ABSTRACT

VENKATAMOHAN, BALAJI. Automated Implementation of Stateful Firewalls in Linux. (Under the direction of Ting Yu.)

Linux Firewalls are the first line of defense for any Linux machine connected to a network and is responsible for implementing an organization's network security policies on individual systems. The GNU/Linux firewall architecture framework, popularly known as Netfilter, is a generalized framework with defined call-back functions in the network stack [4]. This framework provides security at various levels of abstraction starting from the packet level all the way up to the application level. Netfilter also provides stateful firewalling capabilities in which the decision to allow or deny a packet is dependent on whether that packet belongs to an already existing connection or not and also on the connection's past behavior.

The current stateful firewall feature provided by Netfilter, popularly known as conntrack, is useful but is not generic and is not easily configurable as it requires writing new kernel modules for stateful policies corresponding to unsupported protocols. Even for complex protocols that are supported by Netfilter, there exist separate helper modules which assist Netfilter in identifying packets belonging to that protocol. Also, the state machine defined by Netfilter does not allow adding new states for policies. In this thesis, a new model for implementing a stateful firewall policy, based on a state machine, is provided. In our approach to automated stateful policy implementation, we propose an XML-based stateful policy specification language which will capture the state of the policy to be implemented in terms of state variables and the transition conditions for state variable changes and consequently, the decision whether to permit or deny the packet to pass through. Users can focus on specifying policies using the language which is intuitive and can be formally verified. Users need not worry about the implementation details of the firewall as our tool will do the automated translation of a security policy file into a registered Netfilter rule.

Automated Implementation of Stateful Firewalls in Linux

by
Balaji Venkatamohan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Rudra Dutta	Dr. Xiaohui Gu

Dr. Ting Yu
Chair of Advisory Committee

DEDICATION

Dedicated to my family…

BIOGRAPHY

Balaji Venkatamohan was born on May19th, 1985 in Chennai, India. He obtained his Bachelor of Technology degree in Information Technology from St. Joseph's College of Engineering, an engineering institution affiliated to Anna University, Chennai, in 2006. He worked for two years as a software developer at Mahindra Satyam, a leading software services company in India, from 2006 to 2008, which he quit to join North Carolina State University, Raleigh, for his graduate studies in the Computer Science Department. He has been working under the guidance of Dr. Ting Yu, Department of Computer Science for his thesis research. His primary research interest lies in access control policies and policy implementation mechanisms.

ACKNOWLEDGEMENTS

I would like to thank my family for their unending support and motivation throughout my research period.

I would like to thank my advisor, Dr. Ting Yu, for his sincere guidance and support for my research work. His suggestions shaped my thesis work and his critical comments helped improve the quality of my work. I also thank Dr. Rudra Dutta and Dr. Xiaohui (Helen) Gu for kindly agreeing to serve in my thesis advisory committee.

I would also like to thank all my friends for their support and motivation during the course of my research work.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ixx

CHAPTER 1 Introduction 1

1.1 Contribution of this thesis 4

1.2 Organization of this thesis 5

CHAPTER 2 Linux Firewall Architecture 6

2.1 History 6

2.2 iptables 6

2.2.1 iptables tables 7

2.2.2 iptables syntax 9

2.3 Netfilter 10

2.3.1 Netfilter Packet Traversal 12

CHAPTER 3 Stateful Linux Firewall 14

3.1 conntrack 14

3.1.1 conntrack states 14

3.1.2 conntrack entries 15

3.1.3 conntrack Implementation 17

3.2 Issues with conntrack and the need for a new Stateful Firewall model 19

CHAPTER 4 Overview of Our Approach and A Stateful Policy Specification Language 21

4.1 Approach Architecture 21

4.2 Enforceable Security Policies 23

4.2.1 Execution monitoring class of policy enforcement mechanisms 23

4.2.2 Security automata 25

4.3 Applying Security Automata in our policy language 28

4.4 A Stateful Policy Specification Language 32

CHAPTER 5 Implementation 41

5.1 Modules 41

5.1.1 The header file 42

5.1.2 Userspace Module 46

5.1.3 Kernel Module 48

5.2 Compiling and registering the files as Xtables modules 53

CHAPTER 6 Related Work, Conclusions and Future Work 55

6.1 Related Work 55

6.2 Conclusions 58

6.3 Future Work 59

References 62

LIST OF TABLES

Table 4.1 List of keywords used for specifying transition conditions in our policy language file.....35

Table 4.2 List of keywords used for modifying state variable values 37

LIST OF FIGURES

Figure 2.1 A general syntax for MASQUERADE through rules in *nat* table……10

Figure 2.2 Priorities of iptables tables in Netfilter IPV4hooks……...12

Figure 2.3 Packet traversal in Netfilter ……...13

Figure 3.1 An example conntrack entry in the file /proc/ip_conntrack……...15

Figure 3.2 connection tracking structure…….....19

Figure 4.1 Approach architecture……...21

Figure 4.2 Security automata diagram for 'no file write op after file read op' policy.....26

Figure 4.3 guarded command representation of a security automaton for fig 4.2……..27

Figure 4.4 guarded command representation of a security automaton for a pattern count policy……...30

Figure 4.5 State machine for a stateful policy to drop the traffic coming from a UDP connection, if the total count of two specific patterns in the packet have crossed some maximum limit……31

Figure 4.6 skeleton of policy specification language……..33

Figure 4.7 A TCP policy written in our XML-based policy language……39

Figure 5.1 struct xt_new_mtinfo – the binary interfacet between the userspace and kernel modules……43

Figure 5.2 struct st_policy _parameters – this structure holds the IP addresses and port numbers range to which the policy should be applied, along with the TCP header field details, if the packet is of type TCP……...44

Figure 5.3 struct s_global _variables – a structure for holding a policy's state……..44

Figure 5.4 struct st_rules to hold the condition and action field arrays……...45

Figure 5.5 struct st_condition_action to hold the five possible parameters of a rule's condition or action fields……..45

Figure 5.6 structure inside userspace module to initialize the module……46

Figure 5.7 function to register the userspace module……..47
Figure 5.8 structure that is passed to the function xt_register_match which will register the kernel module……..48
Figure 5.9 init function which will is called as soon as our module is loaded in to the kernel. Local structures are allocated memory and static variables are initialized……..49
Figure 5.10 structure local_policy_table_struct which is used for storing policies mentioned in the XML file……...50
Figure 5.11 Structure local_state_table_st which acts as the local state table for connections that are currently being monitored by the tool…….51
Figure 5.12 A generic algorithm of our kernel module's operation for all IP packets……..52
Figure 5.13 An example iptables command that uses our module in the xtables-addons environment……....54
Figure 5.14 Generic Syntax for using our tool in the xtables-addons environment……..54

Chapter 1

Introduction

A firewall is a part of the computer system or network which is designed to block unauthorized access and connections while permitting authorized communications. A firewall is configured to permit or deny network traffic based on the access control policies of the organization of which the firewall is a part of [18]. Firewalls are implemented either as software or in hardware or a combination of both. There are mainly three types of firewalling techniques based on the OSI layer in which the firewall is acting on. They are (1) packet filters, which inspect each packet passing through them, (2) application Gateways which act on specific applications such as FTP, Telnet etc and (3) circuit level Gateways, which apply security mechanisms whenever TCP and UDP connections are established. Firewalls have evolved from being a simple tool performing packet level filtering, into a tool capable of performing deep packet inspections and handling stateful connections.

Linux has an extremely powerful firewall built in, referred to as iptables/Netfilter. iptables is a userspace module which allows users to interact with the firewall through the command line, letting users insert/delete/modify firewall rules. Netfilter is a kernel module, built into the kernel, where the actual filtering of network traffic takes place in accordance with the rules. There are three pre-defined set of tables in iptables to which the firewall rules can be inserted. They are (1) filter table, which contains a list of rules for matching against packets, (2) mangle table, which contains rules for marking or associating a value, specified by netfilter, with a packet and (3) NAT table, which contains a list of rules for modifying a packet's source and destination IP addresses, source and destination port numbers for the purpose of performing source Network Address Translation (SNAT) and destination Network Address Translation (DNAT).

Netfilter is a series of hooks located at various points through which a packet traverses in a network stack. There are five hooks defined in the Netfilter architecture. They are (1) NF_IP_PRE_ROUTING, (2) NF_IP_LOCAL_IN, (3) NF_IP_FORWARD, (4) NF_IP_LOCAL_OUT and (5) NF_IP_POST_ROUTING. After the initial sanity check of incoming packets, like checksum verification etc, to the host system is successful, the packets pass through the NF_IP_PRE_ROUTING hook. If the packets are destined for the local system, they pass through the NF_IP_LOCAL_IN hook. If the packets are not destined for the local system but for another interface instead, they pass through the NF_IP_FORWARD hook. Packets that originated from the host system pass through the NF_IP_LOCAL_OUT hook. All packets that are about to leave the system pass through the NF_IP_POST_ROUTING hook.

A stateful firewall keeps track of the state of network connections such as TCP streams and UDP connections that passes through the firewall. These firewalls are configured to distinguish legitimate packets for various types of connections and can reject packets that are not part of a legitimate connection. A firewall with stateful feature enabled is an extremely useful tool for network administrators, offering them fine grained control of network traffic. For example, without the stateful feature, there is no way to check whether a packet is a part of an existing TCP connection or not. A stateful firewall works by maintaining information about a connection in the system's memory. This information may include the source and destination IP addresses, source and destination port numbers, sequence numbers of the packets traversing the connection.

Linux has a connection tracking machine instead of a state machine. It is called conntrack which is a special kernel framework built on top of Netfilter. When this framework is called, it classifies each packet into one of four connection states of NEW, ESTABLISHED, RELATED and INVALID. Other modules may use this state information of a packet to make decisions as to whether to accept the packet or not. By default, the conntrack engine itself does not allow users to extend the stateful mechanism into

recognizing traffic that are part of a specific protocol such as TCP, UDP, ICMP etc. Instead there exist separate areas within conntrack to enable tracking of such protocols. For even more complex protocols such as FTP, IRC etc, conntrack helper modules are needed to support these protocols. The states of conntrack are also fixed and hence a new module has to be written in case the user needs to use a custom state machine with different states.

A new module has to be written every time the network administrator wishes to implement a new stateful policy feature if it is not supported by Netfilter. This new module should make sure that it implements the stateful policies relating to that protocol correctly because as of now there is no way of enabling a new module to correctly execute the policy for which it is being written. In short, there is no standard stateful model to follow. This puts a burden on the administrators of any Linux system to be dependent on other programmers' coding expertise in implementing a new stateful policy feature or be kernel programmers themselves. An ideal solution would be to have a single module which would take care of any stateful policy written for any protocol so that there is no need to write separate modules for any new stateful functionality or deploy different modules in the kernel for tracking packets of already existing protocols.

In this thesis, we propose a new XML based stateful policy specification language in which the state machine of the policy is captured in accordance with the security automata idea proposed in [1] in which, the state of the system for a set of actions is captured using a group of state variables, and the state changes due to specific actions on the system are captured through a set of transitional conditions. In our language, we are following exactly that and along with it we also mention the action to be taken by the firewall during each transition, if any. Note that in the case of Linux firewall's state machine, there are only two basic events which might change the state of a policy and subsequently the decision to drop or accept the packet. One is incoming packet to the system and the other is outgoing packet from the system. To process this XML based language, we have written a generic kernel

module to implement the stateful policy specified in the proposed language's file. Testing has been performed with various stateful policies with varying degrees of complexity.

Our tool was successfully tested for implementing TCP connection tracking policy, UDP pattern matching policy, ICMP related policies and also policies restricting the number of packets and bytes coming from a connection endpoint. We found that there was no need to change our code to implement the above policies and also found that our tool was working when the state machine for a policy was changed by changing the number of state variables defining the state of a policy. If our tool is improved with the ideas presented in the future works section, it could be claimed as an alternative for conntrack.

1.1 Contribution of this thesis

The following are the contributions made by this thesis,

1. This thesis proposes a new approach to model a Linux stateful firewall.
2. We propose a new policy specification language for specifying a Linux stateful policy. The policy is specified in terms of state variables representing the current state of the policy and the transition conditions of a rule within the policy along with the action that needs to be taken if these conditions are met.
3. We have written Netfilter modules which can translate the policy specified in the policy language file into a Netfilter rule and also apply the rules present in the policy on matching packets hitting our Netfilter module's callback function.
4. We attempt to reduce the burden of Linux firewall administrators by eliminating the need to load additional helper modules along with the conntrack module for tracking certain protocols and also by relieving them from having to remember different iptables syntaxes associated with using these modules.

