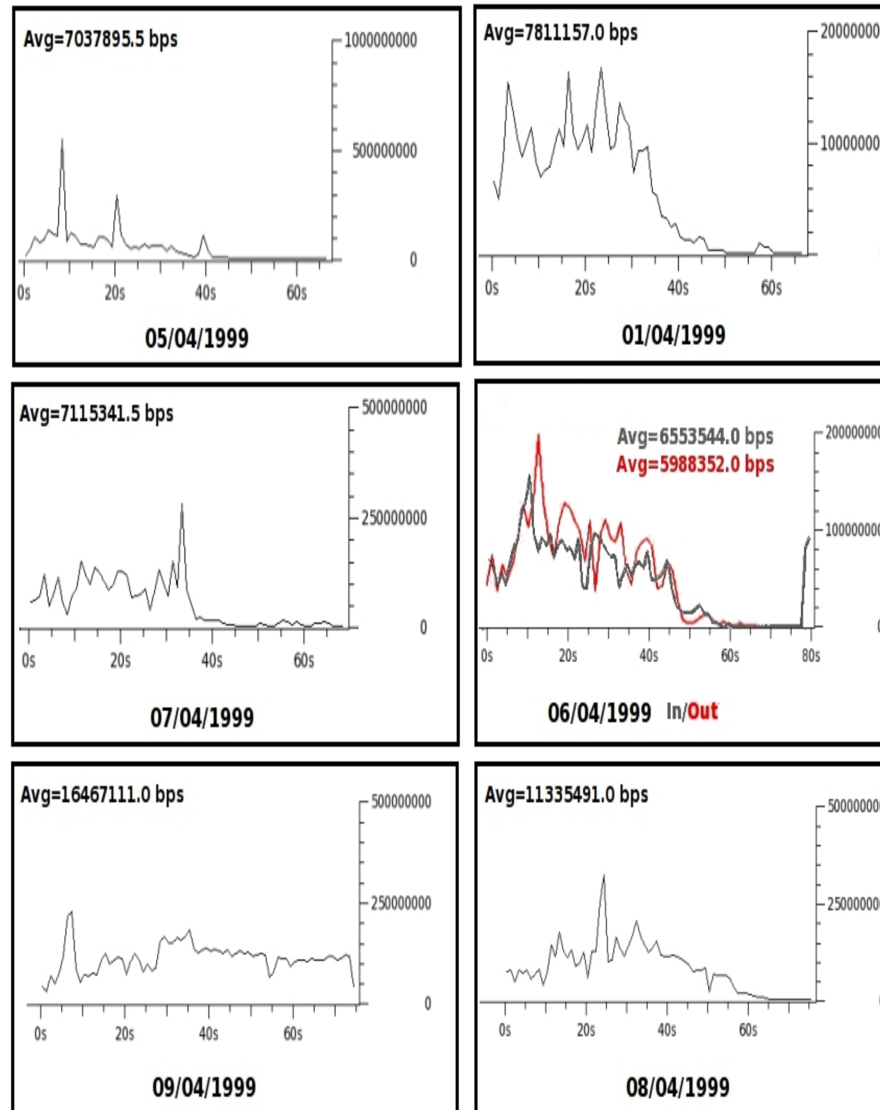


Figure 7.13: Maximum bits/seconds for each data test file

Figure 7.14, we can see that 125,000 packets/sec is sent at 550,000,000 bits/sec which is the max bandwidth reached. Multiplying 125,000 X 160 gives us 20,000,000. Adding this to 550,000,000 gives us 570,000,000 bits/sec.

Capturing the specified attacks without dropping packets at rate up to 570,000 bits/sec and 250,000 packets/sec is really a promising solution toward IDS in high volume traffic networks. For the multiple packet attacks, the number of packets/seconds is very important. This is because of the nature of the attacks in multiple packet attacks. These attacks use multiple packets (arriving in specific order) with certain header values. This means, the more number of packets received by the system the more processing is needed. In Figure 7.12 we can see that in the first half of the experiments in all the data files, an average of 60,000 to 80,000 packets/sec were replayed (this is the rush hours as DARPA recorded the data starting at 8:00 am).

Dreger et al. [25] have experimented with running *BRO* in high volume networks. *BRO* is a well known open source NIDS that works with multiple packets attacks. During their experiments, the best of what *BRO* achieved with using a file that contains a denial of service attacks was 35,000 packets/sec, but with the loss of few packets. They concluded that *BRO* which is highly stateful intrusion detection systems was using memory resources to keep state information and these states could cause the system to run out of memory and crash. Stateful detection means the system is keep tracking of each established session states. So, the detection can be performed by analyzing the information contained in the current packet or from previous packets. Indeed, the system crashed in 2.5 days using normal traffic test data with no attacks.

It was difficult to test *BRO* in our environment as there are no *BRO* script files ready to use that detect the specified attacks in the experiments. Moreover, writing scripts for these attacks is not easy. Nevertheless, we ran a test using four TCP/IP policy files that come with the *BRO* software package. One of these files was the *synflood* attack; the others were policy files that deals with multiple packet attacks. A few seconds after the experiments started and at multiple speed of 1000, the machine running *BRO* crashed. This highlights the advantage using *TeStID*. In *TeStID* the events are kept in memory as long as the temporal distances between these events are valid. Actually, during the experiments less than 400 Mb of memory was used, that is, at the top speed achieved.

We could not perform the same experiments with *SNORT* because *SNORT* does not have the capability to deal with multiple packets attacks. Only signatures for a single packet attacks with or without payload can be specified. *SNORT* could be configured to report single packet attacks that repeat certain times, but it can not report multiple packets attacks either with or without repetition.

We could not make a comparison to *ORCHIDS* or *MONID* which are temporal logic based NIDSs (see Section 3.2). This is because we could not obtain working systems. Furthermore, as these systems were presented as the proof of concept, it is not clear whether these systems have necessary network interfaces to run in a realistic experimental setup.

7.4 Summary

This chapter presented all the results obtained from testing and evaluating the proposed system *TeStID* in misuse attacks. The experiments were conducted to find out the coverage rate of the system, how efficiently it works in high volume environment, and the performance. The experiments covered both single and multiple packet attacks. The experimental setup allow us to replay the data to the sensor machine with the ability to control the volume of the data and to repeat the test. The well known DARPA 1999 evaluation data files and a customized data were used. Using data files with predictable baseline allows us to quantitatively measure the tested systems.

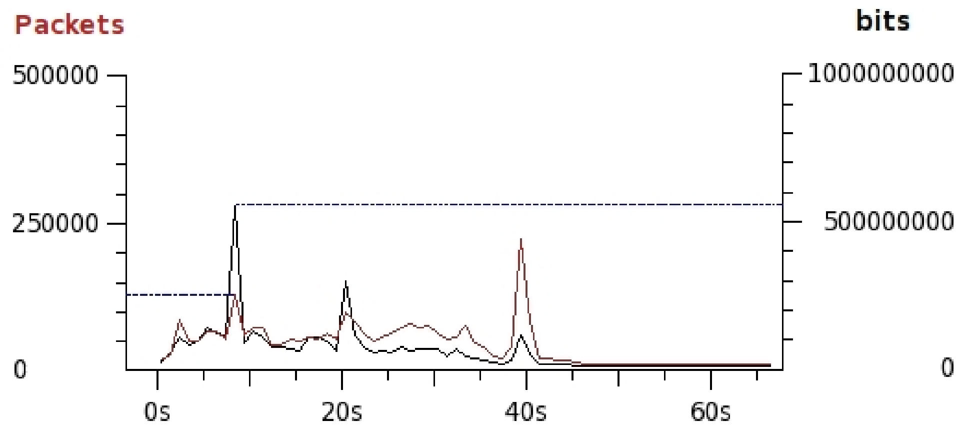


FIGURE 7.14: Packets/Sec and Bits/Sec Overlap graph for 05/04/1999 Data File

For the multiple packet attacks, the results for *TeStID* were promising as it was able to achieve up to 250,000 packets/sec. *SNORT* is not capable of detecting multiple packets, thus no comparison was attempted. *BRO* can detect multiple packets attacks and previously in [25] the best result achieved was 35,000 packets/sec with drop of few packets. Moreover, we tried to run *BRO* with the highest speed *TeStID* achieved using equivalent attacks script files; *BRO* simply crashed. An advantage of *TeStID* is that it does not store session information internally like *BRO*. The events are kept and evaluated in specified temporal distance and discarded when it is no longer needed. This means the events are kept in memory as long as they needed and it will be discarded atuomatically. *ORCHIDS* and *MONID* does not have published results to compare with and we could not obtain the source codes to experiment with.

For single packet attacks with payload, two sets of experiments were conducted with and without the data parallelism features. Also, these tests were carried for *SNORT* and *BRO*. The results for *TeStID* without parallelism was better than *SNORT* but worse than *BRO*. *TeStID* achieved full coverage rate at the multiple speed of 2 but *BRO* achieved it at multiple speed at 4. With the parallelism features, *TeStID* achieved much better results as it only missed one attack at the multiple speed of 48. The fine-grained parallelism features of *StreamBase* raised the performance substantially. In Day and Burns [21] experiments, *SNORT* could not noticeably benefit from multi-cores machine. Intuitively, this is true for *BRO* as it is not written to utilize such architecture [50].

In the next chapter, we look at how *TeStID* can be used to detect attacks using protocol anomaly which is the second part of this thesis. In this method normal behaviour of the communication protocol must be specified and any deviation from these normal behaviour are triggered as possibility of an attack.

Chapter 8

Potential Use of The New System in Anomaly Based IDS

So far all what we have presented in the previous chapters were about the misuse based IDS. As we mentioned in Chapter 2 there are two methods that can be used for intrusion detection. The first method is misuse based and the second is anomaly based. In the proposed system (Figure 5.3), both of theses methods can be implemented. This chapter presents and shows how the same approach and techniques we used for developing the misuse based IDS can also be used to develop protocol anomaly based IDS. This means the syntax and semantics defined in Chapter 5 will be used for specification of normal behaviour and then this specification is translated into *SSQL* using the same tools (Antlr) presented in Chapter 6.

Section 8.1 presents a basic overview of anomaly based network intrusion detection. Section 8.2 presents and discusses the specifications of protocols using temporal logic. The mapping of the formal specifications are presented in Sections 8.3. In Section 8.4, the correctness of the translations are given. Finally, a summary is provided in Section 8.5.

8.1 Anomaly Based Network Intrusion Detection Overview

Anomaly based NIDSs detect intrusions by monitoring the networks for unusual behaviour that differ from normal behaviour. The NIDS resides on the network and examines network traffics packet per packet in real time, or close to real time, to detect deviations from normal behaviour. In some research, a model of normal behaviour is created from the normal ways of network communications, that is, by using the RFC[1] protocol specifications [20, 45] (i.e., by enforcing protocol conformance as a way to detect intrusions). In fact, all the commercial systems mentioned in Section 2.4.2 use protocol anomaly detection. In other research, the normal behaviour is modelled using the traffic

[1]Request for Comments (RFC) is a memorandum published by the Internet Engineering Task Force (IETF) describing methods, behaviours, research, or innovations applicable to the working of the Internet and Internet-connected systems.

characteristics or network behaviour. In these research, different techniques are used such as statistical methods in which the basic models keep averages of some defined variables and detect whether thresholds are exceeded based on the standard deviation of the variable [33, 65]. Other research suggests use of data mining techniques such as clustering and classification [58].

An interesting use of anomaly based detection is suggested by HP in their Tipping Point intrusion prevention system [34]. HP tipping point intrusion prevention system is misuse based and anomaly based NIDS. It provides the ability to detect and prevent network attacks (i.e., filtering). For known attacks it uses signatures misused based detection. For unknown attacks they use vulnerability, protocol anomaly, and traffic anomaly. The vulnerability filters are interesting as it behaves like a network-based "virtual software patch" to protect downstream hosts from network-based attacks on unpatched systems. This is useful for organizations with tens or hundreds of systems to protect vulnerable systems from compromise when patches have not been applied. The vulnerability filter monitors the traffic and blocks it when a specific sequence of events is not met completely. The vulnerability filter is concerned with proprietary application/protocols, whereas the protocol anomaly is concerned with enforcing the RFC protocol conformance as a way to detect intrusions. HP provides Digital Vaccine Services that deliver new filters to their customers on regular basis [35].

Figure 8.1 shows the possible sources of modelling normal behaviour in networks as discussed so far. Actually, the protocol anomaly is a generalization of the proprietary protocols implementations. This means they only differ by the source of the specifications. The protocol anomaly specifications are from the RFC and is publicly announced, but the other is private and not usually announced. Hence from here, we use protocol anomaly to refer to both. Network protocol is a set of rules and messages that can be exchanged between computing systems. These rules and messages of protocols are specified in a set of RFCs. These RFCs describe the syntax, semantics, and the function of the protocol apart from how they should be implemented. This fact may cause the existing of many implementations for a protocol such as the TCP/IP (e.g., Windows, different implementations of Linux/Unix OSs, etc.). This means that an attack may affect some platforms which are not complied by the RFCs. Also, another reason for the success of an attack is that the RFC itself has vulnerability. For instance, the Land attack uses spoofed source address, and so it would be blocked in any implementation that is totally in compliance with RFC 2267 [64]. This attack simulates a TCP connection, but use the victim's own IP address as the source address. The victim computer then attempts to contact itself in order to respond to the simulated connection request. If the target systems are not compliant with RFC 2267, then they may crash or lose services for some time. The point here is that when this attack appeared first time in 1997 it was vulnerability in the original TCP RFCs. So, the attack succeeded on some platforms because of the weakness in the specification and because it was not accounted for by the implementers of the affected platforms. In 1998, the RFC 2267 was released

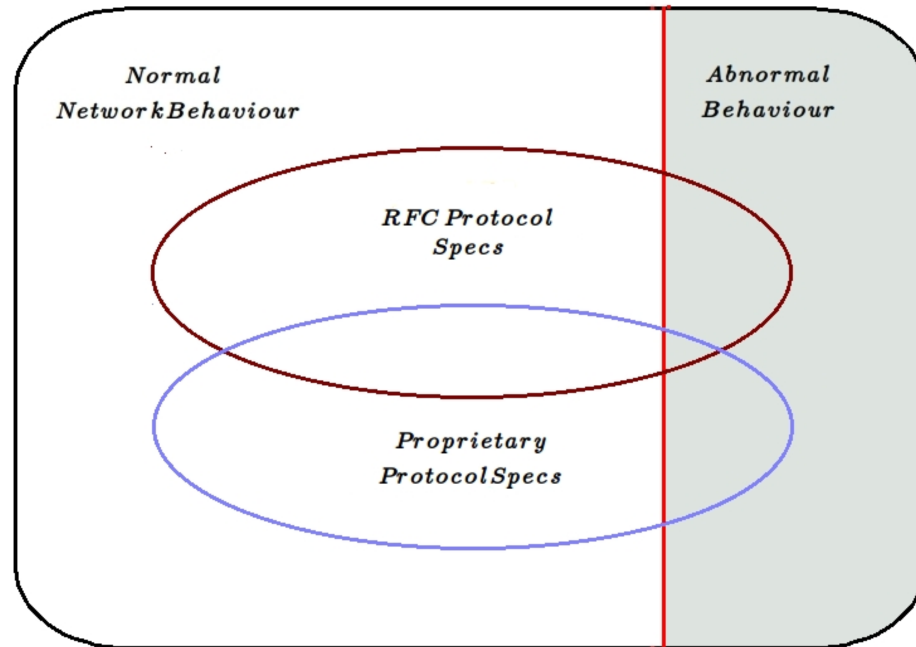


Figure 8.1: Network Anomalies

and the attack affects only the systems that did not implement this specification. It was resurfaced again in 2005 on Windows 2003 and Windows XP SP2 [83] as these operating systems were not compliant with the RFC 2267.

In the rest of this chapter, we continue to use the TCP/IP as our case study. For protocol anomaly, parts of the specifications are used. These parts usually are selected due to the reason of their frequent use (e.g., session establishments and fragmentation/defragmentation of packets) or due to their critical nature (e.g., secure data transfer and authentication).

8.2 Protocol Anomaly Specifications

Formally, we use the *MSFOMTL* to represent parts of the TCP protocol normal specification ϕ and detect any deviation from this specification in the TL models M (incoming events) (i.e., the protocol specification ϕ is not satisfied in M ($M \not\models \phi$). As before, we formally represent a packet as a first order predicate with multi sorts. In TCP/IP a packet has the total number of fields (IP + TCP) of 34. But using 34-arity predicate in the specification is undoubtedly unpleasant for the users (to read or write) and, practically, an error prone task. Hence, it is highly desirable to restrict this only to the required fields. For misuse based intrusion detection since all the attack signatures are known, it is an easy task to identify all the common used fields. Only if a new attack is discovered that uses a new field then this field needs to be included. Many of these signatures have been known for almost 30 years and the chance of using new fields is

unlikely, but is possible. For our experiments, we used 12-arity predicate and it was enough to formally represent all the involved attacks.

In anomaly based intrusion detection, the situation is totally different. We are not dealing with attacks but we are dealing with specifications of normal behaviour. Each function in the specification uses a subset of the fields. But again, using 34-arity is very cumbersome to use. On the other hand, there is no magic number of fields to use. In this chapter, we stick to the 12-arity predicate in our examples. This is adequate to show the concept of using *TeStID* in anomaly detection.

The protocol specifies steps and formulates messages to be exchanged in some order. The way these messages are passed is through packets. Also, the formulation of the packet is self adhere to rules. These rules may specify the acceptable range of some fields or the allowed values of some fields relative to other field value and so on. We consider two categories. The first category covers the normal formation of a single packet. The second category covers the multi step protocol specifications where we model normal behaviour of a fragment of the protocol. More details about these categories are in the subsequent sections.

8.2.1 Single Step Anomalies

In the specifications of protocols, a packet is formulated to exchange a formatted message or information. This information is contained within the fields in such way that they are understandable by the receiving node. Understandable means the combination or use of fields altogether adhere to the protocol specification. So, there are some restrictions on the fields either by limit, range, and relativeness. Limit means there is upper or lower bound values on that field. Range means that the value for the field must be within a certain range. Relativeness refers to a value that is based on another field value using logical comparison and/or arithmetic operation (e.g., $x_6 \geq x_5+1$,). Taking these facts in consideration, the following pseudo code monitors the traffic for single packet anomalies:

Whenever a packet arrives, do:

1. Check the values of the fields that has restrictions.

2. If any value check in step (1) fails, then report this packet as abnormal.

In logic, a packet is represented as a predicate and the fields are the arguments of the predicate. These arguments have different sorts. So, whenever there is a predicate we check that all the arguments with specified conditions on them satisfy their conditions. Formally this can be represented in logic as the following canonical form:

$$\varphi \rightarrow \psi \tag{8.1}$$

where:

- φ is first order predicate (representing a packet).

- ψ is simple first order formula that can be in one of the following forms:
  - simple first order formula in the form (*term* $\langle =, \neq, <, \leq, >, \geq \rangle$ *term*);
  - simple first order formula [$\langle \wedge, \vee \rangle$ simple first order formula]* , where (*) denotes one or more;
  - (formula as any of the previous) $\rightarrow$ (formula as any of the previous).

For single packet anomaly, the user writes a formula that represents normal behaviour using formula 8.1. When this formula is not satisfied, an alarm is raised. Thus, we are looking for $\neg(\varphi \rightarrow \psi)$. There are two forms that the above formula take according to the definition of ψ in formula 8.1. These forms are:

- ψ is simple first order formula or nested simple first order formulae with conjunction or disjunction. So, the negation of formula 8.1 is:

$$\varphi \wedge \neg\psi \tag{8.2}$$

  Here, ψ could be one or more of the conditions that if not satisfied, then an alarm is raised.

- ψ takes the form: $\pi \rightarrow \phi$, where both π and ϕ is a formula as described above (simple first order formula or nested simple first order formulae with conjunction or disjunction). So, the formula 8.1 after replacing ψ: $(\varphi \rightarrow (\pi \rightarrow \phi))$ and the negation of the above formula is:

$$(\varphi \wedge (\pi \wedge \neg\phi)) \tag{8.3}$$

  Here, an alarm is raised if the condition(s) specified in π hold(s) and the condition(s) in ϕ do(es) not hold. So, the user is expected to enter the initial conditions that need to hold in π and the condition(s) that if violated an anomaly alarm is raised in ϕ. Example 7.2 is given to clarify this type of specifications.

