# Retargetable Protocol Conformance Specifications

**Developing a System for Easing the Task of Adding Protocol Conformance Specifications to Stateful Inspection Capable Firewalls**

*By Lars R. Olsen*
**Department of Computer Science**
**Aalborg University**
**2005**

Aalborg University

**Department of Computer Science**

**TITLE:**

Retargetable Protocol Conformance Specifications - Developing a System for Easing the Task of Adding Protocol Conformance Specifications to Stateful Inspection Capable Firewalls

**HAND-IN DATE:**

Master's Thesis
January 2005

**PROJECT GROUP:**

d603a

**GROUP MEMBERS:**

Lars Riis Olsen, *lro@cs.aau.dk*

**PROJECT SUPERVISORS:**

Mikkel Christiansen, *mixxel@cs.aau.dk*
Emmanuel Fleury, *fleury@cs.aau.dk*

**NUMBER OF COPIES:** 4

**REPORT: NUMBER OF PAGES:** 86

**APPENDIX: NUMBER OF PAGES:** 12

**TOTAL: NUMBER OF PAGES:** 98

**SYNOPSIS:**

Over the years the need for a more powerful firewall classification scheme to supplement stateless packet classification has become apparent. As a response to this demand Stateful Inspection (SI) was developed. While significantly more powerful, this scheme has a number of inherent disadvantages. One of the most predominant ones being its inherent dependence on custom made protocol conformance specifications against which the inspected streams can be checked.
Currently, SI capable firewalls implement these specifications by hard-coding them into the firewall using the generic language used to implement the rest of the firewall. While simple, this approach however has a number of disadvantages in terms of complexity and subsequently in terms of the correctness of the implemented specifications. In effect this complexity means that the risk of errors present in these specifications is considerable and as a result the overall level of security imposed by the firewall might be decreased.
In this report we propose, implement, and test a system capable of easing the task of specifying and implementing protocol conformance specifications. Using this system the risk of errors should therefore be reduced and as a result the general level of security should be increased. This is achieved through the introduction of retargetable specifications which can be reused across different firewall implementations while at the same time be implemented using a custom made language. This way, more effort can be put into the development and testing of one shared specification, as opposed to its complete reimplementation on each available firewall.

# Table of Contents

**I Introduction** **1**

**1 Motivation** **3**

1.1 Project Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**2 Stateful Inspection and its Inherent Problems** **7**

2.1 The Conceptual Architecture of a Firewall . . . . . . . . . . . 7

2.2 Stateful Inspection . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Design Problems Introduced by Stateful Inspection . . . . . . 14

2.4 Current Practices in the Implementation of Stateful Inspection 16

2.5 The Proposed Solution - Retargetable Protocol Conformance Specifications . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.6 Project Description . . . . . . . . . . . . . . . . . . . . . . . . 20

**II The Proposed System** **21**

**3 The Underlying Model** **23**

3.1 The Protocol Conformance Model . . . . . . . . . . . . . . . . 23

3.2 The TCP/UDP Specialization . . . . . . . . . . . . . . . . . . 31

**4 The Protocol Oriented Language** **39**

4.1 The PCSL Language . . . . . . . . . . . . . . . . . . . . . . . 39

**5 The Intermediate Representation** **47**

5.1 Requirements to the Intermediate Representation . . . . . . . 47

5.2 The Current Intermediate Representation . . . . . . . . . . . 48

5.3 The Output Generator API . . . . . . . . . . . . . . . . . . . 52

**6 Output Generation** **57**

6.1 Adding Protocol Conformance Specifications to Netfilter . . . 57

6.2 The Netfilter Output Generator . . . . . . . . . . . . . . . . . . 60

**III Test and Conclusion** **65**

**7 Testing The System** **67**

7.1 The Protocol Conformance Specification . . . . . . . . . . . . 67

7.2 Testing the Performance of the Generated Code . . . . . . . . 68

**8 Further Development** **73**

8.1 Support For Application Layer Protocols . . . . . . . . . . . . 73

8.2 Graphical Front-End for the PCSL Language . . . . . . . . . 74

**9 Conclusion** **75**

**A Header Fields Symbol Table** **77**

**B The PCSL Language** **79**

B.1 Abstract Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 79

B.2 Concrete Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 80

B.3 Type System . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

**C Current Status of the Implementation** **85**

**D Summary** **87**

# Preface

This report documents the Master's Thesis by *Lars R. Olsen* written under the research unit of *Distributed Systems and Semantics* at the *Department of Computer Science* at *Aalborg University*. The project is concerned with stateful inspections dependence on custom made protocol conformance specifications, working as overlays against which the inspected streams are checked. To reduce this dependency this report proposes, implements, and tests a system capable of easing the creation and implementation of such specifications, while at the same time making them retargetable so that they can be reused across different firewalls.

The report assumes that the reader has elementary knowledge about basic networking concepts such as packets, routing, the TCP/IP protocol suite, and firewalls in general. It is split into 3 parts. The first, being the introduction, motivates the project, gives an introduction to the current practices in the implementation of stateful inspection, and describes how protocol conformance specifications are currently created for these. With that in place, a system capable of easing the creation of specifications for these implementations is then proposed. In the second part this system is then described in detail. Finally, in the third part, an implementation of the proposed system is tested and a conclusion concerning the advantages and drawbacks of the proposal is drawn.

A homepage containing this report as well as the implementation of the proposed system is located at the following address:

*http://www.cs.aau.dk/~lro/rpcs*

Lars Riis Olsen

# PART I

# Introduction

This part provides an introduction to the project. It starts by motivating the project in Chapter 1; what is stateful inspection, which improvements does it offer compared to stateless packet classification, and which deficiencies in the current way of creating and implementing specifications for it do we want to alleviate. With that in place, Chapter 2 provides a more detailed introduction to stateful inspection and describes how it is implemented and performed by current firewalls. With an outset in this description, some of the problems introduced by stateful inspection are described and our proposal to alleviate some of these problems is introduced. Finally, with this introduction in place, the part concludes with a definition of the final scope and goals of the project.

# Chapter 1

# Motivation

Over the past decade the Internet has grown tremendously. From including only 213 hosts in 1981, it has grown to consist of approximately 233 million hosts as of January 2004[Sur04]. This dramatic increase illustrates the development of the Internet, from a small set of interconnected computers used for scientific and military purposes only, to the general purpose, commercial network that it is today.

A result of this dramatic growth is an equal increase in the demand for technologies to protect and control its users. The firewall is one such technology. A firewall is essentially a selective router which works by intercepting and examining select parts of the protocol headers of all packets sent through it. Based on this examination, commonly known as classification, the packet is either blocked or let through, thereby allowing the firewall administrator to control the traffic passing through it. To further maximize the control, firewalls are usually deployed to act as gateways between networks, thereby allowing for the examination of all traffic passing between them. An example of this setup can be seen in Figure 1.1. Needless to say the effectiveness of

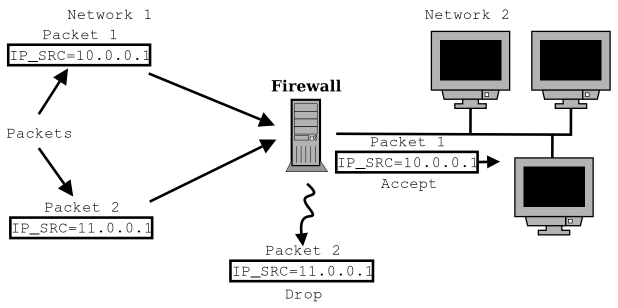


**Figure 1.1.** A firewall acting as a gateway between two networks has received two packets. It has been configured to only accept packets with an *IP source* protocol header field of 10.*.*.*. This means that Packet 1 has been allowed to pass whereas packet 2 has been dropped and therefore removed from the network.

the firewall depends on its ability to classify the intercepted packets. The first firewalls, now referred to as stateless firewalls relied solely on a stateless classification scheme where all packets are classified independently from each other. While fast and simple this scheme however has a number of serious limitations. One of the most predominant ones is the fact that, using this scheme, basing the classification on mutable header fields[1] such as sequence numbers and TCP flags, rarely makes much sense. The result is that the classification in stateless firewalls most often can only be based on a small portion of the packet (the immutable fields), thereby neglecting a lot of information that could otherwise be used to sharpen the classification. An example of the consequences of this limitation is the ACK ping attack which allows an attacker to determine whether an IP address is in use, even though the potential host would placed behind a stateless firewall[tW00]. Where a traditional ping works by sending an ICMP type 8 packet to the address in question[Pos81a], the ACK ping attack works by sending an unsolicited TCP ACK packet (a TCP packet with the ACK flag set) to the victim. If the address is in use, the victim, realizing that the packet is illegal, responds with an RST packet[Pos81b] ultimately telling the attacker that the address is in use. Where a stateless firewall can easily be made to drop all ICMP type 8 packets, thereby disallowing the traditional ping, it has no way of telling the unsolicited packet from a solicited one. The result is that stateless firewalls are not capable of protecting against such attacks as simply dropping all ACK or RST packets would disrupt legal traffic as well. From this example it should therefore be clear that this classification scheme is inadequate and a new, more powerful scheme, is needed.

Stateful Inspection (SI) is one such scheme. It distinguishes itself from the stateless approach in that it incorporates the notion of packet streams, thus making it possible to classify each packet in the context of the stream to which it belongs. In other words, it is capable of behaving very much like the hosts it is trying to protect. It works by storing information about the state of the packet streams existing across the firewall. Every time a packet arrives it is classified using this stored information and a user-defined Protocol Conformance Specification (PCS) specifying a number of requirements that must be met by streams of the type in question (e.g. TCP streams)[2]. As the requirements of the PCS can be made to differ depending on the state of the stream, the inspection can therefore be made stateful by storing the state of the stream in-between inspection of the packets. Based on how the contents of the packet matches the requirements specified for the state in question, a result of the inspection can be obtained (e.g. ok or invalid) and used in the final classification of the packet. Through the use of

[1]Protocol header fields whose correctness depend on the state of the packet stream to which the packet belongs.

[2]Note that there is no universally accepted name for these specifications but that we will refer to them as protocol conformance specifications.

this scheme it is therefore possible for the firewall to base the classification on the state of the stream which in turn enables it to base the classification on mutable fields as well. As a result, through the use of SI, it is possible to protect against state dependent attacks such as ACK Ping as it can easily be established that the unsolicited packet does not belong to any existing stream.

While the introduction of SI clearly increases capabilities of the firewall it also brings about a number of inherent disadvantages. First of all it adds a considerable amount of complexity to the firewall. While SI is conceptually fairly simple, its implementation involves the handling of a number of complex issues such as the efficient storing of information and the implementation of the PCSs. As added complexity always increases the risk of errors being made during development, this factor essentially decreases the overall level of security imposed by the firewall. One of the most significant sources of this added degree of complexity is the fact that SI requires at least one PCS to be devised and implemented for each supported type of stream (TCP, UDP, etc)[3]. In the case of stateless firewalls, adding support for a new type of protocol/packet was simply a matter of getting access to the fields in that packets protocol header(s). For SI on the other hand, the incorporation of the notion of streams means that a specialized PCS must be developed as well. In current firewalls these specifications, which often span several hundred lines of code, are implemented using the same general purpose language used to implement the rest of the firewall[Hom04][Fil04]. As can be seen from the time and effort gone into implementing PCSs for the currently available open source firewalls the result is that adding new PCSs is often a tedious and complicated task. Furthermore, when implementing PCSs using general purpose languages a lot of time is usually spent paying attention to issues not related to the behavior of the streams (avoidance of pointer errors etc.). As firewalls are first and foremost about providing security, and complexity always serves to increase the risk of errors, this approach to the implementation of PCSs is by no means ideal. This project aims to solve this problem by developing a new and more reliable way of implementing PCSs.

## 1.1 Project Goals

The goal of this project is to increase the flexibility and security of SI. This goal is achieved through the development of a retargetable PCS specification system that allows the developer to write the PCSs in a custom made, protocol-oriented, and firewall independent language. This language hides, to the behavior of a protocol, unimportant issues such as the storing and

[3]More than one if you for reasons of security, performance etc. are not content with using the same PCS for all streams of the same type.

retrieving of state information. Doing so, it allows the PCS developer to stop thinking about these issues and instead allows him to focus on what is important - the intended behavior of the packet streams. Secondly, being protocol-oriented means that the language is made exclusively for the task of specifying PCSs. Most notably this means that the language does not contain any unnecessary constructs that can complicate the task at hand. Furthermore, the language being firewall independent allows for the development of compilers that can compile PCSs written in the language into code usable by current and future firewalls. This way different firewalls can reuse the same PCS implementation, thus making it possible to focus on perfecting this single implementation as opposed to manually porting it to the different firewalls. This in turn should strengthen the quality of the PCS and thereby increase the overall level of security imposed by the firewall. Finally, the system eases the job of developers of new firewalls, as all that is needed to add a wide range of PCSs, is to make a compiler for the developed firewall.

Having briefly introduced and motivated the project the next chapter will provide a more in-depth description of SI and its strengths and weaknesses. With this description in place, the proposed retargetable PCS creation system will then be introduced, and the final scope of the project defined.

# Chapter 2

# Stateful Inspection and its Inherent Problems

In the previous chapter it was described how SI, while useful and conceptually simple, has a number of inherent disadvantages when it comes to implementing it. In order to make it clear why this is so, and to further clarify the purpose of the proposed retargetable PCS system, this chapter provides a more in-depth introduction to SI. To fully understand this introduction, one however first need to attain an understanding of the conceptual architecture of a firewall, how it works, and how it goes about incorporating classification schemes such as SI and stateless packet classification. In Section 2.1 a brief introduction to firewalls and their conceptual architecture is therefore given. Then, in Section 2.2 a more thorough introduction to SI and how it is performed is provided. Over the course of that description, the problems surrounding its implementation should become clear, and in Section 2.3 these problems will then be described in greater detail. In Section 2.4 we give a short description to how PCSs are implemented in some current day firewalls, and then in Section 2.5 a more detailed introduction to our retargetable PCS system which aims to ease this task, is given. Finally, in Section 2.6 the introduction is concluded by the definition of the final scope of the project.

## 2.1 The Conceptual Architecture of a Firewall

A firewall can, in short, be described as an advanced selective router. That is, a device which receives packets on a network interface, removes those which are not allowed to pass, and forwards the rest to their proper destination. Exactly which packets are allowed to pass and which are to be blocked is defined in a set of rules created by the administrator of the firewall. More specifically, these rules are created by specifying a number of properties that a class of packets must comply with, along with a description of what must happen to packets belonging to this class. The properties that can be specified depend on the capabilities of the firewall and the following rule illustrates this:

```
rule IP_SRC=10.0.*.* IP_PROTO=6 -a ACCEPT
```

This rule, which is typical for a stateless firewall, defines a class consisting of all packets with the IP_SRC field set to 10.0.*.* and the IP_PROTO field set to 6 (TCP). Furthermore it specifies that all packets belonging to this class must be accepted, thereby allowing all traffic that adheres to these properties to pass through the firewall. As previously described stateless firewalls are restricted to classify all packets independently. More precisely, we define stateless packet classification as performed by the stateless firewalls as follows:

**Definition 1 (Stateless Packet Classification)**
*The task of classifying packets based solely on the contents of the protocol headers of the packet.*

Using this definition implies that only properties concerning the contents of the packets can be specified as properties of the individual classes. SI on the other hand allows for a more high level view. An example of this can be seen in the following rule, which is typical for a firewall capable of using both SI as well as the stateless packet classification:

```
rule IP_SRC=10.0.*.* IP_PROTO=6 -pcs=MyPCS SI=OK -a ACCEPT
```

The difference in this rule compared to the strictly stateless example, is the addition of a new property specifying that the packet, when checked against the *MyPCS* PCS, must result in SI returning *OK*. This way it is no longer sufficient for the TCP packets to have an IP_SRC field of 10.0.*.*. as they now must also make the MyPCS PCS return *OK*. Given a PCS that checks for the correct use of the TCP flags, this rule would therefore protect against all ACK Ping attacks with packets containing an IP_SRC field of 10.0.*.*. With this in mind we define SI as follows:

**Definition 2 (Stateful Inspection)**
*The task of tracking the state of a stream, and based upon this state, checking its packets against a predefined PCS and subsequently returning the inspection result to the rest of the firewall for further classification.*

where a stream is defined as a sequence of related packets, or more specifically as follows:

**Definition 3 (Stream)**
*A sequence of packets related by some relation defined by a protocol used by the packets.*

With this short description of how a firewall is configured using properties and classes in place, the conceptual architecture of a firewall can now be described. This conceptual architecture can be seen in Figure 2.1. In this

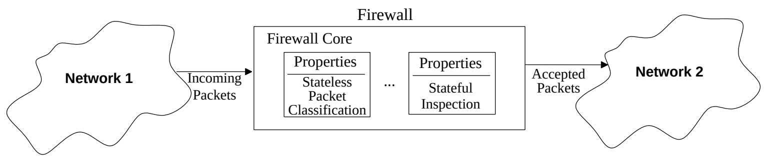


**Figure 2.1.** The conceptual architecture of a firewall where the *Firewall Core* is responsible for the reception, final classification, and forwarding of packets. Similarly, the *Properties* modules are responsible for extracting the values of properties from those packets.

architecture the *Firewall Core* is responsible for the reception, final classification, and forwarding of packets. The classification itself is performed through the use of a number of *Properties* modules which each have a number of properties they can check. For a stateless module this would simply be the values of the different fields in the header, whereas it for an SI module would be the result of the inspection given a specified PCS. Through the use of these modules, it is then the job of the firewall core to retrieve the values of the appropriate properties and determine which rule they match. In relation to this architecture, the task of adding a PCS can therefore be seen either as the task of making an entirely new properties module specifically for this PCS or as the task of extending an existing module with a new property for the new PCS. As it will soon be clear, in this conceptual design, the idea of creating the proposed retargetable PCS system can be seen as the task of developing a retargetable SI properties module capable of being configured, using a protocol oriented language, to return property values reflecting the specified PCS.

## 2.2 Stateful Inspection

In Chapter 1 it was briefly described how SI classifies packets in the context of their streams, thereby allowing for stricter and more protective firewalls. Throughout this section a more thorough description of SI is given. We start by providing a more in-depth introduction to the concept of the PCS.

