CONTENT BASED PACKET FILTERING IN LINUX KERNEL USING DETERMINISTIC FINITE AUTOMATA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TAHİR BİLAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2011

Approval of the thesis:

CONTENT BASED PACKET FILTERING IN LINUX KERNEL USING DETERMINISTIC FINITE AUTOMATA

submitted by **TAHİR BİLAL** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Dr. Onur Tolga Şehitoğlu
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Dr. Atilla Özgit
Computer Engineering Dept., METU _____

Dr. Onur Tolga Şehitoğlu
Computer Engineering Dept., METU _____

Asst. Prof. Dr. Sinan Kalkan
Computer Engineering Dept., METU _____

Asst. Prof. Dr. Murat Manguoğlu
Computer Engineering Dept., METU _____

Dr. Aykut Erdem
Geodesy and Photogrammetry Dept., Hacettepe University _____

Date: **12.09.2011**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: TAHİR BİLAL

Signature :

ABSTRACT

CONTENT BASED PACKET FILTERING IN LINUX KERNEL USING DETERMINISTIC FINITE AUTOMATA

Bilal, Tahir

M.Sc., Department of Computer Engineering

Supervisor : Dr. Onur Tolga Şehitoğlu

September 2011, 68 pages

In this thesis, we present a content based packet filtering architecture in Linux using Deterministic Finite Automata and iptables framework. New generation firewalls and intrusion detection systems not only filter or inspect network packets according to their header fields but also take into account the content of payload. These systems use a set of signatures in the form of regular expressions or plain strings to scan network packets. This scanning phase is a CPU intensive task which may degrade network performance. Currently, the Linux kernel firewall scans network packets separately for each signature in the signature set provided by the user. This approach constitutes a considerable bottleneck to network performance. We implement a content based packet filtering architecture and a multiple string matching extension for the Linux kernel firewall that matches all signatures at once, and show that we are able to filter network traffic by consuming constant bandwidth regardless of the number of signatures. Furthermore, we show that we can do packet filtering in multi-gigabit rates.

Keywords: Firewall, Linux, Packet Filtering, Iptables, Multiple String Matching

ÖZ

SONLU DURUM MAKİNALARI KULLANARAK LİNUX ÇEKİRDEĞİNDE İÇERİK TABANLI PAKET FİLTRELEME

Bilal, Tahir
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Onur Tolga Şehitoğlu

Eylül 2011, 68 sayfa

Bu tezde, Linux için sonlu durum makinaları ve iptables kullanarak içerik tabanlı bir paket filtreleme mimarisi sunuyoruz. Yeni nesil ateş duvarları ve saldırı tespit sistemleri ağ paketlerinin sadece başlıklarını değil içeriklerini de tarıyorlar. Bu sistemler ağ paketlerini taramak için düzgün deyim ya da karakter dizisi biçiminde yazılan bir örüntü kümesi kullanıyorlar. Bu tarama safhası yoğun işlemci kullanan bir süreç ve ağ performansının gerilememesi için olabildiğince hızlı yürütülmesi gerekiyor. Linux çekirdeğinde varolan ateş duvarı, her bir ağ paketini kullanıcı tarafından belirtilen bir örüntü kümesindeki her örüntü için ayrı ayrı tarıyor. Bu yaklaşım ağ performansı için önemli bir darboğaz oluşturuyor. Bu çalışmada bir ağ paketini tüm örüntüler için tek bir seferde tarayan bir paket filtreleme mimarisi geliştirip, örüntü sayısından bağımsız olarak sabit miktarda bant aralığı tüketerek paket filtreleme yapabildiğimizi gösterdik. Ayrıca gigabit katı hızlarda paket filtreleme yapabildiğimizi gösterdik.

Anahtar Kelimeler: Ateş Duvarı, Linux, Paket Filtreleme, Iptables, Çoklu Dizgi Eşleme

to *Siyah Çoraplılar*

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Dr. Onur Tolga Şehitoğlu for his help and guidance during the research and the implementation phases of my thesis. I would also like to express my thankfulness to Dr. Atilla Özgit for his insightful comments during the process.

Finally, I would like to thank to the Scientific and Technical Research Council of Turkey (TÜBİTAK) for the financial support in accordance with the 2228-scholarship program.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

A firewall is an application or a system which inspects all the packets in incoming or outgoing traffic and decides which packets to allow and which packets to disallow based on some rules. These rules are also called *signatures*. The signatures could be written in terms of the header field values of the packet (such as the protocol type of the packet, source address, destination address, tcp/ip source port). Moreover, some modern firewalls and intrusion detection systems also make use of signatures written in the form of regular expressions or plain strings to scan the payload portion of network packets. If the firewall permits the signatures to be written in form of regular expressions, then it has to scan the payload of network packets and decide if it matches any of the regular expression signatures in the signature set. If the signatures are just arbitrary string patterns, then the firewall has to scan the payload of the packets for arbitrary strings that can appear in any location in the packet payload. Scanning the payloads of network packets instead of just examining their header fields is a much more intricate and resource consuming task both in terms of CPU time and bandwidth. The reason is that the header fields make up a small percent of a network packet in comparison with the packet payloads, and the packet headers take place in strict offsets of the network packets.

The Linux kernel firewall, iptables [4], has the ability to scan and filter network packet payloads by matching them against a set of signatures written as plain strings using the *string* extension. The main drawback of iptables is the fact that that it scans a network packet exhaustively for each of the signature strings in the signature set. Scanning network packets using a large signature set costs too much CPU time, reducing network bandwidth. This approach does not utilize the resources well, and is not a scalable way of filtering network traffic content. Intrusion detection and prevention systems like Snort [45] and Bro [43] are able to examine network packet payloads by matching them against regular expressions or plain

strings. Upon detecting an intrusion or a malicious activity, IDS applications produce alerts and reports. These applications can also have defensive capabilities similar to firewalls owing to some extensions, like blocking the address of a detected attacker; however they are not complete replacements for firewalls and they must run in userspace.

There are hardware based solutions using ASIC and FGPA technologies [42][32][49][19] and commercial proxy based systems [1]; however, to the best of our knowledge, there is no scalable firewall implementation targeted for Linux operating system that is able to filter network packets by scanning their payloads efficiently in kernel space.

In order to overcome the aforementioned problem of iptables' inefficient payload scanning, in this thesis we design and implement a packet filtering architecture on top of iptables framework which is able to scan a network packet against a set of string signatures at once. This way, iptables becomes capable of filtering network traffic using constant bandwidth regardless of the size of the signature set.

We make use of Aho-Corasick algorithm, introduced in [11] to convert signature set into a DFA and finite state automata minimization algorithms to implement our packet filtering architecture. There are several multiple string matching algorithms. The reason that we chose to implement Aho-Corasick algorithm is that it examines each character in a network packet payload at most once. This attribute makes it achieve linear worst-case performance of *O(n)* where *n* is the size of a network packet. This property is of crucial importance for any network application, like intrusion detection systems or firewalls. The applications whose worst-case performances are not linear expose vulnerabilities to algorithmic attacks [22]. An attacker might send network packets that will cause the network application to perform in its worst case profile. This may cause the system to hang up and be unresponsive. Some intrusion detection systems choose to ignore the incoming packets in case of such an attack, which renders the system to be open to intruders.