Example 7.1: RFC6335 [100] states that the port range is from 0-65535. Also, it states that the lowest and top bound usually is reserved. So, ports 0 and port 65535 are officially reserved by IANA [101]. To write this normal specification requirements then we would make sure that the packet port is between 0-65535 exclusively. This means there are no packet with a source port (x_2) or a destination port (x_4) with a value less than or equal 0 and greater or equal 65535. Formally using the syntactical formula 8.1, this can be represented as:

$$\begin{aligned}
&((\forall x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \\
&P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \rightarrow \\
&(((x_2 > 0) \wedge (x_2 < 65535)) \wedge ((x_4 > 0) \wedge (x_4 < 65535))))
\end{aligned}$$

The above formula represents the normal specification. The negation form that will be translated into *SSQL* is obtained using formula 8.2 as follows:

$$((\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12})$$
$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \wedge$$
$$(((x_2 \leq 0) \vee (x_2 \geq 65535)) \vee ((x_4 \leq 0) \vee (x_4 \geq 65535))))$$

The translated code for the above specification was run against the test data files that we used in Chapter 7 for the experiments on the misuse based IDS. Interestingly, two anomalies were raised, and by examining them, we found that they belong to an already known attack in Snort which is sid-524 (Appendix B.1). This attack uses TCP port 0 for scanning target machines. Port 0 is outside the range of normal specification and this is why it was caught.

Example 7.2: As another example for single packet, we use the RFC793 for TCP/IP specification. In this specification, the reliability of TC/IP is described. One of the basic requirements for this reliability (not losing data and ability to recover) is assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment from the receiving TCP [40]. This means if the *ack* flag is set (x_7) the acknowledgment number (x_6) must be greater than 0. Formally, using 8.1 this is represented as:

$$(\forall x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12})$$
$$(P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \rightarrow (x_7 = 1 \rightarrow x_6 > 0))$$

Applying formula 8.3 gives us the negation of the above formula as follows:

$$(\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12})$$
$$(P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9.x_{10}, x_{11}, x_{12}) \wedge ((x_7 = 1) \wedge (x_6 \leq 0)))$$

From these examples, we can see how *TeStID* can be used in anomaly based intrusion detection. In the misuse based detection, the user needs to write the signature of the attack using MSFOMTL, but in anomaly based detection, the user needs to enter the normal specification.

8.2.2 Multiple Step Anomalies

To implement the protocol RFCs specifications multiple steps are required. Each step is merely a packet description with a defined message or information. This packet in connection oriented protocol scheme, such as TCP also refers to a certain state which is defined in the specification. For intrusion detection, parts of these specifications can be used for anomaly behaviour detection. This part must be from the observed portion of the specifications of the protocol which is reflected by the information stored in the

packet captured on the wire. The unobserved parts are specifications within one of the communication end such as storing connection requests in some data structure (TCP control block TCB) and using the local connection as pointer to access the TCB for this connection if needed.

The reporting of malicious traffic in anomaly detection is different from reporting in the misuse [20]. When and what to report is important and essential for further study or to analyse the incidents found in the traffic. In misuse based, when the pattern is matched, then we report this incident with the attack identification. Part or even all of the packet contents can also be part of the report. But it is clear that if the whole pattern is not matched, then nothing would be reported. In anomaly based detection, the situation is different. When a specification has multiple steps, what should be reported as anomaly? Should we only report when all the steps fail to complete successfully (e.g., all steps are successful except the last)? Or, should we report on the failure of the second step, if not the third step, and so on? For example, suppose we are specifying the normal three handshake steps of the TCP/IP protocol. These three steps if all successful, then this is a normal situation. Now, suppose the first step was initiated and the second step but not the third; or the first step initiated and not followed by the second. These situations are both abnormal, but do we need to report them? Perhaps, not always, as it may mean for example that the network connections are congested and nothing malicious is happening.

We propose two syntactical forms. The first syntactical form is for weak requirements situation. By weak requirements we mean all the steps must be satisfied. If the last step is not satisfied, then we raise an alarm. The second syntactical form is for strong requirements, that is, when each consecutive step must be satisfied or an alarm is raised.

8.2.2.1 Multiple Step Anomalies for Weak Normal Behaviour Requirements

The syntactical form of this classification is the following:

$$\Box(\varphi \rightarrow \psi) \tag{8.4}$$

where:

- φ is a predicate (representing a packet).
- ψ is either:
  - Formula of the form $\Diamond_{[t_1,t_2]} P_i$, where P_i is first order predicate;
  - the same form as 8.4.

Using the above syntax, we can write formulae for two or more steps as follows (P_i denotes first order predicate):

$$\Box(P_1 \rightarrow \Diamond_{[t_1,t_2]} P_2)$$

$$\Box(P_1 \rightarrow \Box_{[t_1,t_2]}(P_2 \rightarrow \Diamond_{[t_3,t_4]}P_3))$$
$$\Box(P_1 \rightarrow \Box_{[t_1,t_2]}(P_2 \rightarrow \Box_{[t_3,t_4]}(P_3 \rightarrow \Diamond_{[t_5,t_6]}P_4)))$$
.
.
. (8.5)

So, the negation forms of the above formulae are:

$$\Diamond(P_1 \wedge \neg\Diamond_{[t_1,t_2]}P_2)$$
$$\Diamond(P_1 \wedge \Diamond_{[t_1,t_2]}(P_2 \wedge \neg\Diamond_{[t_3,t_4]}P_3))$$
$$\Diamond(P_1 \wedge \Diamond_{[t_1,t_2]}(P_2 \wedge \Diamond_{[t_3,t_4]}(P_3 \wedge \neg\Diamond_{[t_5,t_6]}P_4)))$$
.
.
. (8.6)

Each formula of the above will not hold if the last step does not hold. So, the syntax of formula 8.4 can be used if raising an alarm is required if all the steps fail to complete.

8.2.2.2 Multiple Step Anomalies for Strong Normal Behaviour Requirements

If we need to raise an alarm at certain step of the multiple steps specification, then formula 8.4 can be applied up to that step. For example, if we have four specification steps and the requirement is to have an alarm when the third and the fifth steps failed, then we use formula 8.4 twice. Once to represent the first three steps of the specification and again to represent the full four steps of the specification. If the requirements is to raise at each step, then we would need to use formula 8.4 three times, that is, for the first two, first three, and all the four steps. This means we need to write three formulae that will have syntax similar to the three formulae in 8.5. Also, the negation of these formulae will have similar syntax to the formulae in 8.6. It would be more convenient if this requirement of raising alarm at each step can be entered in one formula. For this, we suggest the second syntactical formula:

$$\Box(\varphi \rightarrow \psi) \tag{8.7}$$

where:

- φ is a predicate (representing a packet).
- ψ is either:
  - Formula of the form $\Diamond_{[t_1,t_2]}P_i$, where P_i is first order predicate;
  - $P_i \wedge \psi$

Using the above syntax, we can write formulae for two or more steps as follows (P_i denotes first order predicate):

$$\begin{aligned}
&\Box(P_1 \rightarrow \Diamond_{[t_1,t_2]} P_2) \\
&\Box(P_1 \rightarrow \Diamond_{[t_1,t_2]}(P_2 \wedge \Diamond_{[t_3,t_4]} P_3)) \\
&\Box(P_1 \rightarrow \Diamond_{[t_1,t_2]}(P_2 \wedge \Diamond_{[t_3,t_4]}(P_3 \wedge \Diamond_{[t_5,t_6]} P_4))) \\
&. \\
&. \\
&.
\end{aligned} \tag{8.8}$$

If we take the last formula above to represent four normal steps specification we can conclude the following about when it will not hold in one of the following cases:

- if P_1 holds but not P_2;
- if P_1 and P_2 holding and not P_3;
- if P_1, P_2, and P_3 holding and not P_4.

We can argue that this way of writing stronger requirements makes it easier in case we need to raise an alarm at each consecutive steps. In the grammar file, the parse and lexical rules for these syntax are shown in Figures 8.2, 8.3, and 8.4.

Example 7.3: In this example we use the TCP simultaneous connection synchronization specification as described in RFC793 [40]. There are two ways for establishing connections in TCP as in the RFC, the three-way handshake and the simultaneous connection synchronization. In the simultaneous connection specification the normal steps between clients A and B for the process as in the RFC is shown in Figure 8.5. In the following we specify the equivalent to observable steps 2,3,5,6 in Figure 8.5:

- Client A sends a packet to B with source IP (x_1), source port (x_2), destination IP (x_3), destination port (x_4), initial sequence number $= S_A$, and sets the *syn* flag.
- Client B just after the time that A is sending the above request, sends a packet to A with source IP (x_3), source port (x_4), destination IP (x_1), destination port (x_2), initial sequence number $= S_B$, and sets the *syn* flag.
- Client A receives the synchronization request from B and responds with a packet that has the acknowledge number $= S_B + 1$ and both the *syn* and *ack* flags set.
- Client B receives the synchronization request from A and responds with a packet that has the acknowledge number $= S_A + 1$ and both the *syn* and *ack* flags set.

To formally represents the above specification, we use the syntax form 8.4 as follows:

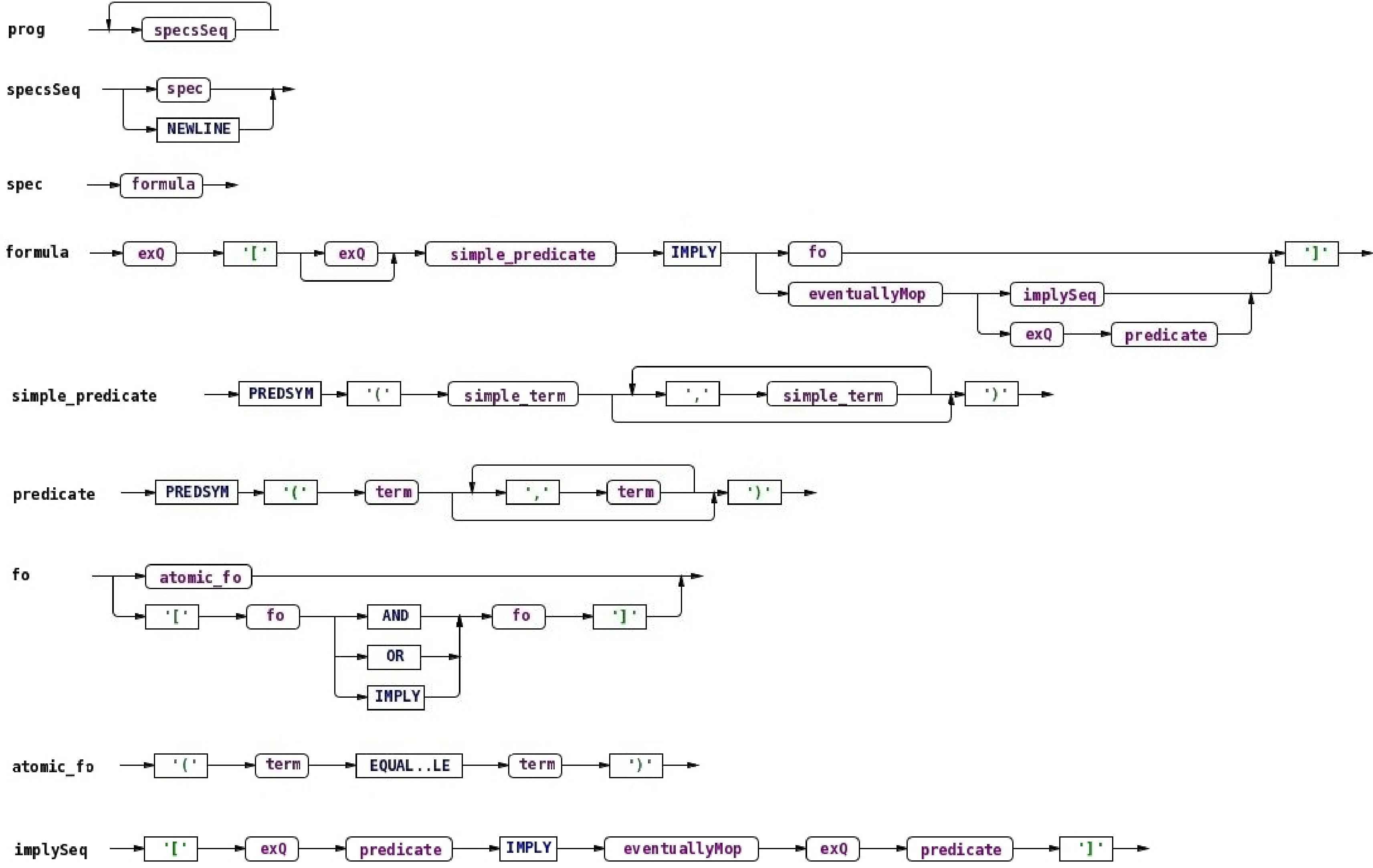


Figure 8.2: Parser Rules (1 of 3)

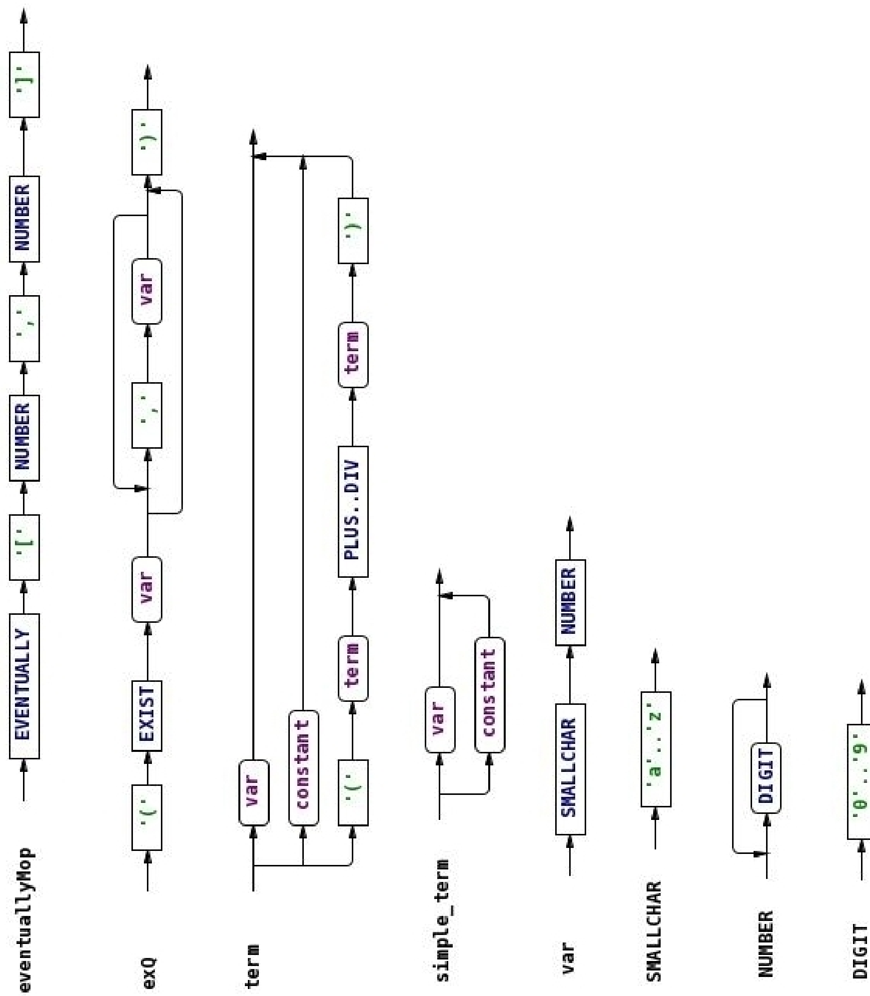


Figure 8.3: Parser Rules (2 of 3) for The Anomaly Based Detection

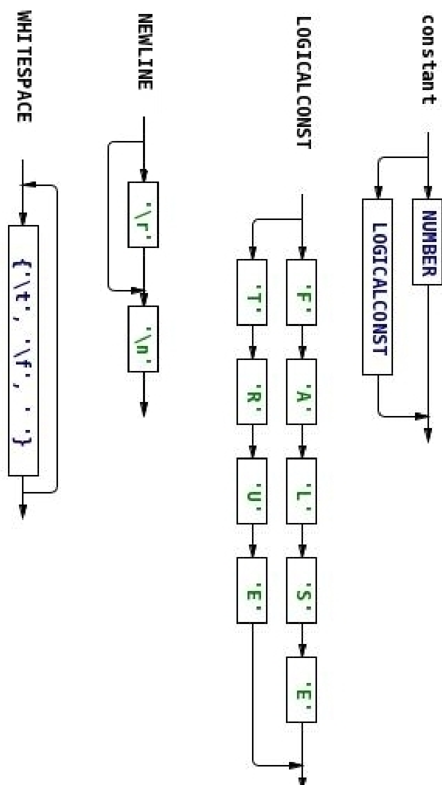


Figure 8.4: Parser Rules (3 of 3) for The Anomaly Based Detection

```
    TCP A                                                              TCP B
1.  CLOSED                                                             CLOSED
2.  SYN-SENT        --> <SEQ=100><CTL=SYN>                             ...
3.  SYN-RECEIVED    <-- <SEQ=300><CTL=SYN>                             <-- SYN-SENT
4.                  ... <SEQ=100><CTL=SYN>                             --> SYN-RECEIVED
5.  SYN-RECEIVED    -->  <SEQ=100><ACK=301><CTL=SYN,ACK>               ...
6.  ESTABLISHED     <-- <SEQ=300><ACK=101><CTL=SYN,ACK>                <-- SYN-RECEIVED
7.                  ... <SEQ=101><ACK=301><CTL=ACK>                    -->    ESTABLISHED
```

FIGURE 8.5: Simultaneous Connection Synchronization (RFC793, September 1981)

$$\begin{aligned}
&\Box((\forall x_1, x_2, x_3, x_4, S_A, S_B)\\
&((\exists y_6, y_7, y_9, y_{10}, y_{11}, y_{12})P(x_1, x_2, x_3, x_4, S_A, y_6, y_7, 1, y_9, y_{10}, y_{11}, y_{12}) \rightarrow\\
&\Box[0,1]((\exists w_6, w_7, w_9, w_{10}, w_{11}, w_{12})P(x_3, x_4, x_1, x_2, S_B, w_6, w_7, 1, w_9, w_{10}, w_{11}, w_{12}) \rightarrow\\
&\Box[0,1]((\exists z_9, z_{10}, z_{11}, z_{12})P(x_1, x_2, x_3, x_4, S_A, S_B + 1, 1, 1, z_9, z_{10}, z_{11}, z_{12}) \rightarrow\\
&\Diamond_{[0,1]}(\exists k_9, k_{10}, k_{11}, k_{12})P(x_3, x_4, x_1, x_2, S_B, S_A + 1, 1, 1, k_9, k_{10}, k_{11}, k_{12})))))
\end{aligned}$$

The above formula is the formula of the specification entered by the user. In the mapping the above formula is translated into the negative form (formula 8.6 as following:

$$\begin{aligned}
&((\exists x_1, x_2, x_3, x_4, S_A, S_B)\\
&\Diamond_{[0,1]}((\exists y_6, y_7, y_9, y_{10}, y_{11}, y_{12})P(x_1, x_2, x_3, x_4, S_A, y_6, y_7, 1, y_9, y_{10}, y_{11}, y_{12}) \wedge\\
&\Diamond_{[0,1]}((\exists w_6, w_7, w_9, w_{10}, w_{11}, w_{12})P(x_3, x_4, x_1, x_2, S_B, w_6, w_7, 1, w_9, w_{10}, w_{11}, w_{12}) \wedge\\
&\Diamond_{[0,1]}((\exists z_9, z_{10}, z_{11}, z_{12})P(x_1, x_2, x_3, x_4, S_A, S_B + 1, 1, 1, z_9, z_{10}, z_{11}, z_{12}) \wedge\\
&\neg\Diamond_{[0,1]}(\exists k_9, k_{10}, k_{11}, k_{12})P(x_3, x_4, x_1, x_2, S_B, S_A + 1, 1, 1, k_9, k_{10}, k_{11}, k_{12})))))
\end{aligned}$$

The above formula is for weak normal requirements specification, which means that it will raise an alert only when the fourth packet or the fourth step of the simultaneous handshake is missing. This formula was translated into *SSQL* code and tested using the custom data file prepared in section 7.1.2. No alert or alarm was raised. In fact, the custom data file has a total of 88458 TCP connection records, but the simultaneous way of TCP connection is used to handle simultaneous requests for a connection which occur rarely.