### 2.2.1 The Protocol Conformance Specification

The PCS is essentially the configuration which defines how SI should behave. It defines, given the reception of any packet of the type for which the PCS was made, and knowledge about the state which the stream is currently in, the property value to be returned to the firewall core as well as the new state of the stream. The PCS can therefore be seen as a state machine and Figure 2.2, which shows a graphical illustration of a PCS capable of detecting the ACK Ping attack described in Chapter 1, illustrates this. In

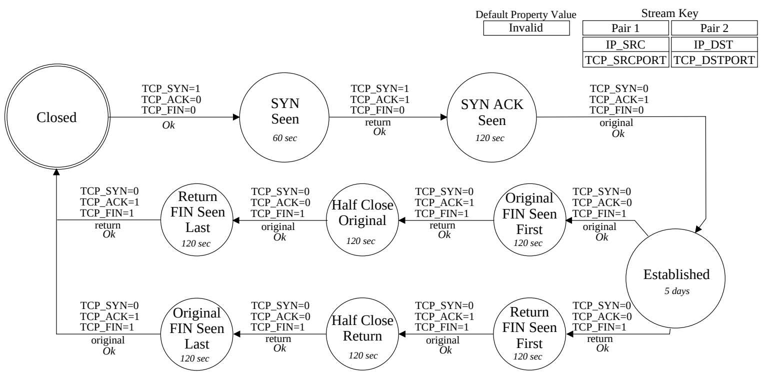
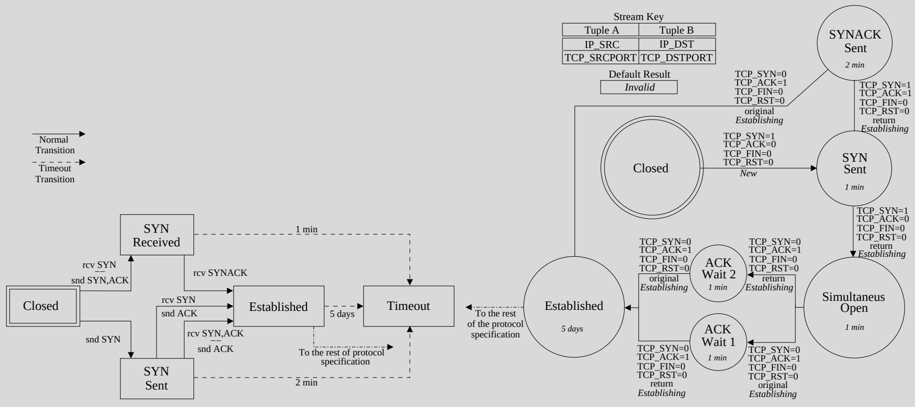

Default Property Value
Invalid
Stream Key
Pair 1
Pair 2
IP_SRC
IP_DST
TCP_SRCPORT
TCP_DSTPORT
Closed
TCP_SYN=1 TCP_ACK=0 TCP_FIN=0 Ok
SYN Seen 60 sec
TCP_SYN=1 TCP_ACK=1 TCP_FIN=0 return Ok
SYN ACK Seen 120 sec
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 original Ok
Established 5 days
TCP_SYN=0 TCP_ACK=0 TCP_FIN=1 original Ok
Original FIN Seen First 120 sec
TCP_SYN=0 TCP_ACK=1 TCP_FIN=1 return Ok
Half Close Original 120 sec
TCP_SYN=0 TCP_ACK=0 TCP_FIN=1 original Ok
Return FIN Seen Last 120 sec
TCP_SYN=0 TCP_ACK=1 TCP_FIN=1 return Ok
TCP_SYN=0 TCP_ACK=0 TCP_FIN=1 return Ok
Return FIN Seen First 120 sec
TCP_SYN=0 TCP_ACK=1 TCP_FIN=1 original Ok
Half Close Return 120 sec
TCP_SYN=0 TCP_ACK=0 TCP_FIN=1 return Ok
Original FIN Seen Last 120 sec
TCP_SYN=0 TCP_ACK=1 TCP_FIN=1 original Ok

**Figure 2.2.** A simplified PCS which checks for the correct use of the TCP- SYN, ACK, and FIN flags and thereby capable of detecting the ACK Ping attack. For a description of the meaning of TCP_SYN, TCP_ACK and TCP_FIN, see Appendix A.

this representation the nodes represent the basic states that the stream can be in. The italic numbers written inside these nodes are the timeout values specifying the amount of time that may pass between the reception of two packets before a timeout occurs. Similarly, the edges (called transitions) describe how a stream can pass between these states. The labels written above the transitions are the requirements, from now on referred to as guards, that must be satisfied by the received packet in order for the stream to go from one state to another. Furthermore, the labels written below the transitions specify which direction the packet must have relative to the first packet of the stream. In the PCS of Figure 2.2 *Original* means that it must be flowing in the same direction as the first, whereas *return* means that it must flow in the opposite direction. The last labels, written below the directions, specify the property value of the inspection. Finally, in the upper right corner, two additional entities are shown. The *default property value*, which specify the property value to be returned for packets not satisfying any of the explicitly specified transitions, and the *stream key*, which specifies the fields used to identify packets belonging to the same stream.

With the specification in Figure 2.2 it should be clear that an ACK Ping attack can be detected as no state information exists about the stream of the unsolicited packet. As a result, the packet is checked against the transitions emerging from the closed (*Closed*) state. As the PCS specifies that the property value *Invalid* is to be returned if the first packet is not a SYN packet, the firewall administrator can protect against ACK Ping attacks by blocking packets returning this value. For an example of how to create a PCS for streams utilizing the TCP/IP protocol, see Example 2.2.1.

**Example 2.2.1 (Simplified PCS for the TCP/IP Protocol Suite)**
TCP is a connection-oriented protocol and therefore by definition goes through a series of states when opening and closing a connection. For the sake of simplicity this example focuses only on the set up phase. In Figure 7.1(a), a transition system showing the states that a TCP/IP connection can pass through during this part of the protocol, can be seen[Pos81b].


Normal Transition
Timeout Transition
SYN Received
1 min
rcv SYN
snd SYN,ACK
rcv SYNACK
Closed
rcv SYN
snd ACK
Established
5 days
Timeout
snd SYN
rcv SYN,ACK
snd ACK
To the rest of protocol specification
SYN Sent
2 min
Stream Key
Tuple A
Tuple B
IP_SRC
IP_DST
TCP_SRCPORT
TCP_DSTPORT
Default Result
Invalid
SYNACK Sent
2 min
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 original Establishing
TCP_SYN=1 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 return Establishing
Closed
TCP_SYN=1 TCP_ACK=0 TCP_FIN=0 TCP_RST=0 New
SYN Sent
1 min
TCP_SYN=1 TCP_ACK=0 TCP_FIN=0 TCP_RST=0 return Establishing
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 original Establishing
ACK Wait 2
1 min
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 return Establishing
To the rest of the protocol specification
Established
5 days
Simultaneus Open
1 min
ACK Wait 1
1 min
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 return Establishing
TCP_SYN=0 TCP_ACK=1 TCP_FIN=0 TCP_RST=0 original Establishing

(a) A transition system depicting the states through which a TCP/IP connection can pass while being set up.

(b) A PCS checking the part of the TCP protocol shown in Figure 7.1(a).

**Figure 2.3.** Transition system for part of the TCP protocol(a) and a corresponding PCS(b).

As that figure shows, a connection is initialized by first sending a SYN packet. The receiving host must then return a SYNACK packet in order to indicate that the first packet has been received. In the event that both hosts try to open a connection at the same time, that is they both send a SYN packet, each host must return an ACK packet to the other before they can both enter the *Established* state. Furthermore, timeouts can occur in all *open* states. An example of this is the *SYN Sent* state. In the event that a packet brings the stream into this state a new SYNACK must arrive within 2 minutes. If this does not happen, a timeout occurs and the stream ceases to exist. Finally, the *stream key* for TCP is made up of the source and destination socket pairs with each pair being an IP address and a port number. That is, if two packets both have the same values stored in these fields they are identified as belonging to the same stream.

With this short description of the first states of the TCP protocol, it is possible to make a PCS that ensures that TCP/IP connections are set up correctly. A graphical illustration of such a PCS, using the previously described notation, can be seen in Figure 7.1(b). As can be seen from that specification any new connection must be initiated by a SYN packet for *New* to be returned to the firewall core. Furthermore, it can also be seen that this packet brings the connection into the *SYN Sent* state. From here two things can happen. Either a *return* SYNACK or a *return* SYN packet is intercepted. In the former case, the connection enters the *SYNACK Sent* state where a further ACK packet will bring the connection into the *Established* state. In the latter case, which is the situation where both hosts simultaneously try to open a connection, a further two ACK packets must be sent before the connection can finally be established. By dropping all packets returning *Invalid* the specifications of the Figure 7.1(a) can therefore be upheld. Similarly, by dropping *New* packets from a network, the establishment of connections by users on this network can be prevented.

### 2.2.2 Performing the Inspection

With the PCS in place it is possible to perform the actual inspection. Currently several different approaches to doing this exists. The most widely used is *Table Based Stateful Inspection*[JSCO02] which is used by most open source firewalls[Fil04][Hom04] and built around a table (the state table) in which the state of all streams currently being inspected is stored. Whenever a packet is received, the state table is consulted for state information about the stream to which the packet belongs, and the packet can subsequently be inspected. The pseudo-code for the *InspectPacket* function, as shown in Algorithm 1, shows how this is done.

As can be seen from this outline the first task upon receiving a packet is to perform a lookup in the state table to determine whether any information is stored about the stream of the packet. Based upon the outcome of this lookup two things can happen:

**Information is found:** If information about the stream is found, this information tells which state the stream was in prior to the reception of the new packet. With this information the content of the packet is then checked against the constraints of the transitions emerging from this state. When a transition whose constraints are satisfied by the packet is found, the property value associated with that transition is delivered to the firewall core. Depending on whether the firewall ends up accepting the packet two things can then happen. If the packet is accepted, the state table is updated to reflect the changes brought by the packet. On the other hand, if the packet is dropped, the state

**Algorithm 1:** Outline of the *InspectPacket* function

```
Data : packet to be inspected
state info ← LookupStreamInTable(packet);
if state info found then
    foreach transition in PCS emerging from stored state do
        if content of packet satisfies transition constraints then
            if property value associated with the transition causes the
            firewall to accept the packet then
                if new state is a closed state then
                    delete state info;
                else
                    update state info;
else
    foreach transition in PCS emerging from closed state do
        if content of packet satisfies transition constraints then
            if property value associated with the transition causes the
            firewall to accept the packet then
                if new state is an open state then
                    create state info for new stream;
```

information remains unchanged to reflect the fact that the packet will never reach its final destination.

**No information is found:** If the lookup yields no information it must be assumed that the received packet is the first in a new stream. In this case the contents of the packet is therefore checked against the transitions emerging from the closed state. When a matching transition is found its associated property value is passed on to the firewall core. As was the case for when state information was found, two things can then happen. If the firewall chooses to accept the packet the aforementioned transition is checked to see if it points to an *open* state. If it does, a new entry is added to the state table to reflect the arrival of this new stream. If on the other hand the firewall chooses to block the packet no entry is added regardless of the new state.

With respect to timeouts these are handled in a slightly different manner. Where the *InspectPacket* function checked the received packets against the transitions in the specification, timeouts only occur when no packets have been received for a predefined amount of time. The normal procedure in table based SI is therefore to periodically check the status of all streams in the state table. If a stream is found for which no packets have been received within the predefined amount of time, the stream has timed out and the entry is deleted. The outline of the *TimeoutHandler* function shown in Algorithm 2 illustrates this.

**Algorithm 2:** Outline of the *TimeoutHandler* function

```
while true do
    foreach stream in state table do
        if stream has timed out then
            delete state info for stream;
    wait X seconds;
```

## 2.3 Design Problems Introduced by Stateful Inspection

With the more detailed description of SI in place, we now turn our attention to a description of some of the problems introduced by it. Some of the most predominant ones will be described next.

### 2.3.1 Adds a Considerable Amount of Complexity to the Firewall

From the description in the previous section it should be clear that performing SI is a relatively complicated task. Inevitably the introduction of SI into an otherwise stateless firewall therefore leads to an increase in the complexity of the firewall. As complexity always increases the risk of errors being made during design and implementation, this is a serious problem. The following paragraphs describe, from a design and implementation point of view, some of the origins of this added complexity.

**Fast storage and access to stream information:** The key aspect differentiating SI from stateless packet classification is SIs ability to view the packets in the context of the streams to which they belong. For this to be possible the firewall needs to be able to store, update, and access information about this context. As should be clear from the outline of the InspectPacket function this information needs to be accessed at least once for every intercepted packet. For these operations not to incur to great a performance penalty, it is therefore paramount to the performance of the firewall, that they can be performed in a fast and efficient manner.

**State information needs to be kept consistent:** With the ability to store information comes the need to keep it consistent with the stream it represents. In particular, this means that two packets belonging to the same stream can not be inspected at the same time as the code in-between the table lookup and the table update is a critical region. That is, if two packets from the same stream are in this section at the same time, the latter is likely to take outset in a wrong state, and therefore likely to be wrongfully inspected.

Secondly, there is the issue of coordinating the operations of the InspectPacket and TimeoutHandler functions. As was described in Section 2.2.2 the normal procedure is to periodically traverse the entire state table in the search for timed out streams. But what if the TimeoutHandler encounters a stream that appears to have timed out only because a packet which arrived within time is still in the previously mentioned critical region. In this situation the connection has clearly not timed out and the stream information should not be deleted.

Both of the above issues can obviously be dealt with through the use of a number of locks. However, as with most multi-threading, the solution and its implementation can quickly become complex when the previously mentioned performance requirements are also taken into account.

**Fast access to protocol conformance specifications:** As with the state table a PCS is accessed every time a packet is received. While the specification does not change during the operation of the firewall, a fast way of accessing it is still needed.

**Efficient handling of timeouts:** While the code needed to deal with timeouts is not very different from that needed for the inspection of packets, it still puts a considerable strain on the state table. Where the InspectPacket function requires a state table with fast access to a single table, the TimeoutHandler requires fast traversal of all entries. As a result, a datastructure capable of performing well in both situations, while still allowing for a great deal of concurrency, is needed.

### 2.3.2 Keeping SI Up-to-date With New Protocols is a Tedious and Error Prone Task

As each PCS is specific to one type of stream at least one PCS is needed for each type of stream supported by the firewall. This means that in order to perform SI on a large number of stream types, you either need an easy way of adding new specifications, or otherwise spend a lot of time adding these. The latter approach however raises a number of issues. First of all it is not likely to be a very feasible long term solution as new protocols and hence new types of streams frequently appear. Secondly, it raises some concerns with regard to the overall security of the firewall. The problem arises with the fact that with each new PCS the amount of code making up the firewall increases. As it is a widely recognized fact that in relation to security, simplicity is a virtue[ea03][ea02], it is a good idea to make the addition of new specifications as easy as possible. This need is further increased by the fact that one PCS for each type of stream might not be enough to suit all needs. An example of this is the PCS for the detection of ACK Ping attacks described in Section 2.2.1. While this specification is indeed capable of detecting such attacks, it does not deal with e.g. the incorrect use of the sequence number fields. Where the specification may therefore be sufficient in some areas of use, it may equally as well be totally insufficient in others. On the other hand, more comprehensive PCSs may in some situations be to strict or indulge to great a performance penalty, and may therefore for some purposes be equally unsuitable. As a result, several PCSs for the same type of stream is likely to be needed.

However, one key factor somewhat easing the need for new PCSs is to be found in the layered layout of the TCP/IP reference model[CK74]. Because of this model, all streams simultaneously make use of several different protocols. In relation to SI this makes it possible to divide these protocols into two groups, each with their own distinct properties. The first group make up the network and transport layer protocols. These protocols are characterized by being few in numbers and hardly ever changing. The second group comprises the application layer protocols, and where only a few network and transport layer protocols exists, new application layer protocols appear on a regular basis. A way of countering the need for many PCSs would therefore be to limit the support to include network and transport layer protocols only. This approach however has one major drawback. While only allowing for SI on network and transport layer protocols is certainly an improvement over the strictly stateless approach, its effect is still limited compared to the full approach where support for all the protocols of the packet is provided. No matter which solution is chosen there is however nothing eliminating the need for easy way of adding new PCSs.

## 2.4 Current Practices in the Implementation of Stateful Inspection

This section provides an introduction to how two currently available firewalls implement SI and try to deal with the previously described problems. It starts by describing the simplest and least flexible approach (used by *OpenBSD PF*[Fil04]), before moving onto the more flexible approach used by the *Linux Netfilter*[Hom04] firewall. Finally, a description of some of the general limitations of the current practices will be given.

### 2.4.1 OpenBSD PF

The implementation of SI in PF has a very monolithic architecture where focus has been put on immediate simplicity as opposed to flexibility and extendability. This means that while its architecture and code tends to be relatively simple, the task of adding new PCSs is comparatively harder. This is most clearly visible in the fact that the firewall provides no functionality for easing the task of adding new PCSs. An example of this is the lack of a built-in state table implementation that automatically handles issues such as performance and concurrency. The result is that the implementor of new specifications is forced to handle these issues himself. Arguably, this is probably also one of the reasons why only very few PCSs have been added to this firewall. Finally, all specifications must be written in C, thereby as described in Chapter 1, further increasing the risk of errors in their implementation.

### 2.4.2 Linux Netfilter

In Netfilter, contrary to what was the case with PF, flexibility and modularity has been a major design goal in all aspects of the development process. This particularly shows in its SI implementation which provides the PCS developer with a number of built-in modules that can be used to efficiently store information, handle concurrency issues etc. While the Netfilter firewall might ease the task of storing state information it does however nothing to make the actual specification of the protocol any easier[1]. This, along with the code integrating it with the built-in modules still has to be written in C. However, in comparison to PF, the addition of these built-in modules is still a big improvement. Furthermore, as these modules are used by virtually all PCSs and are integral parts of the SI subsystem, the number of bugs in these modules is likely to be small. By using these modules, the PCS developer can therefore stop focusing on these issues and instead concentrate more on actual behavior of the stream.

With regards to the protocols supported by Netfilter, the effects of the modular architecture are clear. As of writing, Netfilter ships with several different PCSs for the most commonly used network and transport layer protocols as well as PCSs for a wide range of application layer protocols.

### 2.4.3 Limitations in Current Practices

Although the two firewalls approach to the implementation of SI differ, there is one problem they both share - the job of adding new PCSs is a tedious and unstructured process containing a mix of dealing with the behavior of the stream, as well as writing code integrating it with the firewall. While Netfilter clearly constitutes an improvement over PF it is still insufficient.

[1]In Netfilter PCSs are referred to as *connection tracking helpers*

Most of this stems from the fact the specifications must still be written in C and that this language is simply not designed for this. This makes the task unnecessarily complicated and can be the cause of errors which have nothing to do with the behavior of the streams (pointer errors etc.). Because firewalls are intended to provide security, and many of them (especially hardware firewalls used by private users) are not easily upgradeable once they have been deployed, it is important that they are free of errors. Writing PCSs in general purpose languages is therefore overkill and not a feasible approach. Furthermore, as correctness is paramount to the overall level of security it would be a considerable improvement if it was possible to formally prove the correctness of the PCSs. As proving the correctness of code written in complex languages such as C is generally a very tedious task, this is yet another reason why the current practices are insufficient.

## 2.5 The Proposed Solution - Retargetable Protocol Conformance Specifications

Having described the different problems involved in performing SI and determined the limitation of the current practices, our proposal to a solution to the problem of adding PCSs will now be introduced. We start by giving a brief outline of the proposal thereby describing its core features.

### 2.5.1 Outline of the Proposal

One of the main problems haunting todays implementations of SI is their lack of a simple specialized language designed solely for the task of specifying PCSs. We propose the development of such a language. The main advantage will be that such a language can be made to perfectly hide all issues not directly related to the specification of the PCS. Secondly, having a language that is simple and tailored towards the specification of PCSs should reduce the risk of bugs being present in the final specification. Furthermore, as the behavior of a protocol remains the same no matter in which firewall it is inspected, the language can be kept independent of any particular firewall. The result is that a PCS written in this language can be seen as a universal specification whose deployment is no longer confined to any single firewall. In effect, such a language would therefore allow for the development of a fully retargetable PCS creation system that can be used by current and future firewalls. All that is left to the developer of the firewall is to develop a compiler capable of transforming the universal specifications into code usable by his particular firewall. This way, the same tried and tested PCSs can be reused across different firewalls thereby further strengthening their quality and easing the task of adding them. Finally, the language can be tailored towards easing the task of formally proving the correctness of the specifications. This way, the system will not only ease the implementation of the PCSs, but also make it easier to verify their correctness. In turn, this should thereby reduce the risk of having to issue expensive fixes to already deployed firewalls as a result of erroneous software.

### 2.5.2 The Architecture of the Proposed System

In Figure 2.4 the three phases involved in transforming a universal specification written in the firewall independent language, into the final code usable by a particular firewall, can be seen. The approach is simple and fairly similar to what is used by other compilers. That is, the universal specification is parsed into an intermediate representation from which the different outputs are generated. Using this approach code reuse can be maximized as the implementation created for phases 1 and 2 remains the same for all output generators. This way, all that is left to the firewall developer is to create an output generator for his particular firewall (which itself is a fairly simple task). Finally, as the system is not bound to any specific programming language, the entire system along with the output generators can be implemented in a more high level language which, in turn, should ease the development process.

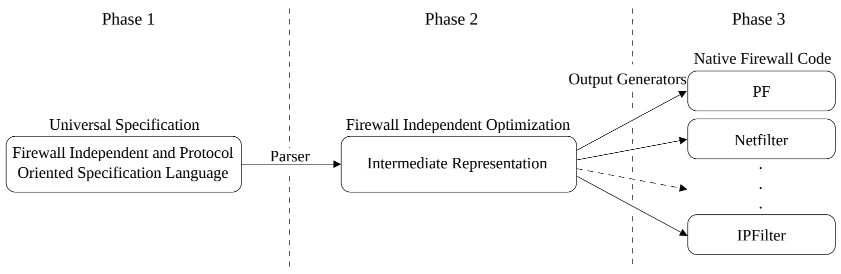

Phase 1
Phase 2
Phase 3
Native Firewall Code
Output Generators
PF
Universal Specification
Firewall Independent Optimization
Netfilter
Firewall Independent and Protocol Oriented Specification Language
Parser
Intermediate Representation
IPFilter

**Figure 2.4.** The three phases involved in transforming a PCS written in the firewall independent language into code usable by a firewall.