1.2 Organization of this thesis

The organization of the thesis is as follows: Chapter 2 provides an introduction to Linux firewalls, particularly, iptables and Netfilter. Chapter 3 provides a detailed explanation on the working of conntrack, Netfilter's connection tracking tool. Chapter 4 discusses our work's approach architecture and our new XML based policy specification language. Chapter 5 discusses how we implemented our tool in Netfilter. Chapter 6 concludes and gives ideas for improving our tool so as to make it more generic and being able to replace conntrack.

Chapter 2

Linux Firewall Architecture

2.1 History

The current iptables/Netfilter mechanism which is used to implement firewall in Linux was introduced only since the 2.4.x versions of Linux. It superseded the IPFW (IPFireWall) mechanism. The term IPFW is used to collectively call the ipchains [6] and the ipfwadm firewalling mechanisms that were used in the previous versions of Linux. ipchains superseded ipfwadm [5] and was introduced in 2.2.x version of Linux. The most important difference between iptables/Netfilter and the IPFW mechanisms lies in the path packets take as they pass through various check points within the operating system. They also differ on how the firewall rule set is constructed. Another notable difference is that IPFW is a stateless firewall whereas stateful functionality was added to the Linux firewall only through the introduction of iptables/Netfilter mechanism. Iptables/Netfilter also fully implements source address translation (SNAT), destination address translation (DNAT) and masquerading, which is a special form of SNAT.

2.2 iptables

iptables is a user space application that allows a system administrator to add firewall rules using one or more registered netfilter kernel modules. It allows a system administrator to configure iptables tables and netfilter hooks. It is also possible to delete rules present in the hooks and affect the order in which rules are called within a hook, using iptables. This application requires root access to operate. On most Linux systems it is installed under the folder /usr/sbin/iptables and is documented in its manpage. The kernel module x_tables is

inclusively referred by iptables for its implementation [8]. iptables preserves the basic ideas introduced in ipfwadm, which is to have one or more lists of rules specifying what to match in a packet and the action that needed to be taken with that packet when a match is found. The difference between ipchains and iptables is that ipchains added chains of rules whereas iptables added tables of rules. iptables has three individual tables: filter, NAT and mangle table.

2.2.1 iptables tables

iptables uses the concept of having separate rule tables for different kinds of packet matching and processing functionalities. The rule tables are implemented as functionally separate modules. The three primary modules are (1) the rule *filter* table, (2) the NAT *nat* table and (3) the specialized packet handling *mangle* table. Unless these table modules and its own associated module extensions are built directly into the Linux kernel, they are dynamically loaded when first referenced [9].

***filter* table**

The *filter* table is the default table. There is no need to specify the table name in an iptables rule. Rules can be applied on the INPUT, OUTPUT and FORWARD chains and on user defined chains. Operations involving IP header match for protocol, source and destination addresses, input and output interfaces, fragment handling, can be specified through rules inserted into this table [9].

***nat* table**

iptables NAT module supports source NAT (SNAT) and destination NAT (DNAT). Rules added to the *nat* table allow modification of a packet's source or destination address and port. It has three built-in chains. The PREROUTING chain specifies destination changes to incoming packets before passing the packet to the routing function. This is known as DNAT. The OUTPUT chain specifies destination changes to packets generated locally and

before the routing decisions are made and they are known as DNAT and REDIRECT. This is usually done to transparently redirect locally generated packets to a local proxy or to port-forward to a different host. The POSTROUTING chain specifies source changes to outgoing packets (SNAT, MASQUERADE). The changes are applied after the routing decision has been made [9].

***mangle* table**

Rules specified in the *mangle* table allows marking or associating a value, specified by netfilter, with a packet as well as making changes to a packet before sending it to its destination. The mangle table has five built-in chains. The PREROUTING chain specifies changes to incoming packets as they arrive at an interface, before any routing decision has been made. After the PREROUTING chain is traversed, the INPUT chain specifies changes to packets as they are processed. For locally generated packets, the OUTPUT chain is traversed first before sending the packets to the POSTROUTING chain, where further changes are made as required by the rules entered in the NAT table. The FORWARD chain specifies changes to packets that are forwarded through the firewall [9].

Apart from the standard set of conditions which iptables provides for matching a packet against a rule and the standard set of actions that can be taken on a packet once there is a match, the three iptables tables can be extended to provide one of match extensions or target extensions or both. *filter* table provides both match and target extensions. The *filter* table match extensions provide access to the fields in the TCP, UDP and ICMP headers, as well as the match features available in iptables. The stateful tracking of packets which Netfilter/iptables provides is implemented through one such match extension called state *filter* table match extension. The *filter* table target extensions include logging functionality and the capability to reject a packet rather than dropping it, which means an ICMP type 3 error messages can be sent to the sender of that packet which was rejected. The *nat* table target extension provides five different target extensions, each providing the rules inserted

into the *nat* table, the ability to implement Source NAT, Destination NAT, MASQUERADE, REDIRECT and BALANCE. The *mangle* table provides two target extensions. One is the mark *mangle* table target extension which is used to set the value of the Netfilter mark value for this packet. The other is the *TOS mangle* table target extension which sets the TOS value in the IP header. There are no match extensions for *nat* table and *mangle* table [9].

2.2.2 iptables syntax

An iptables rule would normally consist of three parts. The table chain to which the rule is being added/modified/deleted is specified first. If the table is not *filter* table, the name of the table must be specified explicitly in the rule. Next the matching parameters for that rule such as the source IP, destination IP, ports etc are specified and finally the action to be taken in case there is a match for that traffic entity is specified. The action to be taken, called as TARGET, is normally a command to accept or to drop the packet. Additional options to log the packet for the userspace as well as marking it for future actions are also available. An iptables table can take more than one rule. Rules are added to each table in a list and an option to specify the priority of these rules within the table is also available [9].

When a packet enters the Linux firewall system, it is matched against the rules present in these tables, starting from the top of the rule list. When a matching rule for the packet is found, the action specified for the traffic in that rule is applied and no further rule is applied to that packet. If there is no match for the packet against all the rules present in the table, the default action is applied to that packet. This is called as the default policy for that table and it may be to either ACCEPT or DROP the packet. When the default policy is to DROP the packet, rules are entered into the table which will specify the conditions under which a packet may be accepted by the firewall. This policy is normally applied to the INPUT table, which controls what traffic is allowed into the host system.

iptables –t nat –A POSTROUTING –out-interface <interface> -j MASQUERADE [--to-ports <port>[--<port>]]

Figure 2.1: A general syntax for MASQUERADE through rules in *nat* table

When the default policy is to ACCEPT the packet, conditions under which a packet may be disallowed by the firewall are specified through a set of rules into that table. This policy is applied to the OUTPUT table, where the firewall trusts the traffic leaving the host system [9]. A general iptables syntax for masquerading is given in figure 2.3.

2.3 Netfilter

Netfilter is a generalized, protocol independent framework with defined callback functions in the network stack. It was founded by Paul "Rusty" Russel during the development of Linux 2.3.x series along with the corresponding user space iptables feature. It comprises of a set of hooks over the Linux network protocol stack. The hooks are hardcoded in layer 3 stacks (IPV4, IPV6, ARP). Multiple kernel modules can register callbacks with each of these hooks. These callback functions will be called for every packet traversing a particular hook in the network stack. iptables, the userspace program which we explained in section 2.2 and netfilter are often confused with each other to mean the same tool. But iptables is built on top of Netfilter framework and they are not same.

Netfilter inserts five hooks into the Linux network stack for IPV4, to perform packet handling at different stages [10]:

NF_IP_PREROUTING hook: All packets hit this hook and they reach this hook before the routing decision is taken and after all the IP header sanity checks are fulfilled. Sanity checks are done to check whether the packet was damaged or changed during its traversal. These checks include checking if the IP checksum was changed during transit, checking if the packet is truncated and checking if it is a promiscuous receive. Network Address Port Translation (NAPT) and Destination NAT (DNAT) are implemented in this hook.

NF_IP_LOCAL_IN hook: All packets that are destined for the local machine reach this hook. This hook is reached after the routing decision has been made.

N_IP_FORWARD hook: Packets that are not intended for the local machine reaches this hook.

NF_IP_LOCAL_OUT hook: This is the first hook in the outgoing packet's path.

NF_IP_POST_ROUTING hook: All packets leaving the local machine reach this hook. This hook is reached after the routing decision has been made. Source NAT (SNAT) operation is performed at this hook.

Kernel modules can register to listen at any of the hooks. A module that registers a function must specify the priority of the function within the hook. When that Netfilter hook is called from the core networking code, each registered module with that hook is called in the order of priorities, and is free to manipulate the packet. The module can then tell Netfilter to take one of the five below mentioned actions on the packet [10]:

NF_ACCEPT: allow the packet in its normal traversal route.

NF_DROP: drop the packet; do not let it continue its traversal.

NF_STOLEN: the packet has been taken over by the module; do not continue traversal.

NF_QUEUE: queue the packet (usually for userspace handling).

NF_REPEAT: call this hook again.

The Netfilter framework applies the final decision of the policy. The modules or functions that register with a hook are called in a particular order within that hook. For example, in the case of IPV4, the order in which the built-in tables are called within each hook is given in the figure 2.4 [10].

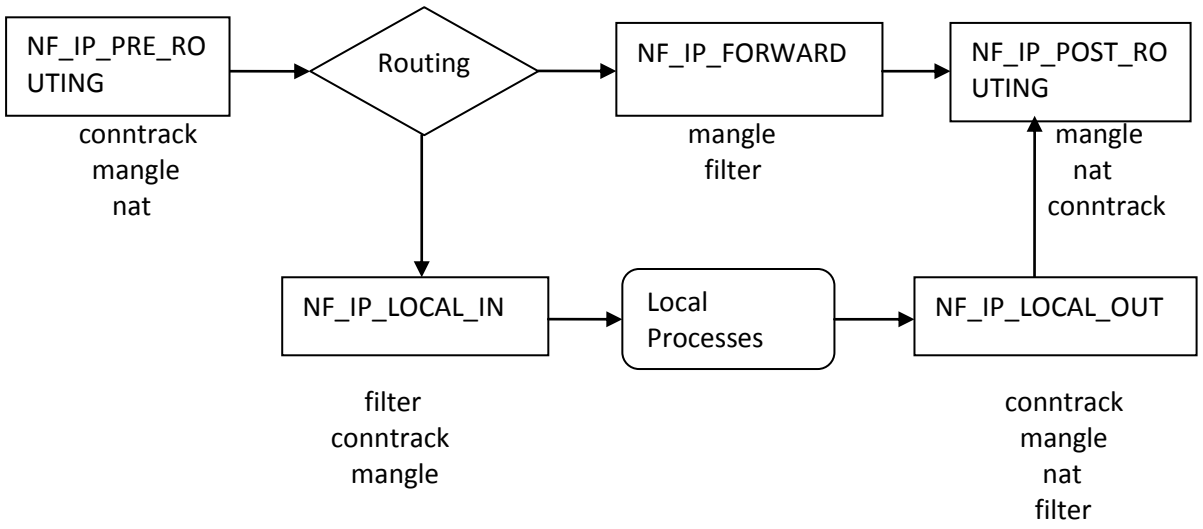

Figure 2.2: Priorities of iptables tables in Netfilter IPV4hooks

2.3.1 Netfilter Packet Traversal

Under Netfilter, built-in INPUT, OUTPUT and FORWARD filter chains are used. Incoming packets pass through the routing function, which determines whether to deliver the packet to the local host's input chain or on to the forward chain. Locally generated packets are passed to the OUTPUT chain's rules. If the packet is accepted, it is sent out of the appropriate interface. Thus each packet is filtered once, except for the loopback packet which is filtered twice. If a locally destined packet is accepted by the INPUT chain's rules, the packet is sent out of the appropriate interface. This packet flow is captured in the figure 2.5 below.