The interesting finding was when we implemented this example (7.3) as strong requirements, which means that an alert will be raised whenever a step (2,3, or 4) is not completed. The result was a total of 88458 alerts. This is because all the connections in the test data file used the three handshake way of tcp connection (explained earlier in Section 5.2). The three way TCP handshake and the simultaneous TCP handshake share the first step but they differ in the second step. This take us back to the reporting

issue of anomaly behaviour in multiple step specifications that we discussed earlier in this chapter.

8.3 Protocol Anomaly Formulae Mapping

To map the syntactical anomaly formulae defined in the previous section, we use the negation of the formula entered by the user.

Single Packet Anomaly: The negated forms we are going to map are the negations of formula 8.1:

$$(\varphi \wedge (\psi \wedge \neg\pi))$$

and

$$(\varphi \wedge \neg\pi)$$

where:

- φ is a first order predicate.
- ψ and π is conjunction or disjunction of Boolean formulae built of the terms of φ.

For the mapping we are concerned with mapping a subset of MSFOMTL (Δ) into a subset of *SSQL* (Θ), the mapping function (M) is:

$$M : \Delta \longrightarrow \Theta$$

We use the same definition for schema, tuple, input stream, output stream, and stream as in Section 6.3. The basic elements of the mapping function is defined as before:

$\boxdot \in \{=, <>, >, <, >=, <=\}$ // $\boxdot$ represents a relational operator

$\boxtimes \in \{+, -, *, /\}$ // $\boxtimes$ represents a mathematical operator

$M1 : P(x_1, x_2, ..., x_n) \mapsto$ " SELECT $x1, x2, ..., xn$ FROM *input_stream*"

$M1 : \wedge \mapsto$ "WHERE " //the first conjunction

$M1 : ci \mapsto ci$ // where c is constant

$M1 : x_i \mapsto xi$ // where x is variable

$M1 : x_i \boxdot \langle c_j | x_j \rangle \mapsto xi \boxdot \langle cj | xj \rangle$

$M1 : \neg(x_i \boxdot \langle c_j | x_j \rangle) \mapsto (xi \bar{\boxdot} \langle cj | xj \rangle)$

$M1 : x_i \boxdot (x_j \boxtimes \langle x_k | c_k \rangle) \mapsto xi \boxdot (xj \boxtimes \langle xk | ck \rangle)$

$M1 : (\neg(x_i \boxdot (x_j \boxtimes \langle x_k | c_k \rangle))) \mapsto (xi \bar{\boxdot} (xj \boxtimes \langle xk | ck \rangle))$

$M : \neg(BF_i \wedge BF_j) \mapsto (\neg BF_i$ or $\neg BF_j)$ // BF=Boolean formula of terms

$M : \neg(BF_i \vee BF_j) \mapsto (\neg BF_i$ and $\neg BF_j)$

$M1 : LF \mapsto$ " INTO outputstream; " //line feed

As an example for mapping, we take the the example 7.1 from Section 7.2.1, and its negation form is as follows:

$$P(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}) \\ \wedge \neg(((x_2 > 0) \wedge (x_2 < 65535)) \wedge ((x_4 > 0) \wedge (x_4 < 65535)))$$

Following the above rules, the mapping is as follows:

$P(x_1, ..., x_{12}) \mapsto$ SELECT $x_1, ..., x_{12}$ FROM *inputstream*

$\wedge \mapsto$ WHERE $/*$ this is the first conjunction in the form

$$\neg(((x_2 > 0) \wedge (x_2 < 65535)) \wedge ((x_4 > 0) \wedge (x_4 < 65535))) \mapsto$$

$$(\neg((x_2 > 0) \wedge (x_2 < 65535)) \text{ or } \neg((x_4 > 0) \wedge (x_4 < 65535))) \mapsto$$

$$((\neg(x_2 > 0) \text{ or } \neg(x_2 < 65535) \text{ or } \neg(x_4 > 0) \text{ or } \neg(x_4 < 65535))) \mapsto$$

$$(((x_2 <= 0) \text{ or } (x_2 >= 65535) \text{ or } (x_4 <= 0) \text{ or } (x_4 >= 65535)))$$

The final mapping or *SSQL* code is:

```
SELECT x_1,...,x_12 FROM inputstream
WHERE (((x_2 = 0) or (x2 >= 65535) or (x4 <= 0) or (x4 >= 65535)));
```

Multiple Steps Anomaly: The negation form of this syntactical form (formula 8.6) is similar to what has been defined for the forward multiple packet attacks (Section 6.3). The pattern operator(s) will be used and the mapping function basic elements are the same.

8.4 Correctness

In this chapter, three syntactical forms have been proposed for specification of normal behaviour. The correctness of the translations of these forms follows from the proof of correctness presented in Section 6.4 of other syntactical forms used for misuse based intrusion detection. Here, for the translation, we use the negation form of the syntactical formula. So, we need to show the correctness of the translations of these negation forms.

The first syntactical form is the single step anomaly (formulae 8.2, 8.3). These syntactical forms have a predicate and some constraints on some of its arguments. This will be mapped to the filter operator exactly as the first syntactical form defined for misuse formula 5.1. The formula 5.1 has the same structure, that is, a predicate and some constraints on its arguments. So, the correctness of the syntactical form (formulae 8.2, 8.3) follows from the proof in Section 6.4 for formula 5.1.

The second syntactical form (formula 8.6) is for multiple step weak normal behaviour requirements. This form will be translated into the pattern operator exactly as the forward multiple packet attacks (formula 5.4). Thus the proof follows the proof in Section 6.4 for formula 5.4.

The last syntactical form is for the multiple step strong requirements (formula 8.8). This form is implemented by using formula 8.6 for weak normal behaviour requirements repeatedly. For example, if this formula is used for three steps specification, then formula 8.6 will be used twice (i.e., reporting the second and third steps failures). This syntax was proposed to make it easier for the user, that is, if he wants to report the failure of each step. The proof of correctness follows from the proof for the formula 8.6.

8.5 Summary

This chapter explored the potential use of the proposed approach of using temporal logic and stream data processing in anomaly based network intrusion detection. A basic overview of anomaly based network intrusion detection is presented. Specifically, the use of the proposed system for protocol anomaly based detection is considered. In protocol anomaly based intrusion detection, deviations from protocol normal specifications are considered suspicious activities. Parts of protocol specifications is normally used. The specifications can be from RFCs, or from proprietary or vendor specific protocol such as Microsoft DCOM protocol. Some syntactical forms for normal specifications are provided with examples. The mapping of these syntactical form into *SSQL* presented and explained. Finally, the correctness of the translations of the syntactical forms are given.

Chapter 9

Conclusion

In this chapter a summary of the research conducted is presented in Section 9.1. The scientific contributions of the research and the significance of the proposed novel approach to develop network based intrusion detection system are presented in Section 9.2. Finally, future work and further research opportunities are suggested in 9.3.

9.1 Summary

An intrusion detection system is one of the security mechanisms that must exist in any IT infrastructure to protect connected systems and networks. The current network intrusion detection systems can not keep up with the constant increase of network speed. The buffers in these systems are filled up with packets very quickly and are dropped before they can be processed. To address this issue of IDS in high volume networks, the research of this thesis has used a combination of temporal logic and stream data processing to develop network based IDS. The main research question is: "Can stream data processing technology be utilised in conjunction with temporal logic to develop a system that works efficiently and accurately in high volume networks?" How the proposed system efficiently uses the available resources and accurately detects all attacks (coverage rate) were addressed in this research.

The research to develop *TeStID* involves the use of temporal logic for specifications of attacks in misuse based method and for specifications of the normal behaviour in the anomaly based method, the process of mapping logical specifications into *SSQL*, and the use of SDP as the attack (or normal behaviour) detection engine. As part of the system an automated translator was built to parse MSFOMTL formulae and then translate into *SSQL*.

The abstract view and modelling of the system presented in Chapter 5. Also, the syntax and semantics of Many Sorted First Order Metric Logic which is used to specify attack patterns or normal behaviour of parts of protocols was defined in this chapter. Chapter 4 presented an overview of SDP technology and the proposed system architecture. The mapping of temporal logic into stream queries and developing the translator were explained in Chapter 6. The experimental setup and results were presented in

Chapter 7. Finally, potential use of the system in anomaly based intrusion detection was given in Chapter 8. In the next section, the contributions of the thesis and findings are presented.

9.2 Contribution

Combining the use of temporal logic and stream data processing technology gives a promising solution and a valuable contribution to the field of intrusion detection. The experiments carried out involved testing and comparing with Snort and Bro on the same machine configuration and with the same data sets. The following highlights the findings:

- *TeStID* was more efficient than Bro and Snort as it achieved higher bandwidth with full coverage rate. In some of the experiments we used the parallel features of *StreamBase* and these features show that it can boost the performance of the system. This has been achieved using the developer edition and not the enterprise edition. The *StreamBase* enterprise edition (or *StreamBase* Server) is an ultra low-latency application server optimized for high production level performance [92], but the results achieved by using the developer version was adequate to show the efficiency of the proposed system.
- Using temporal logic for the specifications has the advantages of giving the users an easy, concise, unambiguous, and transparent (abstract from technical details) way to specify attacks or normal behaviour.
- *TeStID* is easy to maintain. To add, modify, and delete attacks all what is required is to modify the formulae in the text file and run the translator.
- When the experiments run at the top achievable speed by the hardware using TCPREPLAY with top speed option (close to 1 Gbits/Sec), Bro crashed, whereas *TeStID* did not crash but the coverage rate was dropped. Bro is stateful NIDS and maintains information about connections. On the other hand *TeStID* only maintains information as needed, that is, as long as the temporal relations between events hold.

The above results encourage the use of this approach to develop a full intrusion detection system. Although *StreamBase* system was used, the same design and concepts can be used with other commercial or open data stream systems. Further work and more research opportunities are discussed in the next section.

9.3 Future Work and Research

In this section we propose further improvements and research opportunities following from this research.

- Extending the work on misuse based NIDS for other protocols (ICMP, UDP, etc.) should not be difficult following the work in this thesis.
- Extending the initial work on anomaly protocol intrusion detection using TCP and covering other communication protocols.
- NIDS uses network packets as source of input. The packet structure covers all the layers of the network including the application layer. Thus writing application specific intrusion detection is further extension to this research.
- In this research, temporal logic formulae were mapped to *SSQL*. An alternative research opportunity would be to have executable temporal logic using the stream engine low level interface to interact directly with the internal data structure and streaming handling API.
- Correlated distributed events attacks are new attacks that intruders launch to avoid normal IDS. The events may be from several distributed resources such as logs and alerts from firewalls, intrusion detection systems, operating systems, or other software. A research opportunity is to study dealing with these type of attacks with the proposed system.
- Further evaluation of *TeStID* is to install it on an appliance and compare it to appliance based IDSs.
- Using other techniques with SDP for intrusion detection can be investigated such as using neural networks, fuzzy logic, probabilistic logic, etc.
- Detecting intrusions in encrypted traffic.
- Formulating specifications in temporal logic from normal traffic for application or protocol.

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, Jan. 2005.

[2] C. Aggarwal and P. Yu. Outlier detection for high dimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, USA, 2001.

[3] A. Ahmed, A. Lisitsa, and C. Dixon. A misuse-based network intrusion detection system using temporal logic and stream processing. In P. Samarati, S. Foresti, J. Hu, and G. Livraga, editors, *Proceedings of the 5th International Conference on Network and System Security*, pages 1–8, Milan, Italy, 2011. IEEE. ISBN 978-1-4577-0458-1.

[4] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *REX Workshop*, pages 74–106, 1991.

[5] J. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., 1980.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[7] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004. URL `http://ilpubs.stanford.edu:8090/641/`.

[8] D. Barbara, N. Wu, and S. Jajodia. Detecting novel network intrusions using bayes estimators. In *Proceedings of the First SIAM Conference on Data Mining*, Chicago, IL, USA, 2001.

[9] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 44–57, Venice, Italy, 2004.

[10] L. Bic and R. L. Hartmann. AGM: A dataflow database machine. *ACM Trans. Database Syst.*, 14(1):114–146, Mar. 1989. ISSN 0362-5915. doi: 10.1145/62032.62037. URL `http://doi.acm.org/10.1145/62032.62037`.

[11] BroWiki. Reference Manual: Signatures. `http://www-old.bro-ids.org/wiki/index.php/Reference_Manual:_Signatures`, 2009. Accessed on 02 January 2013.

[12] J. Cabrera, J. Gosar, W. Lee, and R. Mehra. On the statistical distribution of processing times in network intrusion detection. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 75 – 80 Vol.1, dec. 2004. doi: 10.1109/CDC.2004.1428609.

[13] CERT/CC. CERT® Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, 2001.

[14] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL databases with temporal logic. *ACM Trans. Database Syst.*, 26(2):145–178, June 2001. ISSN 0362-5915. doi: 10.1145/383891.383892. URL `http://doi.acm.org/10.1145/383891.383892`.

[15] CISCO. Cisco IOS Intrusion Prevention System (IPS). `http://www.cisco.com/en/US/products/ps6634/index.html`, 2012. Accessed on 02 January 2013.

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362384.362685.

[17] K. J. Cox. *Managing Security with Snort and IDS Tools.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. ISBN 0596006616.

[18] C. Cranor, T. Johnson, and O. Spataschek. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651, 2003.

[19] R. Cunningham, R. Lippmann, J. Fried, S. Garfinkel, R. Kendall, S. Webster, D. Wyschogrod, and M. Zissman. Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation. Technical report, Defense Advanced Research Projects Agency, Department of US Defense, 1998.

[20] K. Das. Protocol Anomaly Detection for Network-based Intrusion Detection. `http://www.sans.org/reading_room/whitepapers/detection/protocol_anomaly_detection_for_networkbased_intrusion_detection_349?show\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{3\global\mathchardef\accent@spacefactor\spacefactor}\accent223\egroup\spacefactor\accent@spacefactor49.php&cat=detection`, 2002. Accessed on 02 January 2013.

[21] D. Day and B. Burns. A performance analysis of Snort and Suricata network intrusion detection and prevention engines. In *Proceedings of the ICDS'11, 5th International Conference on Digital Society*, pages 187–192, Gosier, Guadeloupe, France, 2011.

[22] H. Debar, M. Becker, and D. Siboni. A neural network component for intrusion detection system. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 240–250, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[23] S. Dharmapurikar, J. Lockwood, and M. Ieee. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24:2006, 2006.

[24] C. Dowell and P. Ramstedt. The ComputerWatch data reduction tool. In *Proceedings of the 13th National Computer Security Conference*, Washington DC, USA, 1990.

[25] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of the SIGSAC: 11th ACM Conference on Computer and Communications Security (CSS'04)*, pages 2–11, 2004.

[26] Endace Ltd. EndaceAccess™. `http://www.endace.com/`, 2012. Accessed on 02 January 2013.

[27] Enterasys Secure Networks. Enterasys® Intrusion Prevention System: Post-Connect threat analysis, prevention and containment. `http://www.enterasys.com/products/advanced-security-apps/dragon-intrusion-detection-protection.aspx`, 2012. Accessed on 02 January 2013.

[28] EsperTech Inc. Esper - event stream and complex event processing for java. `http://esper.codehaus.org/esper-3.3.0/doc/reference/en/html_single/index.html`, 2009. Accessed on 02 January 2013.

[29] D. M. Gabbay and P. McBrien. Temporal logic & historical databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB

'91, pages 423–430, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishttp://tools.ietf.org/html/rfc6335hers Inc. ISBN 1-55860-150-3. URL http://dl.acm.org/citation.cfm?id=645917.672332.

[30] A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *Proceedings of the Eighth USENIX Security Symposium*, pages 141–151, Washington DC, USA, 1999.

[31] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32 (2):5–14, June 2003.

[32] M. Grzenda. Towards the reduction of data used for the classification of network flows. In E. Corchado, V. Snášel, A. Abraham, M. Woźniak, M. Graña, and S. Cho, editors, *Hybrid Artificial Intelligent Systems*, volume 7209 of *Lecture Notes in Computer Science*, pages 68–77. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28930-9. doi: 10.1007/978-3-642-28931-6_7. URL http://dx.doi.org/10.1007/978-3-642-28931-6_7.

[33] J. Haggerty, Q. Shi, and M. Merabti. Statistical signatures for early detection of flooding denial-of-service attacks. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP Advances in Information and Communication Technology*, pages 327–341. Springer US, 2005. ISBN 978-0-387-25658-0. doi: 10.1007/0-387-25660-1_22. URL http://dx.doi.org/10.1007/0-387-25660-1_22.

[34] Hewlett-Packard Development Company, L.P. A complete set of security solutions that address today's sophisticated security threats at the perimeter and interior of your business. http://h17007.www1.hp.com/us/en/products/network-security/index.aspx, 2012. Accessed on 02 January 2013.

[35] Hewlett-Packard Development Company, L.P. Tipping Point Digital Vaccine Services. http://h17007.www1.hp.com/docs/security/400931-004_DigitalVaccine.pdf, 2012. Accessed on 02 January 2013.

[36] IBM Corp. Introducing IBM Security Network Intrusion Prevention System (IPS) products. http://pic.dhe.ibm.com/infocenter/sprotect/v2r8m0/topic/com.ibm.ips.doc/concepts/landing_page.htm, 2012. Accessed on 02 January 2013.

[37] IBM Corp. WebSphere Business Events. http://www-01.ibm.com/software/integration/wbe/, 2012. Accessed on 02 January 2013.

[38] IBM Corp. RealSecure® Server Sensor. http://pic.dhe.ibm.com/infocenter/sprotect/v2r8m0/topic/com.ibm.legacy.doc/RealSecure_Server_Sensor.htm?resultof=%22%72%65%61%6c%73%65%63%75%72%65%22%20%22%72%65%61%6c%73%65%63%75%72%22%20, 2012. Accessed on 02 January 2013.

[39] IDAPPCOM Ltd. Traffic IQ Library. `http://www.idappcom.com/index.php`, 2012. Accessed on 02 January 2013.

[40] IETF. Transmission Control Protocol Specification. `https://tools.ietf.org/html/rfc793`, 1981. Accessed on 02 January 2013.

[41] X. W. J. Lin and S. Jajodia. Abstraction-based misuse detection: High-level specification and adaptable strategies. In *Proceedings of the 11th Computer Security Foundation Workshop*, pages 190–201, Washington, DC, 1998.

[42] Jacobson, C. Leres, and S. McCannee. *The TCPDUMP Manual Page*. Lawrence Berkeley Laboratory, Berkeley, CA, June, 1989.

[43] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. In *Proceedings of the 19th annual IFIP WG 11.3 working conference on Data and Applications Security*, DBSec'05, pages 1–15, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28138-X, 978-3-540-28138-2. doi: 10.1007/11535706_1. URL `http://dx.doi.org/10.1007/11535706_1`.

[44] D. S. P. Jr., R. R. Muntz, and H. L. Chau. The Tangram stream query processing system. In *ICDE*, pages 556–563. IEEE Computer Society, 1989. ISBN 0-8186-1915-5.

[45] D.-H. Kang, B.-K. Kim, J.-T. Oh, T.-Y. Nam, and J.-S. Jang. FPGA based intrusion detection system against unknown and known attacks. In Z.-Z. Shi and R. Sadananda, editors, *Agent Computing and Multi-Agent Systems*, volume 4088 of *Lecture Notes in Computer Science*, pages 801–806. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-36707-9. doi: 10.1007/11802372_97. URL `http://dx.doi.org/10.1007/11802372_97`.

[46] Keita Fujii. A Java library for capturing and sending network packets. `http://netresearch.ics.uci.edu/kfujii/Jpcap/doc/index.html`, 2007. Accessed on 02 January 2013.