The first step in the process depicted in Figure 2.4 is the firewall independent and protocol oriented language. The first task in the development of this language is to determine what it must be able to represent. That is, what makes up a PCS, which protocols must it be able to represent, and what is needed in order to specify the intended behavior of these protocols. The result of this analysis is the development of a minimal model capable of representing the specifications, and the development of a language capable of representing this model. The next step is to develop an intermediate representation upon which firewall independent optimizations can be performed, and from which the final output can be generated. Doing so, the key issue is to figure out which optimizations are feasible to perform, and in which data structure the intermediate representation should be stored. The final step in the transformation is to generate the actual output. This task is performed by output generators provided by the firewall developers. The job of these generators is to transform the optimized specification stored in the intermediate representation, into something usable by the different firewalls.

## 2.6 Project Description

The goal of this project is to design, implement, and test the model and accompanying retargetable PCS system described in the previous section. As this, to our knowledge is the first project dealing with this issue, we will narrow its focus and limit the system to the support of PCSs for TCP/IP and UDP/IP only. The reason for choosing these protocols is mainly due to their widespread use as the basis for most of todays Internet traffic[Tra01].

As described in Section 2.5 the development of the proposed system includes the design and implementation of the following components:

- Phase 1
  - A model capable of representing protocol conformance specifications for the TCP/IP and UDP/IP protocol suites.
  - A firewall independent and protocol oriented language with an expressive power equivalent to that of the aforementioned model.
  - A parser capable of turning PCSs written in the previously mentioned language into a parse tree acting as an interface between the parser and the intermediate representation.
- Phase 2
  - An intermediate representation capable of optimizing and storing the PCS.
  - An API for use by the output generators, capable of providing access to PCSs stored in the intermediate representation.
- Phase 3
  - To be able to test the proposed system and facilitate its current and future development an output generator for the Netfilter[Hom04] firewall is needed.

# PART II

# The Proposed System

Having briefly introduced the proposed system this part describes, in detail, the different parts of that system. As described in relation to the architecture a number of steps are involved in transforming the PCS written in the firewall independent language into code usable by the individual firewalls. To reflect this architecture this part contains a separate chapter for each of these steps. As a result, Chapter 3 describes the underlying minimal model capable of representing PCSs for the TCP/IP and UDP/IP protocol suites, and developed to act as the foundation for the retargetable system. With that foundation in place, Chapter 4 describes the protocol oriented language developed to be capable of representing that model. With the language in place Chapter 5 concerns the development of the intermediate representation and the corresponding API from which the final output can be generated. Finally, Chapter 6 is concerned with output generation process and provides a hands-on example of how an output can be generated using the aforementioned API.

# Chapter 3

# The Underlying Model

This chapter describes the model underlying the proposed system. To increase flexibility, while at the same time building a solid foundation for the final system, the development of this model was split into two phases. In Section 3.1 the behavior and different pieces of basic functionality needed in a PCS is described, and an abstract model, the *protocol conformance model* (PCM), which captures this behavior and functionality is defined. Through the creation of this abstract model we aim to create a common foundation which, depending on how it is specialized, can be made to represent PCSs for various current and future protocols. Finally, and in tune with this approach, Section 3.2 presents such a specialization capable of representing TCP and UDP PCSs. Throughout the rest of the report, this specialization is then used as the basis for the current version of proposed system.

## 3.1 The Protocol Conformance Model

As should be clear from the PCSs described and illustrated in the previous chapters, the behavior of a protocol, and hence the streams using them, can be described using an automaton. The PCM is therefore a formal definition of the key concepts of the automaton described in those chapters, combined with a definition of the environment in which it operates[1]. We begin by describing and defining the major components in this environment.

### 3.1.1 The Environment

The two elements in the environment in which SI, and therefore the PCSs, operates are those of hosts and streams. In this environment it is the responsibility of the SI system to inspect the contents of the packets and return a

[1]The automaton used in the previous chapters is itself heavily inspired by that of timed automata[AD94].

result to the firewall core. As all packets using the same protocols are usually divided into the same predefined number of parts (each part containing a particular piece of information, IP_SRC, TCP_SRCPORT etc.), a packet can be seen as a predefined set of variables whose valuation depend on the content of the inspected packet. In the PCM the packets checked against the PCS can therefore be defined in terms of such variables:

**Definition 4 (Packet Variables)**
*A finite non-empty set of bounded variables PV contained within all packets assumed to belong to streams of the type being inspected. The value of a packet variable $v \in PV$ as stored in the packet p is denoted pv(v,p).*

A natural consequence of this definition is that only packets with a valuation for each packet variable, can ever be presented to the PCS. As will later be clear this is an essential property as it ensures that all packets, presented to the PCS and legal within the framework of the PCM, can be successfully inspected.

#### 3.1.1.1 Streams

The second influencing element in the environment is the streams. As previously defined, these are sequences of packets logically bound together at each host using a relation defined by one of the protocols of the stream. In the PCM this relation is modeled by the abstract concept of the *stream key* which is defined as follows:

**Definition 5 (Stream Key)**
*The relation used to relate packets belonging to the same stream to each other.*

In tune with the previous description of SI, it is in the PCM the responsibility of the SI system, using the stream key, to investigate any received packet and find the stream to which it is seems to belong. Upon completion of this task it is then checked against the PCS and an inspection result is identified.

### 3.1.2 The Protocol Conformance Specification

The second part of the PCM is the model of the PCS itself. As previously described, this part can be modeled using an automaton where state change can occur either as the result of a packet being received, or as the result of a timeout. In the former case a result is returned to the firewall core signifying the outcome of the inspection. Depending on the final classification performed by the firewall two things can happen. If the packet is allowed to pass, the state information for that stream is updated to reflect the reception of that packet. On the other hand, if the packet is blocked, no updates occur and the packet is simply ignored. To model this behavior a number of different entities need to be incorporated into the automaton of the PCM. The following sections describe and define these entities.

#### 3.1.2.1 Locations and Transitions

Being an automaton the PCS, as modeled in the PCM, consists of a number of locations and transitions. The locations (of the set L) model the basic states in which the inspected streams can be. As was the case for the sample PCSs described in Chapter 2, two different types of locations exists. The first location in any PCS is the *closed* location which represents the state where no packets have been received and therefore no information is stored. Similarly, the *open* locations are the locations representing the intermediate states where information is stored.

Concerning transitions two types, with differing semantics, exists - *update transitions* and *ignore transitions*. The update transitions (UT) are the traditional transitions used whenever the final acceptance of a packet means that the state information needs to be updated. Ignore transitions (IT) on the other hand are self-loops used whenever the acceptance of a packet must not lead to that information being updated (at the very least an update transition will reset the timeout timer). A scenario where ignore transitions are useful is in situations where an invalid packet has been received. In the event that the firewall, regardless of this, chooses to accept the packet it could be useful for SI system to simply ignore the packet and assume that it will be ignored by the receiving host. Had a traditional update transition been used, essential information such as the timeout timer would have been updated and no longer been consistent with the stream.

#### 3.1.2.2 State Information

To keep track of the state of a stream in-between inspections, state information needs to be stored. In the PCM this capability is made possible through the introduction of a number of *stored variables*. These variables are defined as follows:

**Definition 6 (Stored Variables)**
*A finite set of bounded variables SV stored by the SI system for each stream being inspected using the particular PCS. The value of a stored variable $v \in SV$ is denoted sv(v).*

#### 3.1.2.3 Constraints on Transitions

An essential part in the functionality of a PCS is the ability to vary the result of an inspection based upon the contents of the received packets. In the PCM this functionality is implemented by allowing for a number of constraints to be placed on the transitions of the automaton. As firewalls, by their very nature assume the role of intermediary observers, two different types of constraints are needed - *direction constraints* and *content constraints*. To simplify the final system, while at the same time stressing that direction and content constraints, at least conceptually, are two different types of constraints, these are kept as separate entities in the model. The first type of constraints, the direction constraints, are defined as follows:

**Definition 7 (Direction Constraints)**
*Constraints on the direction of the received packet. Being assigned to all transitions not emerging from the closed location, the set of possible directions is denoted $D$ and the direction constraint associated with a transition $t \in UT \cup IT$ is denoted dc(t). Similarly the direction of the packet p is denoted dir(p).*

Similarly, the constraints on the content of the packet, referred to as the *guards*, are defined as follows:

**Definition 8 (Guards)**
*Constraints over the stored and packet variables $G \subseteq SV \cup PV$, assigned to a transition $t \in UT \cup IT$ and obeying the following rules: Letting emit($\chi$) denote the location from where the transition $\chi \in UT \cup IT$ is going out, it must for the guards grd(t) associated with the transition t, never be the case that $\exists t' \in UT \cup IT$ where emit(t)=emit(t') and dir(t)=dir(t') and where the following holds:*

$$\models grd(t) \cap \models grd(t') \neq \{\varnothing\}$$

*where $\models \rho$ denotes the set of all tuples over the values of the stored and packet variables that satisfy the guards $\rho \in G$. Finally, there must for any packet p, from all locations L and directions D, be a transition whose guards are satisfied by the values of the packet variables of the packet.*

As can be seen from this definition all guards must adhere to two basic rules. First of all it must never be the case that the guards assigned to two transitions cause these transitions overlap. If this property was to fail the PCS would become non-deterministic and unable to return a distinct result to the firewall core. Secondly, there must always be an enabled transition for any given packet that, by the stream key, is considered part of the stream. As the result returned to the firewall core depends entirely on which transition is taken, the PCS could, without this requirement, be presented with a packet to which it has no response.

#### 3.1.2.4 Updating the State Information

Upon taking a transition it must be possible to update the state information stored for the stream. To fulfill this requirement variable assignments can be placed on update transitions throughout the PCS. This way, when an update transition is traversed, the stored variables can be updated to reflect this. To ease the formalization of the model, the assignments on all transitions going to the closed location are predefined to assign to the stored variables, their initial value. More specifically these assignments, referred to as *updates*, are defined as follows:

**Definition 9 (Updates)**
*Assignments to the stored variables. Associated with update transitions, the updates are performed whenever the transition to which they are assigned is taken. The set of possible updates is denoted $U$ and finally, the updates associated with transitions going to the closed location are predefined to reset all stored variables to their initial values.*

#### 3.1.2.5 Specifying the Inspection Result

The next concept in the model is that of *property values*. As previously described these are used to specify the result to be returned to the firewall core. In the PCM this is done by assigning a property value to each transition and returning it whenever that transition is taken. In the PCM the property value is defined as follows:

**Definition 10 (Property Value)**
*A value assigned to each transition $t \in UT \cup IT$ and returned to the firewall core whenever that transition is taken. The set of property values is denoted $V$.*

#### 3.1.2.6 Timeouts

The final part of the automaton is the functionality used to model timeouts. For this, two component are needed - the *clock* and the *timeout value*. The clock, which is used to keep track of the time elapsed since the reception of the last accepted packet, is defined as follows:

**Definition 11 (Clock)**
*A variable $C$, ranging over $\mathbb{Z}^*$, whose value $v(C)$ is incremented by 1 each time a second passes and the stream is in an open location.*

The timeout value, that is, the amount of time allowed to pass before a timeout occurs is a value assigned to each open location. More formally it is defined as follows:

**Definition 12 (Timeout Value)**
*A value $x \in \mathbb{Z}^+$ assigned to each open location. Given a location, this value defines the amount of time that may elapse between the reception of two accepted packets.*

With the definitions of the individual components in place the final automaton, named the *protocol conformance automaton*, can be formally defined. In Definition 13 this formal definition can be seen:

**Definition 13 (Protocol Conformance Automaton)**
*A protocol conformance automaton (PCA) is a tuple ($L$, $l_0$, $tval$, $C$, $UT$, $IT$, $dc$, $grd$, $upd$, $pval$) where:*

$L$, *is a finite non-empty set of locations.*

$l_0 \in L$, *is the closed location that is used when no state information is stored.*

$tval : L \setminus \{l_0\} \rightarrow \mathbb{Z}^+$, *is a function which labels each open location with a timeout value.*

$C$, *is a clock.*

$UT \subseteq L \times L \setminus \{(l_0, l_0)\}$, *is a set of update transitions.*

$IT \subseteq L \times L$, *is a set of ignore transitions where for any two connected locations $l, l' \in L$ then $l = l'$.*

$dc : (UT \cup IT) \setminus \{l_0\} \times L \rightarrow D$, *is a function which labels each transition not going from $l_0$ with a direction constraint.*

$grd : UT \cup IT \rightarrow G$, *is a function which assigns to each transition a number of guards of the type, and obeying the rules defined in Definition 8.*

$upd : UT \rightarrow U$, *is a function which assigns to each update transition not going to $l_0$ a number of updates.*

*$pval : UT \cup IT \rightarrow V$, is a function which assigns a property value to each transition.*

Having defined the environment and the syntax of the PCA, the semantics of this automaton can now be defined.

### 3.1.3 Semantics of Protocol Conformance Automata

The semantics of the PCA, and thus the behavior of a PCS, is defined in the form of a transition system $(S, s_0, \rightarrow)$. In this system $S$ is a set of states where each state is a triple $(l, v, t)$ with $l$ being a location, $v$ a valuation of the stored variables, and $t$ a valuation of the clock. $s_0$ is the initial state $(l_0, v_0, t_0)$, where the clock and all stored variables are zero. Finally, $\rightarrow$ is the transition relation defining how to move between the states. To capture the differences in the transitions incurred by the absence of directions on transitions emerging from the closed location, and the absence of a timeout value on the closed location, the transition relation defines two different types of update and ignore transitions. The *open* transitions are transitions not emerging from closed location, whereas the *closed* transitions all emerge from this location. Specifically, the transition system underlying a PCA is defined as follows, where $\models$ is a satisfaction relation between a valuation of packet variables, stored variables, and the set of guards $G$:

**Definition 14 (Transition System Underlying a PCA)**
*The transition system associated with the protocol conformance automaton A, denoted M(A) is defined as $(S, s_0, \rightarrow)$ where:*

$S = \{(l, v, t) \in (L \setminus \{l_0\}) \times sv(SV) \times v(C) \mid t \leq tval(l)\}$

$s_0 = (l_0, v_0, t_0)$ *where* $t_0 = 0$ *and* $v_0(x) = 0$ *for all* $x \in SV$

*the transition relation* $\rightarrow \subseteq S \times (\{uo, uc, io, ic\} \times V \cup \{t\} \cup \mathbb{Z}^+) \times S$ *is defined by the rules :*

*1* $\binom{update}{open}$ : *(l,v,t)* $\xrightarrow{uo,\alpha}$ *(l',v',0) if the following conditions hold:*

*a. e = (l,l') $\in$ UT and $l \neq l_0$*

*b. a packet p is received*

*c. pv(PV,p), v $\models$ grd(e)*

*d. the return of the property value $\alpha \in V$ associated with the transition e, will allow the packet to pass through the firewall*

*e. $v'$ is the valuation of $SV$ after applying upd(e) to v*

*f. t < tval(l)*

*g. dir(p) = dc(e)*

*2* $\binom{update}{closed}$ : $(l_0,v_0,t_0) \xrightarrow{uc,\alpha} (l',v',t')$ *if the following conditions hold:*

*a.* $(l_0,l') \in UT$

*b. rules 1.b, 1.c, 1.d and 1.e hold*

*2* $\binom{ignore}{open}$ : $(l,v,t) \xrightarrow{io,\alpha} (l,v,t)$ *if the following conditions hold:*

*a.* $(l,l) \in IT$ *and* $l \neq l_0$

*b. rules 1.b, 1.c, 1.d, 1.f, and 1.g hold*

*2* $\binom{ignore}{closed}$ : $(l_0,v_0,t_0) \xrightarrow{ic,\alpha} (l_0,v_0,t_0)$ *if the following conditions hold:*

*a.* $(l_0,l_0) \in IT$

*b. rules 1.b, 1.c, and 1.d hold*

*3 (timeout):* $(l,v,t) \xrightarrow{t} (l_0,v_0,t_0)$ *if the following conditions hold:*

*a.* $l \neq l_0$

*b.* $t = tval(l)$

*4 (delay):* $(l,v,t) \xrightarrow{d} (l,v,t+d)$ *for any positive integer d, if the following conditions hold:*

*a.* $l \neq l_0$

*b.* $t+d \leq tval(l)$

*c. no packet is received*

### 3.1.4 Depicting the Protocol Conformance Model

For depicting the protocol conformance model we use the following conventions. With regards to the PCA, circles denote locations and timeout values are written inside these circles. Update transitions are represented using arrows and ignore transitions are denoted using dotted arrows. The guards and updates associated with the update transitions are written above or to the right of the arrows where as direction constraints and property values are written below or to the left. Furthermore, the closed location is depicted using a double lined circle. Finally, for the sake of clarity the different locations are given a name which is written inside the circle.

An example of this graphical representation of the PCM can be seen in Figure 3.1. In this example a stream can get into location $A$ if a packet satisfying $guard_1$ is received, and no information about the stream of the packet is stored. From there two things can happen: either a packet satisfying $guard_2$, $guard_3$, or $guard_4$ and have the appropriate direction is received within 60 seconds, or else a timeout occurs. In the former case where a packet satisfies $guard_2$, the stream remains in location $A$, the stored information is updated according to $update_2$, and the clock is reset. In the case where $guard_4$ is sat-

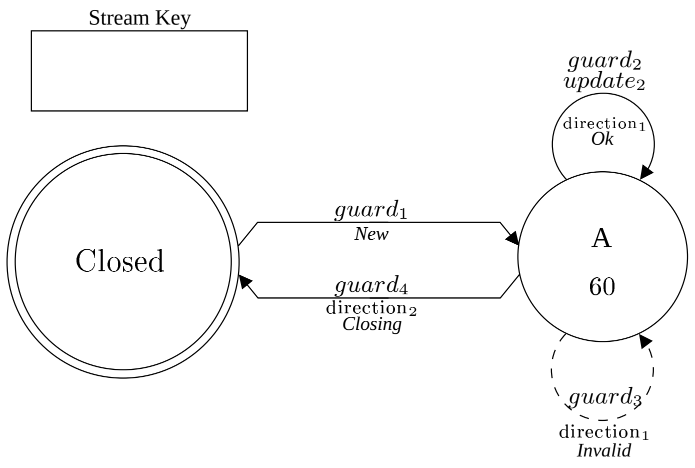


**Figure 3.1.** Graphical representation of a PCS specified in the PCM.

isfied, the stream enters the closed location and all information about it is deleted. Similarly, in the case where $guard_3$ is satisfied by a packet traveling in $direction_1$, the property value *Invalid* is returned and all information, including the clock, is left unchanged. Finally, in the latter case where no packets are received, the stream times out and effectively enters the closed state where the stored information as well as the clock is reset.

Having defined the abstract version of the underlying model it can now be specialized towards the towards the task of representing PCSs for TCP and UDP streams. Most notably, this means that abstract concepts such as packet variables, stream key, and guards must be specialized so that they represent what is needed by these protocols.

## 3.2 The TCP/UDP Specialization

The parts of the PCM which need to be specialized for it to be capable of representing PCSs for TCP and UDP streams are *packet variables*, *stream key*, *stored variables*, *direction constraints*, *guards*, and *updates*. An example of the need for a specialization can be seen in the definition of guards. In the PCM a guard is defined merely as a constraint over the set of stored and packet variables. But what does this mean in terms of representing PCSs for TCP and UDP streams?. That is, what must be possible to use as guards when creating PCSs for these streams. As the answer to this question, along with similar questions for the other parts, depends on the protocol being inspected, and on what the PCS is supposed to check, this question has no exact and final answer. In an attempt to overcome this problem a wide range of existing PCSs for these protocols have therefore been investigated. Using the knowledge gained from this investigation, the PCM has then be specialized to a point where it is sufficiently concrete to be used a the basis for the universal language, while at the same time still expressive enough to represent the investigated PCSs. Through the examination of PCSs that are complicated and in widespread use, we thereby hope to be able to represent most, if not all, PCSs one may want to make.

In the following section the reference PCS used as a basis for the specialization will be briefly described[2]. Following this description we conclude on the requirements of that PCS and formally define the specialized model.

### 3.2.1 The Reference Protocol Conformance Specification

The PCS used as a reference for what must be representable by the specialization is a PCS which checks TCP streams for their correct use of flags and sequence numbers[Roo]. The reason for using this as the reference is that it appears to be the most comprehensive, while at the same time being one of the most widely used PCSs around[3]. The sequence number part essentially places an upper and a lower bound on the $TCP\_SEQ$ and $TCP\_ACKSEQ$ fields. That is, the values of these fields must always fall within the window defined by these bounds.