The packet filtering architecture that we develop in this thesis endows iptables with the following capabilities:

- Filtering of network traffic based on network traffic content including request/reply bodies, i.e Universal Resource Locator (URL) based filtering.
- Defending against possible application level attacks to the systems (cross-side scripting,

script injection, etc),

- Defending against harmful software residing on the network by controlling outgoing and ingoing traffic.
- Providing a more favorable web surfing environment for children by blocking requests to spam web sites and responses with inappropriate keywords.

1.1 Outline of the Thesis

The rest of the thesis is organized as follows. We first present some background information and related work in Chapter 2. The design and implementation details of our content based packet filtering architecture is explained in detail in Chapter 3. Next, experiments to assess the efficiency of our approach and the results are presented in Chapter 4. We state the conclusion of the thesis and present some new ideas for future work in Chapter 5.

CHAPTER 2

BACKGROUND INFORMATION AND RELATED WORK

This chapter gives some background information about the topics that are closely connected with this thesis, and presents the related work in the literature about them.

2.1 Firewall

A **firewall** is a device or an application which is responsible to protect a system by monitoring incoming and outgoing traffic and take necessary actions upon detection of a malicious activity. Firewalls are expected to prevent attacks to the system while permitting reliable network traffic [25]. In the following subsections we will present more information about the types of firewalls and their characteristics.

2.1.1 Packet Filtering

A packet filtering firewall examines the incoming or outgoing network packets according to a set of rules and decides which ones to block and which ones to pass [25]. What distinguishes packet filtering from higher level firewalling approaches is that these rules are written only in terms of the header fields of the packet. Packet filtering firewalls mainly work in the network and transport layers of The Open Systems Interconnection model (OSI model).

Packet filtering firewalls most commonly inspect the following fields in a network packet.

- Source IP address
- Destination IP address

- Source port number
- Destination port number
- Protocol type

Ability to inspect ports allows packet filters to block certain types of applications, since they use well defined ports for communication. For examples, File Transfer Protocol (FTP) communications user port 20, Telnet uses port 23, Simple Mail Transfer Protocol (SMTP) uses 25. However, port based filtering cannot prevent applications using protocols on non-standard port numbers.

The main advantage of packet filtering firewalls is their speed. They analyze very little amount of data, compared to next generation firewalls, hence create less latency and consume less CPU time. Another advantage of packet filtering firewalls is the fact that they are simple enough to be implemented in kernel level. Kernel networking layers parse header of network packets already for processing. Packet filters simply use these fields for watching. The user level packet filtering firewalls run as a separate process so they have to copy the network packets from kernel level to user level in order to process them. This ability, running in kernel level, gets rid of the overhead of context switching between kernel level and user level applications [34] and improve filtering performance.

2.1.2 Stateful Firewalls

The important thing to note in network traffic filtering is that packet filters inspect each packet individually without paying attention to the place of the packets in the whole network stream. Stateful firewalls (also called dynamic packet filtering firewalls) take into account the sequence of the network packets and uses the sequence information in addition to the metadata content of the packet to decide which packets to allow and disallow.

This technology is generally referred to as stateful packet inspection. It maintains a state table which registers information about all connections passing through the firewall. By making use of this table, it is able to determine whether a packet is the start of a new connection, a part of an existing connection, or an invalid packet [25].

Most of the modern systems utilize dynamic packet filtering firewalls since they provide ex-

tra functionality and protocol handling. Linux kernel used to implement static only packet filtering but starting with kernel version 2.4, it is capable of packet dynamic filtering [25].

The earliest packet filtering system implementation is CSPF (CMU/Stanford packet filter) which is presented in [40]. This packet filtering system is mainly designed for a stack based machine, hence does not perform well in RISC CPUs. Another problem with CSPF is that it cannot parse variable length header fields, hence cannot find out encapsulated header fields. Moreover it uses a tree-based filtering engine, which parses all the headers of a packet and then evaluates them. This evaluation schema causes redundant computations and slows down the system.

In order to overcome all these problems, a new packet filter for Unix platform is designed in Berkeley University, which is known as BPF (BSD packet filter) [39]. This packet filter mainly targets BSD operating system but has clones in other UNIX based operating systems, like Linux Socket Filter (LSF) [24] for Linux. Quite a lot of applications use BPF, the most famous one being tcpdump. Unlike CSPF which has a stack based evaluator, BFD uses a register based evaluator, which makes it more suitable for RISC architecture CPUs. It can parse encapsulated header fields, and uses a CFG (Control Flow Graph) based filter evaluator which enables optimizations in filter expressions and increases the overall efficiency of the packet filter. The authors have found out that BPF performs up to 20 times faster than CSPF in a RISC architecture CPU [39].

Another packet filtering system is XPF [34], which tries to further enhance the performance of BSF, by trying to move much of the functionalities of BPF into kernel level from userspace, and trying the reduce the number of context switches between kernel and userspace.

FFPF [17] is another packet filtering platform, which tries to increase the performance of BPF by reducing the number of context switches like XPF does. In addition, FFPF enables different applications to share their packet buffers, which significantly reduces copying of packets for different applications. All the previous approaches before FFPF rely on copying the network packets to userspace for complex processing. FFPF may reduce the number of copies of a packet to zero by permitting processing at lower levels [17].

The Mach Packet Filter (MPF) [28] and BPF+ [16] both try to optimize BSF by recognizing the common expressions of different filters, and evaluating them only once.

Swift packet filtering system [55], handles another problem on BPF, filter update latency. When a new filter is added to existing BPF filters, BPF merges all the filters into one filter, and this recompilation procedure takes place in each update, which takes too long to complete. Likewise BPF, both xPF and FFPF suffer from this problem. Swift performs at least three orders of magnitude faster than BPF in filter update latency [55].

In a recent study [46], SPAF (Stateless FSA-based Packet Filters), the authors dwell upon optimizing filter expressions in packet filters. They make use of finite state automata algorithms. SPAF considers each packet filter as a deterministic finite automata. To illustrate, the packet filter "ip.src = 10.0.0.1 and portno = 99" makes up a DFA of 3 states, in which the first arc is the filter expression "ip.src = 10.0.0.1" and the second one is "portno = 99". SPAF converts all the packet filters to a DFA, and merges these DFAs into one single NFA by adding an empty edge from the head of the merged automata to each packet filter automata. Next step is converting this NFA to a DFA, and applying DFA minimization. Afterwards, the minimized DFA is transformed into C language code. This technique successfully gets rid of redundant filter expressions. If two packet filters have the same filter expression, let's say "ip.src = 10.0.0.1", the resulting minimized DFA only has one edge evaluating this expression. This property makes this approach to be capable of applying filtering by examining each header of a network packet only once. Thus, this technique offers the best redundant filter expression removal technique in literature [46]. On the other hand, this approach lacks support for fast dynamic filter updating. When a new filter is added to the set of filter, all the filters have to be re-compiled and minimized again.

2.2 Intrusion Detection and Prevention Systems

An IDS (intrusion detection system) is a system in the form of a device or a software application that monitors network and system applications and creates reports based on its detection of suspicious activities [33]. Intrusion detection systems primarily focus on identifying possible incidents, logging information about them, and reporting attempts. Intrusion detection systems do not try to prevent malicious activities or policy violations from taking place. Trying to protect the system by examining the alerts and logs that an intrusion detection has produced is the duty of another application.