[47] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990. ISSN 0922-6443. doi: http://dx.doi.org/10.1007/BF01995674.

[48] H. Lai, S. Cai, J. Xi, and H. Li. A parallel intrusion detection system for high-speed networks. *ACNS 2004. LNCS*, 3089:439–451, 2004.

[49] K.-Y. Lam, L. Hui, and S.-L. Chung. A data reduction method for intrusion detection. *J. Syst. Softw.*, 33(1):101–108, Apr. 1996. ISSN 0164-1212. doi: 10.1016/0164-1212(95)00106-9. URL `http://dx.doi.org/10.1016/0164-1212(95)00106-9`.

[50] Lawrence Berkeley National Laboratory. Bro Intrusion Detection System. `http://www.bro-ids.org/`, 2011. Accessed on 02 January 2013.

[51] G. Liepins and H. Vaccaro. Anomaly detection purpose and framework. In *Proceedings of the 12th National Computer Security Conference*, pages 495–504, Baltimore, MD, USA, 1989.

[52] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 12 –26 vol.2, 2000. doi: 10.1109/DISCEX.2000.821506.

[53] P. Loshin. *TCP/IP Clearly Explained.* Morgan Kaufmann, 1999.

[54] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, P. G. Neumann, H. Javitz, A. Valdes, and T. D. Garvey. A real time intrusion detection expert system (IDES). SRI technical report, Defense Advanced Research Projects Agency, Department of US Defense, 1992.

[55] M. Manzano. *Introduction to many-sorted logic.* John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0-471-93485-2.

[56] G. Marcus. *Firewalls.* McGraw-Hill, 2000.

[57] Microsoft. Microsoft Security Bulletin MS01-033 Unchecked Buffer in Index Server ISAPI Extension Could Enable Web Server Compromise. `http://technet.microsoft.com/en-us/security/bulletin/ms01-033`, 2001. Accessed on 02 January 2013.

[58] Z. Miller, W. Deitrick, and W. Hu. Anomalous network packet detection using data stream mining. *J. Information Security*, 2(4):158–168, 2011.

[59] MIT Lincoln Laboratory. DARPA Intrusion Detection Data Sets. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`, 1999. Accessed on 02 January 2013.

[60] MIT Lincoln Laboratory. Intrusion Detection Attacks Database. `http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/docs/attackDB.html`, 1999. Accessed on 02 January 2013.

[61] M. Moorthy and S. Sathiyabama. Article: A hybrid data mining based intrusion detection system for wireless local area networks. *International Journal of Computer Applications*, 49(10):19–28, July 2012. Published by Foundation of Computer Science, New York, USA.

[62] P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, Madrid, Spain, 2004.

[63] S. Navathe and R. Ahmed. Temporal extensions to the relational model and SQL. In *Temporal Databases: Theory, Design, and Implementation*, pages 92–109. Benjamin/Cummings Publishing Company, 1993.

[64] Network Working Group, IETF Org. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. `http://www.ietf.org/rfc/rfc2267.txt`, 1998. Accessed on 02 January 2013.

[65] P. G. Neumann and P. A. Porras. Experience with EMERALD to date. In *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring - Volume 1*, ID'99, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267880.1267888`.

[66] NIST. Vulnerability Summary for CVE-1999-0153. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-1999-0153`, 1997. Accessed on 02 January 2013.

[67] Official LSV Web Site. ORCHIDS: Real-time event analysis and temporal correlation for intrusion detection in informations systems. http://www.streambase.com/products/streambasecep, 2012. Accessed on 02 January 2013.

[68] OISF Open Information System Foundation. SURICATA. `http://www.openinfosecfoundation.org/`, 2012. Accessed on 02 January 2013.

[69] J. Olivain and J. Goubault-Larrecq. The ORCHIDS intrusion detection tool. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, 2005.

[70] OpenESP. What is the IEP SE? `http://wiki.open-esb.java.net/Wiki.jsp?page=IEPSE`, 2009. Accessed on 02 January 2013.

[71] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006. ISBN 1597490733.

[72] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007. ISBN 0978739256.

[73] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999. URL `http://www.icir.org/vern/papers/bro-CN99.pdf`.

[74] J. Peppard. *IT Strategy for Business.* Pitman Publishing, 1993.

[75] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[76] K. Price. Introduction to Intrusion Detection. `http://www.cerias.purdue.edu/about/history/coast_resources/idcontent/introduction.html`, 1999. Accessed on 02 January 2013.

[77] Progress Software. Data streams. `http://www.progress.com/en/data-streams.html`, 2012. Accessed on 02 January 2013.

[78] R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pages 191–233, 1984.

[79] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 220–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=872752.873518`.

[80] J. Ryan, M. Lin, and R. Miikkulainen. Intrusion detection with neural networks. In *Proceedings of the AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 72–77, Providence, RI, USA, 1997.

[81] N. Sarda. Extensions to SQL for historical databases. *IEEE Transactions on Knowledge and Data Engineering*, 2:220–230, 1990. ISSN 1041-4347. doi: http://doi.ieeecomputersociety.org/10.1109/69.54721.

[82] K. Scarfore and P. Mell. Guide to intrusion detection and prevention systems (IDPS). Special Publication 800-94, National Institute of Standards and Technology (NIST), Feb. 2007.

[83] SecurityFocus. Windows Server 2003 and XP SP2 LAND attack vulnerability. `http://www.securityfocus.com/archive/1/392354`, 2005. Accessed on 02 January 2013.

[84] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *In ICDE*, pages 232–239, 1995.

[85] S. E. Smaha. Haystack: An intrusion detection system. In *Proceedings of the IEEE Fourth Aerospace Computer Security Applications Conference*, 1988.

[86] R. Snodgrass and I. Ahn. Temporal databases. *Computer*, 19(9):35–42, Sept. 1986. ISSN 0018-9162. doi: 10.1109/MC.1986.1663327. URL `http://dx.doi.org/10.1109/MC.1986.1663327`.

[87] Sourcefire. SNORT. `http://www.snort.org/`, 2010. Accessed on 02 January 2013.

[88] Sourcefire. Download Snort Rules. http://www.snort.org/snort-rules/, 2012. Accessed on 02 January 2013.

[89] J. H. Spring. *Reflexes: programming abstractions for highly responsive computing in Java*. PhD thesis, IC, Lausanne, 2008. URL http://library.epfl.ch/theses/?nr=4228.

[90] W. Stallings. *Network Security Essentials: Applications and Standards*. Prentice Hall, Upper Saddle River, NJ, 2000.

[91] StreamBase Systems. StreamBase CEP: Complex Event Processing Platform. http://www.streambase.com/products/streambasecep, 2012. Accessed on 02 January 2013.

[92] StreamBase Systems. StreamBase Server. http://www.streambase.com/products-StreamBaseServer.htm, 2012. Accessed on 02 January 2013.

[93] StreamBase Systems. Product Documentation. http://www.streambase.com/support/product-documentation, 2012. Accessed on 02 January 2013.

[94] StreamBase Systems. StreamSQL. http://www.streambase.com/products/streambasecep/streamsql, 2012. Accessed on 02 January 2013.

[95] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB*, page 594. Morgan Kaufmann, 1996. ISBN 1-55860-382-4.

[96] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 13–24, Berkeley, CA, USA, 1998. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268256.1268258.

[97] Sybase Incorporation. Coral8. http://www.sybase.com/products/archivedproducts/coral8, 2012. Accessed on 02 January 2013.

[98] The TCPDUMP Group. TCPDUMP and LIBPCAP. http://www.tcpdump.org/, 2010. Accessed on 02 January 2013.

[99] TIBCO Software Inc. Introducing TIBCO BusinessEvents™3.0. http://www.tibco.com/products/business-optimization/complex-event-processing/businessevents/default.jsp, 2012.

[100] J. Touch, M. Kojo, E. Lear, A. Mankin, K. Ono, M. Stiemerling, and L. Eggert. Request for Comments: 6335. https://tools.ietf.org/html/rfc6335, 2011. Accessed on 02 January 2013.

[101] J. Touch, M. Kojo, E. Lear, A. Mankin, K. Ono, M. Stiemerling, and L. Eggert. Service Name and Transport Protocol Port Number Registry. `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt`, 2012. Accessed on 02 January 2013.

[102] TRAC. Welcome to TCPREPLAY. `http://tcpreplay.synfin.net/`, 2010. Accessed on 02 January 2013.

[103] R. Trost. *Practical Intrusion Analysis: Prevention and Detection for the Twenty-First Century.* Pearson Education, 2009. ISBN 9780321591883. URL `http://books.google.co.uk/books?id=3y2fhCaJJA0C`.

[104] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. In *Proceedings of the sixteenth international conference on Very large databases*, pages 13–23, San Francisco, CA, USA, 1990. Morgan Kaufmann Publish Inc. ISBN 0-55860-149-X. URL `http://dl.acm.org/citation.cfm?id=94362.94390`.

[105] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. doi: 10.1007/978-3-540-87403-4_7. URL `http://dx.doi.org/10.1007/978-3-540-87403-4_7`.

[106] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Midea: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046741. URL `http://doi.acm.org/10.1145/2046707.2046741`.

[107] G. Vigna and R. A. Kemmerer. A network-based intrusion detection approach. *Journal of Computer Security*, 7:37–71, 1999.

[108] G. Wideman. ST condensed – Templates and expressions. `http://www.antlr.org/wiki/display/ST/ST+condensed+--+Templates+and+expressions`, 2009. Accessed on 02 January 2013.

Appendix A

ANTLR Grammar and String Template Group Files

A.1 Description of The Grammar File Structure

```
grammar <grammar file name>;
<optionsSpec>
<tokensSpec>
<attributeScopes>
<actions>
/** For multi line comments*/
// for single line comment

rule1 : ... | ... | ... ;
rule2 : ... | ... | ... ;
...
```

The first line of the grammar file specifies the name of the file as stored in the file system. The options section specify the options set to be used for processing this grammar file. The options syntax is:

```
options {
name1 = value1;
name2 = value2;
...
}
```

For instance, if we set language = JAVA then this will set the target language to be JAVA which will cause ANTLR to generate the lexer and parser analysers in JAVA.

The tokens section allow to assign aliases for string literals or allow to introduce or define imaginary or dummy token. The syntax for the tokens section is:

```
tokens {
token-name1 ;
token-name2 = 'string-literal';
```

```
...
}
```

The attributes section is for defining attributes at each rule and these attributes can pass information between chained rules or deeply nested rules. This is very useful in writing actions that depends on attributes that come from different rules. The syntax for the attributes section is:

```
scope name {
type1 attribute-name1;
type2 attribute-name2;
...
}
```

The actions section allows you to add snippets of code that you write in the target language and embed in your grammar. ANTLR generates a method for each rule in a grammar. The methods are wrapped in a class definition for object-oriented target languages. ANTLR provides named actions so you can insert variables and instance methods into the generated class definition. The syntax is as follows:

```
@action-name { ... }
@action-scope-name::action-name { ... }  // scope can be for
                                                 // parser or lexer
....
```

The named actions has a place in the grammar file and ANTLR will inserts them into the generated recognizer according to their positions relative to the surrounding grammar elements. Some of the common named actions are as follows:

- @header the code here will appear before the class definition (e.g., import statement and package definition).
- @members this section contains codes that are placed outside rules and globally defines instance variables and methods.
- @init is placed before a rule in the grammar file. Its code is executed at the beginning of a matched rule.
- @after is placed before a rule in the grammar file and executed after the rule is matched.

Finally, in the grammar file rules must be specified at the end of the grammar files. These rules can be a lexer rule or a parser rule. Lexer rule start with capital letter and parser rule start with small letter. A simple example for the lexer and parser rules is:

```
decl:  type ID (',' ID)* ; // e.g., "int a", "int a,b"

type:  'int' | 'float'; // match either an int or float keyword

ID :   'a'..'z'+ ; // match one ore more small letters from a to z
```

In the above example, decl and type are parser rules and ID is a lexer rule.