In an environment where packets are sent between hosts $A$ and $B$ and the firewall $F$ is placed in between, the reference PCS defines the upper bound on the TCP_SEQ field as follows:

$$TCP\_SEQ_A + TCP\_LEN_A \leq max\left\{TCP\_ACKSEQ_B + max(TCP\_WINSIZE_B, 1)\right\} \tag{3.1}$$

where the notation $X_Y$ denotes the value of the variable $X$ in a packet sent by $Y$ and seen by $F$. Using that notation the constraint signifies that the sum of the TCP_SEQ and TCP_LEN fields in any packet sent by A, must never exceed the maximum value of the sum of the TCP_SEQ and TCP_ACK fields from packets sent by B and seen by F. Finally, the term $max(TCP\_WIN_B, 1)$ denotes the maximum value of the two arguments and is used in the special case where the window of host $B$ needs to be re-probed after its annunciation of a zero sized window. In the same environment, and using the same notation, the lower bound for the TCP_SEQ field is defined as follows:

$$TCP\_SEQ_A + max\left\{max(TCP\_WIN_B, 1)\right\} \geq max\left\{TCP\_SEQ_A + TCP\_LEN_A\right\} \tag{3.2}$$

Similarly, with respect to the TCP_ACKSEQ field, the upper bound is defined as follows:

$$TCP\_ACKSEQ_A \leq max\left\{TCP\_SEQ + TCP\_LEN\right\} \tag{3.3}$$

[2]Several other PCSs for both TCP and UDP streams have been investigated but none brought about any additional requirements.

[3]Most open source firewalls (PF, Netfilter, IPFilter etc.) implement it.

Finally, the lower bound for the TCP_ACKSEQ field is defined as follows:

$$TCP\_ACKSEQ_A + MAXACKWINDOW \geq max\Big\{TCP\_SEQ_B + TCP\_LEN_B\Big\} \quad (3.4)$$

where MAXACKWINDOW is as a user-defined constant slightly larger than the largest possible TCP window size.

To implement the 4 constraint the reference PCS proposes to use 3 stored variables for each of the two hosts. These are: $X.td\_end$, $X.td\_maxend$, and $X.td\_maxwin$ where X denotes either of the two hosts. $X.td\_end$ is used to hold the maximum value of $max\{TCP\_SEQ_X + TCP\_LEN_X\}$ as used in constraints 3.2, 3.3, and 3.4 where as $X.td\_maxend$ holds the value of $TCP\_ACKSEQ_X + max\{TCP\_WIN_X, 1\}$ as used by constraint 3.1. Finally, $X.td\_maxwin$ is used to hold the value of $max\{max(TCP\_WIN_X, 1)\}$. With regards to the flag checking part of the PCS, it simply enforces the rules defined in RFC793[Pos81b], and is therefore a superset of the PCS previously shown in Figure 2.3(b).

## 3.2.2 The Specialized Protocol Conformance Model

From the reference PCS several things becomes clear with regards to the abstract concepts which need to be specialized. First of all, it is clear that the set of packet variables should be defined as set of all fields present in the headers of the inspected protocols. The reason for this is obvious as all information used by these protocols is present within the protocol headers. Secondly, it should be noted that the TCP protocol makes use of two types of header fields - *normal fields* (e.g. TCP_WIN and TCP_SYN) and *sequence number fields* (TCP_SEQ and TCP_ACKSEQ). The normal fields are normal bounded variables where as the sequence number fields are used for simulating unbounded behavior in the otherwise bounded variables[EB96]. As the semantics of sequence number arithmetics (as prescribed by RFC1982) requires these two types to be kept separate, the PCMs definition of packet variables can be specialized to the following:

**Definition 15 (Packet Variables for TCP and UDP Streams)** *The set of packet variables PV for TCP and UDP streams is the set of fields in the protocol headers of the packets of the type of stream for which the PCS was made. The set PV is divided into two subsets, PSV which is the set of sequence number fields and PNV which is the set of normal fields. For these sets the following must hold:* $PSV \cup PNV = PV$ *and* $PSV \cap PNV = \{\varnothing\}$

Concerning the specialization of the stream key, TCP and UDP relate packets by matching the contents of a few predefined fields. If two packets share the same content in these fields they are said to belong to the same stream. To reflect this, the stream key for TCP and UDP streams is defined in terms of a pair of tuples capable of holding these predefined fields.

**Definition 16 (Stream Key for TCP and UDP Streams)**
*A pair of ordered n-tuples* $a = (a_1, a_2, \ldots, a_n)$ *and* $b = (b_1, b_2, \ldots, b_n)$ *over the set of packet variables* $PV$. *Two packets* $p$ *and* $p'$ *are considered to belong to the same stream if either* $pv(a_i, p) = pv(a_i, p') \wedge pv(b_i, p) = pv(b_i, p')$ *or* $pv(a_i, p) = pv(b_i, p') \wedge pv(b_i, p) = pv(a_i, p')$, *for* $i = 1, 2, \ldots, n$.

With regards to stored variables the reference PCS stores either the values of the fields, or the result of an expression over the set of packet and stored variables. As a packet variable is now defined as a set of fields and as a field is essentially a bounded variable over the domain $\mathbb{Z}^*$ with a lower bound 0, and an upper bound of $2^{\#bits} - 1$, the definition of stored variables can be specialized to the following:

**Definition 17 (Stored Variables for TCP and UDP Streams)**
*The set of stored variables* $SV$ *for TCP and UDP streams is a set of bounded variables over* $\mathbb{Z}^*$. *The set* $SV$ *is divided into two subsets, where* $SNV$ *is the set of stored normal variables and* $SSV$ *is the set of stored sequence number variables. Letting ub(a) denote the upper bound for the variable* $a \in SNV \cup SSV$, *then for any variable* $x \in SNV$ *the following must hold:* $ub(x) = 2^i - 1$ *for* $i \in \{1, 2, \ldots, 32\}$. *Similarly, for any variable* $y \in SSV$ *the following must hold:* $ub(y) = 2^i - 1$ *for* $i \in \{8, 16, 32\}$. *Finally, for the sets* $SNV$ *and* $SSV$ *the following must hold:* $SSV \cup SNV = SV$ *and* $SSV \cap SNV = \{\varnothing\}$

With regards to the direction constraints for TCP and UDP streams the direction of packets in these streams are seen relative to the first packet of the stream to which they belong. Furthermore, the direction depends on the values stored within the fields making up the stream key. In tune with the definition of the stream key for TCP and UDP streams, this type of constraint is therefore specialized to the following:

**Definition 18 (Direction Constraints for TCP and UDP Streams)**
*Either **original** or **return**, the direction of a TCP or UDP packet is seen relative to the first packet of the stream. A received packet* $p$ *is said to be flowing in the **original** direction if for the first packet of the stream* $p'$ *the following holds with regards to the tuples* $a$ *and* $b$ *in the stream key:* $pv(a_i, p) = pv(a_i, p') \wedge pv(b_i, p) = pv(b_i, p')$, *for* $i = 1, 2, \ldots, n$. *Similarly a packet is flowing in the **return** direction if the following holds:* $pv(a_i, p) = pv(b_i, p') \wedge pv(b_i, p) = pv(a_i, p')$, *for* $i = 1, 2, \ldots, n$.

In the PCM the set of guards is defined as a set of constraints over the set of packet variables and stored variables. With the specialization of these two sets this abstract concept can be specialized as well.

As explained in the description of the reference PCS, this PCS specifies a window for the sequence numbers. It therefore uses guards that are boolean expressions over expressions on the members of the sets of packet variables, stored variables, and positive integer constants. Based on this, and RFC1982s definition of sequence number arithmetics, we specialize the set of guards to the following:

**Definition 19 (Guards for TCP and UDP Streams)**
*For the set NEXP of expressions over the sets $PNV \cup SNV$, and for the set SEXP of expressions over the sets $PSV \cup SSV$, the set of guards for TCP and UDP streams is defined according to the following grammar:*

$$G ::= G_1 \wedge G_2 \mid a \sim b \mid x \sim y \mid true$$

**where** $a, b \in NEXP$, $x, y \in SEXP$, *and* $\sim \in \{<, \leq, >, \geq, =\}$

where *true* represents the guard that is always true. Furthermore, with an outset in the reference PCS, we define the set NEXP to be the set of expressions over normal variables using the common operators. More specifically, we define it as follows:

**Definition 20 (Set of Expressions Over Normal Variables)**
*The set $NEXP$ of expressions over normal variables is defined according to the following grammar:*

$$\phi ::= d \mid a \mid b \mid (\phi) \mid (z)(\phi_1 \sim \phi_2)$$

**where** $d \in \mathbb{Z}^+$, $z \in \{1, 2, \ldots, 32\}$, $a \in SNV$, $b \in PNV$, *and* $\sim \in \{+, -, *, /, //\}$

Important to notice from this definition is the structure of the expressions over the basic elements, *positive integer values*, *packet variables*, and *stored variables*. As can be seen, expressions are augmented with a bit value ($z$) specifying the size of the domain in which the evaluation of the expression is to be made. This is especially useful when operating with expressions over different sized variables. Later, in Definition 22, the exact meaning of this bit value will be defined.

Finally, in terms of the set $SEXP$ of expressions over sequence number variables we define this in accordance with the rules of RFC1982:

**Definition 21 (Set of Expressions Over Sequence Number Variables)** *The set $SEXP$ of expressions over sequence number variables is defined according to the following grammar:*

$$\phi ::= \ x \mid y \mid x + a \mid y + a$$

**where** $x \in SSV$, $y \in PSV$, *and* $a \in NEXP$

With regards to the semantics of these two sets, the natural semantics of expressions over normal variables is defined as follows:

**Definition 22 (Evaluation of Expressions over Normal Variables)** *The natural semantics for the evaluation of expressions over normal variables is defined as follows:*

$[num]$ $s \vdash d \to_a n$ ***where*** $n = \mathcal{N}[\![d]\!]$

$[var_{sv}]$ $s \vdash a \to_a n$ ***where*** $n = sv(a)$

$[var_{pv}]$ $s \vdash b \to_a n$ ***where*** $n = pv(b, p)$ *and p is the packet being inspected*

$[parent]$ $\frac{s\vdash\phi\to_a v}{s\vdash(\phi)\to_a v}$

$[add]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3}{s\vdash(z)(\phi_1+\phi_2)\to_a v}$ ***where*** $v = (v_1 + v_2)\ mod\ 2^{v_3}$

$[mult]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3}{s\vdash(z)(\phi_1*\phi_2)\to_a v}$ ***where*** $v = (v_1 * v_2)\ mod\ 2^{v_3}$

$[divf]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3}{s\vdash(z)(\phi_1/\phi_2)\to_a v}$ ***where*** $v = \lfloor \frac{v_1}{v_2} \rfloor mod\ 2^{v_3}$

$[divc]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3}{s\vdash(z)(\phi_1//\phi_2)\to_a v}$ ***where*** $v = \lceil \frac{v_1}{v_2} \rceil\ mod\ 2^{v_3}$

$[sub_1]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3 \quad v_1\geq v_2\to_g tt}{s\vdash(z)(\phi_1-\phi_2)\to_a v}$ ***where*** $v = (v_1 - v_2)\ mod\ 2^{v_3}$

$[sub_2]$ $\frac{s\vdash\phi_1\to_a v_1 \quad s\vdash\phi_2\to_a v_2 \quad \vdash z\to_a v_3 \quad v_1\geq v_2\to_g ff}{s\vdash(z)(\phi_1-\phi_2)\to_a v}$ ***where*** $v = 2^{v_3} - v_2 + 2v_1$

Similarly, based upon RFC1982 the semantics of sequence number expressions is defined as follows:

**Definition 23 (Evaluation of Sequence Number Expressions)** *Letting $size(a)$ denote the number of bits assigned to the variable $a \in PSV \cup SSV$, the natural semantics for the evaluation of sequence number expressions from the set $SEXP$ is as follows:*

$[var_{sv}]$ $s \vdash x \to_x n$ ***where*** $n = sv(x)$

$[var_{pv}]$ $s \vdash y \rightarrow_x n$ ***where*** $n = pv(y,p)$ *and p is the packet being inspected*

$[add_s]$ $\frac{s \vdash x \rightarrow_x v_1 \quad s \vdash a \rightarrow_a v_2}{s \vdash x + a \rightarrow_x v}$ ***where*** $2^{size(x)-1} > v_2$

***and*** $v = (v_1 + v_2) \; mod \; 2^{size(x)}$

$[add_p]$ $\frac{s \vdash y \rightarrow_x v_1 \quad s \vdash a \rightarrow_a v_2}{s \vdash y + a \rightarrow_x v}$ ***where*** $2^{size(y)-1} > v_2$

***and*** $v = (v_1 + v_2) \; mod \; 2^{size(y)}$

With the syntax and semantics of the elements making up a guard in place, the semantics for the evaluation of guards is defined as follows:

**Definition 24 (Evaluation of Guards for TCP and UDP Streams)** *Letting size(a) denote the number of bits assigned to the variable $a \in PSV \cup SSV$, the natural semantics for the evaluation of guards for TCP and UDP streams is as follows:*

$[and_t]$ $\frac{s \vdash G_1 \rightarrow_g tt \quad s \vdash G_2 \rightarrow_g tt}{s \vdash G_1 \wedge G_2 \rightarrow_g tt}$

$[and_f]$ $\frac{s \vdash G_i \rightarrow_g ff}{s \vdash G_1 \wedge G_2 \rightarrow_g ff}$ ***where*** $i \in \{1, 2\}$

$[gt_{nt}]$ $\frac{s \vdash a \rightarrow_a v_1 \quad s \vdash b \rightarrow_a v_2}{s \vdash a > b \rightarrow_g tt}$ ***where*** $v_1 > v_2$

$[gt_{nf}]$ $\frac{s \vdash a \rightarrow_a v_1 \quad s \vdash b \rightarrow_a v_2}{s \vdash a > b \rightarrow_g ff}$ ***where*** $v_1 \leq v_2$

$[eq_{nt}]$ $\frac{s \vdash a \rightarrow_a v_1 \quad s \vdash b \rightarrow_a v_2}{s \vdash a = b \rightarrow_g tt}$ ***where*** $v_1 = v_2$

$[eq_{nf}]$ $\frac{s \vdash a \rightarrow_a v_1 \quad s \vdash b \rightarrow_a v_2}{s \vdash a = b \rightarrow_g ff}$ ***where*** $v_1 \neq v_2$

$[gt_{st}]$ $\frac{s \vdash x \rightarrow_x v_1 \quad s \vdash y \rightarrow_x v_2}{s \vdash x > y \rightarrow_g tt}$ ***where*** $size(x) = size(y)$

***and*** $\begin{array}{c} v_1 < v_2 \;\wedge\; v_2 - v_1 \geq 2^{size(x)-1} \\ \vee \\ v_1 > v_2 \;\wedge\; v_1 - v_2 \leq 2^{size(x)-1} \end{array}$

$[gt_{sf}]$ $\frac{s \vdash x \rightarrow_x v_1 \quad s \vdash y \rightarrow_x v_2}{s \vdash x > y \rightarrow_g ff}$ ***where*** $size(x) = size(y)$

***and*** $\neg \left( \begin{array}{c} v_1 < v_2 \;\wedge\; v_2 - v_1 \geq 2^{size(x)-1} \\ \vee \\ v_1 > v_2 \;\wedge\; v_1 - v_2 \leq 2^{size(x)-1} \end{array} \right)$

$[eq_{st}]$ $\frac{s \vdash a \rightarrow_x v_1 \quad s \vdash b \rightarrow_x v_2}{s \vdash a = b \rightarrow_g tt}$ ***where*** $v_1 = v_2$

$[eq_{sf}]$ $\frac{s \vdash a \rightarrow_x v_1 \quad s \vdash b \rightarrow_x v_2}{s \vdash a = b \rightarrow_g ff}$ ***where*** $v_1 \neq v_2$

$[true]$ $s \vdash true \rightarrow_g tt$

Important to note from this definition is the slight difference in the semantics of the comparison of sequence number variables as opposed to what is prescribed by RFC1982. Where the RFC leaves it free for any implementation to decide on the outcome of comparisons between values where $x - y = 2^{size(x)-1}$, the above semantics defines such comparisons to *true*. This alternate definition is needed as the original definition would introduce ambiguity into the final model. The new definition has been chosen as it matches the semantics of the implementation currently used by both Linux and OpenBSD.
Finally, with regards to updates, building upon the separation of normal and sequence number variables, we specialize them to the following:

**Definition 25 (Updates for TCP and UDP Streams)**
*The set of updates U for TCP and UDP streams is defined as generated by the following grammar:*

$$U ::= \; a := b \mid x := y \mid U_1 U_2$$

**where** $a \in SNV, b \in NEXP, x \in SSV, \;\; and \;\; y \in SEXP$

where their semantics is defined as follows:

**Definition 26 (Evaluation of Updates for TCP and UDP Streams)**
*The set of updates U for TCP and UDP streams is defined as generated by the following grammar:*

$$[ass_{norm}] \; \langle a := b, s\rangle \rightarrow s[a \mapsto v] \; \textbf{\textit{where}} \; s \vdash b \rightarrow_a v$$

$$[ass_{seq}] \; \langle x := y, s\rangle \rightarrow s[x \mapsto v] \; \textbf{\textit{where}} \; s \vdash y \rightarrow_x v$$

$$[comp] \; \frac{\langle U_1, s\rangle \rightarrow s'' \qquad \langle U_2, s''\rangle \rightarrow s'}{\langle U_1 U_2, s\rangle \rightarrow s'}$$

This concludes the development of the model underlying the proposed system. With that in place, the stream oriented and firewall independent language capable of representing the specialized model can now be devised.

Chapter 4

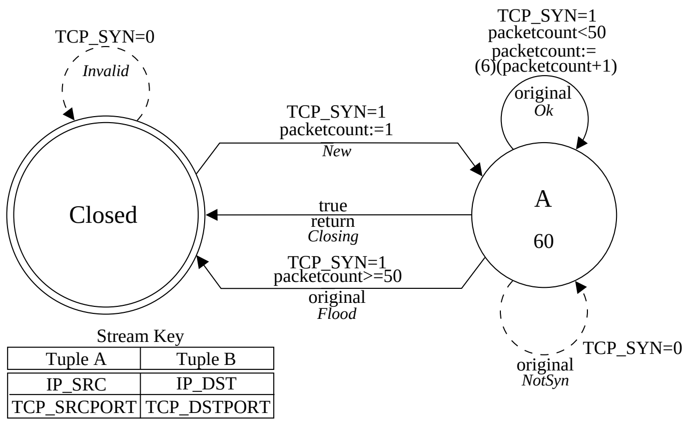
# The Protocol Oriented Language

Having concluded on the requirements for the description of PCSs for TCP and UDP streams, and having devised a model encapsulating those requirements, the protocol oriented language can now be created. In the following section that language, named PCSL for *Protocol Conformance Specification Language*, will be described.

## 4.1 The PCSL Language

The PCSL language is a strongly typed, declarative language inspired by the *ta* language[LPY97] used by the formal verification tool UppAal to describe an extended version of timed automata. This chapter will describe the PCSL language through the use of a simple example. Included in that example will be a graphical illustration of a PCS and the corresponding PCSL code describing it. From this example the abstract syntax of the different constructs in the language, and how they fit into the previously described model, will be described[1]. With regards to the formal semantics of the language we refer to the definition of the specialized PCM as the semantics of all significant constructs are already defined there. As a result the language is merely a way of describing these constructs in a textual way, and formally defining the semantics of the complete language would therefore be superfluous. With that said, the graphical representation of the example PCS used throughout this chapter can be seen in Figure 4.1. Similarly, the corresponding PCSL code used to describe this PCS can be seen in Table 4.1. As can be seen from the figure, this rather naive PCS is capable of detecting SYN floods against a single non-existent host. More specifically, if more than 50 SYN packets are sent through the firewall within

[1]For a full listing of the concrete and abstract syntax of the PCSL language as well as a formal description of its type system, see Appendix B.



| Tuple A | Tuple B |
|---|---|
| IP_SRC | IP_DST |
| TCP_SRCPORT | TCP_DSTPORT |

**Figure 4.1.** The PCS generated by the PCSL specification in Table 4.1.

```
defpropvalue Invalid;

packetnorm IP_SRC 32;
packetnorm IP_DST 32;
packetnorm TCP_SRCPORT 16;
packetnorm TCP_DSTPORT 16;
packetnorm TCP_SYN 1;

storednorm packetcount 6;

keyitem IP_SRC , IP_DST;
keyitem TCP_SRCPORT , TCP_DSTPORT;

clocation Closed;
olocation A 60;

utrans Closed -> A {
 guard TCP_SYN == 1;
 update packetcount := 1;
 propvalue New;
}

utrans A -> A {
 direction original;
 guard TCP_SYN == 1,
       packetcount < 50;
 update packetcount := (6)(packetcount + 1);
 propvalue Ok;
}

itrans A -> A {
 direction original;
 guard TCP_SYN == 0;
 propvalue NotSyn;
}

utrans A -> Closed {
 direction return;
 propvalue Closing;
}

utrans A -> Closed {
 direction original;
 guard TCP_SYN == 1,
       packetcount >= 50;
 propvalue Flood;
}
```

**Table 4.1.** The PCSL code generating the PCS shown in Figure 4.1.

a timeframe of 60 seconds, and no return packet is received, the property value *Flood* is returned to the firewall core. If on the other hand a reply to one of the packets is received, the *Closing* value is returned and the tracking of the stream is terminated.