Intrusion detection and prevention systems are extensions of intrusion detection systems. These applications have all the capabilities of an intrusion detection systems, moreover they could respond to attacks and intrusion by trying to stop them. They could block drop malicious packets, configure the system firewall to block the attacking host, or restart or close the network connection.

There are several intrusion detection systems on the market. Snort [45] and Bro [43] are extensively used since they are open source and free.

The two main detection methods that IDSs utilize to monitor networks are anomaly based detection and signature based detection.

2.2.1 Anomaly Based Detection

In this method, intrusion detection systems make use of the anomalies that they detect in the usage of some protocols. They make use of already compiled statistics about network protocols. For example, NFR [6] IDS protocol anomaly detector for SMTP tracks the entire SMTP command series from start to finish and generates alerts if the clients issues a "MSG From:" command before a "RCTP To:" command, or issued a "HELP" command [10]. Such protocol tracking works very effectively in practise because software applications send emails using definite series of commands, however human users may get the order of the commands wrong while trying to hack the system. Additionally they may intentionally use a wrong order of commands in order to trigger errors [10].

2.2.2 Signature Based Detection

In signature based detection, intrusion detection systems try to match network packets against a set of signatures, like common attack patterns and take the necessary actions that are marked in the matching signatures. Like packet filtering signatures, these signatures could include checks against packet header fields. They could match all the traffic in some specific port, or all the outgoing traffic to some particular IP address. Unlike packet filtering signatures, these signatures could also involve checks based on packet payload content.

Here is an example of a snort signature.

```
activate tcp !$HOME_NET any -> $HOME_NET 22 (content:"/bin/sh"; nocase;
msg:"Possible SSH buffer overflow"; )
```

This signature alerts the message “Possible SSH buffer overflow” for the incoming packets whose destination port is 22, protocol is tcp and that include the string “/bin/sh”. The *nocase* directive tells that the the matching of the content string is done in a case insensitive manner.

2.3 Deep Packet Inspection

Deep Packet Inspection (DPI) is technology used in firewalls which merges the packet inspection capabilities of intrusion detection systems with the prevention capabilities of a stateful firewall [23]. Actually Deep Packet Inspection is just a buzzword term which refers to a stateful firewall accompanied with a IDS like signature set.

In other words, DPI firewalls, could filter the network packets, according to their header fields like a packet filter, according to their payload content by matching against some signatures like and IDS, and also according to the position of a network packet in a connection like a stateful firewall.

Packet filtering systems could tell the difference between different a web request (TCP port 80) and a Telnet request (TCP port 23). However, they cannot differentiate between two web requests. Deep packet inspection systems are also capable of this. In other words, signatures used in deep packet inspection systems could also involve that data in packet payloads.

Deep packet inspection techniques could also be used for other purposes such as packet classification [15]. Packet classification deals with identifying and classifying network packets based on the header fields and packet content. Packet classification can be applied for different purposes, such as security purposes (all packets received from a specific IP could be dropped), output scheduling (VoIP packet are routed to high priority queues), load balancing (routing packets to different subnets) [14]. Two products that use deep packet inspection for packet classification are Pace [9] and opendpi [7]. Actually opendpi is a limited and open source version of Pace. Pace offers improved functionalities over opendpi like IPv6 support and performance optimizations [9].

Tuck et al. [53] proposes a deep packet inspection system for hardware implementations. The authors mainly try to reduce the memory consumption of Aho-Corasick algorithm, which will be explained in detail in Chapter3, so that it becomes suitable to be implemented on a single chip. They work on the string signature set used by Snort. In the process, they take inspiration from IP lookup techniques in the literature, such as applying bitmap compression to the edges of the finite state automata that is generated by Aho-Corasick algorithm. Next they apply path compression to the already bitmapped edges. By using these two optimizations, they reduce the amount of memory required by default snort string signature set, as of 2004, from 53.1 MB down to 1.09 MB, 2% of the original required memory [53]. The problem with this approach is that even though it reduces the memory consumption of Aho-Corasick algorithm, it sweeps away its main superior characteristics, like requiring single memory reference in a state transition. After these optimizations, Aho-Corasick algorithm requires at least one memory reference for any state transition. This result makes this optimized Aho-Corasick algorithm reduce bandwidth more than the original algorithm. Another problem with this approach is the implementation complexity.

Another study that tries to decrease the memory consumption of Aho-Corasick algorithm is proposed in [51]. In this study, the authors build a special purpose hardware architecture that implements Aho-Corasick algorithm for deep packet inspection. Instead of building one DFA that tries to match all the strings in a string signature set, the authors build 8 different tiny state machines each of which are responsible for only one of the eight bits of a network packet character [51]. In the original Aho-Corasick algorithm, each state can have 256 edges to another state. When the FSA is divided into 8 distinct DFAs, each state in the tiny DFAs can have at most 2 edges (0 or 1) to another state. This way, the authors show that, they are able to implement Aho-Corasick algorithm on a chip. Another issue dealt in this paper is the dynamic filter update. When a new signature is added to the signature set, the system backs up the previous composed DFA and uses that DFA to inspect the system, while the new DFA is being compiled. While promising, this implementation has to work a on general architecture computer having at least a 8-core CPU.

In [56], the authors present a deep packet inspection system using regular expressions as signatures. The problem with regular expressions is that they require much more memory than plain strings, and merging them to only one finite automata is not feasible because of excessive memory requirement. The authors put forward the following three steps in order to

provide a solution to the problem in this paper. First, they analyze the characteristics and types of regular expressions that lead to exponential growth in DFAs. Based on this analysis, they put forward 2 rewrite rules for specific regular expressions that lead to exponential growth in DFAs. These rules do not change the expressive power of the applied regular expressions, yet prevent the exponential state space explosion in the the resulting DFA. In the last step, they develop techniques to intelligently combine multiple DFAs each of which function to recognize only one regular expression, into smaller number of DFA groups, each of which are responsible to recognize more than one regular expression. Combining all the DFAs to only one DFA, instead of merging then to small number of groups would require much more memory, that's why they took this approach instead. The authors show that they are able to create a small group of DFAs for the signature sets of Bro and Snort, and the execution of the DFA groups delivers more bandwidth than the state of the art NFA based matching engine.

Kumar et al. [37] presents another deep packet inspection system using regular expressions. The main motive in this paper is to decrease the memory consumption of DFAs, so that they could be implemented on a chip. In order to achieve this, the authors present a method that decreases the number edges per state, using default states. This method works perfectly to decrease memory consumption, however it sacrifices some bandwidth like similar methods do.

Sommer et al. [50] presents another technique to decrease the memory consumption of DFAs created from regular expressions. This method involves creating the DFA incrementally according to the content of network packets. Each time the DFA needs to transit into a state, that is not already constructed, the state is computed and added to the DFA [50]. This approach may lead to an algorithmic attack by an invader who creates packets which forces the DFA to spend most of its time create new states [22].

2.4 Netfilter/Iptables Extensions

One major drawback of iptables is that it tests a network packet against each rule in its rule set sequentially. As a result it does not scale well with the number of rules.