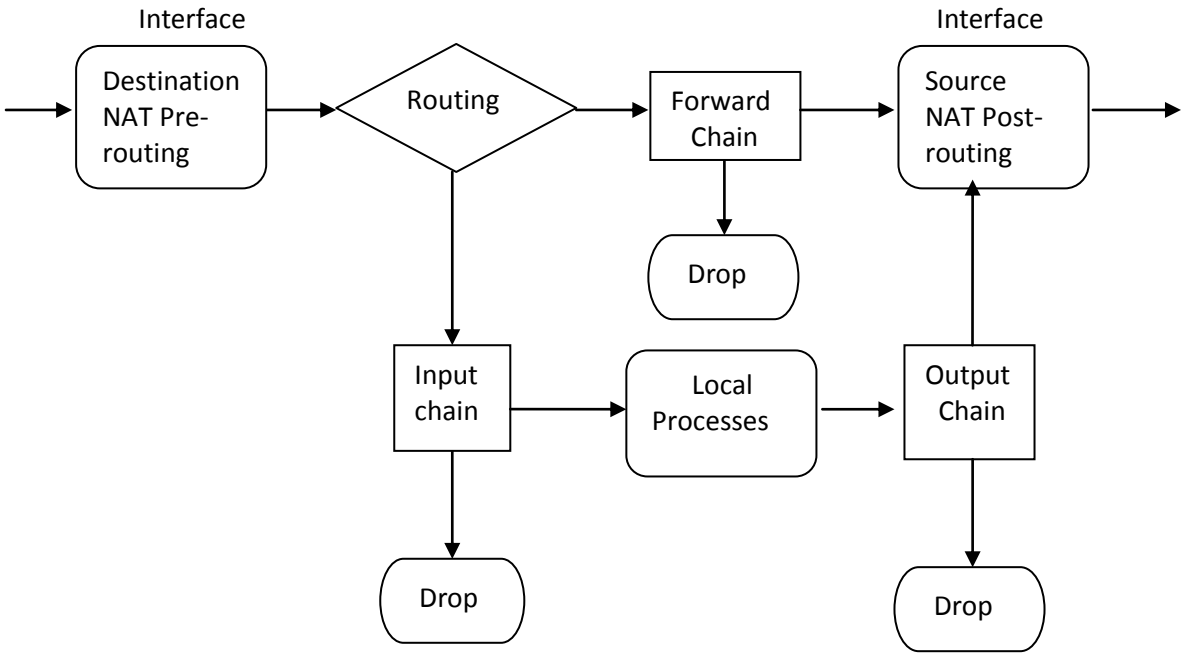

Figure 2.3: Packet traversal in Netfilter

Chapter 3

Stateful Linux Firewall

Linux does not have a state machine built into it. Instead it has a connection tracking machine. Firewalls in Linux which implement stateful filtering of packets and connections using this connection tracking mechanism are called Linux stateful firewalls [11]. Packet filtering based firewall techniques are not sufficient for protection against probes and Denial of Service attacks. This prompted Linux developers to implement a stateful firewall in Linux that could tell whether a packet belongs to an existing connection or not. Connection tracking was conceived during the birth of Netfilter project. This mechanism is implemented by a special kernel framework called conntrack which can be loaded as a module or can be made to be an internal part of the kernel itself. It is implemented on top of Netfilter.

3.1 conntrack

conntrack works by classifying each packet which enters or leaves or passes through the system into one of the four connection tracking states of NEW, ESTABLISHED, RELATED and INVALID. conntrack is handled in the PREROUTING chain of Netfilter hook for incoming traffic. For outgoing traffic, it is handled in the OUTPUT chain.

3.1.1 conntrack states

conntrack defines the above four states to classify whether a packet is a part of a connection or not. All the state changes and calculations are done within the PREROUTING and OUTPUT chains of the NAT table. For locally generated packets, when the initial packet in a stream is sent, the state gets set to NEW within the OUTPUT chain [11]. For example, a SYN packet which is sent from the local system is classified as NEW if it is the first packet

in a connection. If the first packet in a connection is not originated by the local machine, the NEW state is set within the PREROUTING chain. When a corresponding return packet is received, the state gets changed in the PREROUTING chain to ESTABLISHED. For example, ICMP reply messages received can be considered as ESTABLISHED if the local machine had previously sent an ICMP request message.

A connection is considered to be RELATED when it is related to an already ESTABLISHED connection which would have spawned another connection outside the main connection. INVALID state means that the packet cannot be identified. The reasons may vary from the system running out of memory, receiving ICMP error messages that do not correspond to any known connections, etc. iptables/Netfilter advises users to drop packets that are in this state. If a packet is marked within the raw table with NOTRACK target, then this packet will show up as UNTRACKED in the state machine. The raw table was created just for this purpose. It allows for setting a NOTRACK mark for packets so that they are not tracked by conntrack. NOTRACK option can be applied on individual packets and not on individual connections. It can be applied for instances where only incoming and outgoing packets in a heavily trafficked router need to be tracked and not the passing by traffic.

3.1.2 conntrack entries

The conntrack module maintains a list of all the current connections being tracked by the module. This information is stored in the file /proc/net/ip_conntrack. If the ip_conntrack module is loaded, a cat of /proc/net/ip_conntrack will yield entries similar to the example given in the figure 2.8 below [17].

```
tcp      6 90 SYN_SENT src=192.168.1.6 dst=192.168.1.9 sport=32775 \
dport=22 [UNREPLIED] src=192.168.1.9 dst=192.168.1.6 sport=22 \
dport=32775 [ASSURED] use=2
```

Figure 3.1: An example conntrack entry in the file /proc/ip_conntrack

The above example contains an entry which contains enough information to inform the conntrack module about the specific connection state that connection is in. The first parameter in the conntrack entry specifies the protocol which, in this example, is TCP. Next, the name of the protocol is represented in the normal decimal coding. The next parameter specifies the amount of time in seconds the conntrack entry can live within the file. This value is decremented continuously. The fourth parameter specifies the actual state that the connection is in at that instance. This state is different from the conntrack states and is dependent on the type of protocol of that connection. The next four parameters of the conntrack entry specify the source and destination IP addresses and the source and destination ports of the connection. The keyword UNREPLIED in the example means that no return traffic has been received in the opposite direction for that conntrack entry as yet. The next four entries correspond to the ports numbers and IP addresses of the source and destination machine of the packet that is expected by that conntrack entry as a reply. When the machine receives a packet in the opposite direction corresponding to the entry given in the example, conntrack will erase the UNREPLIED flag and will reset it. The ASSURED flag specifies that this connection is assured and that it will not be erased if the maximum possible tracked connections are reached. Thus, connections marked as ASSURED will not be erased. The number of connections that the connection tracking table can hold at any moment depends upon a variable that can be set through the ip-sysctl functions in recent kernels [17]. The default value of this variable depends on the RAM of the machine on which the conntrack module is running. From [17], we understand that on a 128 MB RAM machine, 8192 possible conntrack entries are allowed and on a 256 MB RAM machine, 16376 possible conntrack entries are possible.

3.1.3 conntrack Implementation

Basic Structure of conntrack

conntrack is implemented as an optional modular loadable subsystem. Eventhough it is an optional module, it is required by NAT subsystem. It is implemented with a hash table to perform efficient lookups. Each bucket of the hash table has double linked list of hash tuples. There are two hash tuples for every connection: One for original direction and one for the reply direction. A tuple represents the relevant information for a connection such as the source and destination IP addresses, source and destination port numbers and layer-4 information etc. Such tuples are embedded in a hash tuple. Both structures are defined in the header file nf_conntrack_tuple.h. The two hash tuples that represent a connection are embedded in the structure nf_conn, also called as conntrack. A tuple is embedded in a hash tuple and two related hash tuples are embedded in a conntrack [13]. The result is that there are three layers of embedded structures. Jenkins hash is used for hashing [13] and the input parameters for the hash function are the relevant layer-3 and layer-4 protocol information. There is an upper bound on the number of connections that can be stored in the hash table and it is tunable during conntrack module load or at kernel boot time. If the hash table fills up, the evicted conntrack will be the least recently used of a hash chain. Figure 3.2 gives an indication as to how a connection is represented in a hash table [13].

The *conntrack* creation and lookup process

The callback function for conntrack nf_conntrack_in is registered in the PREROUTING hook [13]. Before this function is called for a packet, sanity checks are performed to ensure that the packet is correct. If the packet clears the sanity check, a matching conntrack entry for that packet is searched for through the function resolve_normal_ct present in nf_conntrack_core.c file. If no matching conntrack is found for the packet, it is created. The newly created conntrack entry will have the CONFIRMED flag unset. This flag will remain unset until the packet arrives at the last hook without being

dropped. The confirmation of a conntrack entry happens through the callback function nf_conntrack_confirm which is registered with the LOCAL_INPUT and the POSTROUTING hooks. The conntrack entry is inserted into the hash table, the CONFIRMED flag is set and the associated timer for the conntrack entry is activated. The association between a packet and a conntrack is established by means of a pointer. If the pointer is null, then the packet belongs to an invalid connection.

Tracking complex protocols with *conntrack* helper modules

Certain protocols are complex and are difficult to track using *conntrack* alone. Some examples are FTP, IRC and TFTP etc. Packets belonging to these protocols carry information within the actual data payload of the packets, and hence require special connection tracking helpers to enable them to function correctly. In both active and passive FTP, for example, the firewall needs to scan the data part of these packets to know about the data port that need to be opened for enabling FTP data sessions. conntrack resolves this problem by adding a special helper to the connection tracking module which will scan through the data in the control connection for specific syntaxes and information [13]. When it finds the relevant information, it will add that specific information as RELATED and the system will be able to track the connection. Protocols such as FTP, IRC and TFTP have already been provided with conntrack helpers.

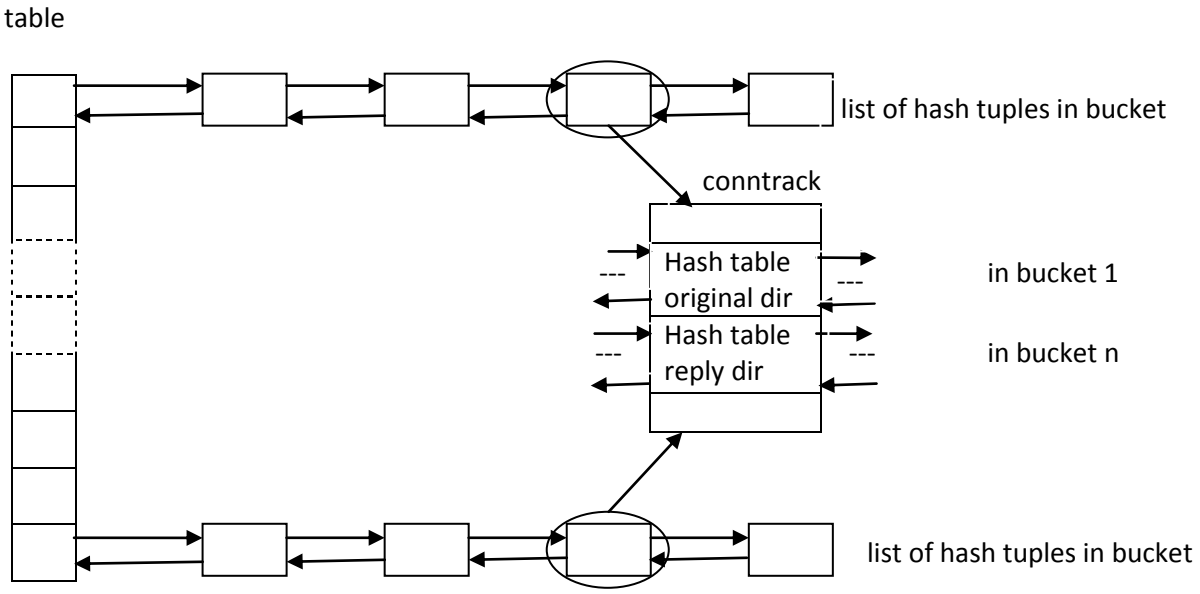

Figure 3.2: connection tracking structure

For protocols unsupported by conntrack, a feature called patch-o-matic, which is a system to apply Netfilter-related patches to the latest kernel and/or iptables sources, might have the helpers. Protocols such as H.323 have helpers defined in patch-o-matic [13]. If patch-o-matic does not have the helpers then the administrators are left to write their own helpers.

3.2 Issues with conntrack and the need for a new Stateful Firewall model

By default, the features of the conntrack mechanism does not allow users to extend the stateful mechanism into recognizing traffic that are part of a specific protocol such as FTP, IRC and TFTP etc. Instead there exist separate modules for each of these protocols, which, when used along with conntrack, will help in determining whether a packet belongs to a specific higher level protocol or not. Therefore, in case there is a need to use the stateful features of Linux firewall to track packets belonging to an entirely new protocol or a custom

developed protocol, a new kernel module has to be written specifically for that protocol. Also, for tracking TCP, ICMP and UDP protocols, conntrack has special segments to deal with these protocols which are hardcoded and are not generic. There is an additional burden of having to make sure that the module implements the stateful policies relating to that protocol correctly. The states of conntrack are fixed and sometimes might not correspond to the actual states of a protocol which uses it. There is no way to define custom states for protocols. The state machine does not follow a clear model through which state changes can be defined. In short, there is no standard stateful model to follow for Linux stateful firewalls. The disadvantage of not having a standard model to allow is that further research and development on stateful Linux firewalls is restricted. Also, different vendors hard code their implementations on existing stateful firewall in Linux in case a new feature is needed. This is because of the absence of a model.