A.2 *TeStID* Grammar File

```
grammar TeStID;
options {
output = template;
language=Java;
}
tokens {
        PLUS            = '+' ;
        MINUS           = '-' ;
        MULT            = '*' ;
        DIV             = '/' ;
        PREDSYM         = 'P';
        AND             = '&';
        OR              = '|';
        ALWAYS          = 'G';
        ALWAYSP         = 'H';
        EVENTUALLY      = 'F';
        NEG             = '~';
        EQUAL           = '=';
        NOTEQUAL        = '<>';
        GT              = '>';
        GE              = '>=';
        LT              = '<';
        LE              = '<=';
        EXIST           = 'E';
        REOCCUR         = 'R';
}
@header {
import org.antlr.stringtemplate.*;
import java.util.ListIterator;
import java.util.List;
import java.lang.String;
}
@members {

public String checkconsistence(List varterms,List vpindx,int formulano) {
String error=null;
Boolean found = false;
int termindex=0;

for (int i=0; i < vpindx.size(); i++) {
 Integer pp = (Integer)vpindx.get(i);
 if (pp == 2) {
  termindex = i;
  found = false;
  for (int j=0; j < vpindx.size(); j++) {
```

```
    Integer p = (Integer)vpindx.get(j);
    if (p == 1)
    if (varterms.get(i).equals(varterms.get(j)))
      found = true;
  }
  if (!found) {
      error = varterms.get(termindex) + " variable is not defined previously
              in formula number " + formulano;
      return error;}
  }
}   //for i
return error;
} //proc

public String checkpredicate(List preds, List terms) {
 String error="Inconsistence number of terms in predicates";

if (terms.size() \% preds.size() == 0)
 return null;
else
 return error;
}

 public List wcond(int formulano,List arity,List filter,List constant){
  Object currentfilter;
  String wcondition;
  List conditions = new ArrayList();
  String strconst = null;

  if (filter.size() >= 1) {
    currentfilter = filter.get(0);
    strconst = constant.get(0).toString();
    if (constant.get(0).equals( "0")) strconst = "false";
    if (constant.get(0).equals("1")) strconst = "true";
      wcondition = "where x"+arity.get(0)+" = "+ strconst;
      for(int i = 1; i < filter.size(); i++) {
      if (filter.get(i).equals(currentfilter)) {
      strconst = constant.get(i).toString();
      if (constant.get(i).equals( "0")) strconst = "false";
      if (constant.get(i).equals("1")) strconst = "true";
      wcondition = wcondition + " and " + "x"+arity.get(i)+" = "+ strconst;
      } else {
        wcondition = wcondition + " INTO out__Filter" + formulano +
                     "_" + currentfilter;
        conditions.add(wcondition);
        currentfilter = filter.get(i);
        strconst = constant.get(i).toString();
```

```
        if (constant.get(i).equals( "0")) strconst = "false";
        if (constant.get(i).equals("1")) strconst = "true";
        wcondition = "where x" + arity.get(i)+ " = " + strconst.get(i);
      }
    }
    wcondition = wcondition + " INTO out__Filter" + formulano +
                 "_"+ currentfilter;
    conditions.add(wcondition);
    return conditions;
  }
  return null;
}
public String select(int cat,int formulano,int patternno,List<Integer> window,
List vars,List vindx,List vpindx,List Neg)){
 String s = "       \n";
 List prefix = new ArrayList();
 String prefix1 = "output1";
 String prefix2 = "output2";
 String tmpprefix = null;

 for(int k = 1; k <= patternno; k++) {
 if (k == 1) {
   prefix.add("output1");
   prefix.add("output2");
   String cond = "WHERE ";

      if (patternno !=  1 ||  cat == 4)
        s =   s + "CREATE STREAM out__Pattern" + formulano + "_" + k +
            "; \n SELECT ";
      else
         s =   s + "SELECT ";

   for (int l =0; l < prefix.size(); l++) {
     for (int j = 1; j < 13; j++) {
        s = s + prefix.get(l) + "." + "x" + j + " AS " + prefix.get(l) +
            "_x" + j +"," ;
     }
   }
   s = s.substring(1,s.length() - 1) + "\n"; // remove last comma
   s = s + "FROM PATTERN (" + "out__Filter" + formulano + "_" + k + " AS "
       + prefix.get(0) + " THEN" + Neg.get(k-1) + "out__Filter" +
       formulano + "_" + (k+1) + " AS " + prefix.get(1) + ")";
   Integer w = (Integer)window.get(0);
   s = s + " WITHIN " + w + " TIME \n";

          for (int i = 0; i < vpindx.size(); i = i + 1) {
             Integer p = (Integer)vpindx.get(i);
```

```
        if (p == 1) {
          for (int j = 0;  j < vpindx.size(); j = j + 1) {
            Integer pp = (Integer)vpindx.get(j);
            if (pp == 2)
            if (vars.get(j).equals(vars.get(i)))
            if (!cond.toString().equals("WHERE "))
              cond = cond + " AND " + prefix.get(1)  + "." + "x" +
                     vindx.get(j) + " = " + prefix.get(0) + "." + "x"
                     + vindx.get(i);
           else
             cond = cond +  prefix.get(1) + "." + "x" + vindx.get(j) +
                    " = " + prefix.get(0) + "." + "x" + vindx.get(i);
          }
        }
      }
    if (patternno == 1 & cat ==2)
      s = s + cond + "\n"+ " INTO OutputStream" + formulano + "; \n \n";
    else
      s = s + cond + "\n"+ " INTO out__Pattern" + formulano + "_" + k +
              "; \n \n";

   }  //if pattern (k)  = 1 end here
   else {
     prefix.add("output" +(k+1));
     for (int m = 0; m < k; m ++) {
       String temp = prefix.get(m).toString();
       prefix.remove(m);
       temp = temp.replace('.' , '_');
       prefix.add(m, "pattern" + (k-1) + "."+ temp);
     }
     if (patternno != k || cat==4)
        s = s + "CREATE STREAM out__Pattern" + formulano + "_" + k +
            "; \n SELECT ";
      else
         s =   s + "SELECT ";

      for (int l =0; l < prefix.size(); l++) {
      if (l == prefix.size() - 1) {
          for (int j = 1; j < 13; j++) {
          s = s + prefix.get(l) + "." + "x" + j + " AS " +prefix.get(l) +
              "_x" + j +"," ;
          }
       }
      else {
         for (int n = 1; n < 13; n++) {
           tmpprefix = prefix.get(l).toString();
           s = s + prefix.get(l) + "_" + "x" + n + " AS " +
```

```
                    tmpprefix.replace('.', '_')  + "_x" + n +"," ;
              }
            }
           }
           s = s.substring(1,s.length() - 1) + "\n"; // remove last comma
           s = s + "FROM PATTERN (" + "out__Pattern" + formulano + "_" +
               (k-1) + " AS " + "pattern" + (k-1) + " THEN" + Neg.get(k-1) +
               "out__Filter" + formulano + "_" +  (k+1) + " AS "
               + prefix.get(k) + ")";
           Integer w = (Integer)window.get(k-1);
           s = s + " WITHIN " + w + " TIME \n";
           String cond = "WHERE ";
            for (int i = 0; i < vpindx.size(); i = i + 1) {
               Integer p = (Integer)vpindx.get(i);
               if (p == k) {
                 for (int j = 0;  j < vpindx.size(); j = j + 1) {
                   Integer pp = (Integer)vpindx.get(j);
                   if (pp == k+1)
                    if (vars.get(j).equals(vars.get(i)))
                     if (!cond.toString().equals("WHERE "))
                       cond = cond + " AND " + prefix.get(1)  + "_" + "x" +
                              vindx.get(j) + " = " + prefix.get(0) + "_" +
                              "x" + vindx.get(i);
                     else
                       cond = cond +  prefix.get(1) + "_" + "x" +
                              vindx.get(j) + " = " + prefix.get(0)
                              + "_" + "x" + vindx.get(i);
                  }
                }
              }
               if (patternno == k &&  cat == 2)
                  s = s + cond + "\n"+ " INTO OutputStream" +
                      formulano + "; \n \n";
                else
                  s = s + cond + "\n"+ " INTO out__Pattern" +
                      formulano + "_" + k + "; \n \n";
        } // end last else
 }
 return s;
}
public String items(List varterms,List vlindx,List vpindx) {
String List= "";

for (int i=0; i < vpindx.size(); i++) {
  Integer p = (Integer)vpindx.get(i);
  if (p == 1) {
    for (int j=i+1; j < vpindx.size(); j++)
```

```
    if (varterms.get(i).equals(varterms.get(j))) {
      List = List + varterms.get(i);
      Integer pos = (Integer)vlindx.get(i);
      switch(pos) {
      case 1:
        List=List+" string" + ", \n";
        break;
      case 2:
       List=List+" int"  + ", \n";
        break;
      case 3:
        List=List+" string" + ", \n";
        break;
      case 4:
        List=List+" int" + ", \n";
        break;
      }
    }
  }
}
List = List.substring(0,List.length() - 3) + "\n) \n";
return List;
}
public List skey(List varterms,List vlindx,List vpindx) {
List keys= new ArrayList();

for (int i=0; i < vpindx.size(); i++) {
  Integer p = (Integer)vpindx.get(i);
  if (p == 1) {
    for (int j=i+1; j < vpindx.size(); j++)
    if (varterms.get(i).equals(varterms.get(j))) {
      keys.add(varterms.get(i));
    }
  }
}
return keys;
}
public String regexpmatch(String regexp) {
String regexpf = "";
regexpf = "regexmatch" + "(\\" + regexp.substring(2,regexp.length()-2) + "\\" +
          regexp.substring(regexp.length()-2,regexp.length()-1) + ", x12)";
                                                             // adding slash for shell
return regexpf;
}
public String preproc(List term1, List term2, List term3, List term4) {
String filter = "allp";
for (int i=0; i < term1.size(); i++) {
```

```
if (term1.get(i).equals("x2")) {
if ((term2.get(i).equals("20")) || (term2.get(i).equals("53")) ||
   (term2.get(i).equals("0"))) {
  filter = "src" + term2.get(i);
  return filter;}
else if  ((term2.get(i).equals("80")) || (term2.get(i).equals("8080")) ||
         (term2.get(i).equals("8000")) || (term2.get(i).equals("8001"))) {
  filter = "srchttp";
  return filter;}
}
if (term1.get(i).equals("x4")) {
if ((term2.get(i).equals("21")) || (term2.get(i).equals("53")) ||
(term2.get(i).equals("79")) || (term2.get(i).equals("1080")) ||
(term2.get(i).equals("15104")) || (term2.get(i).equals("161")) ||
(term2.get(i).equals("139")) || (term2.get(i).equals("162")) ||
(term2.get(i).equals("3128")) || (term2.get(i).equals("705")) ||
(term2.get(i).equals("143")) || (term2.get(i).equals("25"))) {
    filter = "dst" + term2.get(i);
    return filter;}
else if  ((term2.get(i).equals("80")) || (term2.get(i).equals("8080")) ||
(term2.get(i).equals("8000")) || (term2.get(i).equals("8001"))) {
  filter = "dsthttp";
  return filter;}
}
}
for (int i=0; i < term3.size(); i++) {

if (term3.get(i).equals("x2")) {
  if ((term4.get(i).equals("20")) || (term4.get(i).equals("53")) ||
     (term4.get(i).equals("0"))){
       filter = "src" + term4.get(i);
       return filter;}
else if  ((term4.get(i).equals("80")) || (term4.get(i).equals("8080")) ||
  (term4.get(i).equals("8000")) || (term4.get(i).equals("8001"))) {
    filter = "srchttp";
    return filter;}
}
if (term3.get(i).equals("x4")) {
if ((term4.get(i).equals("21")) || (term4.get(i).equals("53")) ||
(term4.get(i).equals("79")) || (term4.get(i).equals("1080")) ||
(term4.get(i).equals("15104")) || (term4.get(i).equals("161")) ||
(term4.get(i).equals("139")) || (term4.get(i).equals("162")) ||
(term4.get(i).equals("3128")) || (term4.get(i).equals("705")) ||
(term4.get(i).equals("143")) || (term4.get(i).equals("25"))) {
  filter = "dst" + term4.get(i);
  return filter;}
```

```
else if  ((term4.get(i).equals("80")) || (term4.get(i).equals("8080")) ||
(term4.get(i).equals("8000")) || (term4.get(i).equals("8001"))) {
  filter = "dsthttp";
  return filter;}
}
}
return filter;
}
public Boolean Allp(String filter) {
if (filter == "allp")
  return true;
else
  return false;
}
public Boolean Src(String filter) {
if (filter.substring(0,3).equals("src"))
  return true;
else
  return false;
}
public Boolean Dst(String filter) {
if (filter.substring(0,3).equals("dst"))
   return true;
else
   return false;
}
}
/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/
prog
scope {
  List formulae;
  List sfilter;
  List constlist;
  List lindx;
  List clindx;
  List pindx;
  int formulaindex;
  List vpindx;
  List attackid;
  Boolean isFormula1;
}
@init {
  $prog::formulae = new ArrayList();
  $prog::lindx = new ArrayList();
  $prog::formulaindex = 1;
```

```
  $prog::attackid = new ArrayList();
  $prog::isFormula1 = false;
}
    : formulaSeq+ -> prog(formulae={$prog::formulae},
                        isFormula1={$prog::isFormula1})
    ;
formulaSeq
scope {
int sfilterindx;
List wc;
String select;
int patternindex;
List window;
List size;
List varterms;
List vlindx;
String table;
String regexp;
List secondary;
List rwindow;
String error;
List Negation;
}
@init {
  $prog::sfilter = new ArrayList();
  $formulaSeq::sfilterindx = 1;
  $prog::clindx = new ArrayList();
  $prog::constlist = new ArrayList();
  $prog::pindx = new ArrayList();
  $prog::vpindx = new ArrayList();
  $formulaSeq::wc = new ArrayList();
  $formulaSeq::select = null;
  $formulaSeq::patternindex = 0;
  $formulaSeq::window = new ArrayList();
  $formulaSeq::varterms = new ArrayList();
  $formulaSeq::vlindx = new ArrayList();
  $prog::pindx = new ArrayList();
  $formulaSeq::table = null;
  $formulaSeq::regexp = null;
  $formulaSeq::secondary = new ArrayList();
  $formulaSeq::size = new ArrayList();
  $formulaSeq::rwindow = new ArrayList();
  $formulaSeq::error=null;
  $formulaSeq::Negation = new ArrayList();
}
    :  '#' 'sid-' NUMBER {$prog::attackid.add("sid-"+$NUMBER.text);} NEWLINE
       formula NEWLINE {$prog::formulae.add($formula.st);}
```

```
       {$prog::formulaindex++;}
    |  NEWLINE
    ;
formula
scope {
List terml1;
List terml2;
List terml3;
List terml4;
List equality;
List operator;
List conjunction;
List conjunction2;
int termindx;
Boolean isVar;
Boolean isFunc;
List isVarL2;
List isVarL5;
List isVarL6;
List isFuncL2;
List isFuncL5;
List isFuncL6;
Boolean isArth;
Boolean isnotArth;
Boolean F;
List isArthL;
String Filler;
Boolean isF1;
Boolean isF2;
Boolean isF3;
List exvar;
int cat;
String filter;
Boolean isAllp;
Boolean isSrc;
Boolean isDst;
}
@init {
  $formula::equality=new ArrayList();
  $formula::operator=new ArrayList();
  $formula::conjunction=new ArrayList();
  $formula::conjunction2=new ArrayList();
  $formula::isVarL2 = new ArrayList();
  $formula::isVarL5 = new ArrayList();
  $formula::isVarL6 = new ArrayList();
  $formula::isFuncL2 = new ArrayList();
  $formula::isFuncL5 = new ArrayList();
```

```
  $formula::isFuncL6 = new ArrayList();
  $formula::isArthL = new ArrayList();
  $formula::isArth = true;
  $formula::isnotArth = false;
  $formula::terml1 = new ArrayList();
  $formula::terml2 = new ArrayList();
  $formula::terml3 = new ArrayList();
  $formula::terml4 = new ArrayList();
  $formula::Filler = "!!";
  $formula::F = false;
  $formula::isF1=false;
  $formula::isF2=false;
  $formula::isF3=false;
  $formula::cat=0;
  $formula::exvar = new ArrayList();
  $formula::isAllp=false;
  $formula::isSrc=false;
  $formula::isDst=false;
}
@after {
$prog::attackid=new ArrayList();
}
    : formula1  -> {$formula1.st}
    | formula2  ->  {$formula2.st}
    | formula3  -> {$formula3.st}
    | formula4  -> {$formula4.st}
    ;
exQ : '(' EXIST var (','  var)* ')';

atomic_formula
@init {
  $formula::termindx = 1;

}
    : PREDSYM '(' p+=term {$prog::lindx.add($formula::termindx++);}
      (',' p+=term {$prog::lindx.add($formula::termindx++);} )*')'
    ;
formula1
    :  exQ '[' exQ  atomic_formula
       {$prog::sfilter.add($formulaSeq::sfilterindx++);} AND
       {$prog::sfilter.add($formulaSeq::sfilterindx);}
       {$formula::conjunction2.add($formula::F);} (disjunction (AND
       {$formula::conjunction2.add("and");} disjunction)* ) ']'
       {$formulaSeq::error=checkconsistence($formulaSeq::varterms,
       $prog::vpindx,$prog::formulaindex);}
       {$formula::filter=preproc($formula::terml1,$formula::terml2,
       $formula::terml3,$formula::terml4);}
```

```
        {$formula::isAllp = Allp($formula::filter);}
        {$formula::isSrc = Src($formula::filter);}
        {$formula::isDst = Dst($formula::filter);}
        {$prog::isFormula1=true;}
         -> category1(term1={$formula::terml1}, term2={$formula::terml2},
            equality={$formula::equality}, operator={$formula::operator},
            isVar2={$formula::isVarL2}, isVar5={$formula::isVarL5},
            isVar6={$formula::isVarL6}, formulaindex={$prog::formulaindex},
            term5={$formula::terml3}, term6={$formula::terml4},
            isArth={$formula::isArthL},regexp={$formulaSeq::regexp},
            isFunc2={$formula::isFuncL2}, isFunc5={$formula::isFuncL5},
            isFunc6={$formula::isFuncL6}, conj={$formula::conjunction},
            attackid={$prog::attackid},conj2={$formula::conjunction2},
            filter={$formula::filter},
            isAllp={$formula::isAllp},isSrc={$formula::isSrc},
            isDst={$formula::isDst},error={$formulaSeq::error})
            ;
formula2
    :  exQ? '[' exQ atomic_formula
       {$prog::sfilter.add($formulaSeq::sfilterindx++);} AND
       {$formulaSeq::Negation.add(" ");} (NEG
       {$formulaSeq::Negation.remove($formulaSeq::patternindex);}
       {$formulaSeq::Negation.add(" NOT ");})?
       EVENTUALLY '[' NUMBER',' N2=NUMBER
       {$formulaSeq::window.add(Integer.parseInt($N2.text));} ']'
       {$formulaSeq::patternindex++;} (formula2 |exQ atomic_formula
       {$prog::sfilter.add($formulaSeq::sfilterindx++);}) ']'
       {$formulaSeq::error=checkpredicate( $prog::sfilter,$prog::lindx);
       $formula::cat=2;
       $formulaSeq::wc = wcond($prog::formulaindex,$prog::clindx,$prog::pindx,
       $prog::constlist);
       $formulaSeq::select = select($formula::cat,$prog::formulaindex,
       $formulaSeq::patternindex, $formulaSeq::window,$formulaSeq::varterms,
       $formulaSeq::vlindx,$prog::vpindx,$formulaSeq::Negation);}
         -> category2(sfilter={$prog::sfilter},wc={$formulaSeq::wc},
             formulaindex={$prog::formulaindex},select={$formulaSeq::select},
             error={$formulaSeq::error})
    ;
formula3
    :  exQ '[' exQ atomic_formula
       {$prog::sfilter.add($formulaSeq::sfilterindx++);} AND
       ALWAYSP '[' NUMBER',' N=NUMBER
       {$formulaSeq::window.add(Integer.parseInt($N.text));}']'
       exQ NEG atomic_formula
       {$prog::sfilter.add($formulaSeq::sfilterindx++);}']'
       {$formulaSeq::error=checkpredicate( $prog::sfilter,$prog::lindx);
       $formulaSeq::table = items($formulaSeq::varterms,$formulaSeq::vlindx,
```

```
        $prog::vpindx);
        $formulaSeq::secondary=skey($formulaSeq::varterms,$formulaSeq::vlindx,
        $prog::vpindx);
        $formulaSeq::wc = wcond($prog::formulaindex,$prog::clindx,$prog::pindx,
        $prog::constlist);}
         -> category3(formulaindex={$prog::formulaindex},
            varterms={$formulaSeq::varterms},
            vlindx={$formulaSeq::vlindx},vpindx= {$prog::vpindx},
            items={$formulaSeq::table}, skey={$formulaSeq::secondary},
            sfilter={$prog::sfilter},wc={$formulaSeq::wc},
            w={$formulaSeq::window},error={$formulaSeq::error})
;
formula4
    : ('(' EXIST var1=var {$formula::exvar.add($var1.text);} (',' var2=var
      {$formula::exvar.add($var2.text);})* ')')?
      REOCCUR  ('[' NUMBER',' N=NUMBER
      {$formulaSeq::rwindow.add(Integer.parseInt($N.text) * 1000);}']' '['
      S=NUMBER {$formulaSeq::size.add(Integer.parseInt($S.text));}']')
      (formula1 {$formula::isF1=true;}|formula2 {$formula::isF2=true;}|formula3
      {$formula::isF3=true;})
      {$formula::cat=4;
      $formulaSeq::wc = wcond($prog::formulaindex,
      $prog::clindx,$prog::pindx,$prog::constlist);
      $formulaSeq::secondary=skey($formulaSeq::varterms,
      $formulaSeq::vlindx,$prog::vpindx);
      $formulaSeq::select = select($formula::cat,$prog::formulaindex,
      $formulaSeq::patternindex, $formulaSeq::window,
      $formulaSeq::varterms,$formulaSeq::vlindx,$prog::vpindx,
      $formulaSeq::Negation);
      $formulaSeq::table = items($formulaSeq::varterms,
      $formulaSeq::vlindx,$prog::vpindx);}
        -> category4(sfilter={$prog::sfilter},
           formulaindex={$prog::formulaindex},
           isF1={$formula::isF1},isF2={$formula::isF2},
           isF3={$formula::isF3},term1={$formula::terml1},
           term2={$formula::terml2}, isFunc2={$formula::isFuncL2},
           isFunc5={$formula::isFuncL5},  isFunc6={$formula::isFuncL6},
           conj={$formula::conjunction}, conj2={$formula::conjunction2},
           filter={$formula::filter},isAllp={$formula::isAllp},
           isSrc={$formula::isSrc},isDst={$formula::isDst},
           equality={$formula::equality}, operator={$formula::operator},
           isVar2={$formula::isVarL2}, isVar5={$formula::isVarL5},
           isVar6={$formula::isVarL6},  term5={$formula::terml3},
           term6={$formula::terml4},isArth={$formula::isArthL},
           w={$formulaSeq::rwindow}, s={$formulaSeq::size},
           exvar={$formula::exvar},wc={$formulaSeq::wc},
           select={$formulaSeq::select},p={$formulaSeq::patternindex},
```

```
            items={$formulaSeq::table},
            skey={$formulaSeq::secondary},error={$formulaSeq::error})
    ;
disjunction

    : ( '(' (term1=term {$formula::terml1.add($term1.text);} (E1=EQUAL
      {$formula::equality.add($E1.text);}|G1=GE
      {$formula::equality.add($G1.text);}
      |G2=GT {$formula::equality.add($G2.text);}|L1=LT
      {$formula::equality.add($L1.text);}|L2=LE
      {$formula::equality.add($L2.text);}|N1=NOTEQUAL
      {$formula::equality.add($N1.text);})
      (term2=term {$formula::terml2.add($term2.text);
      $formula::isVarL2.add($formula::isVar);
      $formula::isFuncL2.add($formula::isFunc);
      $formula::isArthL.add($formula::isnotArth);
      $formula::terml3.add($formula::Filler);
      $formula::terml4.add($formula::Filler);
      $formula::isVarL5.add($formula::F);
      $formula::isVarL6.add($formula::F);
      $formula::isFuncL5.add($formula::F);
      $formula::isFuncL6.add($formula::F);
      $formula::operator.add($formula::Filler);}
      |'(' term5=term {$formula::terml3.add($term5.text);
      $formula::isVarL5.add($formula::isVar);
      $formula::isFuncL5.add($formula::F);;} (Pl1=PLUS
      {$formula::operator.add($Pl1.text);}|Mi1=MINUS
      {$formula::operator.add($Mi1.text);}
      |Mu1=MULT {$formula::operator.add($Mu1.text);}
      |Di1=DIV {$formula::operator.add($Di1.text);}) term6=term
      {$formula::terml4.add($term6.text);
      $formula::isVarL6.add($formula::isVar);
      $formula::isFuncL6.add($formula::F);
      $formula::terml2.add($formula::Filler);
      $formula::isVarL2.add($formula::F);
      $formula::isFuncL2.add($formula::F);} ')'
      {$formula::isArthL.add($formula::isArth);})
      (or=OR {$formula::conjunction2.add($formula::F);}
      {$formula::conjunction.add("or");} term3=term
      {$formula::terml1.add($term3.text);} (E2=EQUAL
      {$formula::equality.add($E2.text);}|G3=GE
      {$formula::equality.add($G3.text);}
      |G4=GT {$formula::equality.add($G4.text);}
      |L3=LT {$formula::equality.add($L3.text);}
      |L4=LE {$formula::equality.add($L4.text);}
      |N2=NOTEQUAL {$formula::equality.add($N2.text);})
      (term4=term {$formula::terml2.add($term4.text);
```

```
      $formula::isVarL2.add($formula::isVar);
      $formula::isFuncL2.add($formula::isFunc);
      $formula::isArthL.add($formula::isnotArth);
      $formula::terml3.add($formula::Filler);
      $formula::terml4.add($formula::Filler);
      $formula::isVarL5.add($formula::F);
      $formula::isVarL6.add($formula::F);
      $formula::isFuncL5.add($formula::F);
      $formula::isFuncL6.add($formula::F);
      $formula::operator.add($formula::Filler);}
      |'(' term5=term {$formula::terml3.add($term5.text);
      $formula::isVarL5.add($formula::isVar);
      $formula::isFuncL5.add($formula::F);} (Pl2=PLUS
      {$formula::operator.add($Pl2.text);}|Mi2=MINUS
      {$formula::operator.add($Mi2.text);}
      |Mu2=MULT {$formula::operator.add($Mu2.text);}
      |Di2=DIV {$formula::operator.add($Di2.text);})
      term6=term {$formula::terml4.add($term6.text);
      $formula::isVarL6.add($formula::isVar);
      $formula::isFuncL6.add($formula::F);
      $formula::terml2.add($formula::Filler);
      $formula::isVarL2.add($formula::F);
      $formula::isFuncL2.add($formula::F);} ')'
      {$formula::isArthL.add($formula::isArth);}))* ))  ')'
    ;
term
    : var   {$formulaSeq::varterms.add($var.text);}
            {$formulaSeq::vlindx.add($formula::termindx);}
            {$prog::vpindx.add($formulaSeq::sfilterindx);}
          -> {new StringTemplate($var.text); $formula::isVar=true;}
    | constant
    | function
    | ('(' term  (PLUS|MINUS|MULT|DIV) term  ')')
    ;
constant
    : (BIGCHAR {$prog::constlist.add($BIGCHAR.text);
      $formula::isVar=false;$formula::isFunc=false;}
    | NUMBER {$prog::constlist.add($NUMBER.text);
      $formula::isVar=false;})+
      {$prog::clindx.add($formula::termindx);}
      {$prog::pindx.add($formulaSeq::sfilterindx);}
    ;
var
    : (SMALLCHAR)(NUMBER)+
    ;
function
    :  'f("' regex '")' {$formulaSeq::regexp = regexpmatch($function.text);}
```

```
       {$formula::isFunc=true;$formula::isVar=false;}
    ;
regex
    : (SMALLCHAR|BIGCHAR|REOCCUR|EXIST|ALWAYSP|PREDSYM|EVENTUALLY|ALWAYS
    | NUMBER|spchar|UNICODE|match)+
    ;
spchar
    : ('.' | '*' | '%' |'^' | '{' |'}' |')' |'(' |'\\' |':' |'<' |'>' |'<>'
      |'~' |'?' |'-' |','|'&'|'=' |'/'  )
    ;
match
    : ('['  '^'? (SMALLCHAR|BIGCHAR|REOCCUR|EXIST|ALWAYSP|PREDSYM|EVENTUALLY
      |ALWAYS|UNICODE)+']')
        ;
UNICODE
    : '\\u0000'   // null
    | '\\u000a'   // LF
    | '\\u0020'   // SP
    | '\\u0025'   // %
    | '\\u0026'   // &
    | '\\u0028'  // (
    | '\\u0029'   // )
    | '\\u002f'  //  / forward slash
    | '\\u002d'  //  - minus
    | '\\u002e'   // . (dot)
    | '\\u0030'   // 0
    | '\\u0031'   // 1
    | '\\u0032'  // 2
    | '\\u0033'   // 3
    | '\\u0034'  //4
    | '\\u0035'  // 5
    | '\\u0038'   // 8
    | '\\u0039' // 9
    | '\\u003a'   // :
    | '\\u003c'   // <
    | '\\u003d'  //  =
    | '\\u003e'  // >
    | '\\u0041'   //A
    | '\\u0042'   // B
    | '\\u0043'   // C
    | '\\u0044'   // D
    | '\\u0045'   //E
    | '\\u0046'   // F
    | '\\u0047'   // G
    | '\\u0048'   // H
    | '\\u0049'   // I
    | '\\u004a'   // J
```

```
    | '\\u004b'   // K
    | '\\u004c'   // L
    | '\\u004d'   // M
    | '\\u004e'   // N
    | '\\u004f'   // O
    | '\\u0050'   // P
    | '\\u0051'   // Q
    | '\\u0052'  // R
    | '\\u0053'   // S
    | '\\u0054'   // T
    | '\\u0055'   // U
    | '\\u0056'  // V
    | '\\u0057'   // W
    | '\\u0058'   // X
    | '\\u0059'   // Y
    | '\\u005a'  // Z
    | '\\u005c'   // \ (backslash)
    | '\\u0061'   // a
    | '\\u0062'   // b
    | '\\u0063'   // c
    | '\\u0064'   // d
    | '\\u0065'   // e
    | '\\u0066'   // f
    | '\\u0067'   // g
    | '\\u0068'   // h
    | '\\u0069'   // i
    | '\\u006a'   // j
    | '\\u006b'   // k
    | '\\u006c'   // l
    | '\\u006d'   // m
    | '\\u006e'   // n
    | '\\u006f'   // o
    | '\\u0070'  // p
    | '\\u0071'  //q
    |  '\\u0072' // r
    | '\\u0073'  // s
    | '\\u0074'  // t
    | '\\u0075'  // u
    | '\\u0076'  // v
    | '\\u0077'  // w
    | '\\u0078'  // x
    | '\\u0079'  // y
    | '\\u007a'  // z
    | '\\u00fc'   //
    ;
BIGCHAR
    : ('A'..'Z')+
```

```
    ;
SMALLCHAR
    : ('a'..'z')
    ;
WHITESPACE
    : ( '\t' | ' '| '\u000C' )+ { $channel = HIDDEN; }
    ;
NUMBER : DIGIT+;
fragment DIGIT : '0'..'9';
NEWLINE  :'\r'? '\n';
```