Heinz [52] presents a packet classification software HiPAC (high performance packet classification) implemented as a Linux kernel module implemented on top of netfilter framework.

The main contributions of the work are presenting a multi-dimensional PCP algorithm providing dynamic operations, a flexible match specification based on ranges of IP header fields such as source port, source IP address, destination port, destination IP address and high classification performance ($O(d \cdot log(w))$) where w is the bit width of the largest packet field and d is dimension [52]. This approach compiles all IP filtering rules into a single data structure and decision to drop or accept a network packet can be made within a single lookup regardless of the number of rules.

There exists a software called fwsnort [2], which transforms the rules in snort ruleset to iptables rules. Moreover, fwsnort includes some scripts to insert the transformed rules to iptables. However, not all of the snort rules can be converted to iptables rules, because iptables does not have all the matching capabilities of snort. For example, iptables does not have any module to process regular expressions and cannot match network packets using them. As of August 2011, fwsnort is able to transform 2028 of 3935 (51.4%) snort rules to iptables rules.

fwsnort heavily makes use of the string extension of iptables which is used to scan a predefined string in the network packets. The main drawback of fwsnort is the parallel to the main drawback of string extension. The rules which are added to the iptables ruleset using string extension are sequentially executed for each network packet that is processed. This property leads to a serious decline in network bandwidth.

Here is an sample snort rule, and the corresponding iptables rule converted by fwsnort. For details on iptables rules see “Linux Packet Filtering HOWTO” [47].

```
alert udp $EXTERNAL_NET any -> $HOME_NET 2140 (msg:"BACKDOOR DeepThroat 3.1
Connection attempt";content:"00"; depth:2; reference:mcafee,98574;
reference:nessus,10053; classtype:misc-activity; sid:1980; rev:4;)
```

```
IPTABLES -A FWSNORT_INPUT -p udp --dport 2140 -m string --string "00" --algo
bm --to 44 -m comment --comment "sid:1980; msg:BACKDOOR DeepThroat 3.1
Connection attempt;classtype:misc-activity; reference:mcafee,98574; rev:4;
FWS:1.1;" -j LOG --log-ip-options --log-prefix "[20] SID1980 "
```

As seen in the previous example, fwsnort converts the snort rule to an iptables rule with a LOG target. This means that if the rule matches a network packet, it does not take any measures, like dropping the network packet, but just logs information about it. This approach

is comprehensible with ease, since snort itself is not a firewall and does not do any filtering of network packets. However, fwsnort also has the ability to convert a snort rule such that it acts like a firewall rule. It does not just log information about network packets, but could drop them. The following iptables rule is an example. Notice that, only the target value of the rule is different from the previous converted rule (DROP instead of LOG).

```
IPTABLES -A FWSNORT_INPUT -p udp --dport 2140 -m string --string "00" --algo
bm --to 44 -m comment --comment "sid:1980; msg:BACKDOOR DeepThroat 3.1
Connection attempt;classtype:misc-activity; reference:mcafee,98574; rev:4;
FWS:1.1;" -j DROP --log-ip-options --log-prefix "[20] SID1980 "
```

2.5 Multiple String Matching

Well known string matching algorithms like Boyer-Moore [18] and Knuth-Morris-Pratt algorithm [36] scan a text in order to find out if it matches a single pattern. These algorithms are called single pattern matching algorithms. There is another class of string matching algorithms which try to scan a text against several string patterns. These algorithms are called multiple string matching or multiple pattern string matching algorithms.

Aho-Corasick [11] is the most widely used algorithm for multiple string matching in networking applications. We present the details of the algorithm in the following chapters. Roughly speaking, the algorithm builds a Deterministic Finite Automaton that is able to recognize all the strings that are requested to be matched. The most important aspect of the algorithm is the fact that it operates in $O(n)$ in the worst case where n is the length of the text to be searched. It inspects each character in the text at most once, regardless of the number of strings that are being scanned in the text. This is the most crucial property of an algorithm that is to be used in networking applications. In addition to reduced performance, the algorithms that do not have a linear worst time complexity expose security flaws by making the system susceptible to algorithmic attacks [22].

A survey of the string matching algorithms used for networking applications can be found in [38]. Most of the other multiple string matching algorithms are extensions of Boyer-Moore algorithm [18]. Boyer Moore algorithm is an efficient algorithm for single string matching. It has sublinear complexity for the average case, meaning that it does not have to inspect all the characters in the text on the average case to find a match. An interesting aspect of the

algorithm is that character comparisons are made from right to left starting at the end of a pattern instead of the beginning [27].

Boyer-Moore algorithm makes use of the bad character heuristic to shift unnecessary characters while matching. Suppose that T is the text to be searched and P is the pattern which is searched in the text. Let e be the endpoint of T and m be the endpoint of P. Suppose further that when searching for the jth character from the end of the pattern a mismatch is found. In other words the character T_{e-j} is different from the character P_{m-j}. If the rightmost occurence of character T_{e-j} in P is q characters away from the end of P, then P is shifted $q-j$ characters to the right such that T_{e-j} aligns with the same character in P for matching. This causes the endpoint of P to jump to the character at the index $m+q-j$. An example is as follows

```
           1
  12345678901234
T:abxdehgxijlkzt
P:  nhgbgxi
```

Here m and e are equal to 9. P and T match for 3 characters “gxi” , starting from end of P, but differ in the next character, so j is equal to 3. The rightmost occurrence of character “h” in P occurs in 5th character from the end of P, so that q equals 5. Consequently the pattern P is shifted 2 (5-3) characters to the right such that the character “h” in T and P collide with each other.The endpoint of P becomes the 11th character. If the rightmost occurrence of the mismatched character in the pattern lies in the right of the mismatch point then this heuristic does not work. The q value, also called bad character value, the distance of the rightmost occurrence of a character from the end of the pattern is calculated with the following formula [26].

$$B(c) = min\{q \mid P_{m-q} = c\}$$

For each character c that does not occur in P $B(c)$ is set to m [26].

Another heuristic that is used in the Boyer-Moore algorithm is the good suffixes heuristic [26]. When bad character heuristic and good suffixes heuristic is used together the worst case running time complexity of Boyer-Moore algorithm becomes $O(3n)$ [20]. On the other hand, to achieve a sublinear average case running time complexity, making use of only bad character heuristic is adequate [31].

Fisk et al [26][27] present a multiple string matching algorithm called Set-wise Boyer-Moore using the ideas from Boyer-Moore algorithm. The algorithm provides an improvement over bad character heuristic to be used for multiple string matching. A character's all bad character values for each pattern is calculated. The minimum of these values,which is calculated with the following formula, is used as the bad character value for that character.

$$B(c) = min\{B_l(c), 1 \leq l \leq k\}$$

$B_l(c)$ represents the bad character value of character c for the lth pattern, whereas $B(c)$ represents the global bad character value of character c. In order to accelerate comparisons of the text with the pattern strings, the algorithm builds a trie (prefix tree) using the reversed pattern strings. When a character in the text does not match the next character in the trie, global bad character value of the character is consulted [26]. The authors show that if number of patterns that are searched in the text are less than 100, this algorithm performs better than Aho-Corasick algorithm on the average and on the worst-case scenarios. However, when the number of patterns exceed 100, Aho-Corasick algorithm surpasses set-wise Boyer-Moore algorithm [26].