In this thesis, we attempt to provide a model of a Linux stateful firewall which can be made generic for all protocols. We also attempt to provide a complete state machine with constants, variables and transitions of these variables that are responsible for the state changes, in accordance to the security automata defined in [1]. The next chapter explains the security automata of [1] in detail and also explains the new XML based stateful policy specification language which we arrived at using the idea of security automata.

Chapter 4

Overview of Our Approach and A Stateful Policy Specification Language

This chapter deals with the approach architecture that we have designed and discusses the concepts in [1], especially the concept of security automata. Our new XML based policy specification language, along with its syntax is also discussed in this chapter.

4.1 Approach Architecture

The main idea of this thesis is to reduce the burden on Linux administrators trying to implement a new stateful policy. Our approach requires an administrator only to have a thorough understanding of the stateful policy and the ability to imagine the policy in terms of state changes in a state machine. The administrator should also be proficient in XML. These being the only constraints on the administrator, the architecture of our thesis's approach are depicted in Figure 4.1

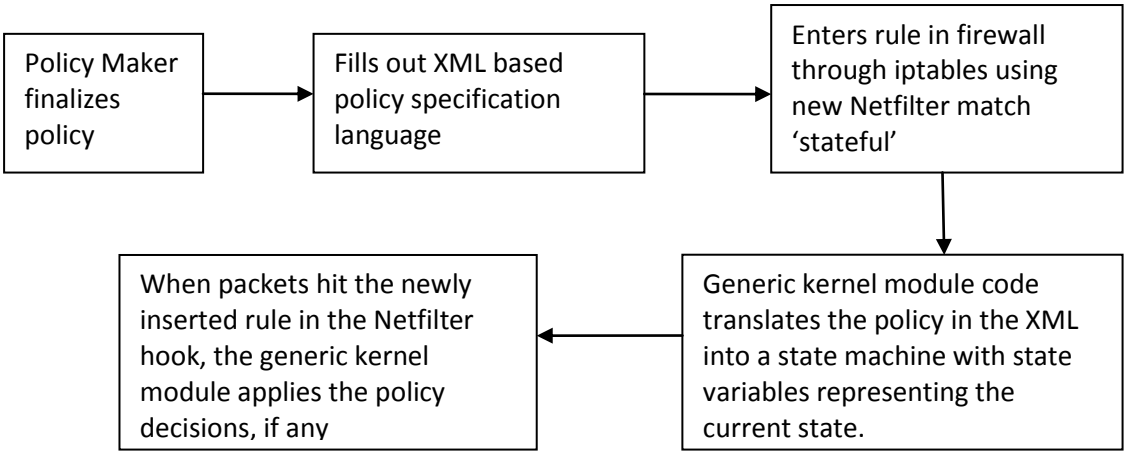

Figure 4.1 Approach architecture

After the policy makers agree on the stateful policy that has to be implemented by the firewall, for example, restricting the number of TCP connections with a particular host or a network, the policy makers come up with a state machine for the policy by identifying the various states possible in the policy along with variables that represent a state, constants and the state transition conditions. The technique of identifying the states in the state machine and the associated variables, constants and conditions are borrowed from the idea of security automata proposed in [1]. We chose the concept explained in [1] because it was the closest model of a stateful policy that we came across. A summary of security automata is described in section 4.2. After the policy's state machine is identified, it is captured in the new XML based policy specification language.

Netfilter/iptables architecture allows coders to add new features to its architecture. Using this flexibility provided by Netfilter, we have created a new Netfilter match extension to handle our implementation and we have named it as stateful match. We have written one userspace module and one kernel module to handle this new match condition and we have used the Xtables add-ons package to load these modules into the kernel. Once successfully loaded, our new match condition is now ready to handle stateful policy specified in the policy specification XML file. iptables is used to insert a new policy using this match condition and an example iptables command for the stateful match is given below:

```
iptables –A input –m stateful --xml pattern_match.xml --srcip 10.0.2.15 --dstip   34.45.56.67
–sport 34000  --dport 45550
```

The userspace module is responsible for parsing this command and passing on relevant policy information contained in the XML file to the kernel module. The kernel module implements this policy on incoming packets/connections, by matching the relevant parameters in the packet against the policy's rules' conditions. If there is a match with a rule's conditions, the actions specified in the rule are applied. Further implementation details are explained in chapter 5. The next section describes the ideas in [1] based on which we have come up with our policy specification language.

4.2 Enforceable Security Policies

General purpose security policies might concern access control to allow only authorized users, information flow to allow only permissible read and writes, availability of the entire system to its users, etc. These days application dependent and special purpose security policies attract the most attention. For example, the security policies to prevent information leakage in mobile code like java, the security policies for e-commerce transactions are application specific security policies. A precise characterization of security policies that are enforceable by monitoring the execution of the system and a security automaton for precisely this class of security policies is introduced in [1]. Definitions of security policies are also given. [1] stresses on the principle of least privileges and explains the value of application dependent and special purpose security policies. The policy enforcement mechanism depends on the state of a system and the attributes of the system to make the decision.

4.2.1 Execution monitoring class of policy enforcement mechanisms

The practicality of a security policy is evaluated in [1]. It decides whether a security policy is enforceable or not and if it is enforceable, at what cost it is enforceable. This work in [1] specifically deals with a class of enforcement mechanism called Execution Monitoring or EM in short. EM may include security kernels, reference monitors, firewalls and operating systems. The system whose execution steps are monitored is called as a Target. It may include objects, modules, processes, sub systems. The execution steps that may be monitored include fine grained actions such as memory access and higher level operations such as method calls and operations that change the security configuration of the target, resulting in halting of further executions on that target.

Enforcement mechanisms that can be classified as EM are limited. Mechanisms which use more information than those obtained by observing only the steps of a target's execution are excluded from being considered as EM mechanisms [1]. Hence, enforcement

mechanism tools such as compilers, theorem provers, and other tools which depend on predicting the future steps that a target might take, cannot be considered as EM mechanisms. This is because information regarding the future steps of a target is not available to mechanisms that can be considered as EM mechanisms [1]. Formally characterizing what can and cannot be accomplished using mechanisms in EM has both practical and theoretical usage. From a theoretical perspective, such a characterization gives a class of security policies that is based on a mathematical semantics of programs [1].

Defining a security policy

An attempt is made in [1] to give a formal characteristic to EM class of enforcement mechanism. Given ψ, a universal set of all possible finite and infinite sequences of target execution, $\sum_s$, a subset of sequences of ψ defining the execution of a target S, Schneider defines a security policy by giving a predicate on the set of execution steps for a target [1]:

A target *S* satisfies a policy *P* if and only if *P* ($\sum_s$) is true.

The requirements for the definition of a security policy demands that such a definition be broad enough to include things that are security policies and the definition be independent of how EM is defined. Given a security policy *P* and sets of executions $\sum$ and π, if *P* ($\sum$) is true, it is not necessary that *P* (π) is true even if π is a subset of $\sum$. This is because imposing such a condition will eliminate information flow from being considered a security policy [1].

Property

Alpern and Schneider define a term called property in [19]. A property, σ, is a set of finite and infinite states:

$$\sigma = S_0, S_1, S_2, \ldots$$

where S_0, S_1, …, can be considered as being the states that a target can take and where each state S_i results from an atomic action taken on a previous state S_{i-1}. A set of executions is called a property if set membership is determined by each execution step alone and not by

other members of a set [19]. The paper [1] suggests that all security policies must be a property if that policy is to have an enforcement mechanism in EM. For example, if a security policy that can be enforced by EM has a predicate of the form *P* (π), then each and every step α of π must individually satisfy the predicate ṗ (α) and should not be dependent on the values from the executions that happened before or will happen after it. Not every security policy is a property. An example is information flow. This is because to decide whether information can flow from A→B depends upon what value B takes in other possible executions. Therefore, a predicate to specify the sets of executions for an information flow cannot be written using predicates defined on single executions individually [1].

Safety Property

Alpern and Schneider give a definition to a term called safety property [19]. For a property *P* to be a safety property, if *P* does not hold for a target's execution, that is, if the set of sequences in that target's execution are not contained in the set of states defined by *P*, some undesirable event will arise as a result of it but the point at which such an undesirable event will happen must be identifiable [19]. Only then the property *P* is considered to be a safety property. Examples of safety properties are mutual exclusion, deadlock freedom and first come first serve. In mutual exclusion, the undesirable event is two processes executing inside the critical section at the same time and in first come first serve the undesirable event occurs when servicing a request that was made after another request that has not been service yet [19].

4.2.2 Security automata

Enforcement Mechanisms in EM terminates target executions that do not satisfy the predicate defined for the policy being enforced on the target. It also requires that all members of the execution set satisfy the predicate and that they all be safety properties. Recognizers for the sets of execution for safety properties serve as the basis for enforcement mechanisms

in EM. A class of Buchi automata that recognizes safety properties is called as security automata [1]. A security automata is defined by

- A countable set of automaton states
- A countable set of initial automaton states
- A countable set of input symbols and
- A transition function on the input symbols that act on the target.

Let us consider an example to explain security automata. The security policy for which we are designing the security automata states that a system should not allow a write operation immediately following a file read operation. Look at figure 4.2 for this example [1].

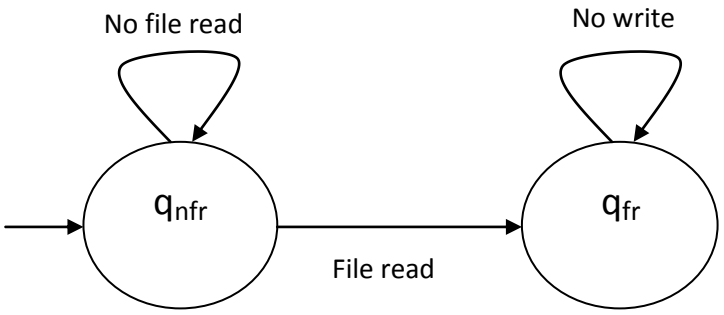

Figure 4.2 Security automata diagram for 'no file write op after file read op' policy

The above figure represents the security automata diagram for the example policy. The intention is to block file write operations immediately following a file read operation. The state q_{fr} represents the state of the system after a file read operation has been carried out and q_{nfr} represents the state of the system when no file read operation has taken place. Assuming that q_{nfr} is the initial state of the system, when no file read operation takes place, the state of the system remains the same. When a file read operation takes place, the state changes from q_{nfr} to q_{fr}. When the system is in the state q_{fr}, only those operations where no writes are involved are allowed. In the diagram there is no transition given for a file write

operation when the system is in q_{fr} state. Therefore, when a file write operation is issued, it will not be executed thus making sure that a write operation does not take place immediately following a file read operation.

In the above example, the number of states is just two and hence drawing the security automata is easy. For complex policies having many states, drawing such a security automata diagram can be difficult to draw, read and understand. To resolve this issue Schneider proposes the use of guarded commands [1]. As an example, in the guarded command B → S, the state transition defined by a program fragment S occurs whenever predicate B is satisfied by the current input symbol and the current state of the automata. B is the guard and it is a predicate which can refer only to the current input symbol and to the variables encoding the current state of the automata. S is the command and it is a computation that updates only the variables encoding the current state of the automation.

Figure 4.3 gives a guarded command representation of the security automata captured in the figure 4.2 [1].

State – vars:
State :{0,1} initial 0

Transitions:
Not file read ^ state=0 → SKIP
File read ^ state=0 → state:=1
No write ^ state=1 → SKIP

Figure 4.3: guarded command representation of a security automaton for fig 4.2

Some important factors to consider before employing a security automata in EM enforcement mechanisms are cost factors in employing the automata, space requirements and whether the automata itself can be compromised by input symbols or not. Once it has been decided to go

ahead with the automata, it is extremely important that a suitable representation for the current automaton state is employed and that the guarded commands correctly captures all the transitions that are allowed in the system in accordance with the policy.

4.3 Applying Security Automata in our policy language

The security automata idea in [1] is used to give a structure to our XML based policy language and the syntax for that language is explained in section 4.4. The idea here, as explained in the previous section, is to visualize the stateful policy in terms of a state machine, where a state is represented by one or more variables and the transition functions for the state changes is given in terms of changes to the policy variables. The transition functions which we call conditions and the changes to the policy variables which we call actions are both wrapped in a rule element. We allow a policy to have multiple rules. When a policy is applied to a packet, we check against all the rules in the policy as long as the packet has not satisfied a rule whose action specifies that the packet be dropped. The ordering of rules present in a policy should be decided by the policy maker and it is important because it will affect how well the policy is applied on matching packets. Our tool does not have the intelligence to check the ordering of these rules so as to make sure that the policy is implemented effectively. Our idea is best explained by using one or more stateful firewall policies and its corresponding state machines.