A.3 *TeStID* String Template Group File

```
group SB;
prog(formulae,isFormula1) ::= <<
<if (isFormula1)>

echo "CREATE OUTPUT STREAM allStream ;

APPLY JAVA \"TCP_W_Payload\" AS TCP_W_Payload (
schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>\n\
<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
```

```
\n\</schema>\n\"
)
INTO allStream;" > allp.ssql

echo "CREATE STREAM allStream ;

APPLY JAVA \"TCP_W_Payload\" AS TCP_W_Payload (
    schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>\n\
<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO allStream;

CREATE OUTPUT STREAM src20;
CREATE OUTPUT STREAM src53;
CREATE OUTPUT STREAM src0;
CREATE OUTPUT STREAM srchttp;

SELECT * FROM allStream
WHERE x2 = 20 INTO src20
WHERE x2 = 53 INTO src53
WHERE x2 = 0 INTO src0
WHERE x2 = 80 or x2 = 8000 or x2 = 8001 or x2 = 8080 INTO srchttp
```

```
;" > src.ssql

echo "CREATE STREAM allStream;
APPLY JAVA \"TCP_W_Payload\" AS TCP_W_Payload (
    schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>\n\
<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO allStream;
CREATE OUTPUT STREAM dst21;
CREATE OUTPUT STREAM dst53;
CREATE OUTPUT STREAM dst79;
CREATE OUTPUT STREAM dst1080;
CREATE OUTPUT STREAM dst15104;
CREATE OUTPUT STREAM dst161;
CREATE OUTPUT STREAM dst139;
CREATE OUTPUT STREAM dst3128;
CREATE OUTPUT STREAM dst162;
CREATE OUTPUT STREAM dst705;
CREATE OUTPUT STREAM dst143;
CREATE OUTPUT STREAM dst25;
CREATE OUTPUT STREAM dsthttp;
```

```
SELECT * FROM allStream
WHERE x4 = 21 INTO dst21
WHERE x4 = 53 INTO dst53
WHERE x4 = 79 INTO dst79
WHERE x4 = 1080 INTO dst1080
WHERE x4 = 15104 INTO dst15104
WHERE x4 = 161 INTO dst161
WHERE x4 = 139 INTO dst139
WHERE x4 = 3128 INTO dst3128
WHERE x4 = 162 INTO dst162
WHERE x4 = 705 INTO dst705
WHERE x4 = 143 INTO dst143
WHERE x4 = 25 INTO dst25
WHERE x4 = 80 or x4 = 8000 or x4 = 8001 or x4 = 8080 INTO dsthttp
;" > dst.ssql

echo "CREATE STREAM allp;" > main.ssql
echo "APPLY PARALLEL MODULE \"allp.ssql\" AS allpm INTO allStream = allp ;"
\>> main.ssql

echo "CREATE STREAM out_src20 ;
CREATE STREAM out_src53 ;
CREATE STREAM out_src0 ;
CREATE STREAM out_srchttp ;

APPLY PARALLEL MODULE \"src.ssql\" AS srcm
    INTO src20 = out_src20, src53 = out_src53, src0 = out_src0 ,
    srchttp = out_srchttp;

CREATE STREAM out_dst21 ;
CREATE STREAM out_dst53 ;
CREATE STREAM out_dsthttp ;
CREATE STREAM out_dst79 ;
CREATE STREAM out_dst1080 ;
CREATE STREAM out_dst15104 ;
CREATE STREAM out_dst161 ;
CREATE STREAM out_dst139 ;
CREATE STREAM out_dst3128 ;
CREATE STREAM out_dst162 ;
CREATE STREAM out_dst705 ;
CREATE STREAM out_dst143 ;
CREATE STREAM out_dst25 ;

APPLY PARALLEL MODULE \"dst.ssql\" AS dstm
    INTO dst21 = out_dst21, dst53 = out_dst53, dsthttp = out_dsthttp ,
    dst79 = out_dst79, dst1080 = out_dst1080, dst15104 = out_dst15104,
    dst161 = out_dst161, dst139 = out_dst139, dst3128 = out_dst3128,
```

```
    dst162 = out_dst162, dst705 =  out_dst705, dst143 = out_dst143,
    dst25 = out_dst25
    ;" \>> main.ssql
<endif>
<formulae; separator="\n">
>>
category1(term1,term2,equality,operator,isVar2,isVar5,isVar6,formulaindex,
term5,term6,isArth,regexp,isFunc2,isFunc5,isFunc6,conj,attackid,conj2,
filter,isAllp,isSrc,isDst,error) ::= <<

<if (error)>
******** <error> *******
<else>
echo "CREATE INPUT STREAM Filter<formulaindex> (
    x1 string,
    x2 int,
    x3 string,
    x4 int,
    x5 long,
    x6 long,
    x7 boolean,
    x8 boolean,
    x9 boolean,
    x10 boolean,
    x11 boolean,
    x12 string
);" > m<formulaindex>.ssql

echo "CREATE OUTPUT STREAM Filtero<formulaindex> ;" \>> m<formulaindex>.ssql
echo "CREATE STREAM out_Filter<formulaindex> ;" \>> m<formulaindex>.ssql
echo "SELECT * FROM Filter<formulaindex> " \>> m<formulaindex>.ssql
echo "WHERE (<term1,equality,operator,term2,isVar2,isVar5,isVar6,isFunc2,
isFunc5,isFunc6,term5,term6,isArth,conj,conj2:{t1,eq,op,t2,isVar2,isVar5,
isVar6,isFunc2,isFunc5,isFunc6,t5,t6,isArth,conj,conj2|<if(!isArth)>
<if(conj2)>) <conj2> (<else><endif><if(isVar2)><t1><eq><t2> <conj>
<elseif(isFunc2)><regexp> <conj> <else><t1> <eq> <t2> <conj> <endif><else>
<t1> <eq><if(isVar5)><t5> <op> <else> <t5> <op> <endif><if(isVar6)><t6> <conj>
<else><t6><conj><endif><endif>};separator ="">) INTO out_Filter<formulaindex>
;" \>> m<formulaindex>.ssql

echo "SELECT
    out_Filter<formulaindex>.x1,
    out_Filter<formulaindex>.x2,
    out_Filter<formulaindex>.x3,
    out_Filter<formulaindex>.x4,
    out_Filter<formulaindex>.x12,
    \"<attackid>\" AS attack
```

```
FROM   out_Filter<formulaindex>
INTO   Filtero<formulaindex>
;" \>> m<formulaindex>.ssql
echo "CREATE OUTPUT STREAM Output<formulaindex>;" \>> main.ssql

<if (isAllp)>echo "APPLY  USING CONCRETE 5 MODULE \"m<formulaindex>.ssql\" AS
allpm<formulaindex> FROM Filter<formulaindex> = <filter> USING round_robin
INTO Filtero<formulaindex> = Output<formulaindex>;" \>> main.ssql
<elseif (isSrc)>echo "APPLY  USING CONCRETE 5 MODULE \"m<formulaindex>.ssql\"
AS srcm<formulaindex> FROM Filter<formulaindex> = out_<filter> USING
round_robin INTO Filtero<formulaindex> = Output<formulaindex>;" \>> main.ssql
<elseif (isDst)>echo "APPLY  USING CONCRETE 5 MODULE \"m<formulaindex>.ssql\"
AS dstm<formulaindex> FROM Filter<formulaindex> = out_<filter>
USING round_robin INTO Filtero<formulaindex> = Output<formulaindex>;"
\>> main.ssql<endif>
<endif>
>>
category2(sfilter,wc,formulaindex,select,error)::= <<

<if (error)>
******** <error> *******
<else>
echo "CREATE OUTPUT STREAM OutputStream<formulaindex>;

CREATE STREAM out__InputAdapter2_<formulaindex> ;

APPLY JAVA \"TcpSniffer\" AS InputAdapter<formulaindex> (
    schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>
\n\<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
```

```
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO out__InputAdapter2_<formulaindex>;
<sfilter:{CREATE STREAM out__Filter<formulaindex>_<it>}; separator=";\n">;

SELECT * FROM out__InputAdapter2_<formulaindex>
<wc; separator="\n">;

<select>
" > b<formulaindex>.ssql
<endif>
>>

category3(formulaindex,varterms,vlindx,vpindx,items,skey,
          sfilter,wc,w,items,skey,error)::= <<
<if (error)>
******** <error> *******
<else>
echo "CREATE OUTPUT STREAM OutputStream<formulaindex>;
CREATE MEMORY TABLE QueryTable<formulaindex>
(
packetno long,
predicate_time double,
<items>
PRIMARY KEY (packetno) USING BTREE
SECONDARY KEY (predicate_time,<skey; separator=",">) USING HASH;

CREATE STREAM out__InputAdapter3_<formulaindex>;

APPLY JAVA \"TcpSniffer\" AS InputAdapter<formulaindex> (
     schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>
\n\<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
```

```
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO out__InputAdapter3_<formulaindex>;

CREATE STREAM out__Map3_<formulaindex> ;
SELECT
    out__InputAdapter3_<formulaindex>.* AS *,
    to_milliseconds(now()) AS predicate_time
FROM   out__InputAdapter3_<formulaindex>
INTO   out__Map3_<formulaindex>
;

<sfilter:{CREATE STREAM out__Filter<formulaindex>_<it>}; separator=";\n">;

SELECT * FROM out__Map3_<formulaindex>
<wc; separator="\n">;

CREATE STREAM out__Query2_<formulaindex> ;
SELECT out__Filter<formulaindex>_1.x1, out__Filter<formulaindex>_1.x2,
out__Filter<formulaindex>_1.x3, out__Filter<formulaindex>_1.x4,
out__Filter<formulaindex>_1.x5, out__Filter<formulaindex>_1.x6,
out__Filter<formulaindex>_1.x7, out__Filter<formulaindex>_1.x8,
out__Filter<formulaindex>_1.x9, out__Filter<formulaindex>_1.x10,
out__Filter<formulaindex>_1.x11, out__Filter<formulaindex>_1.x12
, out__Filter<formulaindex>_1.predicate_time,
QueryTable<formulaindex>.packetno AS tablepacketno,
QueryTable<formulaindex>.predicate_time AS tablepredicate_time,
<skey:{QueryTable<formulaindex>.<it> AS table<it>}; separator=",">
 FROM out__Filter<formulaindex>_1 OUTER JOIN QueryTable<formulaindex>

WHERE QueryTable<formulaindex>.predicate_time >=
(out__Filter<formulaindex>_1.predicate_time - <w>) and
```

```
QueryTable<formulaindex>.predicate_time
\<= out__Filter<formulaindex>_1.predicate_time and
<skey:{QueryTable<formulaindex>.<it> = out__Filter<formulaindex>_1.<it>};
separator = " and ">
LIMIT 1
INTO out__Query2_<formulaindex>;

DECLARE sequence1id long DEFAULT 0;
 -- seqIdSetter
CREATE STREAM gen__seqIdSetter<formulaindex> (
    packetno long
);

CREATE STREAM out__Sequence<formulaindex>_1 ;
SELECT sequence1id + 1 AS packetno FROM out__Filter<formulaindex>_2 INTO
gen__seqIdSetter<formulaindex>;
SELECT *, sequence1id AS packetno FROM out__Filter<formulaindex>_2 INTO
out__Sequence<formulaindex>_1;

CREATE STREAM out__Filter<formulaindex>_3 ;
SELECT * FROM out__Query2_<formulaindex>
WHERE isnull(tablex1) INTO OutputStream<formulaindex>
WHERE true INTO out__Filter<formulaindex>_3
;

REPLACE INTO QueryTable<formulaindex> (packetno, predicate_time, <skey:{<it>};
separator=","> )
    SELECT packetno, predicate_time, <skey:{<it>}; separator=",">
    FROM out__Sequence<formulaindex>_1
;
DELETE FROM QueryTable<formulaindex>
USING out__Filter<formulaindex>_3
WHERE (QueryTable<formulaindex>.predicate_time \<
out__Filter<formulaindex>_3.predicate_time - <w>) or
(<skey:{QueryTable<formulaindex>.<it> = out__Filter<formulaindex>_3.<it>};
separator=" and ">)
;

UPDATE sequence1id FROM (SELECT * FROM gen__seqIdSetter<formulaindex>);"
> c<formulaindex>.ssql

<endif>
>>
 category4(sfilter,formulaindex,isF1,isF2,isF3,term1, term2, isFunc2, isFunc5,
           isFunc6, conj, conj2, filter,isAllp, isSrc, isDst,equality,
           operator,isVar2, isVar5,isVar6, term5, term6,isArth,w,s,exvar,
           wc,select,p,error)::= <<
```

```
<if(isF1)><formula1()><elseif(isF2)><formula2()><elseif(isF3)><formula3()>
<endif>
>>
formula1() ::= <<

<if (error)>
******** <error> *******
<else>
echo "CREATE INPUT STREAM Filter<formulaindex> (
    x1 string,
    x2 int,
    x3 string,
    x4 int,
    x5 long,
    x6 long,
    x7 boolean,
    x8 boolean,
    x9 boolean,
    x10 boolean,
    x11 boolean,
    x12 string
);" > d<formulaindex>.ssql

echo "CREATE OUTPUT STREAM Filtero<formulaindex> ;" \>> d<formulaindex>.ssql
echo "CREATE STREAM out_Filter<formulaindex> ;" \>> d<formulaindex>.ssql
echo "SELECT * FROM Filter<formulaindex> " \>> d<formulaindex>.ssql
echo "WHERE (<term1,equality,operator,term2,isVar2,isVar5,isVar6,isFunc2,
             isFunc5, isFunc6,term5,term6,isArth,conj,
             conj2:{t1,eq,op,t2,isVar2,isVar5,isVar6,isFunc2,isFunc5,
             isFunc6,t5,t6,isArth,conj,conj2| <if(!isArth)><if(conj2)>)
              <conj2> (<else><endif><if(isVar2)><t1><eq><t2> <conj>
             <elseif(isFunc2)><regexp> <conj> <else><t1> <eq> <t2> <conj>
             <endif><else><t1> <eq><if(isVar5)><t5> <op> <else> <t5>
             <op> <endif><if(isVar6)><t6> <conj><else>
             <t6><conj><endif><endif>};separator ="">)
             INTO out_Filter<formulaindex>
             ;" \>> d<formulaindex>.ssql

echo "CREATE STREAM out_Map<formulaindex> ;
SELECT
    out_Filter<formulaindex>.* AS *,
    to_milliseconds(now()) AS predicate_time
FROM   out_Filter<formulaindex>
INTO   out_Map<formulaindex>
;
CREATE STREAM out_PatternSum_<formulaindex> ;
 -- Calculates how many orders for the submitted category
```

```
CREATE WINDOW sumOfPattern<formulaindex>(SIZE <s> ADVANCE 1 TUPLES);
SELECT
    <if(exvar)><exvar;separator = " , ">,<endif>
    count() AS Numberpackets,
    firstval(predicate_time) AS FirstPatternT,
    lastval(predicate_time) AS LastPatternT,
    firstval(*) AS input_*
FROM out_Map<formulaindex>[sumOfPattern<formulaindex>]
<if(exvar)>GROUP BY <exvar;separator = " , "><endif>
INTO out_PatternSum_<formulaindex>;

SELECT * FROM out_PatternSum_<formulaindex>
WHERE ((LastPatternT - FirstPatternT) \< <w>) INTO Filtero<formulaindex>
;
" \>> d<formulaindex>.ssql
echo "CREATE OUTPUT STREAM Output<formulaindex>;" \>> main.ssql

<if (isAllp)>echo "APPLY MODULE \"d<formulaindex>.ssql\" AS allpm<formulaindex>
FROM Filter<formulaindex> = <filter>
INTO Filtero<formulaindex> = Output<formulaindex>;"
\>> main.ssql
<elseif (isSrc)>echo "APPLY MODULE \"d<formulaindex>.ssql\"
AS srcm<formulaindex>
FROM Filter<formulaindex> = out_<filter> INTO Filtero<formulaindex> =
Output<formulaindex>;" \>> main.ssql
<elseif (isDst)>echo "APPLY MODULE \"d<formulaindex>.ssql\"
AS dstm<formulaindex>
FROM Filter<formulaindex> = out_<filter> INTO Filtero<formulaindex> =
Output<formulaindex>;" \>> main.ssql<endif>
<endif>
>>

formula2() ::= <<

<if (error)>
******** <error> *******
<else>
echo "CREATE OUTPUT STREAM OutputStream<formulaindex>;

CREATE STREAM out_InputAdapter4_<formulaindex> ;

APPLY JAVA \"TcpSniffer\" AS InputAdapter<formulaindex> (
     schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>
\n\<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
```

```
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO out_InputAdapter4_<formulaindex>;
<sfilter:{CREATE STREAM out__Filter<formulaindex>_<it>}; separator=";\n">;

SELECT * FROM out_InputAdapter4_<formulaindex>
<wc; separator="\n">;

<select>

CREATE STREAM out_Map<formulaindex> ;
SELECT
    out__Pattern<formulaindex>_<p>.* AS *,
    to_milliseconds(now()) AS predicate_time
FROM   out__Pattern<formulaindex>_<p>
INTO   out_Map<formulaindex>
;

CREATE STREAM out_PatternSum_<formulaindex> ;
 -- Calculates how many orders for the submitted category
CREATE WINDOW sumOfPattern<formulaindex>(SIZE <s> ADVANCE 1 TUPLES);
SELECT
    <if(exvar)><exvar;separator = " , ">,<endif>
    count() AS Numberpackets,
    firstval(predicate_time) AS FirstPatternT,
    lastval(predicate_time) AS LastPatternT,
```

```
    firstval(*) AS input_*
FROM out_Map<formulaindex>[sumOfPattern<formulaindex>]
<if(exvar)>GROUP BY <exvar;separator = " , "><endif>
INTO out_PatternSum_<formulaindex>;

SELECT * FROM out_PatternSum_<formulaindex>
WHERE ((LastPatternT - FirstPatternT) \< <w>) INTO OutputStream<formulaindex>
;

" > d<formulaindex>.ssql
<endif>
>>
formula3() ::= <<

<if (error)>
******** <error> *******
<else>
echo "CREATE OUTPUT STREAM OutputStream<formulaindex>;
CREATE MEMORY TABLE QueryTable<formulaindex>
(
packetno long,
predicate_time double,
<items>
PRIMARY KEY (packetno) USING BTREE
SECONDARY KEY (predicate_time,<skey; separator=",">) USING HASH;

CREATE STREAM out__InputAdapter4_<formulaindex>;

APPLY JAVA \"TcpSniffer\" AS InputAdapter<formulaindex> (
    schema0 = \"\<?xml version=\\\\\"1.0\\\\\" encoding=\\\\\"UTF-8\\\\\"?>
\n\<schema name=\\\\\"schema:TCP_W_Payload\\\\\">\n
\<field description=\\\\\"\\\\\" name=\\\\\"x1\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x2\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x3\\\\\"
type=\\\\\"string\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x4\\\\\"
type=\\\\\"int\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x5\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x6\\\\\"
type=\\\\\"long\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x7\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x8\\\\\"
type=\\\\\"bool\\\\\"/>\n
```

```
\<field description=\\\\\"\\\\\" name=\\\\\"x9\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x10\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x11\\\\\"
type=\\\\\"bool\\\\\"/>\n
\<field description=\\\\\"\\\\\" name=\\\\\"x12\\\\\"
type=\\\\\"string\\\\\"/>
\n\</schema>\n\"
)
INTO out__InputAdapter4_<formulaindex>;

CREATE STREAM out__Map4_<formulaindex> ;
SELECT
    out__InputAdapter4_<formulaindex>.* AS *,
    to_milliseconds(now()) AS predicate_time
FROM   out__InputAdapter4_<formulaindex>
INTO   out__Map4_<formulaindex>
;

<sfilter:{CREATE STREAM out__Filter<formulaindex>_<it>}; separator=";\n">;

SELECT * FROM out__Map4_<formulaindex>
<wc; separator="\n">;

CREATE STREAM out__Query2_<formulaindex> ;
SELECT out__Filter<formulaindex>_1.x1, out__Filter<formulaindex>_1.x2,
out__Filter<formulaindex>_1.x3, out__Filter<formulaindex>_1.x4,
out__Filter<formulaindex>_1.x5, out__Filter<formulaindex>_1.x6,
out__Filter<formulaindex>_1.x7, out__Filter<formulaindex>_1.x8,
out__Filter<formulaindex>_1.x9, out__Filter<formulaindex>_1.x10,
out__Filter<formulaindex>_1.x11, out__Filter<formulaindex>_1.x12,
out__Filter<formulaindex>_1.predicate_time,
QueryTable<formulaindex>.packetno AS tablepacketno,
QueryTable<formulaindex>.predicate_time AS tablepredicate_time,
<skey:{QueryTable<formulaindex>.<it> AS table<it>}; separator=",">
 FROM out__Filter<formulaindex>_1 OUTER JOIN QueryTable<formulaindex>

WHERE QueryTable<formulaindex>.predicate_time >=
(out__Filter<formulaindex>_1.predicate_time - <w>) and
QueryTable<formulaindex>.predicate_time \<=
out__Filter<formulaindex>_1.predicate_time and
<skey:{QueryTable<formulaindex>.<it> =
out__Filter<formulaindex>_1.<it>}; separator = " and ">
LIMIT 1
INTO out__Query2_<formulaindex>;
```

```
DECLARE sequence1id long DEFAULT 0;
 -- seqIdSetter
CREATE STREAM gen__seqIdSetter<formulaindex> (
    packetno long
);

CREATE STREAM out__Sequence<formulaindex>_1 ;
SELECT sequence1id + 1 AS packetno FROM out__Filter<formulaindex>_2
INTO gen__seqIdSetter<formulaindex>;
SELECT *, sequence1id AS packetno FROM out__Filter<formulaindex>_2
INTO out__Sequence<formulaindex>_1;

CREATE STREAM out__Filter<formulaindex>_3 ;
CREATE STREAM out__Filter<formulaindex>_4 ;
SELECT * FROM out__Query2_<formulaindex>
WHERE isnull(tablex1) INTO out__Filter<formulaindex>_3
WHERE true INTO out__Filter<formulaindex>_4
;

REPLACE INTO QueryTable<formulaindex> (packetno, predicate_time, <skey:{<it>};
separator=","> )
    SELECT packetno, predicate_time, <skey:{<it>}; separator=",">
    FROM out__Sequence<formulaindex>_1
;
DELETE FROM QueryTable<formulaindex>
USING out__Filter<formulaindex>_3
WHERE (QueryTable<formulaindex>.predicate_time \<
out__Filter<formulaindex>_3.predicate_time - <w>) or
(<skey:{QueryTable<formulaindex>.<it> =
out__Filter<formulaindex>_3.<it>}; separator=" and ">)
;
CREATE STREAM out__PatternSum_<formulaindex> ;
 -- Calculates how many orders for the submitted category
CREATE WINDOW sumOfPattern<formulaindex>(SIZE <s> ADVANCE 1 TUPLES);
SELECT
    count() AS Numberpackets,
    firstval(predicate_time) AS FirstPatternT,
    lastval(predicate_time) AS LastPatternT,
    firstval(*) AS input_*
FROM out__Filter<formulaindex>_3[sumOfPattern<formulaindex>]
INTO out__PatternSum_<formulaindex>;

SELECT * FROM out__PatternSum_<formulaindex>
WHERE ((LastPatternT - FirstPatternT) \< <w>) INTO OutputStream<formulaindex>
;

UPDATE sequence1id FROM (SELECT * FROM gen__seqIdSetter<formulaindex>);"
```

```
> d<formulaindex>.ssql
<endif>
>>
```