Another multiple string matching algorithm that takes inspiration from Boyer-Moore algorithm is Wu-Manber algorithm [54]. This algorithm starts by building three tables, namely shift, hash, and prefix tables. Shift table is akin to the bad character shift function except the fact that it computes and records the shift values of substrings instead of individual characters. Whenever shift table fails prefix and hash tables are consulted to find a match. This algorithm performs better than Aho-Corasick algorithm on the average case, however in the worst case the algorithm requires for every character of input a memory access to the shift and hash table, followed by as many string compares as the number of patterns [53]. Thus, it is not a suitable algorithm for networking applications.

There are other multiple string matching algorithms similar to Wu-Manber [21][35][29]. However, they suffer from the same design choice. They are algorithms constructed for the average case and perform poorly in worst-case scenarios.

CHAPTER 3

DESIGN AND IMPLEMENTATION

This chapter first gives an overview of the technologies and algorithms that are used in this thesis. Afterwards, the design and implementation details of our approach to implement content based packet filtering architecture in Linux kernel using Deterministic Finite Automata are presented.

3.1 Netfilter

Netfilter is a Linux kernel module and library that provides a framework for inspecting and manipulating network packets at certain positions in a protocol stack. These positions are also called *hooks* in netfilter terminology. Current netfilter implementation provides hooks for IPv4, IPv6,DECnet and ARP protocol stacks [48][52].

Several kernel functions can register to a certain hook. These functions should specify their priority while registering. When a network packet arrives at a particular hook, the registered functions at that hook are called sequentially based on their priorities. The called functions are free to inspect and manipulate the network packets if they need to. After the inspection phase is over, the inspecting functions have to return one of the following verdicts to netfilter [48][52].

1. NF_ACCEPT: the packet is accepted; continue traversal
2. NF_DROP: drop the packet; do not continue traversal
3. NF_STOLEN: the packet is taken over by the current function; do not continue traversal.

4. NF_QUEUE: queue the packet (the packet is sent to userspace for processing. if there is no process waiting to fetch network packets, the packet is dropped).

5. NF_REPEAT: call this hook again.

Depending on the type of the verdict, it is decided if the network packet in inspection continues its traversal path or not. If the verdict is NF_ACCEPT the packet continues its traversal along the path as shown in Figure 3.1.

3.1.1 Netfilter Hooks and Ipv4 Traversal Path

Figure 3.1 displays the hooks in the IPv4 protocol stack and the traversal paths of incoming, outgoing, and forwarded network packets. The netfilter hooks for Ipv4 protocol stack, which are drawn in ovals, are NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD, NF_IP_LOCAL_OUT and NF_IP_POST_ROUTING.

Incoming IPv4 packets from outside network first visit NF_IP_PRE_ROUTING hook. This hook is usually used for NAT (Network Address Translation) operations. Next they enter *Routing Decision* step. In this step, the destination address of the packet is checked to see if it matches the network address of the local machine or not. In case it matches, meaning that the packet is destined for the local machine, the packet is forwarded to NF_IP_LOCAL_IN hook. After further inspection in this hook, the packet is directed to the user process expecting it.

If the incoming packet is found to be sent to to another host apart from the local machine, it is checked if packet forwarding is enabled in kernel or not. If it is not enabled, the packet is dropped. Otherwise the packet is sent to NF_IP_FORWARD hook, which inspects and directs it to the NF_IP_POST_ROUTING hook. This hook inspects and forwards the packet to the next host.

Outgoing packets travel through 2 hooks. First they visit NF_IP_LOCAL_OUT and then NF_IP_POST_ROUTING hook. This last hook inspects and forwards the packet to the the outside network.

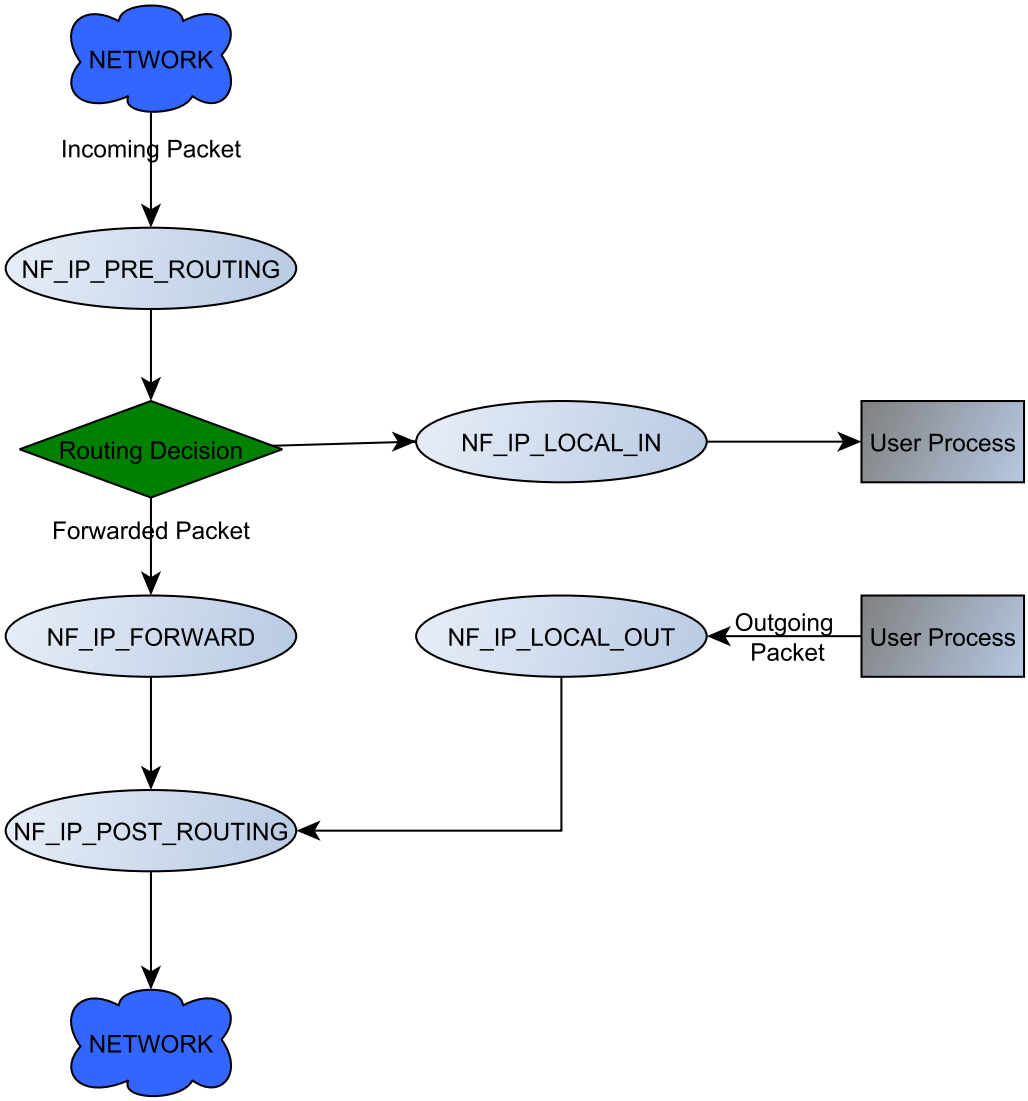


Figure 3.1: Netfilter Hooks

3.2 Iptables

Iptables is a software platform implemented on top of netfilter framework, which provides packet filtering, stateful packet inspection, packet mangling, and network address translation for Ipv4 packets. There is also another software platform named ip6tables which provides some of these functionalities for IPv6 packets.