### 4.1.1 The Overall Structure of the PCSL Language

As can be seen from the syntax in Table 4.2 a PCSL specification consists of 5 basic parts. These are, *default property value declaration*, *variable declarations*, *stream key declaration*, *location declarations*, and *transition decla-*

*rations.*

| |
|---|
| **1. Syntactic categories** |
| $PCS \in Protocol\ Conformance\ Specifications$ |
| $DD \in Default\ property\ value\ declaration$ |
| $VD \in Variable\ declarations$ |
| $SD \in Stream\ key\ declaration$ |
| $LD \in Location\ declarations$ |
| $TD \in Transition\ declarations$ |
| **2. Definitions** |
| $PCS\ ::=\ DD\ \ VD\ \ SD\ \ LD\ \ TD$ |

**Table 4.2.** The 5 parts of any PCSL specification.

In the following sections, with an outset in the example, the purpose and syntax of each part is described.

### 4.1.2 Default Property Value Declaration

The first part of any PCSL specification (line 1 in Table 4.1) is a construct used to simplify the code needed to describe a specification. As described in the previous chapter, for a PCS to be legal, it must contain an enabled transition for every possible packet to which it may be presented. This way, no matter which packet is received by the firewall, the SI system will always be able to take a transition and thereby return a result. To ease the task of adding these transitions the *default property value* has been introduced. Instead of adding a lot of *ignore transition* covering all the packets that are to be ignored by the firewall, this default value can be used to specify which value must be returned when no transition is enabled. This way, the developer is left with declaring the transitions for the *special cases* and can leave the rest to the default value. An example of the benefits of this construct can be seen in the example PCS (Figure 4.1) where the ignore transition emerging from the *Closed* location is not explicitly declared in the PCSL code. Instead, it is merely a result of the default property value being *Invalid*. This way, the declaration of a lot of trivial transitions can be omitted. The syntax for the specification of the default property value can be seen in Table 4.3.

### 4.1.3 Variable Declarations

The next part in a PCSL specification is the declaration of the variables used throughout the PCS. In the example code this part spans the lines from 3 to 9, and as can be seen from the syntax depicted in Table 4.4, a variable declaration consists of 3 parts. The first part is the *variable type*

**1. Syntactic categories**

$DD \in Default\ property\ value\ declarations$

$p \in Property\ values$

**2. Definitions**

$DD$ ::= **defpropvalue** $p$;

Table 4.3. Abstract syntax for the declaration of the default property value.

**1. Syntactic categories**

$VD \in Variable\ declarations$

$sn \in Stored\ normal\ variables$

$ss \in Stored\ sequence\ number\ variables$

$pn \in Packet\ normal\ fields$

$ps \in Packet\ Sequence\ number\ fields$

$n \in Numerals$

**2. Definitions**

$VD$ ::= **storednorm** $sn$ $n$; | **storedseq** $ss$ $n$; | **packetnorm** $pn$ $n$;
| **packetseq** $ps$ $n$; | $VD_1$ $VD_2$

Table 4.4. Syntax for variable declarations.

where the name of each type should speak for itself. The second part is the *name* of the variable. For *stored variables* the name is, as in other languages, simply a way of identifying the variable for use later in the code. For *packet variables* on the other hand it acts as an interface to the fields in the inspected packets. In Appendix A the name-to-field mapping table used by the system can seen. As an example of this system, when naming a packet variable *TCP_SYN*, it means that this variable must always contain the value of the *SYN* field in the received TCP packets. This way, all information stored in the headers of the packet can be accessed using the appropriate variable names. The final part of the declaration is the number of bits assigned to hold the variable. As defined in the specialized PCM (Definition 15), all variables are bounded variables over $Z^*$ with a lower bound of 0. The upper bound however varies from variable to variable and therefore needs to be declared. An example of such a declaration can be seen in line 5 where the packet variable *TCP_SRCPORT* has been declared with an upper bound of 65535.

### 4.1.4 Stream Key Declaration

The next part, which covers lines 11 and 12, is the declaration of the stream key. The syntax for this part can be seen in Table 4.5. As described in

**1. Syntactic categories**

$SD \in Stream\ key\ declaration$

$pn \in Packet\ normal\ fields$

$ps \in Packet\ sequence\ number\ fields$

$VLIST \in Variable\ lists$

**2. Definitions**

$SD ::= \textbf{keypair}\ VLIST\ \textbf{;} \mid SD_1\ SD_2$
$VLIST ::= pn\ \textbf{,}\ pn \mid ps\ \textbf{,}\ ps$

**Table 4.5.** Syntax for stream key declarations.

the model (Definition 16) the stream key is made up of two tuples of packet variables. As also described in that definition, every member in each tuple is paired with a member in the other tuple, and depending upon how the contents of these pairings match up, the direction of the packet relative to the stream can be determined. In PCSL these two tuples and pairings are specified using the *keypair* construct which is made up of two comma separated parts. In this respect the first part specifies a member in tuple $A$, whereas the second part specifies its corresponding pairing in tuple $B$. Using this syntax it is thereby always assured that the tuples are of the same size, while at the same time assuring for the easy specification of e.g. the two socket pairs used by TCP/IP streams.

### 4.1.5 Location Declarations

The fourth part is the declaration of locations. In the example PCS this part covers lines 14 and 15 and its syntax can be seen in Table 4.6. As can be

**1. Syntactic categories**

$LD \in location\ declarations$

$cl \in Closed\ locations$

$ol \in Open\ locations$

$n \in Numerals$

$CLD \in Closed\ location\ declarations$

$OLD \in Open\ location\ declarations$

**2. Definitions**

$LD ::= CLD;\ OLD;$
$CLD ::= \textbf{clocation}\ cl$
$OLD ::= \textbf{olocation}\ ol\ n \mid OLD_1\ \textbf{;}\ OLD_2$

**Table 4.6.** Syntax for location declarations.

seen from that syntax this part is itself split into separate 2 parts. The first part is for the declaration of the *closed* location and consists of the reserved word *clocation* and a name for that location. The second part is a sequence of declarations of *open* locations with each declaration consisting of the reserved word *olocation* along with a location name and a timeout value. The timeout value corresponds to the timeout value in the PCM and describes how many seconds may pass before a timeout occurs. In the example 2 locations are declared. The closed location is named *Closed*, where as the open location is named *A*. Finally, the timeout value for the open location is 60 seconds.

### 4.1.6 Transition Declarations

The final and most dominating part is the specification of the transitions. This part spans from line 17 onto the end of the example and its syntax can be seen in Table 4.7.

**1. Syntactic categories**

$TD \in Transition\ declarations$
$pn \in Packet\ normal\ fields$
$ps \in Packet\ sequence\ number\ fields$
$sn \in Stored\ normal\ variables$
$ss \in Stored\ sequence\ number\ variables$
$cl \in Closed\ locations$
$ol \in Open\ locations$
$n \in Numerals$
$p \in Property\ values$
$D \in Directions$
$GD \in Guard\ declarations$
$G \in Guards$
$UD \in Update\ declarations$
$U \in Updates$
$PD \in Property\ value\ declatations$
$NEXP \in Normal\ expressions$
$SEXP \in Sequence\ number\ expressions$
$BOP \in Boolean\ operators$
$AOP \in Arithmetic\ operators$

**2. Definitions**

$TD$ ::= **itrans** *cl* -> *cl* $\{GD\ \ PD\}$
| **itrans** *ol* -> *ol* $\{D\ \ GD\ \ PD\}$
| **utrans** *cl* -> *ol* $\{GD\ \ UD\ \ PD\}$
| **utrans** *ol* **->** *ol* $\{D\ \ GD\ \ UD\ \ PD\}$

*continued on the next page*

| *continued from the previous page* | | |
|---|---|---|
| | | $\mid$ **utrans** $ol$ -> $cl$ $\{D\ \ GD\ \ PD\}$ |
| | | $\mid TD_1\ \ TD_2$ |
| $D$ | ::= | **direction original;** $\mid$ **direction return;** |
| $PD$ | ::= | **propvalue** $p$**;** |
| $GD$ | ::= | **guard** $G$**;** $\mid \epsilon$ |
| $G$ | ::= | $NEXP\ BOP\ NEXP \mid SEXP\ BOP\ SEXP \mid G_1$ , $G_2$ |
| $NEXP$ | ::= | $n \mid sn \mid pn \mid (n)(NEXP\ AOP\ NEXP) \mid (NEXP)$ |
| $BOP$ | ::= | $< \mid > \mid <= \mid >= \mid ==$ |
| $SEXP$ | ::= | $ps + NEXP \mid ss + NEXP$ |
| $AOP$ | ::= | $+ \mid - \mid * \mid / \mid //$ |
| $UD$ | ::= | **update** $U$**;** $\mid \epsilon$ |
| $U$ | ::= | $sn$ **:=** $NEXP \mid ss$ **:=**$SEXP \mid U_1$ , $U_2$ |

**Table 4.7.** Syntax for transition declarations.

As can be seen from this syntax the declaration of transitions is fairly straight forward and built around the grammar defined in the previous chapter. In tune with the definitions of the model two types of transitions exists, *update* and *ignore*. To reflect this, every declaration begins with one of the reserved words, **utrans** or **itrans**, signifying the type of the transition. Following this reserved word are the names of the locations that are to be connected and a block of declarations assigning constraints, updates, and a property value to the transition. Furthermore, as can be seen from the example the syntax of the guards and updates is equal to that of the model, with the exception of the *true* guard being left out and represented by leaving out the guard declaration. Finally, to ensure that no two transitions have overlapping constraints, and thereby ensuring that the rules defined in relation to guards in the underlying model are upheld, precedence is given to the transitions in the order they are declared. Specifically, this means that in the event that the reception of a packet causes two transitions to be enabled, the top most transition as declared in the PCSL code is chosen. While it is possible to explicitly check that no transitions are overlapping it is an NP-complete problem and this simple approach of giving precedence is therefore used instead.

This concludes the description of the language capable of representing the functionality of the specialized protocol conformance model. In the next chapter the intermediate representation capable of storing, optimizing, and offering the PCS to the output generators, will be described.

Chapter 5

# The Intermediate Representation

As described in Chapter 2 the role of the intermediate representation is two-fold. First of all it is the place in which firewall independent optimizations are performed, and secondly, it is in charge of providing the output generators with an easy-to-use interface (the output generator API), giving them access to the PCSs. In the first two sections we analyze what is required from the intermediate representation to represent PCSL specifications, and describe how the current implementation of the system meets those requirements. With that in place, Section 5.3 gives a brief introduction to the interface currently offered to the output generators[1].

## 5.1 Requirements to the Intermediate Representation

As should be clear from the previous chapters the transitions are the cornerstones of any PCS. The property value to be returned, the updates to be applied, and the new state of the stream, are all factors determined by the transitions. Because of this, the intermediate representation, and hence the requirements to it, can be split into two parts - the requirements to the representation of *transitions* and the requirements to the representation of *everything else*. With regards to the transitions the intermediate representation should meet the following requirements:

**Should make it easy to find the enabled transition:** A major part in performing SI is the task of determining which transition is enabled by a given packet. To make it easier for the output generators to create an output capable of doing this, the information provided through

[1]For a complete description of the output generator API see the documentation accompanying the current implementation.

the output generator API should be structured in a way that easily accommodates this task. As a result, to ease the implementation of this API, the intermediate representation should store the information regarding the constraints on the transitions, in a way that eases this.

**Should reduce the amount of guard checks needed:** This requirement relates to the fact that the intermediate representation is the place where firewall independent optimizations are performed. As an extension to the last requirement the task of finding the enabled transition should therefore be made as efficient as possible. From a firewall independent point of view, efficiency is mainly influenced by the number of guards which needs to be checked before the right transition is found. As a result, the intermediate representation should optimize the PCS to minimize this number of guard checks.

**Should retain the C like syntax used for guards and updates:** This final requirement relates to how the individual pieces of information in the transitions must be represented. As the vast majority of current firewalls are implemented in C and most output generators therefore are likely to generate C code, the structure of the information given to them by the output generator API should ease the generation of C code. Consequently, the intermediate representation should retain the C-like syntax of the guards and updates in the PCSL language and refrain from compiling them into another syntax.

Finally, concerning the representation of the rest of the PCS only one requirement exist. As most output generators must generate C code, the output generator API should be geared towards supporting this. To ease the task of implementing that API the intermediate representation should therefore store this part of the PCS in a way that simplifies this.

## 5.2 The Current Intermediate Representation

The intermediate representation currently used in the retargetable PCS creation system is fairly simple. The reason for this being that this version of the implementation serves mainly as a test and development platform for the retargetable concept. In terms of the intermediate representation, focus has therefore been put on meeting the requirements concerning the ease of finding the enabled transition and creating a stable output generator API. Because of this, the description provided in this report regarding the structure of the intermediate representation is rather brief.

### 5.2.1 Representing Transitions

Currently transitions are represented using decision diagrams (one for each location and direction) in which the guards associated with the different transitions are encoded. In Figure 5.1 examples of these diagrams, along with the code for the transitions they represent, can be seen. The advantages of

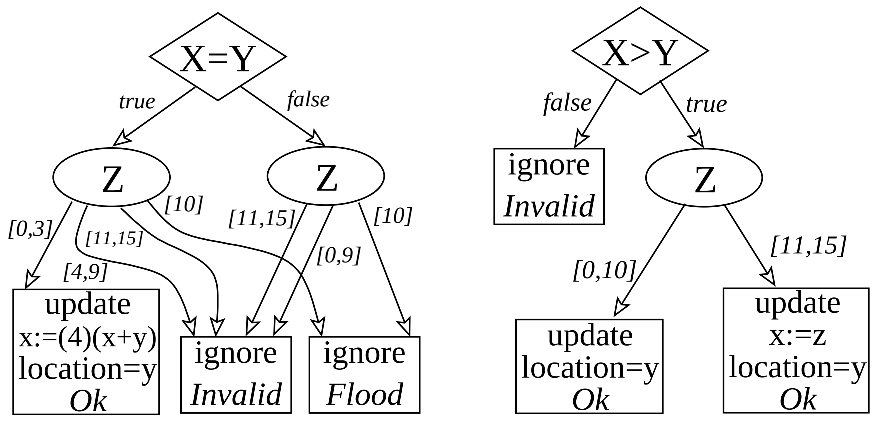
```
defpropvalue Invalid;
...snip...

utrans X -> Y {
 direction original;
 guard x==y, z<4;
 update x:=(4)(x+y);
 propvalue Flood;
}
itrans X -> Y {
 direction original;
 guard z==10;
 propvalue Flood;
}

utrans X -> Y {
 direction return;
 guard x>y, z<=10;
 propvalue Ok;
}

utrans X -> Y {
 direction return;
 guard x>y, z>8;
 update x:=z;
 propvalue Ok;
}
```

(a) The PCSL code generating the PCS shown in Figure 4.1.



(b) The decision diagram representing the *original* transitions in Figure 5.1a.

(c) The decision diagram representing the *return* transitions in Figure 5.1a.

**Figure 5.1.** A partial PCSL specification and the decision diagrams representing its transitions.

using decision diagrams are many. First of all they ease the task of picking the right transition as this becomes merely a matter of traversing a tree. Secondly, as the terminals can hold all the information associated with the transition they represent, generating code for picking the right transitions becomes comparatively simple. As a result, this task amounts to nothing more than traversing the diagram and generating conditional statements for each node. Upon arriving at a terminal the actions associated with that node can be performed and the packet has then been inspected. Using decision diagrams therefore enables the intermediate representation to meet the first requirement described in the previous section. Another reason for using decision diagrams is to be found in the way they ease the task of optimizing the PCS as prescribed by requirement 2. As all transitions applicant to a single packet are encoded in the same diagram, minimizing the number of guard checks amounts to nothing more than reducing the diagram thus potentially reducing its height[2].

#### 5.2.1.1 The Decision Diagram

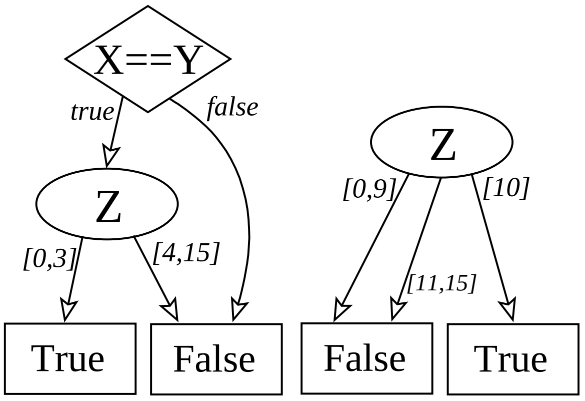
For the purpose of representing the transitions of the PCSL language a new type of decision diagram had to be created. The reason for this being that none of the investigated diagrams could properly handle the potentially overlapping transitions of the PCSL language, while at the same time allowing for the use of (boolean) expressions over variables as the test associated with each non-terminal. The result was a new type of diagram which, as should be clear from Figure 5.1, lends its basic structure from the intermediate representation of BPF+[ABG99] and its nodes from that of BDDs[FMY97] and IDDs[ST98]. The diagram has two distinguishing features. First of all the tests associated with its non-terminals are capable of mirroring the guards of the specialized PCM. This way it meets the aforementioned requirement of being able to retain the syntax of the guards of the PCSL language. Secondly, it incorporates the priorization of transitions defined in PCSL. This allows for the construction of the diagram without having to go through the costly process of identifying and altering any overlapping guards. The result is that the time complexity of the construction process is linear to the number of guard elements being encoded, thus allowing for the fast transformation of PCSs from PCSL code to the final output[3].

**Constructing the Diagram**

The construction of the diagram is a two phase process. In the first phase an intermediate boolean diagram for the guards of each transition to be included in the final diagram is created. These diagrams are constructed using the recursive *CreateID* algorithm outlined in Algorithm 3, and examples of these diagrams can be seen in Figure 5.2. This construction algorithm, which takes as input a list of the guard elements to be represented and returns the corresponding boolean diagram, has three parts. If the input list is empty, all non-terminal nodes have been created and the "true" terminal is returned. If it is not empty, two things can happen depending on the guard element at the head of the input list. Either the element is a boolean comparison of two variables (or expressions containing variables) in which case a *boolean* node with the partitions "true" and "false" is created. On the other hand, if the list head contains a comparison of a variable/expression and an integer

[2]Due to the experimental status of the implementation only a very limited amount of optimization is performed on the diagrams. Well known predicate elimination techniques similar to those described in [ABG99] could however be applied.

[3]It should be noted that the current diagram is very limited in the logic it is able to represent, but that it is perfectly capable of representing that of the guards in the specialized model.



(a) Intermediate diagram for the guard x==y, z<4.

(b) Intermediate diagram for the guard z==10.

**Figure 5.2.** The intermediate boolean diagrams for the diagram depicted in Figure 5.1b.

**Algorithm 3:** Outline of the *CreateID* algorithm

```
Data   : list of guard elements
Result: diagram for guard element list
if empty list then
    return "true" terminal;
if list head contains comparison of two variables then
    node ← boolean node;
    child(node, "true") = CreateID(consecutive list elements);
    child(node, "false") = "false" terminal;
else
    node ← valuation node;
    child(node, satisfied interval) = CreateID(consecutive list elements);
    foreach unsatisfied interval i do
        child(node, i) = "false" terminal;
return node;
```

constant, a *valuation* node with the corresponding intervals as partitions is created.