Appendix B

Single Packet Attacks With Payload Signatures Files

B.1 *SNORT* Single Packet Signatures File

```
# LOCAL RULES
# ----------------
alert tcp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"sid-255 DNS zone
transfer TCP"; flow:to\_server,established; content:"|00 00 FC|";
offset:15; metadata:policy security\-ips drop; reference:arachnids,212;
reference:cve,1999\-0532; reference:nessus,10595;
classtype:attempted-recon; sid:255; rev:17;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 79 (msg:"sid-323 FINGER root query";
flow:to_server,established; content:"root"; reference:arachnids,376;
classtype:attempted-recon; sid:323; rev:7;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-334 FTP .forward";
flow:to_server,established; content:".forward"; reference:arachnids,319;
classtype:suspicious-filename-detect; sid:334; rev:8;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-361 FTP SITE EXEC
attempt"; flow:to_server,established; content:"SITE"; nocase;
content:"EXEC"; distance:0;nocase; pcre:"/^SITE\s+EXEC/smi";
reference:arachnids,317; reference:bugtraq,2241; reference:cve,1999-0080;
reference:cve,1999-0955; classtype:bad-unknown; sid:361;
rev:18;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1919 FTP CWD overflow
attempt"; flow:to_server,established; content:"CWD"; nocase; isdataat:180,
relative; pcre:"/^CWD(?!\n)\s[^\n]{180}/smi"; reference:bugtraq,11069;
reference:bugtraq,1227; reference:bugtraq,1690; reference:bugtraq,6869;
reference:bugtraq,7251; reference:bugtraq,7950; reference:cve,1999-0219;
reference:cve,1999-1058; reference:cve,1999-1510; reference:cve,2000-1035;
reference:cve,2000-1194; reference:cve,2001-0781; reference:cve,2002-0126;
```

```
reference:cve,2002-0405; classtype:attempted-admin; sid:1919; rev:28;)

alert tcp $HOME_NET any <> $EXTERNAL_NET any (msg:"sid-241 DDOS shaft
synflood"; flow:stateless; flags:S,12; seq:674711609;
reference:arachnids,253; reference:cve,2000-0138;
classtype:attempted-dos; sid:241; rev:10;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 15104 (msg:"sid-249 DDOS mstream
client to handler"; flow:stateless; flags:S,12; reference:arachnids,111;
reference:cve,2000-0138; classtype:attempted-dos; sid:249; rev:8;)

alert tcp $EXTERNAL_NET 20 -> $HOME_NET :1023 (msg:"sid-503 DELETED MISC Source
Port 20 to <1024"; flow:stateless; flags:S,12; reference:arachnids,06;
classtype:bad-unknown; sid:503; rev:8;)

alert tcp $EXTERNAL_NET 53 -> $HOME_NET :1023 (msg:"sid-504 DELETED MISC source
port 53 to <1024"; flow:stateless; flags:S,12; reference:arachnids,07;
classtype:bad-unknown; sid:504; rev:8;)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"sid-497
ATTACK-RESPONSES file copied ok"; flow:established;
content:"1 file|28|s|29| copied"; nocase; metadata:policy balanced-ips drop,
policy security-ips drop; reference:bugtraq,1806; reference:cve,2000-0884;
classtype:bad-unknown; sid:497; rev:14;)

alert tcp $EXTERNAL_NET any <> $HOME_NET 0 (msg:"sid-524 DELETED BAD-TRAFFIC
tcp port 0 traffic"; flow:stateless; classtype:misc-activity; sid:524; rev:10;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 161 (msg:"sid-1418 SNMP request tcp";
flow:stateless; reference:bugtraq,4088; reference:bugtraq,4089;
reference:bugtraq,4132; reference:cve,2002-0012; reference:cve,2002-0013;
classtype:attempted-recon; sid:1418; rev:15;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1497 DELETED
WEB-MISC cross site scripting attempt"; flow:to_server,established;
content:"|3C 53 43 52 49 50 54|"; classtype:web-application-attack; sid:1497;
rev:10;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"sid-530 NETBIOS NT NULL
session"; flow:to_server,established; content:"|00 00 00 00|W|00|i|00|n|00
|d|00|o|00|w|00|s|00| |00|N|00|T|00| |00|1|00|3|00|8|00|1";
reference:arachnids,204; reference:bugtraq,1163; reference:cve,2000-0347;
classtype:attempted-recon; sid:530; rev:11;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 3128 (msg:"sid-618 DELETED SCAN Squid
Proxy attempt"; flow:stateless; flags:S,12; classtype:attempted-recon; sid:618;
rev:11;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 1080 (msg:"sid-615 DELETED SCAN SOCKS
Proxy attempt"; flow:stateless; flags:S,12; reference:url,
help.undernet.org/proxyscan/; classtype:attempted-recon; sid:615; rev:11;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 8080 (msg:"sid-620 DELETED SCAN Proxy
Port 8080 attempt"; flow:stateless; flags:S,12; classtype:attempted-recon;
sid:620; rev:12;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"sid-622 SCAN ipEye SYN
scan"; flow:stateless; flags:S; seq:1958810375; reference:arachnids,236;
classtype:attempted-recon; sid:622; rev:8;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"sid-623 DELETED SCAN NULL";
flow:stateless; ack:0; flags:0; seq:0; reference:arachnids,4;
classtype:attempted-recon; sid:623; rev:8;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-974 WEB-IIS
Directory transversal attempt"; flow:to_server,established; content:"..|5C|..";
fast_pattern:only; reference:bugtraq,2218; reference:cve,1999-0229;
classtype:web-application-attack; sid:974; rev:15;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-978 WEB-IIS
ASP contents view"; flow:to_server,established;
content:"%20"; content:"&CiRestriction=none"; nocase;
content:"&CiHiliteType=Full"; fast_pattern:only; reference:bugtraq,1084;
reference:cve,2000-0302; reference:nessus,10356;
reference:url,www.microsoft.com/technet/security/bulletin/MS00-006.mspx;
classtype:web-application-attack; sid:978; rev:17;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-981 DELETED
WEB-IIS unicode directory traversal attempt"; flow:to_server,established;
content:"/..%c0%af../"; nocase; reference:bugtraq,1806;
reference:cve,2000-0884; reference:nessus,10537;
classtype:web-application-attack; sid:981; rev:13;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-983 DELETED
WEB-IIS unicode directory traversal attempt"; flow:to_server,established;
content:"/..%c1%9c../"; nocase; reference:bugtraq,1806;
reference:cve,2000-0884; reference:nessus,10537;
classtype:web-application-attack; sid:983; rev:13;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1002 WEB-IIS
cmd.exe access"; flow:to_server,established; content:"cmd.exe"; fast_pattern;
nocase; http_uri; metadata:policy balanced-ips drop, policy connectivity-ips
drop, policy security-ips drop, service http; classtype:web-application-attack;
sid:1002; rev:14;)
```

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1042 WEB-IIS
view source via translate header"; flow:to_server,established;
content:"Translate|3A| F"; fast_pattern:only; reference:arachnids,305;
reference:bugtraq,14764; reference:bugtraq,1578; reference:cve,2000-0778;
reference:nessus,10491; classtype:web-application-activity; sid:1042; rev:17;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1070
WEB-MISC WebDAV search access"; flow:to_server,established; content:"SEARCH ";
depth:8; nocase; reference:arachnids,474; reference:bugtraq,1756;
reference:cve,2000-0951; classtype:web-application-activity;
sid:1070; rev:12;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1112 DELETED
http directory traversal"; flow:to_server,established; content:"..|5C|";
reference:arachnids,298; classtype:attempted-recon; sid:1112; rev:9;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1113 DELETED
WEB-MISC http directory traversal"; flow:to_server,established; content:"../";
reference:arachnids,297; classtype:attempted-recon; sid:1113; rev:7;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1122
WEB-MISC /etc/passwd"; flow:to_server,established; content:"/etc/passwd";
nocase; http_uri; classtype:attempted-recon; sid:1122; rev:9;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1136
WEB-MISC cd.."; flow:to_server,established; content:"cd.."; nocase;
classtype:attempted-recon; sid:1136; rev:8;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1147
WEB-MISC cat%20 access"; flow:to_server,established; content:"cat ";
nocase; http_uri; reference:bugtraq,374; reference:cve,1999-0039;
classtype:attempted-recon; sid:1147; rev:11;)

alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"sid-1201
ATTACK-RESPONSES 403 Forbidden"; flow:from_server,established; content:"403";
http_stat_code; classtype:attempted-recon; sid:1201; rev:8;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"sid-1239 NETBIOS RFParalyze
Attempt"; flow:to_server,established; content:"BEAVIS"; content:"yep yep";
metadata:policy balanced-ips drop, policy connectivity-ips drop,
policy security-ips drop; reference:bugtraq,1163; reference:cve,2000-0347;
reference:nessus,10392; classtype:attempted-recon; sid:1239; rev:11;)

alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"sid-1292 ATTACK-RESPONSES
directory listing"; flow:established; content:"Volume Serial Number";
classtype:bad-unknown; sid:1292; rev:9;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1378 TP wu-ftp bad file
completion attempt"; flow:to_server,established; content:"~";
content:"{"; distance:0; reference:bugtraq,3581; reference:bugtraq,3707;
reference:cve,2001-0550; reference:cve,2001-0886; reference:nessus,10821;
classtype:misc-attack; sid:1378; rev:20;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 162 (msg:"sid-1420 SNMP trap tcp";
flow:stateless; reference:bugtraq,4088; reference:bugtraq,4089;
reference:bugtraq,4132; reference:cve,2002-0012; reference:cve,2002-0013;
classtype:attempted-recon; sid:1420; rev:15;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 705 (msg:"sid-1421 SNMP AgentX/tcp
request"; flow:stateless; reference:bugtraq,4088; reference:bugtraq,4089;
reference:bugtraq,4132; reference:cve,2002-0012; reference:cve,2002-0013;
classtype:attempted-recon; sid:1421; rev:15;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1529 FTP SITE overflow
attempt"; flow:to_server,established; content:"SITE"; nocase; isdataat:100,
relative; pcre:"/^SITE(?!\n)\s[^\n]{100}/smi"; reference:cve,1999-0838;
reference:cve,2001-0755; reference:cve,2001-0770; classtype:attempted-admin;
sid:1529; rev:14;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1530 DELETED FTP format
string attempt"; flow:to_server,established; content:"%p"; nocase;
reference:bugtraq,1387; reference:bugtraq,2240; reference:bugtraq,726;
reference:cve,1999-0997; reference:cve,2000-0573; reference:nessus,10452;
classtype:attempted-admin; sid:1530; rev:14;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1734 FTP USER overflow
attempt"; flow:to_server,established; content:"USER"; nocase; isdataat:100,
relative; pcre:"/^USER(?!\n)\s[^\n]{100}/smi"; reference:bugtraq,10078;
reference:bugtraq,10720; reference:bugtraq,1227; reference:bugtraq,1504;
reference:bugtraq,15352; reference:bugtraq,1690; reference:bugtraq,22044;
reference:bugtraq,22045; reference:bugtraq,4638; reference:bugtraq,7307;
reference:bugtraq,8376; reference:cve,1999-1510; reference:cve,1999-1514;
reference:cve,1999-1519; reference:cve,1999-1539; reference:cve,2000-0479;
reference:cve,2000-0656; reference:cve,2000-0761; reference:cve,2000-0943;
reference:cve,2000-1035; reference:cve,2000-1194; reference:cve,2001-0256;
reference:cve,2001-0794; reference:cve,2001-0826; reference:cve,2002-0126;
reference:cve,2002-1522; reference:cve,2003-0271; reference:cve,2004-0286;
reference:cve,2005-2123; reference:cve,2005-3683; classtype:attempted-admin;
sid:1734; rev:40;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1807
WEB-MISC Chunked-Encoding transfer attempt"; flow:to_server,established;
content:"Transfer-Encoding|3A|"; nocase; http_header; content:"chunked";
nocase; http_header; reference:bugtraq,4474; reference:bugtraq,4485;
```

```
reference:bugtraq,5033; reference:cve,2002-0071; reference:cve,2002-0079;
reference:cve,2002-0392; reference:nessus,10932;
classtype:web-application-attack; sid:1807; rev:15;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (msg:"sid-1930 IMAP auth literal
overflow attempt"; flow:established,to_server; pcre:"/.* [aA][uU][tT][hH].*/";
metadata:policy balanced-ips drop, policy connectivity-ips drop, policy
security-ips drop, service imap; reference:bugtraq,21724;
reference:cve,1999-0005; reference:cve,2006-6424; classtype:misc-attack;
sid:1930; rev:11;)