3.2.1 Iptables Terminology

In order to understand the following chapters, there are a few general terms and expressions that need to be defined. This is a list of the most common terms used in the upcoming chapters.

- **Match** - A *match* is a test for the header fields or the contents of a network packet. A sample *match* could be “is the source ip of this packet is 10.0.10.1”
- **Target** - A *target* tells what to do with a network packet. A sample *target* would be “drop the packet”.
- **Rule** - A *rule* is a set of matches that has to match a network packet all together along with a target. A sample rule would be “if the source ip of the packet is 10.0.0.1 and the destination ip of the packet is 10.0.0.2, then drop the packet”.
- **Chain** - A *chain* is a set of rules used for a particular purpose and an application area. The purpose of a chain is determined by the table it belongs to. The application area is determined by the netfilter hook it is attached to. All the rules in a chain may only be attached to a certain hook. When a network packet travels through a netfilter hook, all the rules in the chain attached to that hook are called one by one.
- **Table** - A *table* is a set of chains, each of which are attached to different netfilter hooks. Each table serves a specific purpose, like packet filtering, or network address translation.
- **Jump** - A *jump* is an extended form of *target*. A jump could tell what to do with a network packet as a target, or could direct the packet to a user defined chain for further processing. User defined chains will be explained in the following sections.

3.2.2 Iptables Tables

There are 4 main tables used tables in iptables framework, which are *FILTER*, *NAT*, *MANGLE* and *RAW*.

- **FILTER Table** - This table is used for stateless or stateful packet filtering. There are extensions in this table which provides the means to apply deep packet inspection

techniques. This table is comprised of 3 chains INPUT, FORWARD and OUTPUT, which are attached to the netfilter hooks NF_IP_LOCAL_IN, NF_IP_FORWARD and NF_IP_LOCAL_OUT respectively. Each packet can be inspected at only one chain in this table. Incoming packets are inspected in INPUT chain, forwarded packets are inspected in FORWARD chain, and outgoing packets are inspected in OUTPUT chain.

- **NAT Table** - This table is used for network address translation. It is possible to apply SNAT or DNAT using this table. SNAT (Source Network Address Translation) modifies the source address of the IPv4 header whereas DNAT (Destination Network Address Translation) is used to modify the destination address of the IPv4 header. For TCP and UDP packets, it is also possible to modify the source port (SNAT) and the destination port (DNAT) of the TCP-IP header [52]. This table is comprised of 3 chains PREROUTING, OUTPUT and POSTROUTING, which are attached to the netfilter hooks NF_IP_PRE_ROUTING, NF_IP_OUTPUT and NF_IP_POST_ROUTING respectively.

- **MANGLE Table** - This table is used for packet mangling, in other words, to alter the payload and header fields of network packets. This table makes use of 5 chains, PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING which are attached to the corresponding 5 netfilter hooks.

- **RAW Table** - This table is used for specifying exemptions from connection tracking, in other words stateful packet inspection. This table registers itself at the NF_IP_PRE_ROUTING and NF_IP_OUTPUT hooks with PREROUTING and OUTPUT chains, respectively.

A complete picture of all the tables and the chains they use are depicted in Figure3.2.

Since packet filtering and deep packet inspection operations are implemented in FILTER table, the rest of the thesis will only focus on this table.

3.2.3 iptables tool

Iptables framework comes with the *iptables* tool. This tool works in userspace to modify the data structures of the iptables framework in kernel space. The supported operations are of three types, table operations, chain operations and rule operations. The rule operations are:

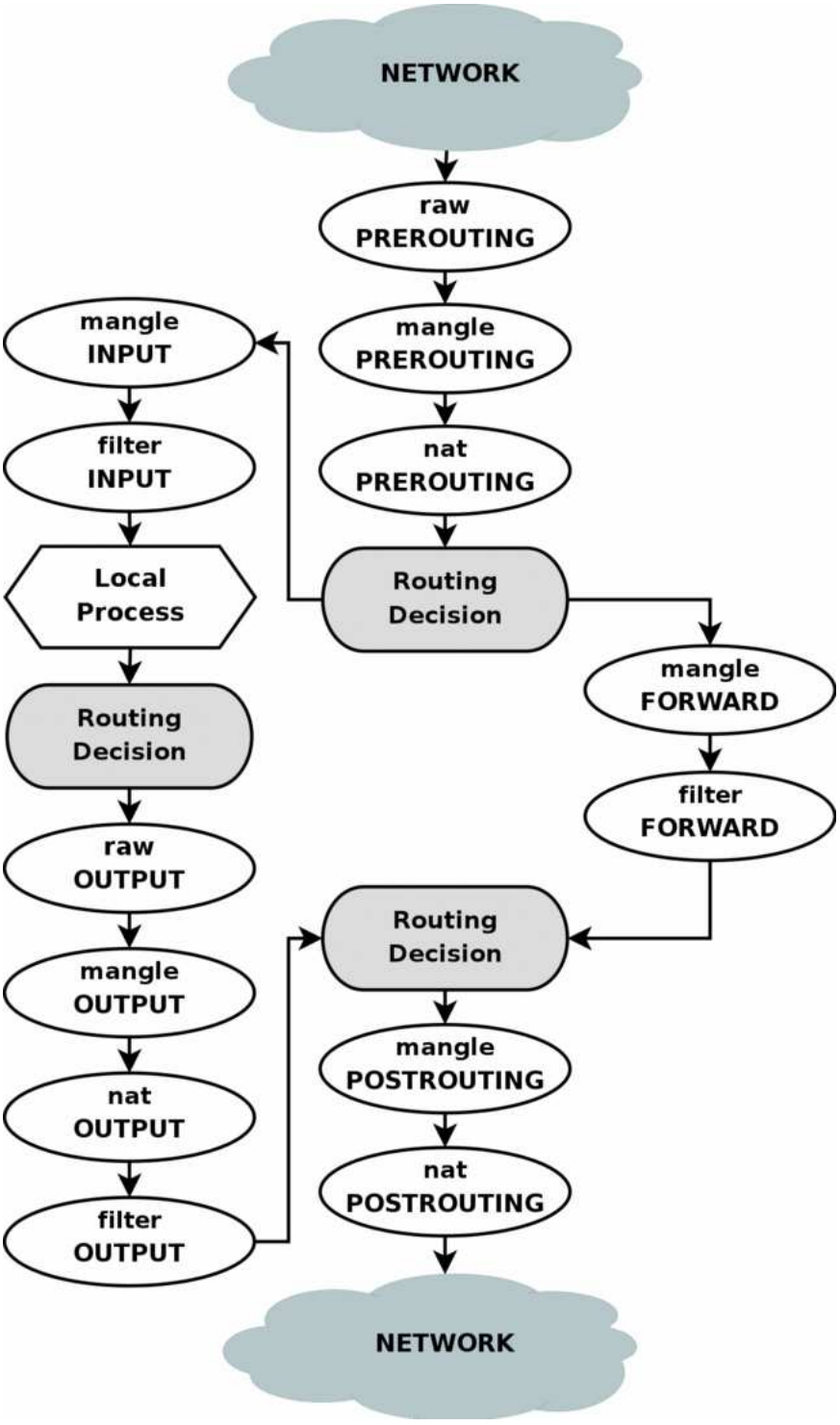


Figure 3.2: Iptables Tables and Chains, adopted from [12]

append, insert, delete, replace a single rule. The chain operations are: add, delete, rename a chain, list and delete all the rules in a chain. The table operations are: delete all the rules in a table.