Upon completing the first phase the different intermediate diagrams need to be merged. This is done using the *Append* algorithm outlined in Algorithm 4. This algorithm takes as input two diagrams (A and B) and merges them, so that in cases of overlaps, the transition(s) of diagram A are

**Algorithm 4:** Outline of the *Append* algorithm

```
Data  : diagram A, diagram B
Result: new diagram
if A is a "false" terminal then
    return B;
if A is a "true" terminal then
    return A;
if A=B then
    Mutually decompose A and B into their greatest common intervals;
    foreach resulting interval i do
        child(A,i) = Append(child(A,i), child(B,i));
else
    foreach interval i of A do
        child(A,i) = Append(child(A,i), B);
return A;
```

preferred. It ensures this property by only attaching the B diagram (or parts of it) to the intervals pointing to the “false” terminals in the A diagram. This way, only if the transitions represented by the first diagram are unsatisfiable, will the transition of the next diagram be considered. An example of this can be seen in Figure 5.1b where diagram B has replaced the “false” terminal pointed to by the $X == Y$ node in diagram A. By merging the diagrams in the order of their priority in the PCSL specification the final diagram can therefore be created.

Having described how the current implementation internally represent the important parts of the PCS, the next section provides a description of the output generator API offered by the intermediate representation.

## 5.3 The Output Generator API

The output generator API is a C interface containing 33 functions split into 5 groups, with each group making available different parts of the PCS. In this section the basic principles behind these groups along with the functions they encompass are described[4].

### 5.3.1 The Initialization Group

The initialization group contains only one function.

[4]For a complete description of the output generator API see the electronic documentation accompanying the current implementation.

```
rpcshandler *procSpec(char *pcs)
```

It takes as input a PCS written in the PCSL language, processes it, and returns to the caller a *rpcshandler* to an instance of the intermediate representation created for that PCS. With an instance of the intermediate representation in place, it is then the purpose of the functions in the rest of API, to give access to the information stored within that instance.

### 5.3.2 The Variable Group

The variable group contains the functions needed to obtain information about the variables declared in the PCS. They are as follows:

```
int     getNumVars(rpcshandler *handler)
varType getVarType(rpcshandler *handler, int varId)
int     getVarSize(rpcshandler *handler, int varId)
char   *getVarName(rpcshandler *handler, int varId)
```

Common for all of them is that they as their first parameter take a rpcshandler as returned by *procSpec*(). Secondly, each individual variable is indexed and accessed using its own unique $varId$ taken from the pool of integers between 0 and $getNumVars() - 1$. Using this function, all variables can therefore be identified and information such as *type* (eg. packetnorm), $varsize$ (the number of bits assigned to the variable), and $name$ can be determined.

### 5.3.3 The Stream Key Group

As the name suggests the stream key group is home to the functions needed to obtain information about the stream key defined in the PCS. The functions in this group are as follows:

```
int   getNumKeyPairs(rpcshandler *handler)
char *getPairVariable(rpcshandler *handler, int pairId, pEl mode)
```

As was the case with the variables, each keypair is accessed using its own unique Id. Using these Ids, the $getPairVariable()$ can be used to obtain the names of the variables used in the different keypairs by specifying the *mode* parameter (either TUPLE_A or TUPLE_B) which describes which part of the keypair that should be returned.

### 5.3.4 The Location Group

The fourth group is the location group. This group encompasses all the functions needed to obtain information about the locations declared in the PCS. The functions contained within this group are as follows:

```
int      getNumLocations(rpcshandler *handler)
locType  getLocType(rpcshandler *handler, int locId)
char    *getLocName(rpcshandler *handler, int locId)
int      getTimeout(rpcshandler *handler, int locId)
```

The functions in the location group are conceptually very similar to those found in the variable and key groups. As was the case with variables and keys, the individual locations are indexed and accessed using their own unique Id's, and using the different functions in the group, information such as *type* (open or closed), *timeout value*, and finally the *name* of the location, can be obtained.

### 5.3.5 The Transition Group

The last and largest group is the transition group. This group is itself split into 3 subgroups. The functions in the first subgroup are used to get hold of the different decision diagrams created from the transitions in the PCS. They are as follows:

```
dd *getClosed(rpcshandler *handler)
dd *getOpen(rpcshandler *handler, int locId, directions dir)
```

As no direction constraints are associated with transitions emerging from the closed location, the *getClosed*() function takes as its only input the rpcshandler representing the PCS in question. *getOpen*(), on the other hand, takes as input also a location Id along with a direction, and returns the corresponding decision diagram as described in Section 5.2.1.

The second subgroup holds the functions needed to traverse the transition decision diagrams returned by the above functions. They are as follows:

```
nType  getNodeType(rpcshandler *handler, dd *ddNode)
uint   getNumPartitions(rpcshandler *handler, dd *ddNode)
uint   getLBound(rpcshandler *handler, dd *ddNode, uint partId)
uint   getUBound(rpcshandler *handler, dd *ddNode, uint partId)
dd     getNextNode(rpcshandler *handler, dd *ddNode, uint partId)
apTree *getGuardAPTree(rpcshandler *handler, dd *ddNode)
gType  getGuardType(rpcshandler *handler, apTree *aptNode)
```

To traverse the diagram 5 different functions are available. *getNodeType()* is used to determine the type of a node in the diagram (non-terminal or terminal) and similarly *getNumPartitions()* returns the number of associated partitions. As with the previous groups the unique *partIds* can then be used to obtain the upper and lower bound of the partitions using the *getLBound()* and *getUBound()*. Finally, an abstract parse tree encoding the test associated with a given node can be obtained using the *getGuardAP-Tree()* function. Furthermore, to obtain the type of the guard represented by that tree (boolean or expression), and thereby determine whether the node storing it in the decision diagram is a boolean or expressional node, *getGuardType()* can be used. Finally, to traverse the abstract parse tree, the following 4 functions are available and the specifics to how they correspond to the different constructs of the language can be found in the electronic documentation accompanying the current implementation:

```
apTree *getTokenName(apTree *aptNode)
apTree *getFirstChild(apTree *aptNode)
apTree *getSecondChild(apTree *aptNode)
apTree *getThirdChild(apTree *aptNode)
```

As should be clear from the name *getTokenName* is used to get the name of a node in the abstract parse tree. Depending on the outcome of this function a number of children are assigned to the node thereby making up the structure of the tree. These children can subsequently be obtained through the use of the 3 *get*Child* function.

The third and last subgroup contain the functions needed to obtain the information stored in the terminals of the decision diagram. They are as follows:

```
tType *getTransType(rpcshandler *handler, dd *ddTerminal)
int    getNumUpds(rpcshandler *handler, dd *ddTerminal)
int    getUpdVar(rpcshandler *handler, dd *ddTerminal, int updId)
apTree *getUpdAPTree(rpcshandler *handler, dd *ddTerminal, int updId)
int    getNewLoc(rpcshandler *handler, dd *ddTerminal)
char  *getPropValue(rpcshandler *handler, dd *ddTerminal)
```

From their names the purpose of each of these functions should be clear. As previously described, associated with each terminal is a number of updates (possible 0). Using the three functions *getNumUpds()*, *getUpdVar()*, and *getUpdAPTree()* these updates can be obtained in a way similar to what was the case with guards in the previous subsection. Finally, the *getTransType()*, *getNewLoc()*, and *getPropValue()* functions can be used to get the *type* of the transition (ignore or update), the location to which the transition points, and finally the *property value* associated with the transition in question.

This concludes the description of the requirements to the intermediate description, the intermediate representation used by the current implementation of the retargetable PCS system, and the output generator API. In the next chapter, an example as to how to use this API to create a simple output generator for the Netfilter firewall will be given.

# Chapter 6

# Output Generation

This chapter describes the output generation phase of the proposed system. As described in Section 2.5.2 this is the phase where the PCS, obtained from the intermediate representation, is transformed into something usable by a particular firewall. Obviously, as what is usable by one firewall is most likely useless to everything else, the development of these generators is left entirely to the firewall developers wanting to use the system. However, to give the reader a feel for the process involved in making such a generator, this chapter describes the design and implementation of a sample output generator for a slightly modified version Netfilter[1]. In Section 6.1 a brief introduction to the SI part of this firewall is given thus making it clear what needs to be produced by the generator. With that in place, Section 6.2 provides a description of the actual generator.

## 6.1 Adding Protocol Conformance Specifications to Netfilter

As described in Section 2.4.2 Netfilter is an open-source firewall where a lot of infrastructure has been added to ease the development of new functionality. Due to this added infrastructure, adding an additional PCS to the firewall amounts to implementing a C interface of *6 functions* and creating a *table entry struct* specifying the variables to be stored in the state table.

As described in Section 2.2.2 table based SI works by storing the state information in a table. Whenever a packet is received, a lookup is performed to determine whether information is stored about the stream of the packet. If a matching entry is found the information stored in this entry is used in the inspection of the packet. If not, the packet is checked against the transitions

[1]Currently SI does not distinguish between the packets that are to be blocked and those that are to be accepted. As a result the state table is updated regardless of the future of the packet. To circumvent this, a few minor alterations have been made to the standard firewall, thus in effect, easing the development of the generator.

emerging from the closed state of the PCS. In Netfilter this inspection process is carried out using a number of functions, each with their own well defined area of responsibility. By implementing these functions a new PCS can therefore be added to the firewall. More specifically, the process that the SI subsystem goes through whenever a packet is received can be seen in Algorithms 5 and 6, where the functions to be implemented for each PCS are written in italic. In Netfilter the *InspectPacket* function listed in

**Algorithm 5:** Outline of the Netfilter *InspectPacket* function

```
Data  : packet to be inspected
entry ← LookupStreamInTable(packet);
if entry found then
    $packet(entry, packet, direction)$;
else
    entry ← create empty entry;
    $pkt\_to\_tuple(buffer, entry)$;
    $invert\_tuple(buffer, entry)$;
    $new(entry, packet)$;
```

Algorithm 5 is used to perform the actual inspection. Depending on whether the packet is eventually accepted or dropped it is then the responsibility of the *Commit* function listed in Algorithm 6 to apply the updates to the state table[2].

**Algorithm 6:** Outline of the Netfilter *Commit* function

```
Data  : packet to be inspected
if packet is to be accepted then
    if packet->delete then
        DeleteEntryFromTable(packet);
    else
        tableEntryForPacket=packet->updatedEntry;
```

As can be seen from the *InspectPacket* function the first function to be used is the *packet* function which has the following prototype:

```
int packet(struct ip_conntrack *ct, struct iphdr *iph, size_t len,
           enum ip_conntrack_info ctinfo)
```

[2]The introduction of the *Commit* function is one of the modifications made to the standard firewall.

This function is called whenever a packet for whose stream, an entry in the state table exists. With the *entry*, *packet*, and the *direction* as parameters, it is the job of this function to perform the actual inspection. To make it possible to delay the actual updating of the state table until it has been determined whether the packet is to be accepted or not, the updates are written into a temporary table entry struct attached to the packet along with the property value of the satisfied transition. Just before the packet is either accepted or dropped the updates can then be applied using the *Commit* function.

Next, is the *pkt_to_tuple* function:

```
int pkt_to_tuple(const void *datah, size_t datalen,
                 struct ip_conntrack_tuple *tuple)
```

As can be seen from the listing this function is called after an empty table entry has been created. Given the *packet* and the new *entry* it is the responsibility of this function to extract from the packet, the values of the variables making up the stream key, and write them in the new entry. When performing lookups in the table the values of these stored variables can then be used to identify the appropriate entry.

The third function in the interface is *invert_tuple*:

```
int invert_tuple(struct ip_conntrack_tuple *tuple,
                 const struct ip_conntrack_tuple *orig)
```

The purpose of this function is very similar to that of *pkt_to_tuple*. As with that function, *invert_tuple* is used to fill the fields used to store the stream key. However, where the previous function simply extracted the values from the packet, this function stores an inverted version of those variables such that they reflect how they would look had the packet been flowing in the *return* direction. This way, the direction of a received packet can be determined simply by checking which version of the stream key is matched by the packet.

The fourth and final function used by *InspectPacket* is the *new* function. Its prototype looks as follows:

```
int new(struct ip_conntrack *conntrack, struct iphdr *iph,
        size_t len)
```

As with the *packet* function this function is responsible for performing the actual inspection. However, as can be seen from *InspectPacket* this function is called upon the reception of the first packet in a stream. Apart from this, the responsibilities of *packet* and *new* are the same.

In addition to the 4 functions used by *InspectPacket*, two secondary functions used to print the content of the state table entry must also be implemented. The prototype of the first function is as follows:

```
unsigned int print_tuple(char *buffer,
                         struct ip_conntrack_tuple *tuple)
```

The purpose of this function is to print the contents of the stream key into the buffer provided in the parameter list. This buffer is then used to provide the user with information about the streams flowing across the firewall.

The last function is the *print_ conntrack* function:

```
unsigned int print_conntrack(char *buffer,
                             struct ip_conntrack *conntrack)
```

Similar to *print_ tuple* this function is responsible for printing information about the values stored in the state table entry to the given buffer. However, contrary to that function *print_ conntrack* is responsible for printing the values of any additional variables not already covered by *print_ tuple*.

The last piece of the interface to be implemented is the table entry struct. As previously described this struct declares the variables that are to be stored within each entry in the state table. It is to instantiations of this struct updates are made using the *packet* and *new* functions.

## 6.2 The Netfilter Output Generator

Having identified the interface which needs to be implemented the output generator can now be created. As this generator is responsible for mapping the different constructs of the protocol conformance model to the operations of Netfilter using the above mentioned interface, the output generator will be described in that order.

### 6.2.1 Stored Variables

The code for generating the table entry struct, thereby mapping the stored variables of the PCM to a PCS in Netfilter firewall, is fairly straight forward. An outline of the code capable of generating this struct can be seen in Algorithm 7 [3]. As can be seen from this outline a variable is simply declared for each stored variable in the PCS. To achieve the bounded behavior defined for variables in the model, each variable is declared to the exact size specified in the PCS. As a result the normal integer primitives are used for variables of size 8,16, and 32 whereas bit-fields are used for everything else. Finally, as can be seen from the last line of the outline an additional integer is declared to be used for storing the location Id. This variable can then be used by the *new* and *packet* functions to track the location of the stream.

[3]The *handler* parameter in the output generator API calls has been left out for brevity.

**Algorithm 7:** Outline of the *generate_entry_struct* function

```
foreach stored variable varId do
    if getVarSize(varId) != 8,16 or 32 then
        declare bit-field of size $getVarSize(varId)$
    else
        declare unsigned int of size $getVarSize(varId)$
declare location variable integer
```

### 6.2.2 Stream Key

As previously described the code for implementing the stream key in Netfilter is split over two functions, *pkt_to_tuple* and *invert_packet*. As the current system, and Netfilter on the transport layer level, currently only supports streams utilizing the TCP and UDP protocols, and as the stream keys for these protocols are always the same, the code for generating these functions is trivial. An outline of this code is shown in Algorithm 8. As can be seen from this outline the stream key generator simply traverses the keypairs specified in the PCS. As Netfilter handles all network layer parts of the stream key transparently, pairs related to this layer are simply ignored and the traversal continues. Eventually, upon reaching a TCP or UDP stream key pair (identified on the prefix of the variable) the code corresponding to that protocol is generated and the generation stops.

**Algorithm 8:** Outline of the *generate_streamkey* function

```
foreach stream keypair do
    switch packet variable type do
        case TCP packet variable
            generate pkt_to_tuple function for TCP
            generate invert_tuple function for TCP
            exit generator
        case UDP packet variable
            generate pkt_to_tuple function for UDP
            generate invert_tuple function for UDP
            exit generator
        otherwise
            continue
```

### 6.2.3 Transitions and Locations

In terms of mapping transitions and determining which transition to take for a given packet, this is done in very much the same way described in relation to the intermediate representation. As previously described two functions need to be implemented, *new* and *packet*. An outline of the generator for the *new* function can be seen in Figure 9. Initially called with the root

**Algorithm 9:** Outline of the *generate_new* function

```
Data : ddNode from the getClosed diagram
switch getNodeType(ddNode) do
    case NON-TERMINAL
        foreach partId association with ddNode do
            generate conditional for partition partId
            generate_new(getNextNode(ddNode, partId));
    case TERMINAL
        if getTransType(ddNode) == UPDATE then
            foreach update associated with terminal do
                generate update code
            generate code for updating location variable
            generate code for updating timeout value
        generate code for returning associated property value
```

of the diagram returned by the *getClosed* function, this recursive function is capable of traversing this diagram and generate the *new* function. It does this by first determining the type of the node in the diagram. In case of it being a terminal, the type of the transition represented by this terminal determines what happens next. If it is an update transition, code for each associated update is generated along with code for updating the location and timeout values. Finally, regardless of the transition type, code for returning the property returned by *getPropVal* is generated. In case of the given node being a non-terminal, conditional statements for each partition are generated. Upon completion of the generation for each partition, the *generate_new* function is then called with the child of that partition.

With regards to the code for the *packet* function, the generator just described can be reused. However, instead of calling it with the *getClosed* diagram, *packet* requires it to be called with each diagram associated with the open locations. Similarly, conditionals for checking the location variable along with the direction of the received packet must be generated as well.

### 6.2.4 Additional Functionality

As previously described Netfilters PCS interface contains an additional two functions, *print_tuple* and *print_conntrack*. The purpose of these are to print the contents of a given table entry to a buffer which can subsequently be displayed in a `/proc` entry. The first function, *print_tuple*, is responsible for printing the contents of the non-IP parts of the stream key. An outline of the structure of the generated buffer can be seen in Figure 10. Because

**Algorithm 10:** Outline of the structure of the buffer generated for the *print_tuple* function

```
sprintf(buffer,“
foreach stream keypair pairId do
    if getPairVariable(pairId, TUPLE_A) != “IP_*” then
        getPairVariable(pairId,TUPLE_A) :%u
        getPairVariable(pairId,TUPLE_B) :%u
”
foreach stream keypair pairId do
    if getPairVariable(pairId, TUPLE_A) != “IP_*” then
        , getPairVariable(pairId,TUPLE_A) ,
        getPairVariable(pairId,TUPLE_B)
);
```

Netfilter handles all network layer stream key information transparently the generator simply picks out the non-IP pairs of the stream key. For each variable the name along with its value is then printed to the buffer.

With regards to the buffer generated for the *print_conntrack* function this is very similar to what just described. However, where the *print_tuple* buffer printed the value of the stream key variables this buffer prints the names and values of the stored variables. An outline of the structure of this buffer can be seen in Algorithm 11. As can be seen from this outline the buffer is generated simply by traversing the stored variables of the PCS. As the generator for the table entry struct declared these variables using the same names as in the PCS these names can be used directly.

This concludes the detailed description of the proposed system. In the next part the current implementation of the system is tested, and a conclusion concerning the system as a whole is drawn.

**Algorithm 11:** Outline of the structure of the buffer generated for the *print_ conntrack* function

```
sprintf(buffer,“
foreach stored variable varId do
  └ $getVarName(handler, varId)$ :%u
state : %s”,
foreach stored variable varId do
  └ $getVarName(handler, varId)$
stateid_to_name(stateid)
);
```

# PART III

# Test and Conclusion

Having spent the last part describing the proposed retargetable PCS system in detail, this part finalizes and concludes on the project. First, in Chapter 7 the current implementation of the proposed system is tested to ensure that the possible performance penalty incurred by the high-level approach of the retargetable system does not hinder any practical use. With that in place, Chapter 8 points out a number of directions for the further development of the system and finally, in Chapter 9, the project is finalized with a conclusion on the advantages and drawbacks of the system and its usefulness in general.

# Chapter 7

# Testing The System

In order to investigate the feasibility of the proposed system and facilitate its current and future development, the system described in this report has been implemented along with the output generator described in the previous chapter[1]. In this chapter we test the performance of the code generated by the Netfilter output generator and through that, indirectly test the correctness of the implementation in general. The tests involves measuring the performance of the code generated by the Netfilter output generator from a PCSL specification of the TCP PCS currently used by Netfilter. The result is then compared to the performance of the "native" version thus giving a sense of the performance hit accompanying the retargetable system. In Section 7.1 we start by describing the PCS used in the tests and then in Section 7.2 the actual tests and their results are described.

## 7.1 The Protocol Conformance Specification

The TCP PCS used by Netfilter is more geared towards the task of tracking the presumed connection state of a stream rather than that of detecting illegal packets. Because of this the PCS is very liberal in the kind of packets it allows and bares little resemblance to the official TCP specification[Pos81b]. The result is the very large PCS depicted in Figure 7.1. Because of its size this PCS provides a good basis for the tests. First of all its size will result in quite a large PCSL specification. This should, in turn, provide a good test for the usability of the PCSL language. Secondly, the size will result in a lot of code being generated by the output generator. As a lot of processing on the packets takes place outside the code affected by the retargetable system, a proportionally small amount of overhead could easily be overshadowed. Testing on a large PCS with a large amount of code should increase this ratio between inside and outside processing and hopefully reduce this problem.