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"sid-1945 DELETED
WEB-IIS unicode directory traversal attempt"; flow:to_server,established;
content:"/..%255c.."; nocase; reference:bugtraq,1806; reference:cve,2000-0884;
reference:nessus,10537; classtype:web-application-attack; sid:1945; rev:8;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1971 FTP SITE EXEC format
string attempt"; flow:to_server,established; content:"SITE"; nocase;
content:"EXEC"; distance:0; nocase; pcre:"/^SITE\s+EXEC\s[^\n]*?%[^\n]*?%/smi";
reference:bugtraq,1387; reference:bugtraq,1505; classtype:bad-unknown;
sid:1971; rev:9;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1973 FTP MKD overflow
attempt"; flow:to_server,established; content:"MKD"; nocase; isdataat:150,
relative; pcre:"/^MKD(?!\n)\s[^\n]{150}/smi"; reference:bugtraq,11772;
reference:bugtraq,15457; reference:bugtraq,39041; reference:bugtraq,612;
reference:bugtraq,7278; reference:bugtraq,9872; reference:cve,1999-0911;
reference:cve,2005-3683; reference:cve,2009-3023; reference:cve,2010-0625;
reference:nessus,12108; reference:url,www.kb.cert.org/vuls/id/276653;
reference:url,www.microsoft.com/technet/security/bulletin/MS09-053.mspx;
classtype:attempted-admin; sid:1973; rev:22;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1972 FTP PASS overflow
attempt"; flow:to_server,established; content:"PASS"; nocase; isdataat:100,
relative; pcre:"/^PASS(?!\n)\s[^\n]{100}/smi"; reference:bugtraq,10078;
reference:bugtraq,10720; reference:bugtraq,15457; reference:bugtraq,1690;
reference:bugtraq,22045; reference:bugtraq,3884; reference:bugtraq,8601;
reference:bugtraq,9285; reference:cve,1999-1519; reference:cve,1999-1539;
reference:cve,2000-1035; reference:cve,2002-0126; reference:cve,2002-0895;
reference:cve,2005-3683; classtype:attempted-admin; sid:1972; rev:24;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1975 FTP DELE overflow
attempt"; flow:to_server,established; content:"DELE"; nocase; isdataat:100,
relative; pcre:"/^DELE(?!\n)\s[^\n]{100}/mi"; reference:bugtraq,15457;
reference:bugtraq,2972; reference:bugtraq,46922; reference:cve,2001-0826;
reference:cve,2001-1021; reference:cve,2005-3683; reference:cve,2010-4228;
reference:nessus,11755; classtype:attempted-admin; sid:1975; rev:17;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"sid-1992 FTP LIST directory
traversal attempt";flow:to_server,established;pcre:"/.*LIST.{1}.*...{1}.*../";
reference:bugtraq,2618; reference:cve,2001-0680; reference:cve,2002-1054;
reference:nessus,11112; classtype:protocol-command-decode; sid:1992; rev:11;)

alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg:"sid-2087 SMTP From
comment overflow attempt"; flow:to_server,established; content:"From|3A|";
nocase;content:"<><><><><><><><><><><><><><><><><><><><><><><>"; distance:0;
content:"|28|"; distance:1; content:"|29|"; distance:1;
reference:bugtraq,6991; reference:cve,2002-1337;
reference:url,www.kb.cert.org/vuls/id/398025; classtype:attempted-admin;
sid:2087; rev:10;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 143 (msg:"sid-2105 IMAP authenticate
literal overflow attempt"; flow:established,to_server; content:"AUTHENTICATE";
fast_pattern:only; nocase; pcre:"/\sAUTHENTICATE\s[^\n]*?\{/smi";
byte_test:5,>,256,0,string,dec,relative; reference:bugtraq,21724;
reference:cve,1999-0042; reference:cve,2006-6424; reference:nessus,10292;
classtype:misc-attack; sid:2105; rev:11;)
```

B.2 *BRO* Single Packet Signatures File

```
signature sid-255 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 53
  event "DNS zone transfer TCP"
  tcp-state established,originator
  payload /.{14}.*\x00\x00\xFC/
  }

signature sid-323 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 79
  event "FINGER root query"
  tcp-state established,originator
  payload /.*root/
  }

signature sid-334 {
```

```
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP .forward"
  tcp-state established,originator
  payload /.*\.forward/
  }

signature sid-361 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP site exec"
  tcp-state established,originator
  payload /.*[sS][iI][tT][eE] .*.{0}.*[eE][xX][eE][cC] /
  }

signature sid-241-a {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  header tcp[13:1] & 255 == 2
  header tcp[4:4] == 674711609
  event "DDOS shaft synflood"
  }

signature sid-241-b {
  ip-proto == tcp
  src-ip == local_nets
  dst-ip != local_nets
  header tcp[13:1] & 255 == 2
  header tcp[4:4] == 674711609
  event "DDOS shaft synflood"
  }

signature sid-249 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 15104
  header tcp[13:1] & 255 == 2
  event "DDOS mstream client to handler"
  }

signature sid-503 {
```

```
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  src-port == 20
  dst-port >= 0
  dst-port <= 1023
  header tcp[13:1] & 255 == 2
  event "MISC Source Port 20 to <1024"
  }

signature sid-504 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  src-port == 53
  dst-port >= 0
  dst-port <= 1023
  header tcp[13:1] & 255 == 2
  event "MISC source port 53 to <1024"
  }

signature sid-497 {
  ip-proto == tcp
  src-ip == http_servers
  dst-ip != local_nets
  src-port == http_ports
  event "ATTACK-RESPONSES file copied ok"
  tcp-state established,responder
  payload /.*1 [fF][iI][lL][eE]\([sS]\) [cC][oO][pP][iI][eE][dD]/
  }

signature sid-524-a {
  ip-proto == tcp
  src-ip == local_nets
  dst-ip != local_nets
  src-port == 0
  event "BAD-TRAFFIC tcp port 0 traffic"
  }

signature sid-524-b {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 0
  event "BAD-TRAFFIC tcp port 0 traffic"
  }
```

```
signature sid-1418 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 161
  event "SNMP request tcp"
  }

signature sid-1497 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC cross site scripting attempt"
  tcp-state established,originator
  payload /.*<[sS][cC][rR][iI][pP][tT]>/
  }

signature sid-530 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 139
  event "NETBIOS NT NULL session"
  tcp-state established,originator
  payload /.*\x00\x00\x00\x00\x57\x00\x69\x00\x6E\x00\x64\x00\x6F\x00\x77\x00
          \x73\x00\x20\x00\x4E\x00\x54\x00\x20\x00\x31\x00\x33\x00\x38\x00\x31/
  }

signature sid-618 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 3128
  event "SCAN Squid Proxy attempt"
  header tcp[13:1] & 255 == 2
  }

signature sid-615 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 1080
  header tcp[13:1] & 255 == 2
  event "SCAN SOCKS Proxy attempt"
  }
```

```
signature sid-620 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 8080
  event "SCAN Proxy (8080) attempt"
  header tcp[13:1] & 255 == 2
  }

signature sid-622 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  header tcp[13:1] & 255 == 2
  header tcp[4:4] == 1958810375
  event "SCAN ipEye SYN scan"
  }

signature sid-623 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  header tcp[8:4] == 0
  header tcp[13:1] & 255 == 0
  header tcp[4:4] == 0
  event "SCAN NULL"
  }

signature sid-974 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS .... access"
  tcp-state established,originator
  payload /.*\x2e\x2e\x5c\x2e\x2e/
  }

signature sid-978 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS ASP contents view"
  tcp-state established,originator
  payload /.*%20/
  payload /.*&[cC][iI][rR][eE][sS][tT][rR][iI][cC][tT][iI]
```

```
            [oO][nN]=[nN][oO][nN][eE]/
  payload /.*&[cC][iI][hH][iI][lL][iI][tT][eE][tT][yY][pP]
            [eE]=[fF][uU][lL][lL]/
  }

signature sid-981 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS unicode directory traversal attempt"
  tcp-state established,originator
  payload /.*\/\.\.%[cC]0%[aA][fF]\.\.\//
  }

signature sid-983 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS unicode directory traversal attempt"
  tcp-state established,originator
  payload /.*\/\.\.%[cC]1%9[cC]\.\.\//
  }

signature sid-1002 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS cmd.exe access"
  tcp-state established,originator
  payload /.*[cC][mM][dD]\.[eE][xX][eE]/
  }

signature sid-1042 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS view source via translate header"
  tcp-state established,originator
  payload /.*[tT][rR][aA][nN][sS][lL][aA][tT][eE]\x3a [fF]/
  }

signature sid-1070 {
  ip-proto == tcp
```

```
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC WebDAV search access"
  tcp-state established,originator
  payload /.{0,1}[sS][eE][aA][rR][cC][hH] /
  }

signature sid-1112 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC http directory traversal"
  tcp-state established,originator
  payload /.*\.\.\\/
  }

signature sid-1113 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC http directory traversal"
  tcp-state established,originator
  payload /.*\.\.\//
  }

signature sid-1122 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC /etc/passwd"
  tcp-state established,originator
  payload /.*\/[eE][tT][cC]\/[pP][aA][sS][sS][wW][dD]/
  }

signature sid-1136 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC cd.."
  tcp-state established,originator
  payload /.*[cC][dD]\.\./
  }
```

```
signature sid-1147 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC cat%20 access"
  tcp-state established,originator
  payload /.*[cC][aA][tT]%20/
  }

signature sid-1201 {
  ip-proto == tcp
  src-ip == http_servers
  dst-ip != local_nets
  src-port == http_ports
  event "ATTACK-RESPONSES 403 Forbidden"
  tcp-state established,responder
  payload /HTTP\/1\.1 403/
  }

signature sid-1239 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 139
  event "NETBIOS RFParalyze Attempt"
  tcp-state established,originator
  payload /.*BEAVIS/
  payload /.*yep yep/
  }

signature sid-1292 {
  ip-proto == tcp
  src-ip == local_nets
  dst-ip != local_nets
  event "ATTACK-RESPONSES directory listing"
  tcp-state established,responder
  payload /.*Volume Serial Number/
  }

signature sid-1378 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP wu-ftp bad file completion attempt {"
```

```
  tcp-state established,originator
  payload /.*~.{1}.*\{/
  }

signature sid-1420 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 162
  event "SNMP trap tcp"
  }

signature sid-1421 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 705
  event "SNMP AgentX/tcp request"
  }

signature sid-1529 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP SITE overflow attempt"
  tcp-state established,originator
  payload /.*[sS][iI][tT][eE] [^\x0a]{100}/
  }

signature sid-1530 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP format string attempt"
  tcp-state established,originator
  payload /.*%[pP]/
  }

signature sid-1734 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP USER overflow attempt"
  tcp-state established,originator
```

```
  payload /.*[uU][sS][eE][rR] [^\x0a]{100}/
  }

signature sid-1807 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-MISC Transfer-Encoding: chunked"
  tcp-state established,originator
  payload /.*[tT][rR][aA][nN][sS][fF][eE][rR]-[eE][nN]
           [cC][oO][dD][iI][nN][gG]:/
  payload /.*[cC][hH][uU][nN][kK][eE][dD]/
  }

signature sid-1919 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP CWD overflow attempt"
  tcp-state established,originator
  payload /.*[cC][wW][dD] [^\x0a]{100}/
  }

signature sid-1930 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 143
  # Not supported: byte_test: 5,>,256,0,string,dec,relative
  event "IMAP auth overflow attempt"
  tcp-state established,originator
  payload /.* [aA][uU][tT][hH]/
  payload /.*\{/
  }

signature sid-1945 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == http_servers
  dst-port == http_ports
  event "WEB-IIS unicode directory traversal attempt"
  tcp-state established,originator
  payload /.*\/\.\.%255[cC]\.\./
  }
```

```
signature sid-1971 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP SITE EXEC format string attempt"
  tcp-state established,originator
  payload /.*[sS][iI][tT][eE].*.{0}.*[eE][xX][eE][cC] .{1}.*%.{1}.*%/
  }

signature sid-1973 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP MKD overflow attempt"
  tcp-state established,originator
  payload /.*[mM][kK][dD] [^\x0a]{100}/
  }

signature sid-1972 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP PASS overflow attempt"
  tcp-state established,originator
  payload /.*[pP][aA][sS][sS] [^\x0a]{100}/
  }

signature sid-1975 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP DELE overflow attempt"
  tcp-state established,originator
  payload /.*[dD][eE][lL][eE] [^\x0a]{100}/
  }

signature sid-1992 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 21
  event "FTP LIST directory traversal attempt"
  payload /.*LIST.{1}.*\.\..{1}.*\.\./
```

```
  }

signature sid-2087 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == smtp_servers
  dst-port == 25
  event "SMTP From comment overflow attempt"
  tcp-state established,originator
  payload /.*From:.*.{0}.*<><><><><><><><><><><><><><><><><><><><><><><><>.{1}
           .*\(.{1}.*\)/
  }

signature sid-2105 {
  ip-proto == tcp
  src-ip != local_nets
  dst-ip == local_nets
  dst-port == 143
  # Not supported: byte_test: 5,>,256,0,string,dec,relative
  event "IMAP authenticate literal overflow attempt"
  tcp-state established,originator
  payload /.* [aA][uU][tT][hH][eE][nN][tT][iI][cC][aA][tT][eE] .*.{0}.*\{/
  }
```

B.3 *TeStID* Single Packet Signatures File

```
# sid-255
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
          y11,x12) & (x4 = 53) & (x12 = f(".{14}.*\u0000\u0000\u00fc.*"))]
# sid-323
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
          y11,x12) & (x4 = 79) & (x12 = f("(?s).*root.*"))]
# sid-334
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f(".*\\\\.forward"))]
# sid-361
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 =
         f("(?ms).*[sS][iI][tT][eE] .*.{0}.*[eE][xX][eE][cC] .*"))]
# sid-241
(Ex5,x8)[(Ey1,y2,y3,y4,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,y4,x5,y6,y7,x8,y9,y10,
        y11,y12) & (x5 = 674711609) & (x8 = TRUE)]
# sid-249
(Ex4,x8)[(Ey1,y2,y3,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
        y12) & (x4 = 15104) & (x8 = TRUE)]
# sid-503
```

```
(Ex2,x4,x8)[(Ey1,y3,y6,y7,y9,y10,y11,y12) P(y1,x2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
           y12) & (x2 = 20) & (x4 >= 0) & (x4 <= 1023) & (x8 = TRUE)]
# sid-504
(Ex2,x4,x8)[(Ey1,y3,y6,y7,y9,y10,y11,y12) P(y1,x2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
           y12) & (x2 = 53) & (x4 >= 0) & (x4 <= 1023) & (x8 = TRUE)]
# sid-497
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x2 = 80 | x2 = 8000 | x2 = 8001 | x2 = 8080) & (x12 =
         f("(?ms).*1 [fF][iI][lL][eE]\\\\([sS]\\\\) [cC][oO][pP][iI][eE][dD].*"
         ))]
# sid-524
(Ex4)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11,y12) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
     y11,y12) & (x2 = 0)]
# sid-1418
(Ex4)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11,y12) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
     y11,y12) & (x4 = 161)]
# sid-1497
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) & (x12 =
         f("(?ms).*<[sS][cC][rR][iI][pP][tT]>.*"))]
# sid-530
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 139) & (x12 = f("(?ms).*\u0000\u0000\u0000\u0000
         \u0057\u0000\u0069\u0000\u006e\u0000\u0064\u0000\u006f\u0000\u0077
         \u0000\u0073\u0000\u0020\u0000\u004e\u0000\u0054\u0000\u0020\u0000
         \u0031\u0000\u0033\u0000\u0038\u0000\u0031.*"))]
# sid-618
(Ex4,x8)[(Ey1,y2,y3,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
        y12) & (x4 = 3128) & (x8 = TRUE)]
# sid-615
(Ex4,x8)[(Ey1,y2,y3,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
        y12) & (x4 = 1080) & (x8 = TRUE)]
# sid-620
(Ex4,x8)[(Ey1,y2,y3,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,x4,x5,y6,y7,x8,y9,y10,y11,
        y12) & (x4 = 8080) & (x8 = TRUE)]
# sid-622
(Ex5,x8)[(Ey1,y2,y3,y4,y6,y7,y9,y10,y11,y12) P(y1,y2,y3,y4,x5,y6,y7,x8,y9,y10,
        y11,y12) & (x5 = 1958810375) & (x8 = TRUE)]
# sid-623
(Ex5,x6,x7,x8,x9,x10,x11)[(Ey1,y2,y3,y4,y12) P(y1,y2,y3,y4,x5,x6,x7,x8,x9,x10,
        x11,y12) & (x5 = 0) & (x6 = 0) & (x7 = FALSE) & (x8 = FALSE) &
        (x9 = FALSE) & (x10 = FALSE) & (x11 = FALSE)]
# sid-974
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) & (x12 =
        f("(?ms).*\\\\.\\\\.\\\\\u005c\\\\.\\\..*"))]
# sid-978
```

```
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*%20.*&[cC][iI][rR][eE][sS][tT][rR][iI][cC][tT][iI]
        [oO][nN]=[nN][oO][nN][eE].*&[cC][iI][hH][iI][lL][iI][tT][eE][tT][yY]
        [pP][eE]=[fF][uU][lL][lL].*"))]
# sid-981
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*/\\\\.\\\\.%[cC]0%[aA][fF]\\\\.\\\\./.*"))]
# sid-983
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*/\\\\.\\\\.%[cC]1%9[cC]\\\\.\\\\./.*"))]
# sid-1002
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*[cC][mM][dD]\\\\.[eE][xX][eE].*"))]
# sid-1042
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*[tT][rR][aA][nN][sS][lL][aA][tT][eE]\u003a [fF].*"))]
# sid-1070
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f(".{0}[sS][eE][aA][rR][cC][hH] .*"))]
# sid-1112
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*\\\\.\\\\.\\\\\u005c.*"))]
# sid-1113
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*\\\\.\\\\./.*"))]
# sid-1122
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*/[eE][tT][cC]/[pP][aA][sS][sS][wW][dD].*"))]
# sid-1136
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*[cC][dD]\\\\.\\\\..*"))]
# sid-1147
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
        y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
        (x12 = f("(?ms).*[cC][aA][tT]%20.*"))]
# sid-1201
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
```

```
         y11,x12) & (x2 = 80 | x2 = 8000 | x2 = 8001 | x2 = 8080) &
         (x12 = f("(?s)HTTP/1\\\\.1 403.*"))]
# sid-1239
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 139) & (x12 = f("(?ms).*BEAVIS.*yep yep.*"))]
# sid-1292
(Ex12)[(Ey1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,
      y11,x12) & (x12 = f("(?ms).*Volume Serial Number.*"))]
# sid-1378
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f(".*~.{1}.*\\\\{"))]
# sid-1420
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 162)]
# sid-1421
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 705)]
# sid-1529
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 =
         f("(?ms).*[sS][iI][tT][eE] [^\u000a]{100}.*"))]
# sid-1530
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*%[pP].*"))]
# sid-1734
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 =
         f("(?ms).*[uU][sS][eE][rR] [^\u000a]{100}.*"))]
# sid-1919
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 =
         f("(?ms).*[cC][wW][dD] [^\u000a]{100}.*"))]
# sid-1807
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
         (x12 = f("(?ms).*[tT][rR][aA][nN][sS][fF][eE][rR]-[eE][nN][cC][oO][dD]
         [iI][nN][gG]:.*[cC][hH][uU][nN][kK][eE][dD].*"))]
# sid-1930
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 143) &
         (x12 = f("(?ms).* [aA][uU][tT][hH].*\\\\{.*"))]
# sid-1945
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 80 | x4 = 8000 | x4 = 8001 | x4 = 8080) &
         (x12 = f("(?ms).*/\\\\.\\\\.%255[cC]\\\\.\\\\..*"))]
# sid-1971
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
```

```
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*[sS][iI][tT][eE].*.{0}.*[eE]
         [xX][eE][cC] .{1}.*%.{1}.*%.*"))]
# sid-1973
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*[mM][kK][dD]
          [^\u000a]{100}.*"))]
# sid-1972
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*[pP][aA][sS][sS]
          [^\u000a]{100}.*"))]
# sid-1975
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*[dD][eE][lL][eE]
          [^\u000a]{100}.*"))]
# sid-1992
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 21) & (x12 = f("(?ms).*LIST\\\\..*"))]
# sid-2087
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 25) & (x12 = f("(?ms).*From:.*.{0}.*<><><><><><><><>
         <><><><><><><><><><><><><><><>.{1}.*\\\\(.{1}.*\\\\).*"))]
# sid-2105
(Ex4,x12)[(Ey1,y2,y3,y5,y6,y7,y8,y9,y10,y11) P(y1,y2,y3,x4,y5,y6,y7,y8,y9,y10,
         y11,x12) & (x4 = 143) & (x12 = f("(?ms).* [aA][uU][tT][hH][eE][nN]
         [tT][iI][cC][aA][tT][eE] .*.{0}.*\\\\{.*"))]
```