Consider the state machine given in the figure 4.4. The figure represents a state machine for stateful firewall policy which will match on two patterns within an IP packet and will increment one of the two state variables p1 and p2, depending which pattern matched against the IP packet. When the value of p1 is greater than 40 and the value of p2 is greater than 20, the packet will be dropped if there is any further match with either one of the pattern for all packets in that connection. Assuming that p1 is incremented when, say, a pattern of ‘bittorrent’ is found in a packet belonging to a UDP connection and assuming that p2 is incremented when, say, a pattern of ‘limewire’ is found in a packet belonging to the same

UDP connection, the guarded command representation for such a policy is shown below in figure 4.4. The two state variables are p1 and p2 with both having initial value as zero. If the value of p1 is less than 40, the policy is said to be in one state and if the value of p1 is greater than 40 the policy is in another state. Similarly, if the value of p2 is less than 20, the policy is said to be in one state and if the value of p2 is greater than 20 the policy is in another state. Hence there can be four states for the policy by combining these two state variables. The first state is when p1 and p2 are less than 40 and 20 respectively. The policy moves to another state when p1 is less than 40 but p2 is greater than 20. When p1 is greater than 40 and p2 is lesser than 20, the policy moves to a new state. When p1 and p2 are greater than 40 and 20 respectively, the policy is said to have been violated. These logical combinations of p1 and p2 constituting a state is understood by the transition conditions given in the automaton. As it can be seen in figure 4.4, when the policy is not violated, the automaton instructs the enforcement mechanism, in our case, the Linux stateful firewall, to allow the packet take its normal course. When the policy is violated, the automaton instructs the enforcement mechanism to drop the packet.

State-Vars:

State: p1{ <40, >40}: initial p1=0
State: p2 {<20, >20}:initial p2=0

Transitions:

Pattern_match (bittorrent) ^ (p1<40) ^ (p2<20) → p1++; ACCEPT
Pattern_match2 (limewire) ^ (p1<40) ^ (p2<20) → p2++; ACCEPT

Pattern_match1 (bittorrent) ^ (p1>40) ^ (p2<20) → p1++; ACCEPT
Pattern_match2 (limewire) ^ (p1>40) ^ (p2<20) → p2++; ACCEPT
Pattern_match1 (bittorrent) ^ (p1<40) ^ (p2>20) → p1++; ACCEPT
Pattern_match2 (limewire) ^ (p1<40) ^ (p2>20) → p2++; ACCEPT

Pattern_match1 (bittorrent)^ (p1>40) ^ (p2>20) → DROP
Pattern_match12 (limewire) ^ (p1<40) ^ (p2>20) → DROP

Figure 4.4: guarded command representation of a security automaton for a pattern count policy

Figure 4.5 captures the transition between states for the guarded command given in figure 4.4. We were able to write any stateful firewall policy in terms of guarded commands and we could come up with a state transition diagram for each of them. We decided on having an XML based policy specification language to represent the important features of the security automata such as state variables, transition conditions and actions. The next section will explain the syntax of this language.

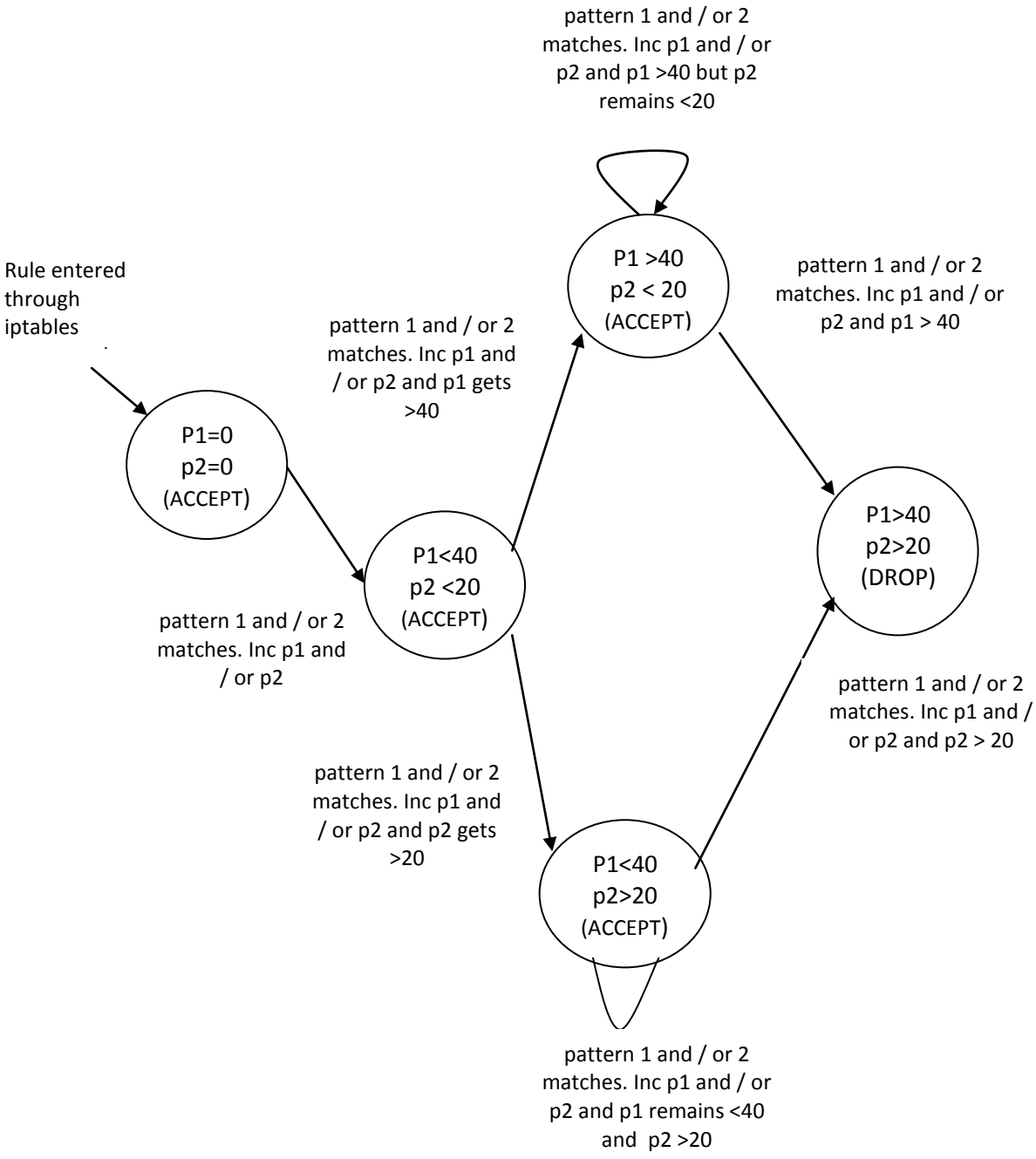

Figure 4.5 State machine for a stateful policy to drop the traffic coming from a UDP connection, if the total count of two specific patterns in the packet have crossed some maximum limit

4.4 A Stateful Policy Specification Language

A generic skeleton for the policy language is given in the figure 4.6. The root element <policy> indicates the beginning of the policy. The <state-vars> element indicates the beginning of definitions of state variables for that policy. All the state variables are specified under the <variable> element. The name of the variable is specified using the <name> element. Our language syntax requires that the state variable names do not have any spaces in them. Special characters in a state variable name are allowed. The variable type is specified under the <type> element. A variable should be of type 'char' or 'int' meaning we allow a state variable to have only characters or numbers as its value at any moment respectively. The value of the state variable at any current instance is stored in the <value> element. The above explained three elements of <name>, <init> and <type> collectively represents a single state variable of the policy and it should be unique for each variable. The policy can have many state variables but it should be finite. Hence a <state-vars> element can have multiple <variable> elements.

The <transition> element indicates the beginning of transition conditions of the policy which should be met by the current state of the policy and the corresponding actions that needs to be taken on the state variables if these conditions are met. An action can also tell the policy to drop the packet if a transition condition is met. The transition conditions can also be pre defined keywords such as PATTERN_MATCH or BYTE_COUNT which will look into the packet for a specific pattern to be present or count the number of bytes in the packet. We have defined a set of keywords for checking certain conditions against the contents of the received IP packet's data and header fields which will be described under the <condition> element. We have grouped the transition conditions and their corresponding actions into rules so that the policy maker can specify all the state changes that can take place in a policy separately. Hence, a single <rule> element will capture a single state change that can occur in a policy. A <transition> element can have many but finite number of <rule> elements.

```
<! – skeleton of the XML stateful policy specification language -- >
<policy>  <!-- root node>

   <state-vars>

         <variable>
                  <name> </name>
                   <init> </init>
                   <type> </type>
         </variable>
        .
        .
        . <!—A state-vars element can have many number of variable elements -->
    </state-vars>

   <transition>
   <!-- start of transition section. Each rule within the policy must be specified within
   <rule> tag -->
      <rule>
        <!-- specify conditions, state transitions and packet actions for each rule -->
        <condition> </condition>
         .
         .
         . <!-- A  rule element can have many number of conditions -->
         <action> </action>
         .
         .
         . <! -- A rule element can have many numbers of actions -->
      </rule>
      .
      .
      .<! – A transition element can have many numbers of rules -- >
   </transition>
</policy>
```

Figure 4.6: skeleton of policy specification language

A single state change is captured in terms of individual transition conditions that should be met and the actions that need to be taken if these conditions are met. The conditions are specified under <condition> element and the actions are specified under <action> element. A rule can have many number of <condition> or <action> elements. The values which a <condition> element can take are logically separated into two types. The first type is checking a state variable's value to see if it is within a range or if it is equal to some fixed value and the second type is checking the received packet's header and data fields to see if it matches the condition specified through a keyword. The first type of condition has a syntax which is generically captured below.

<condition> (variable_name) (keyword_var_cond) (value) </condition>

The (variable_name) should be one of the state variable's name and it must match the name given to a state variable under <name> element of the <variable> parent element. If the names do not match, the rule will not be processed further. The (keyword_var_cond) is a collection of keywords for specifying operators such as LT (less than), LTE (less than or equals), GT (greater than), GTE (greater than or equals), EQ (equals). (value) specifies the value against which the (keyword_pkt_cond) keyword must be applied on the variable values. All the three fields are mandatory. Note that the keywords must be from one of the five keywords mentioned above. If the variable is of type char, then only = (equals) keyword can be used and the [value] field must also be of type char. The second type of condition also has syntax and it is shown below.

<condition> (keyword_pkt_cond) [optional parameter] </condition>

This type of condition is specified when we want the packet received by our module to be checked for certain characteristics, either in the header fields or data field or both. The keyword is specified from a set of predefined keywords and only these actions can be taken on a packet. Some of these actions are to check if the SYN flag is set in the TCP header, check if the packet is of type ICMP response etc. The complete list of keywords and its

corresponding actions on the IP packet is shown in table 4.1 below. Some of these keywords need an additional parameter to work with. For example, the keyword PATTERN_MATCH needs to be complemented with the pattern that needs to be checked in an IP packet.