An example insert rule operation is as follows:

```
iptables -t FILTER -A INPUT -s 10.0.10.1 -p tcp -j DROP
```

This rule tells iptables to add a rule to the INPUT chain of FILTER table which says “if the packet’s protocol type is tcp and source address is 10.0.10.1 jump to the DROP target”. Jumping to the DROP target results in dropping the packet.

For more information on using iptables tool, you could refer to “Linux Packet Filtering HOWTO ” [47] and “Iptables Tutorial 1.2.2” [12].

3.2.4 Connection Tracking

It is possible to do connection tracking using iptables, owing to the capabilities of the iptables *state* extension. This extension associates a connection tracking state with each network packet. There are 4 such states, as described in the following table.

- **NEW** - This state indicates that the packet is requesting to initiate a new connection.
- **ESTABLISHED** - This state indicates that the packet is a part of an already established connection (i.e. a reply packet, or outgoing packet on a connection which has seen replies [47]).
- **RELATED** - This state indicates that the packet is related to an already established connection, however it is not a part of it,such as an ICMP error. Another example is that ftp communication requires two different connections, data connection and control connection. Hence, all the packets that are part of the data connection are marked as *established* to the data connection and *related* to the control connection and vice versa.
- **INVALID** - This state indicates that the packet could not be identified. This can happen due to several factors, memory shortage, packet being too small, or if the transport layer protocol rejected the packet [48].

The following iptables command adds a rule to the INPUT chain of FILTER table which drops all the packets trying to initiate a new connection

```
iptables -t filter -A INPUT -m state --state NEW -j DROP
```

3.2.5 Chains and Targets

When a network packet arrives at at hook that a iptables chain is attached, the rules in the chain start to inspect the network packet one by one. If one of them matches, that rules target is evaluated. If none of the rules in the chain match and decide the fate of the packet, then chain policy is applied. Chain policy is could be seen as the default target of the chain, and could be one of the netfilter verdicts, NF_ACCEPT or NF_DROP.

Iptables targets may be seen as extensions of netfilter verdicts as explained in 3.1. Some of them just maps directly to a corresponding netfilter verdict, while others could also serve different purposes.

An iptables target could also be another chain, which is created by a user. These chains are called user-defined chains and they are written in lower-case letters to be distinguished from built-in chains. When a packet matches a rule whose target is a user defined chain, the packet begins traversing the rules in that user-defined chain. If none of the rules in that chain decide the fate of the packet, then traversal continues from the next rule in the current chain. This is also depicted in Figure 3.3

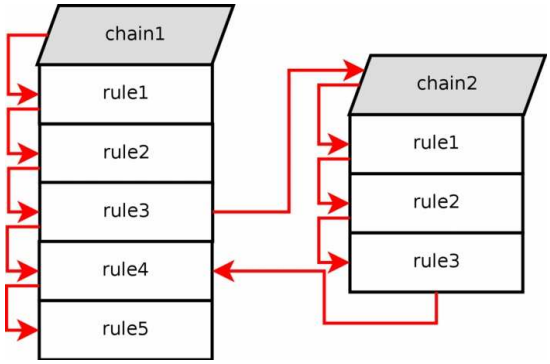


Figure 3.3: User Defined Chains, adopted from [12]

The following iptables commands are demonstrate usage of user defined chains. The first iptables command creates a new chain named “new_chain” in filter table. The second one adds a rule to the INPUT chain of filter table which directs the network packets arrived at the INPUT to the newly created chain fur further processing.

```
iptables -t filter -N new_chain
iptables -t filter -A INPUT -j new_chain
```

A list of most important iptables targets are the following .

- **DROP** - drop the packet immediately (NF_DROP)
- **REJECT** - drop the packet and convey information to the sender that the packet is dropped.
- **ACCEPT** - accept the packet (NF_ACCEPT)
- **LOG** - turn on kernel logging of network packets.
- **RETURN** - skip the remaining rules in the chain. if the chain is a built-in chain, chain policy is applied, otherwise the inspection continues from the previous chain that made a jump here.
- **QUEUE** - send the packet to userspace for processing (NF_QUEUE)

3.2.6 Iptables String Extension

Iptables string extension enables iptables to match the payloads of network packets against predefined strings.

Iptables string extension has the following options.

- –from=from offset of the string to start searching from
- –to=to offset of the string to start searching up to
- –algo=algo the algorithm to use when doing string matching, could be either bm (Boyer Moore) or kmp (Knuth Morris Pratt)
- –icase=[yes/no] if yes string match is done in a case insensitive manner
- –string=string used to enter the match string
- –hex-string=string used to enter the match string using hexadecimal characters

The following command demonstrates the use of string extension.

```
iptables -t filter -A INPUT -m string --string google -algo kmp -j DROP
```

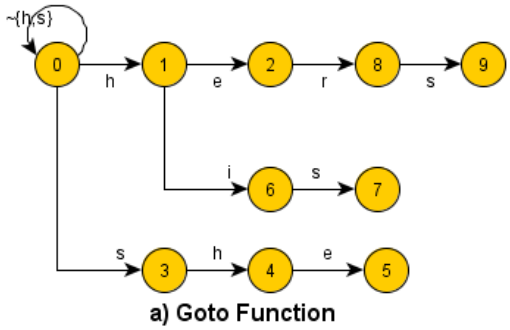
This command adds a rule to INPUT chain of iptables filter table which tries to match the payloads of network packets against the string “google” using Knuth-Morris-Pratt algorithm. Any network packet that includes the string “google” is sent to the DROP target, which causes the packet to de dropped.

3.3 Aho-Corasick Algorithm

Aho-Corasick algorithm is a multiple string match algorithm developed by Alfred V. Aho and Margaret J. Corasick in the paper [11]. The algorithm constructs a DFA (deterministic finite state automaton) which is able to recognize multiple strings. The running time of the algorithm is linear on the order of the length of the text to be searched, and independent of the number of string patterns to be searched in the text. Another characteristic of the algorithm is that it inspects each character in the text to be searched at most once. This characteristic renders the algorithm to have a bounded worst case performance against any text and any number of string patterns to be searched. As mentioned before, having a bounded worst case running time complexity is the most important property of a networking application.

The purpose of Aho-Corasick algorithm is to create a DFA which is able to recognize multiple strings. Let $K = y_1, y_2, \ldots, y_k$ be a finite set of strings and x be the text string to be searched. Aho-Corasick algorithm starts with building a pattern matching machine which is able to locate and identify all substrings of x which are patterns in K [11]. A pattern matching machine to locate the strings {he, hers, she, his} is depicted in Figure 3.4.