[1]For a brief description of the current status of this implementation, see Appendix C.

## 7.2 Testing the Performance of the Generated Code

The purpose of the performance tests are to establish how much of a performance penalty is likely to be incurred by the high level approach of the retargetable system. Given the unoptimized state of the Netfilter output generator the results obtained from these tests should provide an upper bound for that overhead. Therefore, if the results are acceptable we should be able to conclude that the overhead is manageable.

### 7.2.1 Test Setup

The tests have been performed on a small laboratory network consisting of 2 traffic generators separated by a firewall and connected through a Cisco 3500 XL gigabit switch. The topology of this network can be seen in Figure 7.2 and details on the computers and their software are listed in Table 7.1.

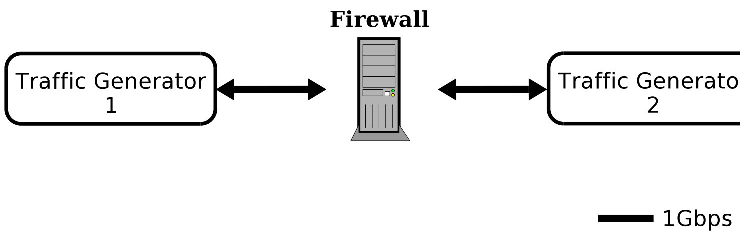


**Figure 7.2.** The topology of the network used for the tests.

| |
|---|
| Processor: Athlon 2000+ |
| NICs: SysKonnect 9821 v. 2.0 |
| Kernel: Linux 2.4.20 |
| NIC driver: sk98lin v6.04 |

(a) Configuration of the traffic generators.

| |
|---|
| Processor: Athlon 2000+ |
| NICs: SysKonnect 9821 v. 2.0 |
| Kernel: Linux 2.6.7 |
| NIC driver: sk98lin v6.23 |

(b) Configuration of the firewall.

**Table 7.1.** Configurations of the computers in the test network.

While this setup does not represent any real world scenario it should be sufficient as we are only interested in a comparative measure between native and the PCSL approach. Furthermore, as we are interested in increasing the ratio between inside and outside processing, this simple setup is preferable as having only a single stream across the firewall reduces the processing required by outside parts (table lookups etc.).

### 7.2.2 Traffic and Test Data

The tests were performed using *Iperf v2.0.1* as this tool is capable of producing bidirectional traffic and allows for the entire test to be performed using a single stream. In accordance with the recommendations of RFC1944[BM96] the tests were conducted using a number of different frame sizes - 56, 128, 512, and 1518 bytes. During these tests a sequence of packets, all belonging to the same stream, are sent across the firewall. This stream will cause the firewall, upon the arrival of the first packet, to store information about that stream. Throughout the rest of the test that information will then be used in the inspection process. This way the task of adding this information is performed only once and the consequent table lookups are simplified. As described above the result is that the ratio between inside and outside processing is increased thus giving a more accurate picture of the overhead. Finally, the firewall is configured using only a single rule. This way the time spent traversing the rule lists are also minimized. For the tests the following rule is used:

```
iptables -A FORWARD -p tcp -m state –state NEW,ESTABLISHED,INVALID -j accept
```

### 7.2.3 Test Procedure

All tests are conducted by first clearing the rule-set of the firewall and then configuring it with the previously described rule. Upon completion of this task the Iperf server is initiated on *Traffic generator 2* using the options: *-s -P 1*. Finally, the Iperf client is initiated on *Traffic generator 1* using the options: *-c traffic_generator_1 -l frame_size -t 300 -i 5*. Upon completion the average of the throughput measurements reported by Iperf is calculated and noted as the result of the test.

### 7.2.4 Results and Conclusions

The results of the tests can be seen in Table 7.2. As an be seen from these

| **Framesize (bytes)** | **64** | **128** | **512** | **1518** |
|---|---|---|---|---|
| **No SI (Mbps)** | 340 | 543 | 579 | 579 |
| **Native (Mbps)** | 337 | 492 | 499 | 499 |
| **Native Overhead (%)** | *1* | *9* | *14* | *14* |
| **Retargetable (Mbps)** | 336 | 493 | 495 | 498 |
| **Retargetable Overhead (%)** | *1* | *9* | *15* | *14* |

**Table 7.2.** Results of the performance tests.

numbers the overhead incurred by the retargetable approach is negligible and within 1% of the native version. It can therefore be concluded that any overhead incurred by the generated code is overshadowed by the rest of the packet processing code. As a result we can conclude that the performance of the generated output does not hinder the practical use of the system. Furthermore, this can be concluded despite the preliminary status of the intermediate representation and the output generator. Given the negligible overhead it is therefore doubtful whether any significant increase in performance would be gained from improving these two components. Using the tests we can therefore also conclude, that from a practical point of view, the current states of these components are quite sufficient.

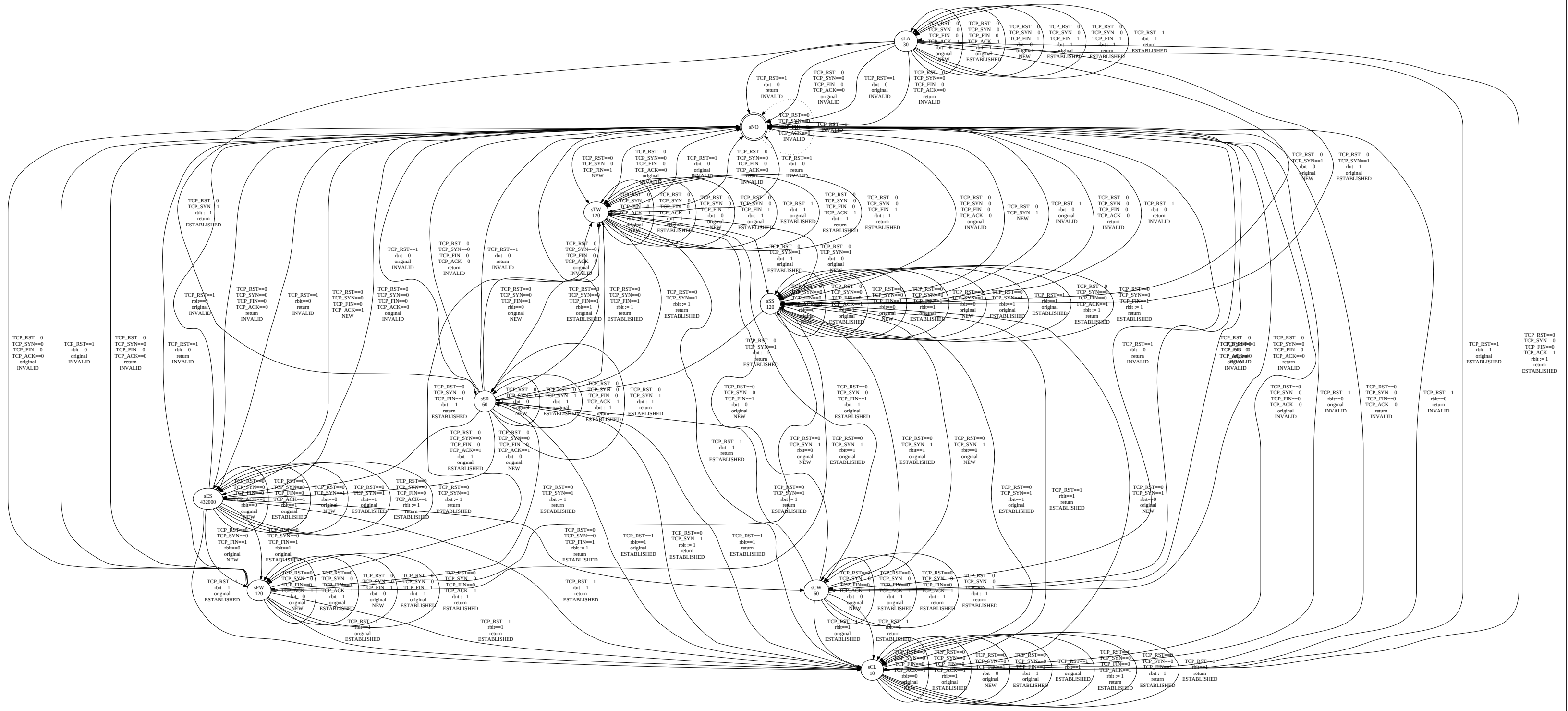


Figure 7.1. The TCP PCS used for testing the system.

# Chapter 8

# Further Development

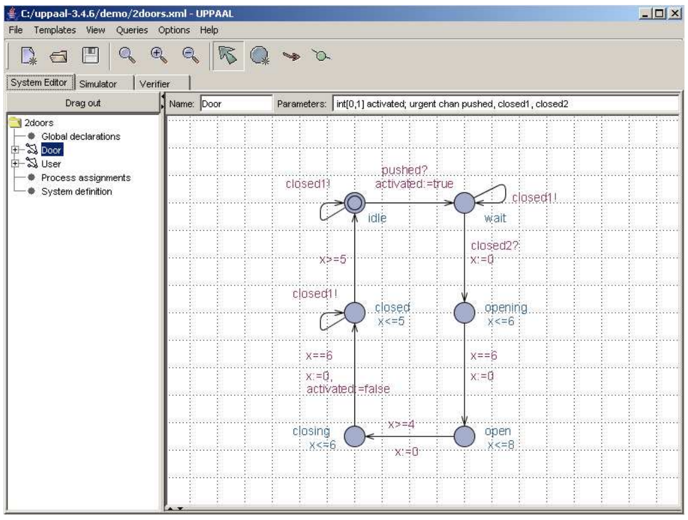
As described in the introduction this project is, to our knowledge, the first one to deal with the development of a retargetable PCS system. As a result, the system presented in this report is by no means complete. In this chapter we point out a number of directions for further developing the concept.

## 8.1 Support For Application Layer Protocols

This project has been limited to support for TCP and UDP streams only. As described in Chapter 2 most traffic also uses an application layer protocol (e.g. HTTP, FTP). An obvious improvement would therefore be to extend the current system with the capability of representing PCSs for such protocols. The task involved in this would amount to the creation of a new specialization of the protocol conformance model. Presumably the biggest change compared to the TCP/UDP specialization would be the retrieval of information from the packets. Where TCP and UDP use fixed size fields to store the relevant header information, some application layer protocols use variable sized fields. An example of this is FTP. Where IPv4 stores the host addresses in 32 bit fields, the FTP protocol stores such information in plain ASCII. The result is that it no longer suffices simply to look at a fixed part of the packet as the addresses 10.0.0.1 and 10.0.0.10 take up 8 and 9 bytes respectively. To make things even worse, the information stored after these two addresses is shifted accordingly. This means that you no longer have constant time access to the information stored in the packets. In other words, if you want to get to some information towards the end of the FTP header you are forced to parse the entire header up to that point. In effect this means that a more flexible, than the current way of specifying and accessing such fields, is needed.

## 8.2 Graphical Front-End for the PCSL Language

As can be seen from the TCP PCS used in the test chapter the PCSs easily become quite large. The result is that the number of lines needed to implement any useful PCS quickly adds up. While each line in itself is simple and straight-forward, one can quickly get lost in the large amount of almost identical transitions. To counter this problem, and thereby make it even easier to write PCSs, we propose the development of a graphical front-end to the PCSL language. The purpose of this front-end is to allow the user to describe PCSs using the same graphical representation used in this report. From this representation the front-end can generate the appropriate code thereby hiding the PCSL language from the user. Examples of this concept are UppAal[LPY97] and YAGCS[ea01] which both provide graphical front-ends for an underlying textual language. In Figure 8.1 a screenshot of the UppAal GUI can be seen. The main advantage of such a tool is that it



Name: Door Parameters: int[0,1] activated; urgent chan pushed, closed1, closed2

Figure 8.1. The formal verification tool UppAal whose GUI is essentially a graphical frontend for specifying models in the *xta* language.

tends to increase the readability of the specification thereby easing the task of creating and maintaining it.

Important to note however is that the need for such a tool should not in any way be seen as a sign of a weakness in the system presented in this report. The fact that it is possible to make such a graphical front-end is a testament to the contributions of the system.

Chapter 9

# Conclusion

With the introduction of SI the task of developing and maintaining a firewall has become harder and more complex. One of the main reasons for this is SIs inherent dependence on protocol conformance specifications against which the inspected streams can be checked. In current implementations of SI these specifications are hard coded into the SI subsystem using the same generic language used to implement the rest of the firewall. Unfortunately, this approach has the disadvantage that the specifications are prone to containing errors as these generic languages are not very well suited for the task. As firewalls are primarily meant to provide security, and errors tend to lessen security, this is by no means an ideal approach. Furthermore, as the software of many firewalls is not easily upgradeable one they have been deployed, the need for a system that minimizes the risk of errors is apparent.

In this project we proposed, developed, implemented, and tested such a system. This system, which introduces the notion of retargetable PCSs, allows the firewall developer to implement PCSs in a firewall independent manner using a custom made, protocol oriented language. This way, the implementation of the PCS is simplified and the chance of it being correct increased.

The proposal has included the development of a number of components. First, an abstract model, the *protocol conformance model*, encompassing the functionality needed in a PCS was made. This provided a common foundation for current and future versions of the system. With that foundation in place, a specialization capable of representing PCSs for streams of two of the most commonly used protocols - TCP and UDP, was made. The basis of this specialization was an investigation of the requirements of a wide range of currently available TCP and UDP PCSs. This ensured that, even though it is impossible to definitively determine the expressive power needed to represent every possible TCP and UDP PCS, this specialization has the expressive power to represent most, if not all, current and future PCSs for these protocols. This way, the major potential drawback associated with a less than touring complete model was alleviated.

Having completed the model and thus created the foundation for the retargetable PCSs, a system capable of transforming these specifications into usable code was made. This included the development of a simple language capable of expressing the specialized model as well as an intermediate representation capable of storing the retargetable PCSs.

Finally, through the evaluation of an implementation of the proposed system, we have shown that the amount of performance overhead incurred by the retargetable approach is negligible. This is despite the fact that very little optimization was performed on the retargetable PCS by the intermediate representation and the Netfilter output generator. Based on this we can therefore conclude, that also in practice, the use of retargetable specifications is a feasible approach and that the system and its accompanying implementation is fully usable.

# Appendix A

# Header Fields Symbol Table

This appendix contains a list of the different fields in the headers of the IP, UDP, and TCP protocols along with the symbol by which they are referenced in this report and the current implementation of the system.

| **Symbol** | **Description** |
|---|---|
| IP_VERSION | Version |
| IP_IHL | Internet Header Length |
| IP_TOS | Type of Service |
| IP_TOTLEN | Total Length |
| IP_ID | Identification |
| IP_FRAG | Fragment Offset |
| IP_TTL | Time To Live |
| IP_PROTOCOL | Protocol |
| IP_CHECKSUM | Checksum |
| IP_SRC | Source IP Address |
| IP_DST | Destination IP Address |

**Table A.1.** IP Fields

| **Symbol** | **Description** |
|---|---|
| UDP_SRCPORT | Source Port |
| UDP_DSTPORT | Destination Port |
| UDP_LEN | Length |
| UDP_CHECKSUM | Checksum |

**Table A.2.** UDP Fields

| Symbol | Description |
|---|---|
| TCP_SRCPORT | Source Port |
| TCP_DSTPORT | Destination Port |
| TCP_SEQ | Sequence Number |
| TCP_ACKSEQ | Acknowledgment Number |
| TCP_DOFF | Data Offset |
| TCP_FIN | FIN Flag |
| TCP_SYN | SYN Flag |
| TCP_RST | RST Flag |
| TCP_PSH | PSH Flag |
| TCP_ACK | ACK Flag |
| TCP_URG | URG Flag |
| TCP_ECE | ECE Flag |
| TCP_CWR | CWR Flag |
| TCP_WINSIZE | Window Size |
| TCP_CHECKSUM | Checksum |
| TCP_URGPTR | Urgent Pointer |

**Table A.3.** TCP Fields

# Appendix B

# The PCSL Language

## B.1 Abstract Syntax

The abstract syntax of the PCSL language is as follows:

---

**1. Syntactic categories**

| | |
|---|---|
| $VD \in$ *Variable declarations* | $ol \in$ *Open locations* |
| $SD \in$ *Stream key declarations* | $p \in$ *Property values* |
| $LD \in$ *Location declarations* | $D \in$ *Directions* |
| $TD \in$ *Transitions* | $GD \in$ *Guard declarations* |
| $p \in$ *Property values* | $G \in$ *Guards* |
| $sn \in$ *Stored normal variables* | $UD \in$ *Update declarations* |
| $pn \in$ *Packet normal fields* | $U \in$ *Updates* |
| $ps \in$ *Packet Sequence number fields* | $PD \in$ *Property value declatations* |
| $n \in$ *Numerals* | $NEXP \in$ *Normal expressions* |
| $VLIST \in$ *Variable lists* | $BOP \in$ *Boolean operators* |
| $cl \in$ *Closed locations* | $AOP \in$ *Arithmetic operators* |
| $OLD \in$ *Open location declarations* | $CLD \in$ *Closed location declarations* |

$ss \in$ *Stored sequence number variables*

$SEXP \in$ *Sequence number expressions*

$DD \in$ *Default property value declarations*

$PCS \in$ *Protocol Conformance Specifications*

**2. Definitions**

$PCS ::= DD\ VD\ SD\ LD\ TD$

$DD ::= \textbf{defpropvalue}\ p\textbf{;}$

$VD ::= \textbf{storednorm}\ sn\ n\textbf{;} \mid \textbf{storedseq}\ ss\ n\textbf{;} \mid \textbf{packetnorm}\ pn\ n\textbf{;}$
$\quad \mid \textbf{packetseq}\ ps\ n\textbf{;} \mid VD_1\ VD_2$

$SD$ ::= **keypair** $VLIST$ **;** | $SD_1$ $SD_2$
$VLIST$ ::= $pn$ , $pn$ | $ps$ , $ps$
$LD$ ::= $CLD$; $OLD$;
$CLD$ ::= **clocation** $cl$
$OLD$ ::= **olocation** $ol$ $n$ | $OLD_1$ **;** $OLD_2$
$TD$ ::= **itrans** $cl$ -> $cl$ {$GD$ $PD$}
| **itrans** $ol$ -> $ol$ {$D$ $GD$ $PD$}
| **utrans** $cl$ -> $ol$ {$GD$ $UD$ $PD$}
| **utrans** $ol$ **->** $ol$ {$D$ $GD$ $UD$ $PD$}
| **utrans** $ol$ -> $cl$ {$D$ $GD$ $PD$}
| $TD_1$ $TD_2$
$D$ ::= **direction original;** | **direction return;**
$PD$ ::= **propvalue** $p$**;**
$GD$ ::= **guard** $G$**;** | $\epsilon$
$G$ ::= $NEXP$ $BOP$ $NEXP$ | $SEXP$ $BOP$ $SEXP$ | $G_1$ **,** $G_2$
$NEXP$ ::= $n$ | $sn$ | $pn$ | $(n)(NEXP\ AOP\ NEXP)$ | $(NEXP)$
$BOP$ ::= $<$ | $>$ | $<=$ | $>=$ | $==$
$SEXP$ ::= $ps + NEXP$ | $ss + NEXP$
$AOP$ ::= $+$ | **-** | ***** | / | //
$UD$ ::= **update** $U$**;** | $\epsilon$
$U$ ::= $sn$ **:=** $NEXP$ | $ss$ **:=**$SEXP$ | $U_1$ **,** $U_2$