Table 4.1: List of keywords used for specifying transition conditions in our policy language

Keyword_pkt_cond	What it does?	Optional Parameters?
PATTERN_MATCH	Checks for a specified pattern in the data section of an IP packet	Pattern to be matched against
BYTE_COUNT	Count the number of bytes of data excluding the header fields in the IP packet	The state variable which needs to be incremented with the number of bytes counted
PING_REQ	Check if the IP packet has a ICMP ping request message	The variable whose value should be incremented if the packet is a ping request packet.
PING_RESP	Check if the IP packet has a ICMP ping response message	The variable whose value should be incremented if the packet is a ping response packet.
PING_DEST_UN	Check if the IP packet has a ICMP destination unreachable message	The variable whose value should be incremented if the packet is a ping destination unreachable packet.
PING_TIME_EXCEEDED	Check if the IP packet has a ICMP time exceeded message	The variable whose value should be incremented if the packet is a ping time exceeded packet.
PING_TIMESTAMP_REQ	Check if the IP packet has a ICMP time stamp request	The variable whose value should be incremented if

	message	the packet is a ping timestamp request packet.
PING_TIMESTAMP_RESP	Check if the IP packet has a ICMP time stamp response message	The variable whose value should be incremented if the packet is a ping timestamp response packet.
PING_INFO_REQ	Check if the IP packet has a ICMP information request message	The variable whose value should be incremented if the packet is a ping information request packet.
PING_INFO_RESP	Check if the IP packet has a ICMP information response message	The variable whose value should be incremented if the packet is a ping information response packet.
PING_ADDR_REQ	Check if the IP packet has a ICMP address request message	The variable whose value should be incremented if the packet is a ping address request packet.
PING_ADDR_RESP	Check if the IP packet has a ICMP address response message	The variable whose value should be incremented if the packet is a ping address response packet.
SYN_SET	Check if the TCP header of the IP packet has just the SYN flag set	None
NO_SYN_SET	Check if the TCP header of the IP packet does not have the SYN flag set	None
SYNACK_SET	Check if the TCP header of the IP packet has just the SYN flag and the ACK flag	None

	set	
NO_SYNACK_SET	Check if the TCP header of the IP packet does not have both the SYN and ACK flags set	None
ACK_SET	Check if the TCP header of the IP packet has just the ACK flag set	None
NO_ACK_SET	Check if the TCP header of the IP packet does not not have the ACK flag set	None

If the conditions for a rule are met, the actions to be taken or the transitions for that rule are then applied. The actions to be taken in case the transition conditions are met are specified in the <action> element. There are two types of actions or transitions possible. One is to change the values of the state variables and the other is two accept or drop the packet. Both these types of actions can be specified for a single rule. The syntax for the first type of transition is

<action> (variable_name) (keyword_var_act) (value) </action>

This is similar to the syntax for type 1 of <condition> element. The difference being that the keywords that can be applied here are different. The keywords present in (keyword_var_act) are given in table 4.2 below.

Table 4.2: List of keywords used for modifying state variable values

Keyword_var_act	What it does?
INC	increment the variable value by 1
DEC	decrement the variable value by 1
ADD	add to the variable value the value specified in (value)

SUB	Subtract from the variable value the value specified in (value).
ASSIGN	Assign the value in (value) to the variable; if the variable is of type char, this is the only keyword that can be used.

The <action> element is also used to another type of transition, which is to accept or drop the packet if the conditions of a rule or met. The keyword ACCEPT is used to indicate that the packet can be accepted, that is, allowed in its normal flow. The keyword DROP is used to indicate that the packet be dropped from the network subsystem.

A policy for allowing only established incoming TCP packets and connection establishing incoming TCP packets is given in Figure 4.7 below.

It is important that a policy language expert or a system administrator is familiar with the concepts of security policy, security automata with guarded commands in order to be able to understand our XML based policy language. It is equally important to use the keywords explained in this section in the policy language file. Also care should be taken to logically order the rules within the <transition> element. Also, it is important to consider the syntaxes for <condition>, <action> and <variable> elements so that our tool can understand what the user wants. If any logical errors are found in the policy language file, our tool will not be able to understand the error and will simply skip over that rule if the combinations of keywords and values do not make sense.

Once the policy language is written adhering to the syntax mentioned in this section, the user now has to pass this policy file to our generic module along with the end points of the connection upon whose packets the policy given in the file will be applied. The user then has to enter the iptables command through the command line interface. This iptables rule is then processed by our generic module. The next chapter explains in detail, how we have implemented our generic module in the Linux kernel.

Figure 4.7 A TCP policy written in our XML-based policy language

```
<policy>
        <state-vars>
            <variable>
                        <name> current_TCP_state </name>
                        <init> UNDEF </init>
                        <type> char </type>
            </variable>
            <variable>
                        <name> direction </name>
                        <init> 0 </init>
                        <type> int </type>
            </variable>
        </state-vars>
        <transition>
            <rule>
                    <condition> direction EQ 0 </condition>
                    <condition>current_TCP_state  EQ UNDEF </condition>
                    <condition>NO_SYN_SET </condition>
                    <action>DROP</action>
            </rule>
            <rule>
                    <condition> direction EQ 0 </condition>
                    <condition> current_TCP_state EQ UNDEF </condition>
                    <condition> SYN_SET </condition>
                    <action> current_TCP_state EQ  INIT </action>
                    <action> direction ASSIGN 1 </action>
                    <action> ACCEPT</action>
            </rule>
            <rule>
                    <condition> direction EQ 1 </condition>
                    <condition> current_TCP_state EQ INIT </condition>
                    <condition> NO_SYNACK_SET </condition>
                    <action> DROP</action>
           </rule>
```

```
        <rule>
                        <condition> direction EQ 1 </condition>
                        <condition> current_TCP_state EQ INIT </condition>
                        <condition> SYNACK_SET </condition>
                        <action> current_TCP_state ASSIGN INTER </action>
                        <action> direction ASSIGN 0 </action>
                        <action> ACCEPT</action>
        </rule>

       <rule>
                        <condition> direction EQ 0 </condition>
                        <condition> current_TCP_state EQ INTER </condition>
                        <condition> NO_SYNACK_SET </condition>
                        <action> DROP</action>
        </rule>
        <rule>
                        <condition> direction EQ 0 </condition>
                        <condition> current_TCP_state EQ INTER </condition>
                        <condition> ACK_SET </condition>
                        <action> current_TCP_state ASSIGN EST </action>
                        <action> ACCEPT</action>
        </rule>
        <rule>
                        <condition> current_TCP_state EQ EST </condition>
                        <action> ACCEPT</action>
         </rule>
         </transition>
</policy>
```

Chapter 5

Implementation

The tool that takes care of translating the stateful firewall policy language into a Netfilter rule has been implemented as a Netfilter match extension. The duty of a match module is to inspect each packet received and check whether it matches the conditions of the rules specified in the match module. If there is a match for the conditions with the packet received, the appropriate action is taken on that received packet as specified in the module. A match module can be as simple as matching against a source IP address or a destination IP address to as advanced as matching on concurrent number of connections. A match module does not modify the contents of the received packet's buffer and the packet buffer passed to the kernel code corresponding to the match module, namely skb buffer, is passed as a constant buffer [2]. Our match module parses the iptables command using which the user invokes our stateful match module and executes the conditions and actions specified in the policy language file.

5.1 Modules

We have implemented our tool as a new Xtables match extension called stateful. Three files are needed to register a new match module. They are the header file, the userspace module and the kernel module. The kernel module is the one with the xt_ prefix. Userspace module is prefixed with the libxt_ prefix. Most modules must adhere to this prefix requirement because both the Makefiles and the iptables codebase responsible for loading plugins have the prefixes hardcoded in them [2].

5.1.1 The header file

The binary interface between the userspace and kernel modules is specified in the header file. The header file is the first file written because this is where the parameters for the match are specified along with the information needed by the kernel module. As far as our stateful match extension is concerned, we do not know beforehand, the total number of rules within the policy and the total number of conditions and actions in each of these rules within the policy. This information is contained in the policy language file but it needs to be shared between our userspace module and our kernel module. So, we have come up with a structure that is generic enough to hold all the essential features of a policy, like the total number of rules, conditions and actions, which will be common for all stateful firewall policies. This structure, stateful_filter_struct, will hold three other structures with each of them holding the information corresponding to the policy endpoint parameters, global variables present in the policy and the rules present in the policy. Policy end point parameters correspond to the source and destination IP address ranges, source and destination port number ranges of the two ends of a network connection, on which the policy will be applied. These policy end points parameters are stored in a structure called st_policy_parameters. The state variables defined in the policy is stored in an array of the structure s_global_variables which is given in figure 5.3. The rules present in the policy are stored in an array of the structure st_rules which can be seen in figure 5.4. A counter is also present to track the total number of rules defined and the total number of global variables present. The structure stateful_filter_struct is given in figure 5.1 below.

The structure st_policy_parameters is used to hold the matching conditions that an IP packet must satisfy in order for the policy to be applied to that packet. The conditions are end point1 and end point2 IP addresses ranges, end point1 and end point 2 port numbers ranges and IP packet type (TCP or UDP or ICMP). A flag is also defined which will keep track of which parameters have been specified through the iptables command while entering the new iptables rule. This structure is given in figure 5.2 below. The TCP header information such as

which flag is set along with the sequence and acknowledgement numbers are stored in a separate structure called st_tcp_struct within st_policy_parameters. This structure is filled only when the policy has to be applied for TCP packets and an incoming packet received by our match module's kernel module satisfies the parameters of the policy. The ICMP packet type information is stored in another structure st_icmp_st within st_policy_parameters. This structure is filled only when the packet type is ICMP.

```
typedef struct stateful_filter_struct
{
  policy_parameters policy_params;
/*The above structure is used to hold IP addresses end points, port numbers end points
and layer 4 protocol information of the connection*/

  global_variables glob_variables[MAX_NO_OF_VARIABLES];
//The above array is used to hold the state variables of the policy

  __u8 global_variable_count;
 //The above count will hold the total number of state variables defined in the policy

  rules rule[MAX_NUMBER_OF_RULES];
//The above array is used to hold all the rules defined in the policy.

  __u8 tot_rule_count; // total no of rules defined in the policy
}stateful_filter;
```

Figure 5.1: struct xt_new_mtinfo – the binary interfacet between the userspace and kernel modules

The third important structure is the st_rules structure declared as an array within the structure st_policy_parameters. The structure is given in figure 5.4. It is used to hold the contents of every transition condition and action that needs to be taken in a rule if the condition matches. A counter is also present to count the number of conditions and actions in each rule.

```
typedef struct st_policy_parameters
{
union nf_inet_addr ep1_ul_ip; // Source (end point 1)upper limit IP address
union nf_inet_addr ep1_ll_ip; // Source (end point 1) lower limit IP address
union nf_inet_addr ep2_ul_ip; // Destination (end point 2)upper limit IP address
union nf_inet_addr ep2_ll_ip; // Destination (end point 2) lower limit IP address

__u16 ep1_ul_port; // Source (end point 1) upper limit port number
__u16 ep1_ll_port; // Source (end point 1) lower limit port number
__u16 ep2_ul_port; // Destination (end point 2) upper limit port number
__u16 ep2_ll_port; // Destination (end point 2) lower limit port number

__u8  ip_packet_type; // Can be UDP, TCP or ICMP

__u8 flags;   // This tells if the src, dest, and max count is set

tcp_structure tcp_params; // structure to hold TCP header details if packet_type=TCP

}policy_parameters;
```

Figure 5.2: struct st_policy _parameters – this structure holds the IP addresses and port numbers range to which the policy should be applied, along with the TCP header field details, if the packet is of type TCP

```
typedef struct s_global_variables
{
  char variable_name[MAX_VARIABLE_NAME_LEN]; // variable name
  __u32 variable_value; // used to hold variable value if type = int
  char variable_type[MAX_VARIABLE_NAME_LEN]; //int or char
  char variable_array[MAX_VARIABLE_NAME_LEN]; // used to hold variable value if
type = char
}global_variables;
```

Figure 5.3: struct s_global _variables – a structure for holding a policy's state

As it can be seen from figure 5.4, both the transition conditions and actions share the same structure. This structure called st_condition_action is shown in figure 5.5. It consists of five parameters. The parameter called type is used to store the different types of conditions and actions possible. We have defined this type for our own purposes of handling the code effectively in the kernel module. The other four parameters are used to store the variable name, keywords in condition or action and two fields to hold the variable's value.

It is important that the structures used in this section be understood in case any future work on improving this generic module further needs to be done.

```
  condition_action condition[MAX_NO_OF_CONDITIONS]; //array to hold condition
structure
  __u8 tot_cond_count;
  condition_action action[MAX_NO_OF_ACTIONS]; //array to hold action structure
  __u8 tot_act_count; //total no of actions defined for the rule
  __u8 is_reverse; // value set to 1 if this rule should be applied for packets in opposite
dir
}rules;
```

Figure 5.4: struct st_rules to hold the condition and action field arrays

```
typedef struct st_condition_action
{
 __u8 type; // 0,1 or 2 for condition and 0 or 1 for action
 char value[MAX_VALUE_LENGTH];
char variable_name[MAX_VARIABLE_NAME_LEN]; //used to store var name
 char keyword[MAX_KEYWORKD_NAME_LEN]; //LT,GT,LTE,GTE,EQ,INC,DEC,ADD,SUB
 __u16 int_value;  // if var type is int, used to store integer values
}condition_action;
```

Figure 5.5: struct st_condition_action to hold the five possible parameters of a rule's condition or action fields.