## B.2 Concrete Syntax

The following listing shows the concrete syntax for the PCSL language:

| | | |
|---|---|---|
| *Alpha* | → | `a` \| ... \| `z` \| `A` \| ... \| `Z` |
| *Digit* | → | `0` \| ... \| `9` |
| *Num* | → | *Digit* \| *Num Digit* |
| *AlphaNum* | → | *Alpha* \| *Num* |
| *Ident* | → | *Alpha* \| *Ident AlphaNum* |
| *Start* | → | *Dd Vd Sd Ld Td* |
| *Dd* | → | `defpropvalue` *Ident* |
| *Vd* | → | *Vd VariableType Ident Num* `;` \| $\epsilon$ |
| *VariableType* | → | `storednorm` \| `packetnorm` \| `storedseq` \| `packetseq` |
| *Sd* | → | *Sd* `keypair` *Vlist* `;` \| $\epsilon$ |
| *Vlist* | → | *Ident* `,` *Ident* |
| *Ld* | → | *Cld Old* |
| *Cld* | → | `closed location` *Ident* `;` |
| *Old* | → | *Old* `open location` *Ident Num* `;` \| $\epsilon$ |

| | | |
|---|---|---|
| *Td* | $\rightarrow$ | *Td TransType Ident* `->` *Ident* `{` *D GD UD PD* `}` |
| *TransType* | $\rightarrow$ | `itrans` \| `utrans` |
| *D* | $\rightarrow$ | `direction original` \| `direction return` |
| *Gd* | $\rightarrow$ | `guard` *G* `;` \| $\epsilon$ |
| *Pd* | $\rightarrow$ | `propvalue` *Ident* `;` |
| *G* | $\rightarrow$ | *Exp Bop Exp* \| *G* `,` *Exp Bop Exp* |
| *Exp* | $\rightarrow$ | *Num* \| *Ident* \| `(` *Num* `)` `(` *Exp* `)` \| `(` *Exp* `)` |
| *Bop* | $\rightarrow$ | `==` \| `<` \| `<=` \| `>` \| `>=` |
| *Aop* | $\rightarrow$ | `+` \| `-` \| `*` \| `/` \| `//` |
| *Ud* | $\rightarrow$ | `update` *U* `;` \| $\epsilon$ |
| *U* | $\rightarrow$ | *Ident* `:=` *Exp* \| *U* `,` *Ident* `:=` *Exp* |

## B.3 Type System

Using the notation of [Car97] the judgments of the formalized type system of PCSL are as follows:

| | |
|---|---|
| $\Gamma \vdash \diamond$ | $\Gamma$ is a well-formed environment |
| $\Gamma \vdash A$ | $A$ is a well-formed type in $\Gamma$ |
| $\Gamma \vdash M : A$ | $M$ is a well-formed term of type $A$ in $\Gamma$ |
| $\Gamma \vdash A <: B$ | $A$ is a subtype of $B$ in $\Gamma$ |
| $\Gamma \vdash D \therefore S$ | $D$ is a well-formed declaration of signature S in $\Gamma$ |
| $\Gamma \vdash DD$ | $DD$ is a well-formed default property value declaration in $\Gamma$ |
| $\Gamma \vdash SD$ | $SD$ is a well-formed keypair declaration in $\Gamma$ |
| $\Gamma \vdash D$ | $D$ is a well-formed direction declaration in $\Gamma$ |
| $\Gamma \vdash GD$ | $GD$ is a well-formed guard declaration in $\Gamma$ |
| $\Gamma \vdash U$ | $U$ is a well-formed update element in $\Gamma$ |
| $\Gamma \vdash UD$ | $UD$ is a well-formed update in $\Gamma$ |
| $\Gamma \vdash PD$ | $PD$ is a well-formed property value declaration in $\Gamma$ |
| $\Gamma \vdash TD$ | $TD$ is a well-formed transition declaration in $\Gamma$ |

With respect to the type rules a few additions are made to the notation of [Car97] to ease their description. This due to each variable being declared with a bitsize denoting its upper bound. Because these bounds influence how the variable can be used in guards, assignments etc. two variables with the differing upper bounds are seen to be of different types. In this view the language contains a large number of types, most of them being variations of the "basic" types *packetnorm*, *packetseq*, *storednorm*, and *storedseq*. To avoid having to define and deal with each of these individually the following notation is used : $BasicType \cdot \#bits$ where $\#bits$ is the number of bits assigned to the variable and $BasicType$ is the type it is a variation of. Using this notation $packetnorm \cdot 6$ would be the type of a variable declared with type *packetnorm* and the number of bits set to 6. With this in mind the following table shows the type rules of the PCSL language:

---

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env M)
$$\frac{\Gamma \vdash A \quad M \notin dom(\Gamma)}{\Gamma, M{:}A \vdash \diamond}$$

(Type Bool)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool}$$

(Type Cloc)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Clocation}$$

(Type Oloc)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Olocation}$$

(Type Propval)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Propvalue}$$

(Type Packetnorm)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Packetnorm{\cdot}n} \ (n = 1, 2, \ldots, 32)$$

(Type Packetseq)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Packetseq{\cdot}n} \ (n = 8, 16, 32)$$

(Type Storednorm)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Storednorm{\cdot}n} \ (n = 1, 2, \ldots, 32)$$

(Type Storedseq)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Storedseq{\cdot}n} \ (n = 8, 16, 32)$$

(Type Norm)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Norm{\cdot}n} \ (n = 1, 2, \ldots, 32)$$

(Type Seq)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash Seq{\cdot}n} \ (n = 8, 16, 32)$$

(Sub Refl)
$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

(Sub Trans)
$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

(Sub Subsumption)
$$\frac{\Gamma \vdash a{:}A \quad \Gamma \vdash A <: B}{\Gamma \vdash a{:}B}$$

(Sub Snorm)
$$\frac{\Gamma \vdash Storednorm{\cdot}x \quad \Gamma \vdash Norm{\cdot}y}{\Gamma \vdash Storednorm{\cdot}x <: Norm{\cdot}y} \ (x \leq y)$$

(Sub Pnorm)
$$\frac{\Gamma \vdash Packetnorm{\cdot}x \quad \Gamma \vdash Norm{\cdot}y}{\Gamma \vdash Packetnorm{\cdot}x <: Norm{\cdot}y} \ (x \leq y)$$

(Sub Sseq)
$$\frac{\Gamma \vdash Storedseq{\cdot}x \quad \Gamma \vdash Seq{\cdot}y}{\Gamma \vdash Storedseq{\cdot}x <: Seq{\cdot}y} \ (x \leq y)$$

(Sub Pseq)
$$\frac{\Gamma \vdash Packetseq{\cdot}x \quad \Gamma \vdash Seq{\cdot}y}{\Gamma \vdash Packetseq{\cdot}x <: Seq{\cdot}y} \ (x \leq y)$$

(PCS)
$$\frac{\emptyset \vdash DD \quad \emptyset \vdash VD.\therefore.(V{:}A) \quad V{:}A \vdash SD.\therefore.(S{:}B) \quad V{:}A, S{:}B \vdash LD.\therefore.(L{:}C) \quad V{:}A, S{:}B, L{:}C \vdash TD}{\emptyset \vdash DD\ VD\ SD\ LD\ TD}$$

(Defpropval)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \textbf{defpropvalue}\ p\ \textbf{;}}$$

(VDecl Sequence)
$$\frac{\Gamma \vdash VD_1 \ \therefore\ (M{:}A) \quad \Gamma, M{:}A \vdash VD_2}{\Gamma \vdash VD_1\ VD_2}$$

(VDecl Snorm)
$$\frac{\Gamma, sn{:}Storednorm{\cdot}n \vdash \diamond}{\Gamma \vdash \textbf{storednorm}\ sn\ n\textbf{;}\ \therefore\ (sn{:}Storednorm{\cdot}n)}$$

(VDecl Sseq)
$$\frac{\Gamma, ss{:}Storedseq{\cdot}n \vdash \diamond}{\Gamma \vdash \textbf{storedseq}\ ss\ n\textbf{;}\ \therefore\ (ss{:}Storedseq{\cdot}n)}$$

(VDecl Pnorm)
$$\frac{\Gamma, pn{:}Packetnorm{\cdot}n \vdash \diamond}{\Gamma \vdash \textbf{packetnorm}\ pn\ n\textbf{;}\ \therefore\ (pn{:}Packetnorm{\cdot}n)}$$

(VDecl Pseq)
$$\frac{\Gamma, ps{:}Packetseq{\cdot}n \vdash \diamond}{\Gamma \vdash \textbf{packetseq}\ ps\ n\textbf{;}\ \therefore\ (ps{:}Packetseq{\cdot}n)}$$

(Keypair Sequence)
$$\frac{\Gamma \vdash SD_1 \quad \Gamma \vdash SD_2}{\Gamma \vdash SD_1\ SD_2}$$

(Keypair)
$$\frac{\Gamma \vdash I_1{:}A \quad \Gamma \vdash I_2{:}A \quad A \in \{Packetnorm{\cdot}n, Packetseq{\cdot}n\}}{\Gamma \vdash \textbf{keypair}\ I_1\textbf{,}\ I_2\ \textbf{;}}$$

(VDecl Cold)
$$\frac{\Gamma \vdash CLD \ \therefore\ (M{:}A) \quad \Gamma, M{:}A \vdash OLD}{\Gamma \vdash CLD\ OLD}$$

(LDecl Cloc)
$$\frac{\Gamma, cl{:}Clocation \vdash \diamond}{\Gamma \vdash \textbf{clocation}\ cl\ \textbf{;}\ \therefore\ (cl{:}Clocation)}$$

(LDecl Oloc)

$$\frac{\Gamma, ol{:}Olocation \vdash \diamond}{\Gamma \vdash \mathbf{olocation}\ ol\ \mathbf{;}\ \therefore\ (ol{:}Olocation)}$$

(LDecl Sequence)

$$\frac{\Gamma \vdash OLD_1\ \therefore\ (M{:}A) \quad \Gamma, M{:}A \vdash OLD_2}{\Gamma \vdash OLD_1\ OLD_2}$$

(Trans Iclcl)

$$\frac{\Gamma \vdash cl_1{:}Clocation \quad \Gamma \vdash cl_2{:}Clocation \quad \Gamma \vdash GD \quad \Gamma \vdash PD}{\Gamma \vdash \mathbf{itrans}\ cl_1\ \texttt{->}\ cl_2\ \mathbf{\{}GD\ PD\mathbf{\}}}$$

(Trans Iolol)

$$\frac{\Gamma \vdash ol_1{:}Olocation \quad \Gamma \vdash ol_2{:}Olocation \quad \Gamma \vdash D \quad \Gamma \vdash GD \quad \Gamma \vdash PD}{\Gamma \vdash \mathbf{itrans}\ ol_1\ \texttt{->}\ ol_2\ \mathbf{\{}D\ GD\ PD\mathbf{\}}}$$

(Trans Uclol)

$$\frac{\Gamma \vdash cl{:}Clocation \quad \Gamma \vdash ol{:}Olocation \quad \Gamma \vdash GD \quad \Gamma \vdash UD \quad \Gamma \vdash PD}{\Gamma \vdash \mathbf{utrans}\ cl\ \texttt{->}\ ol\ \mathbf{\{}GD\ UD\ PD\mathbf{\}}}$$

(Trans Uolol)

$$\frac{\Gamma \vdash ol_1{:}Olocation \quad \Gamma \vdash ol_2{:}Olocation \quad \Gamma \vdash D \quad \Gamma \vdash GD \quad \Gamma \vdash UD \quad \Gamma \vdash PD}{\Gamma \vdash \mathbf{utrans}\ ol_1\ \texttt{->}\ ol_2\ \mathbf{\{}D\ GD\ UD\ PD\mathbf{\}}}$$

(Trans Uolcl)

$$\frac{\Gamma \vdash ol{:}Olocation \quad \Gamma \vdash cl{:}Clocation \quad \Gamma \vdash D \quad \Gamma \vdash GD \quad \Gamma \vdash PD}{\Gamma \vdash \mathbf{utrans}\ ol\ \texttt{->}\ cl\ \mathbf{\{}D\ GD\ PD\mathbf{\}}}$$

(Trans Sequence)

$$\frac{\Gamma \vdash TD_1 \quad \Gamma \vdash TD_2}{\Gamma \vdash TD_1\ TD_2}$$

(Dir Orig)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{direction\ original;}}$$

(Dir Ret)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{direction\ return;}}$$

(Propval)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{propvalue}\ p\ \mathbf{;}}$$

(Guard)

$$\frac{\Gamma \vdash G{:}Bool}{\Gamma \vdash \mathbf{guard}\ G\ \mathbf{;}}$$

(Guardel$_n$)

$$\frac{\Gamma \vdash NEXP_1{:}Norm{\cdot}a \quad \Gamma \vdash NEXP_2{:}Norm{\cdot}b}{\Gamma \vdash NEXP_1 \sim NEXP_2 : Bool} \quad \sim = \{<, >, <=, >=, ==\}$$

(Guard Sequence)

$$\frac{\Gamma \vdash G_1{:}\ Bool \quad \Gamma \vdash G_2{:}\ Bool}{\Gamma \vdash G_1\ \mathbf{,}\ G_2 : Bool}$$

(Guardel$_s$)

$$\frac{\Gamma \vdash SEXP_1{:}Seq{\cdot}a \quad \Gamma \vdash SEXP_2{:}Seq{\cdot}a}{\Gamma \vdash SEXP_1 \sim SEXP_2 : Bool} \quad \sim = \{<, >, <=, >=, ==\}$$

(Numeral)

$$\frac{\Gamma \vdash A \quad A{=}Norm{\cdot}\lceil log_2(n{+}1)\rceil}{\Gamma \vdash n : A}$$

(Nexp)

$$\frac{\Gamma \vdash NEXP_1{:}Norm{\cdot}a \quad \Gamma \vdash NEXP_2{:}Norm{\cdot}b \quad n{\in}\{1,2,...,32\}}{\Gamma \vdash (n)(NEXP_1 \sim NEXP_2) : Norm{\cdot}\ n}$$

(Paren)

$$\frac{\Gamma \vdash NEXP{:}A}{\Gamma \vdash (NEXP) : A}$$

(Sexp)

$$\frac{\Gamma \vdash sv{:}Seq{\cdot}a \quad \Gamma \vdash NEXP{:}Norm{\cdot}b}{\Gamma \vdash sv + NEXP : Seq{\cdot}a} \ (a \geq \tfrac{b}{2})$$

(Update)

$$\frac{\Gamma \vdash U}{\Gamma \vdash \mathbf{update}\ U\ \mathbf{;}}$$

(Updateel)

$$\frac{\Gamma \vdash U_1 \quad \Gamma \vdash U_2}{\Gamma \vdash U_1\ \mathbf{,}\ U_2}$$

(Updateel$_n$)

$$\frac{\Gamma \vdash sn{:}Storednorm{\cdot}a \quad \Gamma \vdash NEXP{:}Norm{\cdot}b}{\Gamma \vdash sn\ \mathbf{:=}\ NEXP} \ (a \geq b)$$

(Updateel$_s$)

$$\frac{\Gamma \vdash ss{:}Storedseq{\cdot}a \quad \Gamma \vdash SEXP{:}Seq{\cdot}a}{\Gamma \vdash ss\ \mathbf{:=}\ SEXP}$$

# Appendix C

# Current Status of the Implementation

As should be clear from the report the current implementation of the system should be seen mainly as an experimental tool used during development, and as a platform for testing and evaluating the retargetable concept. With that being said, the implementation is however stable, fully functional, and the output generator API fully documented[1]. Currently the following three output generators exists:

**Netfilter Generator:** An output generator capable of generating unoptimized code for the Netfilter firewall. The specifics of this generator is described in Chapter 6.

**PCS Illustrator:** An output generator capable of depicting a PCSL specification using the graphical notation described in Section 3.1.4. The graphical illustration of Netfilters standard TCP PCS depicted in Figure 7.1 is created using this generator.

**Diagram Illustrator:** An output generator capable of depicting the decision diagrams stored by the intermediate representation. The graphical notation is similar to that of Figure 5.1.

To ease the development of new generators, and in tune with the initial architecture previously depicted in Chapter 2, phases 1 and 2 have been implemented as an external library. Using that approach, integrating new generators with the current implementation is merely a matter of linking against that library and accessing it using the output generator API. For an illustration of this relationship between phases 1 and 2 and the output generators, see Figure C.1.

[1]For a complete description of the output generator API, see the documentation accompanying the implementation.

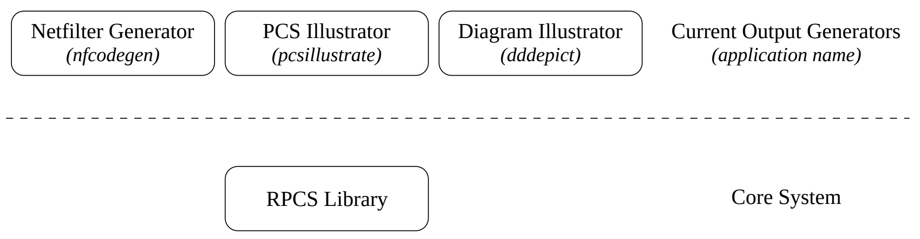


**Figure C.1.** The relationship between the core of the retargetable system (phases 1 and 2) and the output generators. Each individual generator makes up its own application and is linked against the RPCS library which provide access to the system using the output generator API.

Appendix D

# Summary

With the introduction of SI the task of developing and maintaining a firewall has become harder and more complex. One of the main reasons for this is SIs inherent dependence on protocol conformance specifications against which the inspected streams can be checked. In current implementations of SI these specifications are hard coded into the SI subsystem using the same generic language used to implement the rest of the firewall. Unfortunately, this approach has the disadvantage that the specifications are prone to containing errors as these generic languages are not very well suited for the task. As firewalls are primarily meant to provide security, and errors tend to lessen security, this is by no means an ideal approach. Furthermore, as the software of many firewalls is not easily upgradeable one they have been deployed, the need for a system that minimizes the risk of errors is apparent.

In this project we have proposed, developed, implemented, and tested such a system. This system, which introduces the notion of retargetable PCSs, allows the firewall developer to implement PCSs in a firewall independent manner using a custom made, protocol oriented language. This way, the implementation of the PCS is simplified and the chance of it being correct is increased.

This proposed system has included the development of a number of components. First, an abstract model, the *protocol conformance model*, encompassing the functionality needed in a PCS has been made. This provides a common foundation for current and future versions of the system. With that foundation in place, a specialization capable of representing PCSs for streams of two of the most commonly used protocols - TCP and UDP, has been created. The basis of this specialization is an investigation of the requirements of a wide range of currently available TCP and UDP PCSs. This ensures that, even though it is impossible to definitively determine the expressive power needed to represent every possible TCP and UDP PCS, this specialization has the expressive power to represent most, if not all, current and future PCSs for these protocols. This way, the major potential drawback associated with a less than touring complete model has been alleviated.

Having completed the model and thus created the foundation for the retargetable PCSs, a system capable of transforming these specifications into usable code has been made. This includes the development of a simple language capable of expressing the specialized model as well as an intermediate representation capable of storing the retargetable PCSs.

Finally, through the evaluation of an implementation of the proposed system, we have shown that the amount of performance overhead incurred by the retargetable approach is negligible. This is despite the fact that very little optimization was performed on the retargetable PCS by the intermediate representation and the Netfilter output generator. Based on this we therefore conclude, that also in practice, the use of retargetable specifications is a feasible approach and that the system and its accompanying implementation is fully usable.

# Bibliography

[ABG99] Steven McCanne Andrew Begel and Susan L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. 1999.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BM96] S. Bradner and J. McQuaid. RFC 1944: Benchmarking methodology for network interconnect devices, May 1996. Status: INFORMATIONAL.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[CK74] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Trans. on Commun., vol. COM-22, pp. 637-648*, May 1974.

[ea01] L. R. Olsen et. al. *Developing YAGCS - Source Packaging for the 3DVDM Framework*. 2001.

[ea02] M. Howard et al. *Writing Secure Code*. Microsoft Press, 2002.

[ea03] W. R. Cheswick et al. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, second edition, 2003.

[EB96] R. Elz and R. Bush. RFC 1982: Serial number arithmetic, August 1996.

[Fil04] OpenBSD Packet Filter. *http://www.benzedrine.cx/pf.html*. 2004.

[FMY97] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, 1997.

[Hom04] The Netfilter Homepage. *http://www.netfilter.org*. 2004.

[JSCO02] M. Dalum J. S. Christensen and L. R. Olsen. *MTIDD Based Firewalls - Can An MTIDD Based Firewall Accomodate Stateful Packet Filtering And Mangling In A Practical Way.* 2002.

[LPY97] K. G. Larsen, P. P., and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[Pos81a] J. Postel. RFC 792: Internet Control Message Protocol, September 1981.

[Pos81b] J. Postel. RFC 793: Transmission control protocol, September 1981.

[Roo] G. V. Rooij. Real stateful tcp packet filtering in ip filter.

[ST98] K. Strehl and L. Thiele. Symbolic modelchecking using interval diagram techniques. 1998.

[Sur04] Internet Software Consortium Domain Survey. *http://www.isc.org/ds.* 2004.

[Tra01] Internet Technical Ressources: Traffic. *http://www.cs.columbia.edu/∼hgs/internet/traffic.html.* 2001.

[tW00] Tapping On the Walls. *http://smallbusiness.itworld.com/4378/swol-1117-buildingblocks/page_1.html.* 2000.