5.1.2 Userspace Module

The purpose of an iptables extension is basically to interact with the user. It will handle the arguments the user wants the kernel part to take into consideration. The user space module is named 'libxt_stateful.c'. We have followed the Xtables's coding conventions to register the module and initialize the module. The module is initialized as given below in the figure 5.6 [2].

```
static struct xtables_match stateful_mt_reg = {
   .version=XTABLES_VERSION,
   .name = "stateful",
   .revision = 0,
   .family = NFPROTO_IPV4,
   .size = XT_ALIGN(sizeof(stateful_filter)),
   .userspacesize = XT_ALIGN(sizeof(stateful_filter)),
   .save = stateful_mt_save,
   .print = stateful_mt_print,
   .parse = stateful_mt_parse,
   .help = stateful_mt_help,
   //.init = ipmatch_mt_init,
   .final_check = stateful_mt_check,
   .extra_opts = stateful_mt_opts,
};
```

Figure 5.6: structure inside userspace module to initialize the module

.save, .print, .parse, .help, .init ,.final_check are the names of functions present inside the userspace module. help is called whenever a user enters 'iptables -m stateful -h' in the command line .parse is called when a user enters a new rule through iptables for this match; its duty is to validate the arguments. print is invoked by 'iptables -L' to show previously inserted rules. It is possible to omit init, final_check, save functions. help and parse functions must be defined [2]. name refers to the name of the module. family denotes what family this match operates on, in this case IPv4 (NFPROTO_IPV4). the <*name*, *revision*, *address family*> tuple is used to uniquely lookup the corresponding kernel module [2]. size specifies the size of our private structure in total. userspacesize specifies the part of the structure that is

relevant to rule matching when replacing or deleting rules. The function to register the module is given in the figure 5.7. The basic structures that can be seen in each of the function definitions are Xtables structures and its definition can be found in [2].

```
static void_init (void){
xtables_register_match(&stateful_mt_reg);
}
```

Figure 5.7: function to register the userspace module

parse function

This function is called every time a user enters an iptables command, for example,

iptables –A INPUT –m stateful -- policy TCP.xml -- srcip 192.168.3.4 -- dstip 192.168.3.67 -- sport 34000 – dport 34566

through the command line to insert a rule using the stateful match extension. Its function is to parse the XML file and get the data that is required to fill the data structure present in the binary interface so that the corresponding kernel module knows what to match a packet against and if there is a match what to do with the packet. Along with parsing the XML, the source and destination IP address ranges, the source and destination port number ranges are also parsed and filled in the structure that will be shared by both the kernel and userspace module. This structure is stateful_filter_struct which we saw in the previous section. We have written our own XML parse to parse the XML file. Note that the data for structures such as st_tcp_st which is used for storing TCP parameters information and st_icmp_st which is used for storing ICMP packet type information and the ip_packet_type field are not filled in the user space module because these information are not visible to the user space and are filled only in kernel module after inspecting the packet received. If there is a problem with parsing, such as incorrect iptables syntax or incorrect IP address format or missing XML files, the parser will exit after showing an error on the command line through which the user entered the iptables command and the policy is not inserted into Netfilter. If the parsing is

successful a message is displayed to the user in the command line that the parsing was successful.

5.1.3 Kernel Module

The purpose of the kernel module is to analyze each received packet and check whether it is a part of a connection upon which the new stateful policy, entered through iptables, should be applied. If there is a match, the transition conditions specified in the policy language file are applied to the packet. As a transition in the policy language consists of many numbers of rules, each rule is applied to the packet to check if the conditions of a rule are satisfied. If it does satisfy the transition conditions, the action given in that rule is taken. The action can be either to update the state variables defining the current state of a connection or to accept/drop the packet itself. If the action to be taken is to accept the packet, other rules are applied on that packet in the order in which they are specified. If the action to be taken is to drop the packet, the module does not process any further rules and immediately drops the packet. Note that apart from accepting or dropping the packet, Netfilter allows the user to log the packet or save it for future processing. In this thesis, we have included only two packet decisions which are either to accept or drop the packet.

The kernel module is initialized using the structure xt_match. The structure is given in the figure 5.8.

```
static struct xt_match stateful_mt_reg __read_mostly = {
        .name = "stateful",
        .revision = 0,
        .family = NFPROTO_IPV4,
        .match = stateful_mt,
        .checkentry = stateful_mt_check,
        .destroy = stateful_mt_destroy,
        .matchsize = XT_ALIGN(sizeof(stateful_filter)),
        .me = THIS_MODULE,
};
```

Figure 5.8: structure that is passed to the function xt_register_match which will register the kernel module

Both userspace and kernelspace must agree on the same <*name*, *revision*, *address family*, *size*> 4-tuple for an xt_match to be successfully used [2]. The fields table and hooks can limit where the match may be used in an iptables table and netfilter hooks respectively. If no value is provided for the two fields, as in our code, those restrictions will not be applied. There are no Xtables matches we know of that are limited to a specific table, but the field is there for completeness. The fields match, checkentry and destroy are callbacks that the framework will use. The field match callback is called when a packet is passed to our module, checkentry and destroy are called on rule insertion and removal, respectively. The matchsize field specifies the size of the private structure used inside our kernel module [2].

The checkentry callback function is called whenever a new rule is inserted through iptables. Apart from the userspace's parse function, this is the last callback which is invoked to make sure the parameters passed through the iptables command matches the requirements and are error free [2]. Once this function clears the iptables rule entered, the new rule is actually inserted into Netfilter's hooks.

The init function in our kernel module is called when our module is first loaded in the kernel. The code for init function is shown in figure 5.9. In function, we set the initial values for static variables and also allocate memory for internal tables and structures.

```
static int __init stateful_mt_init(void)
{
 memset(&policy_table, 0,(sizeof(local_policy_table)*MAX_NUMBER_OF_RULES));
 current_number_of_rules_added=0;
 memset(&state_table, 0,(sizeof(local_state_table)*MAX_NUMBER_OF_CONN));
 current_number_of_connections_added=0;
 return xt_register_match(&stateful_mt_reg);
}
```

Figure 5.9: .init function which will is called as soon as our module is loaded in to the kernel. Local structures are allocated memory and static variables are initialized

Structures used within the kernel

Our kernel module has two important local tables. One is an array of the structure local_state_table_st which acts as the local state table which is used for storing connections which are currently being monitored by our policy. The other is an array of the structure local_policy_table_struct which acts as a storage table for policies entered by the user through the command line. This structure was created for future purposes where in the user can specify two or more iptables rules using our module and the kernel can then store both these policies in its local structure and apply both of them on all matching incoming packets. In our implementation, even though we have given a provision for storing more than one policy, at a particular instance of the Linux firewall, only one policy can be run using our tool. Attempting to use our module for more than one stateful policy will result in unexpected behaviour as of now. This structure is shown in figure 5.10.

```
typedef struct local_policy_table_struct{
  __u8 curr_count;
  __u8 used;
  stateful_filter local_stateful_filter;
}local_policy_table;
```

Figure 5.10: structure local_policy_table_struct which is used for storing policies mentioned in the XML file

The maximum no of policies that can be presently stored is specified using a constant MAX_NUMBER_OF_RULES in the kernel.

The structure local_state_table_st is used for storing a connection which is currently being monitored by the tool. A new entry to this table is made when a packet is found to be matching all the end point parameters of the policy and matching the transition conditions of at least one rule in the policy, if an entry for that connection is not already present. If an entry

is already present for that connection, and if the packet matches at least one of the rules' transition conditions, the action is taken on that packet and the state table is updated for that connection if the state variable of the connection, to which the packet belongs, has changed due to the action taken. This structure is shown in figure 5.11.

```
typedef struct local_state_table_st{
  __u8 used;
  union nf_inet_addr ep1_ip; // Source (end point 1)uppler limit IP address
  union nf_inet_addr ep2_ip; // Destination (end point 2)uppler limit IP address
  __u16 ep1_port;
  __u16 ep2_port;
  __u8  ip_packet_type; // Can be UDP, TCP or ICMP
  global_variables curr_state[MAX_NO_OF_VARIABLES];
  __u8 check_reverse;
  __u8 policy_index_in_storage;
}local_state_table;
```

Figure 5.11: Structure local_state_table_st which acts as the local state table for connections that are currently being monitored by the tool

stateful_mt match function

This is the callback function which is registered in the Netfilter hook and this function is called whenever an IP packet, traversing the Linux firewall's Netfilter hooks, reaches our registered module. This function receives as parameters, the private structure shared between the userspace and the kernel space along with the IP packet's contents in a buffer which cannot be modified by our module. The private structure contains the information of the policy that needs to be checked against the packet received in our module.

Figure 5.12: A generic algorithm of our kernel module's operation for all IP packets

- Check if policy's end point parameters match the end point parameters of the packet received.
- If there is no match, return false, meaning the packet does not satisfy the conditions of our match and hence allow the packet to pass through and exit the function.
- If there is a match, get the index of the policy from the kernel's local structure. If the policy is not stored in the local table, insert the policy if there is an available spot in the policy storage array.
- Analyze the packet's IP header and retrieve the layer 4 protocol information.
- The received packet's source IP, destination IP, source port, destination port and layer 4 protocol is matched against the connections' end point parameters present in the connection table. Get the index of the connection from the connection table, to which the packet belongs.
- If no such connection is found, mark the packet as new connection and continue. If there is a match return the index of the connection in the connection table.
- For all the rules present in the policy, apply each rule in turn.
- For each rule, apply all the transition conditions on the packet and on the state variable of the connection to which the packet belongs.
- If all the conditions within a rule match, if the packet is marked as a new connection, insert the packet's details in the connection table as a new connection and return the index.
- If even one among all the conditions within a rule do not match, the action for that rule is skipped and the next rule is processed.

- If all the condition matches, the action for that rule is applied on the state variables of the connection in the state table and if the action is to drop the packet, the function returns true and skips all further rules in the policy. By returning true, the module is indicating to Netfilter that the packet has violated some rule specified in our module and Netfilter will just drop the packet.
- After all the rules have been applied and if there is no action indicating to drop the packet is taken, the function simply returns false.

Figure 5.12 explains the working algorithm for our kernel module.

5.2 Compiling and registering the files as Xtables modules

Both the userspace and kernel space programs can be compiled as standalone packages or as in-tree modifications to the kernel or using Xtables-addons package. Xtables-addons is an open source package and can be downloaded from the internet [3]. The steps to use the Xtables-addons for creating the kernel module are given in [2]. Xtables-addons package was developed so that new extensions can be easily added without having to bother about the build infrastructure complexities. We have used the Xtables-addons package version 1.22 to create, build and deploy our kernel module. All the three files should be placed under /xtables-addons-1.22/extensions/ folder. Note that the depending on the version of xtables-addons, the folder names will vary. Since we started working on our tool, new versions of xtables-addons have been released.

After placing all the three files inside the correct folder and after adding one line of command in the build files, Xtables-addons then simply needs to be compiled using ‘Make’ command. The complete instructions on compiling the source files are available in [2]. The addons package will take care of creating the kernel module xt_stateful.ko. Once we have the

kernel module created for us, it needs to be inserted into to kernel using the command ‘insmod xt_stateful.ko’. Now we are ready to use our module. An example iptables command that uses our module in the xtables-addons environment is given in figure 5.13.

```
XTABLES_LIBDIR=$PWD:/usr/libexec/xtables:/lib/xtables iptables -A INPUT
 –m stateful --xml pattern_count2.xml --srcip 192.168.1.4 --dstip 192.168.1.60
--sport 40000 --dport 50000
```

Figure 5.13: An example iptables command that uses our module in the xtables-addons environment

If only a single IP address or port has to be matched, it is enough to enter only that IP address or port number instead of a range. Also, the XML file should be present in the same folder as that of the three source files. A generic syntax for using our tool in the xtables-addons environment is given in the figure 5.14.

```
XTABLES_LIBDIR=$PWD:/usr/libexec/xtables:/lib/xtables iptables -A INPUT
  -m stateful   --xml [file_name.xml]    --srcip [upper_range]-[lower_range]
 -- dstip [upper_range]-[lower_range]  --sport [upper_range]-[lower_range] --
dport[upper_range]-[lower_range]
```

Figure 5.14: Generic Syntax for using our tool in the xtables-addons environment

Chapter 6

Related Work, Conclusions and Future Work

6.1 Related Work

Except for conntrack, the current stateful firewall tool in Netfilter, no other attempts have been made to have a state machine based stateful firewall tool for Linux. We have already discussed the working of conntrack in chapter 3 of this thesis. Attempts have been made to create a model for a stateful firewall in general. For example, [12], claims to have come up with the first ever model of a stateful firewall. In the model presented in [12], each stateful firewall consists of a stateful section and a stateless section. Each stateful firewall has an associated variable set called the state of the firewall, which stores all the packets that the firewall had processed previously and needs to remember in the future. When a packet arrives at a stateful firewall, it is processed in two steps. In the first step, the firewall adds an additional field called the tag to the packet and uses the stateful section of the firewall to compute the value of this field, according to the current state of the firewall. In the second step, the firewall compares the packet together with its tag value against a sequence of rules in the stateless section. When a packet matches a rule in the stateless section, the action specified in that rule is applied on that packet and no further rule processing is done [12].

The ideas presented in [12] are theoretical in nature and so whether it is practically applicable in Linux firewalls is not known. For example, [12] assumes that a packet is augmented with a keyword called tag but does not elaborate on exactly when this tag is appended to a packet. Also it is not clear whether the implementation of the ideas presented in [12] can be automated or not.

Alex Liu and Mohamed Gouda argue that redundant rules in a firewall significantly degrade the efficiency and performance of a firewall and they have come up with a tool to check for redundancy in firewall rules [24]. This tool works by building a firewall decision trees from the list of all firewall rules and attempts to find upward and downward redundant rules from the firewall decision tree. Redundant rules in a stateful policy specified by a policy maker are a concern for us since our tool does not check for redundancies in a stateful policy. Hence it would be wise to use a redundancy checking tool, as presented in [24], along with our tool to make sure that the stateful policy, which our tool will process, is redundant free. Alex Liu and Mohamed Gouda have come up with another technique to check whether a firewall's rules have a consistent order, are compact, and are complete [25]. In this technique, a firewall is checked for consistency and completeness by drawing a firewall decision diagram and then applying an algorithm systematically on the decision diagram to check for inconsistencies [25]. Since this technique checks for all the three vulnerabilities that a firewall configuration might suffer from, we can use this technique in [25] on the stateful firewall ruleset given by the policy maker to our tool.

A method to find inconsistencies and irregularities in conntrack is presented in [20]. The main focus of the work in [20] is to check whether a stateful firewall ruleset is consistent, that is, to check whether a matching rule in the ruleset that will drop a packet is shadowed by rules preceding it which might accept that packet. This condition arises because some firewalls stop matching a packet against a set of rules in a ruleset as soon a matching rule is found. The work in [20] argues that checking for such inconsistencies in a stateful firewall is not difficult and it is enough to check for inconsistency in a ruleset for just one state of the stateful policy and that the result can be applied across all the states that a firewall might take. Since our tool is used to process stateful policies, we would like to use the tool in [20], along with our tool, to check if the policy our tool is about to execute contains inconsistent rules or not.

Similar to the tool mentioned in [20], FIREMAN is a tool developed for checking inconsistencies in firewall configurations [21]. It applies static analysis methods for detecting policy violations, inconsistencies, and inefficiencies of a firewall [21]. Since our tool does not check for consistency of a stateful firewall policy, it would be interesting to use tools such as FIREMAN to see whether the rules in a policy are consistent or not.

In our work, we do not formally check whether our implementation reacts correctly to the rules in a stateful firewall policy. Assuming that the Linux firewall administrator comes up with a logically correct stateful policy, we simply execute all the rules in that policy in the order in which they are listed in that policy unless we come across a matching rule which wants the packet to be dropped. An automatic method to formally check whether a stateful firewall reacts correctly to a stateful policy written in a high level declarative language is presented in [21]. It would be interesting to use the technique described in [21] for formally verifying our implementation.

Eronen and Zitting [23] have developed a tool based on constraint logical programming (CLP) to check for inconsistencies in firewall policies. Eronen and Zitting [23] suggest that the ordering of rules in a firewall policy is very important for the firewall to work properly, which is exactly what we argue. We want the administrator to list the rules in a stateful policy in the correct order so that the firewall is not inconsistent. Hence, it would be interesting to use the tool given in [23] to test whether the policy maker using our tool, lists the rules in a stateful policy in a consistent manner or not.

openflow is a tool developed for enabling researches in campuses for testing their custom developed protocols in campus networks itself, instead of having to test it directly on the internet or developing custom hardware tools such as routers and switches for testing their new ideas [7]. It was developed to encourage researchers test their ideas on real networks amidst real network traffic. openflow works by separating real network traffic, also called as production traffic, from research traffic caused by researchers testing their

applications. It is based on an Ethernet switch with internal flow table to route the two type of traffic to their respective ports so that both the research traffic and the production traffic do not interfere with one another. The flow table is used to record traffic paths and one can also enter or delete flows or connections into the table. The tool uses <match,action> pair to route traffic based on the header information which is entered in the match clause. If the header information of a packet matches the match clause in the <match,action> pair, the corresponding action is applied on the packet [7]. Normally the action would request the packet be sent to a particular port depending on its higher layer protocol or dropped or sent to the tool's controller. Apart from routing decisions, the action clause would also instruct the packet be sent out from the router or switch at a particular bit rate or append a mask on top f the packet's header [7]. This <match, action> pair is similar to how our tool processes stateful policies. It checks a set of conditions in a rule and if the conditions in a rule match, the corresponding action is taken on that packet.

Although a number of stateful firewall products have been available and deployed on the Internet for some time, such as Cisco's Adaptive Security Appliance (ASA) [16], CheckPoint's VPN-1 [14], no model for specifying stateful firewalls exists.

6.2 Conclusions

Stateful firewall in Linux is an essential feature for tracking packets belonging to suspicious connections and to prevent the system against network attacks. conntrack, Netfilter's connection tracking system, only classifies a packet as belonging to a particular connection or not. To make that classification, conntrack depends on other kernel modules to make that decision. It is not based on a state machine and it is not automated. In this thesis, we propose a generic Linux stateful firewall tool which will accept a stateful firewall policy written in a high-level XML-based policy as the input along with the end point source IP address range, destination IP address range, source and destination port number range. The XML-based policy specification language was developed based on the idea of security

automata for enforceable security policies given in [1]. The policy specification language allows Linux network administrators to define their own state machines for a stateful policy and represent the state machine in terms of state variables. The rules present in the policy can be defined in terms of transition conditions and actions to be taken if the conditions are met. Our tool also enables the users to drop a packet if it is found violating the stateful policy.

To process the policy language XML file, we have implemented an iptables match extension module and named it as stateful. We have written three modules which will translate the stateful policy present in the XML file into an iptables rule and apply the rules present in the policy on IP packets. The userspace module will parse the iptables command entered by the user through the command line to extract the relevant policy information from the XML file and the end points of the connection that needs monitoring. The parsed information is stored in a common structure shared between the userspace module and the kernel module and this common structure is defined in a header file to which both the userspace and the kernel module will have access.

Our tool was successfully tested for implementing TCP connection tracking policy, UDP pattern matching policy, ICMP related policies and also policies restricting the number of packets and bytes coming from a connection endpoint. We found that there was no need to change our code to implement the above policies and also found that our tool was working when the state machine for a policy was changed by changing the number of state variables defining the state of a policy. If our tool is improved with the ideas presented in the future works section, it could be claimed as an alternative for conntrack.

6.3 Future Work

Currently, our tool cannot detect layer 7 protocol information to which a packet belongs. Hence, we cannot extend our stateful firewall tool support for tracking packets belonging to HTTP, FTP, SSH and other layer 7 protocols. Inclusion of regular expressions for pattern matching on an IP packet's data will enable our tool to detect a packet's layer-7

protocol. A tool called layer-7 filter classifier [15] is available which will classify packets based on their application layer protocol. This tool uses regular expressions on packets hitting the tool's kernel module to detect the layer 7 information contained in them. The layer-7 filter classifier is a protocol identification tool and it does not have the capability to accept or drop a packet based on the information gained. It works along with Netfilter to make this decision. However, according to Netfilter programming experts, regular expression are not the cheapest, both time and memory-wise. This might cause the system using such a tool to operate less efficiently and slowly because the Netfilter callbacks are called in IRQ interrupt context. We could not confirm or deny this advice because we did not try regular expressions in our tool. But the inclusion of layer-7 identification capability to our tool will make it more generic and eliminate the need to use other open source projects to provide layer-7 protocol identification.

For the purpose of demonstrating our tool's capabilities, we have introduced only a limited set of keywords for matching packets against layer-4 protocol information. It is possible to introduce keywords to check for matching conditions against all the currently available layer-4 protocols. But doing so might increase the total number of keywords and hence will make the job of Linux administrator harder because that person has to remember those keywords and fill in the policy file. Hence, it would make sense to classify the keywords based on protocol information and have a consistent copy of this classification in a repository from where the administrator can search for a relevant keyword. Our tool can also be improved by including a timer for each policy that has to be implemented.

conntrack is essential for Linux firewalls not only for helping in connection tracking but also for improving the efficiency and speed of NAT. The NAT module of a Linux firewall depends on conntrack to classify packets tracked by conntrack as belonging to a particular connection. The NAT logic is then applied on the connection as a whole rather than on individual packets and thus improving its efficiency. For our tool to be considered as an alternative to conntrack, it is essential that our tool is made to work in the pre-routing

chain on Netfilter, without using NAT. Currently our tool depends on NAT to provide destination address translation.

To ease the job of a Linux administrator further, we can develop a GUI application where in the administrator fills in a form specifying the state variables and all the required rules instead of having to fill in an XML file. We have not tested our tool by inserting the modules present in our tool directly in the kernel tree structure. It would be interesting to see the performance of our module once it is placed directly in the kernel tree.

References

[1] Fred B. Schneider. Enforceable security policies.
ACM Transactions on Information and System Security, Vol. 3, No. 1,
Pages 30 –50, February 2000.

[2] Jan Engelhardt and Nicolas Bouliane. Writing Netfilter modules.
Revised, February 07, 2011.

[3] Xtables-addons – Netfilter/Xtables (iptables) extensions.
http://xtables-addons.sourceforge.net/. Date of access: July 27, 2011.

[4] Netfilter/iptables project homepage – The netfilter.org project.
http://www.netfilter.org/. Date of access: July 27, 2011.

[5] IPFWADM basics.
http://www.fwtk.org/ipfwadm/faq/ipfwadm-faq-2.html. Date of access: July 27, 2011.

[6] ipchains – Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Ipchains. Date of access: July 27, 2011.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford,
S.Shenker, and . Turner. OpenFlow: Enabling innovation in campus networks.
ACM SIGCOMM Computer Communication Review 38, 2 (April 2008).

[8] iptables – Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Iptables. Date of access: July 27, 2011.

[9] Steve Suehring and Robert L. Ziegler. Linux Firewalls. Third Edition.
Pages 68-98.

[10] Linux netfilter Hacking HOWTO: Netfilter Architecture.
http://netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html.
Date of access: July 27, 2011.

[11] The state machine
http://security.maruhn.com/iptables-tutorial/c4219.html. Date of access: July 27, 2011.

[12] Mohamed G. Gouda and Alex X. Liu. A Model of Stateful Firewalls and its Properties. *In proc of the 2005 International Conference on Dependable Systems and Networks,* Page 1.

[13] Netfilter's connection tracking system.Pablo Neiro Ayuso.
http://people.netfilter.org/pablo/docs/login.pdf. Date of access: July 27, 2011.

[14] Check Point Software: VPN – 1 Power
http://www.checkpoint.com/products/vpn-1_power/. Date of access: July 27, 2011.

[15] L7- filter Pattern writing
http://l7-filter.sourceforge.net/Pattern-HOWTO. Date of access: July 27, 2011.

[16] Cisco ASA 5500 Series Adaptive Security Applicances – Products & Services
http://www.cisco.com/en/US/products/ps6120/index.html.
Date of access: July 27, 2011.

[17] The conntrack entries
http://security.maruhn.com/iptables-tutorial/x4273.html. Date of access: July 27, 2011.

[18] Firewall (computing) – Wikipedia, a free encyclopedia
http://en.wikipedia.org/wiki/Firewall_%28computing%29. Date of access: July 27, 2011.

[19] Alpern and Schneider. Defining Liveness. February, 1985.

[20] Levente Buttyan, Gabor Pek, Ta Vinh Thong. Consistency verification of Stateful firewall is not harder than the stateless case.
http://www.hit.bme.hu/~buttyan/publications/ButtyanPT09ht-en.pdf. Date of access: July 28, 2011.

[21] Nihel Ben Youssed and Adel Bouhoula. Dealing with stateful firewall checking. Digital Information and Communication Technology and Its Applications. Year 2011. Pages 493-507.

[22] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In IEEE Symposium on Security and Privacy, pages 199-213, 2006.

[23] P. Eronen and J. Zitting. An expert system for analyzing firewall rules. *In Proc. 6th Nordic Worksh. Secure IT Systems, 2001.*

[24] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. *In Proc of 19th Annual IFIP Conference on Data and Applications Security, 2005.*

[25] M. G. Gouda and X.-Y. A. Liu. Firewall design: consistency,completeness and compactness. *In Proc. ICDCS 24, Mar 2004.*