The pattern matching machine is represented by the functions $g(goto)$, $f(failure)$ and $output$. The goto function is represented in Figure3.4(a). The function is represented as a state machine. The states are shown by numbers and the input symbols are written on the edges. The start state is 0. The goto function maps a pair consisting of a state and an input symbol into a state or a the message *fail*. To illustrate, the edge labeled e from 1 to 2 indicates that $g(1, w) = 2$. The lack of an edge signals that the result is fail. This signals that $g(1, y) = fail$. If the goto function signals fail, then fail function is consulted. This function maps a state to another state, it is used as a wildcard transition for non-matching input symbols. The output function records a set of string associated with each state. If the pattern matching machine comes to a state whose output is not empty, it means that the machine found some patterns in



a) Goto Function

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

b) Failure function

i	output(i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

c) Output function

Figure 3.4: Pattern matching Machine for {he, she, his, hers}, adopted from [11]

the scanned text.

Assume that our input is the string “shis”. Without using the failure function the pattern machine makes the following transitions.

```
  s    h    i
0 -> 3 -> 4 -> fail
```

Even though the “shis” includes one of the patterns “his”, using only *goto* and *output* functions is not enough to recognize this pattern. Taking the failure function into account results in the following transitions.

```
  s    h                    i    s
0 -> 3 -> 4 -> (fail -> 1 ) -> 6 -> 7
```

In this case, the pattern matching machine is able to recognize the pattern string “his”. As illustrated by this example the fail functions actually acts like skipping over an empty transition in a NFA (Nondeterministic Finite State Automaton). In other words, we can represent the pattern machine represented by the 3 functions, using a NFA instead, as illustrated by the Figure 3.5.

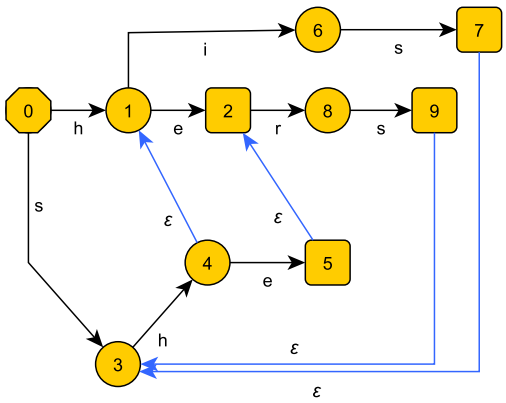


Figure 3.5: Nondeterminitic Finite State Automata

The final states are drawn using squares instead of circles. ε marks the empty transitions. 0 is the start state.

The algorithm for matching a text string against a set of patterns using the pattern matching machine is illustrated in Algorithm 1, which is adopted from[11].

Algorithm 1 Pattern matching machine

Input. A text string $x = a_1a_2...a_n$ where each a_i is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function $output$

Output. Locations at which patterns occur in x.

```
1:  state ← 0
2:  for i ← 1 until n do
3:      while g(state, a_i) = fail do
4:          state ← f(state)
5:      end while
6:      state ← g(state, a_i)
7:      if output(state) ≠ empty then
8:          print(i)
9:          print(output(state))
10:     end if
11: end for
```

This algorithm can also be seen as a NFA parsing algorithm. The outmost for block inspects all the characters in the input text one by one. Each pass through the for loop represents one operating cycle of the machine. The inner while block on the other hand, handles empty

transitions.

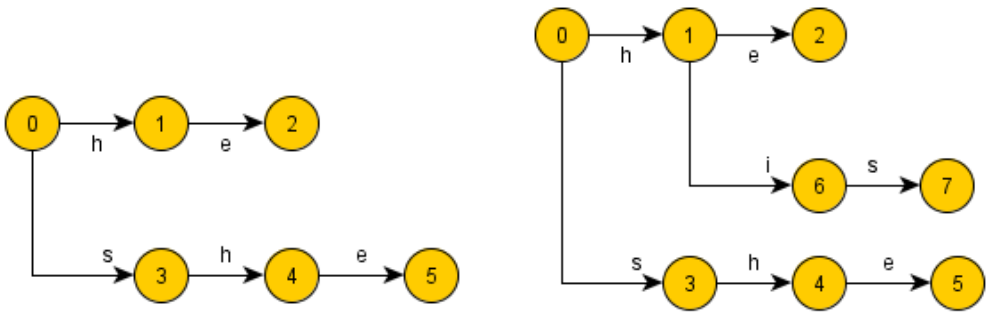
The main problem with this algorithm similar to the problem of NFA parsing. In fact, the algorithm may make more than one state transition per input character, due to the empty edges. This results in deteriorated running time.

The complexity of Algorithm 1 is $O(2n)$ where n is length of the text being scanned. The details of the proof is in [11].

3.3.1 Construction of Goto, Output and Failure Functions

This section deals with how goto, output and failure functions are constructed in the first place.

Algorithm 2, which is adopted from[11], depicts the construction of goto function. This algorithm also partially construct output function. Initially, goto function is represented as a graph of only one node standing for the start state (state no 0). Afterwards, the edges and states representing each pattern are added to the graph beginning from the start state. A catch of the algorithm is the fact that if two patterns have common prefixes, the states and the edges representing the common prefixes are entered the graph only once. This is enabled by the while loop in the match function of Algorithm 2. An example of this step in Figure 3.6.



(a) the goto graph after adding the patterns {he, she}, adopted from [11]

(b) the goto graph after adding the patterns {he, she, his}, adopted from [11]

Figure 3.6: Goto Graph

Figure 3.6(a) shows the goto graph after adding the patterns {he, she}. Figure 3.6(b) shows the goto graph after adding the pattern “his” to the previous set of patterns. As demonstrated by the latter graph, the edge h which connects state 0 to state 1 is shared by “her” and “his”.

Algorithm 2 Construction of *goto* function

Input. Set of patterns $K = y_1, y_2, ..., y_k$

Output. Goto function g and a partially computed output function $output$

Method. We assume $output(s)$ is empty when state s is first created

$g(s, a) = fail$ if a is undefined or if $g(s, a)$ has not yet been defined.

The procedure $enter(y)$ inserts into the goto graph a path that spells out y.

```
1:  newstate ← 0
2:  for i → k do
3:      enter(y_i)
4:  end for
5:  for each symbol a such that g(0, a) = fail do
6:      g(0, a) ← 0
7:  end for
8:
9:  procedure ENTER(a_1 a_2 ... a_m)
10:     state ← 0
11:     j ← 1
12:     while g(state, a_j) ≠ fail do
13:         state ← g(state, a_j)
14:         j ← j + 1
15:     end while
16:     for p ← j until m do
17:         newstate ← newstate + 1
18:         g(state, a_p) ← newstate
19:         state ← newstate
20:     end for
21:     output(state) ← a_1 a_2 ... a_m
22: end procedure
```

Algorithm 3 Construction of $failure$ function

Input. Goto function g and a partially computed output function $output$

Output. Failure function f and output function $output$

1: $queue \leftarrow empty$
2: **for** each symbol a such that $g(0, a) = s \neq 0$ **do**
3: $queue \leftarrow queue \cup \{s\}$
4: $f(s) \leftarrow 0$
5: **end for**
6: **while** $queue \neq empty$ **do**
7: $r \leftarrow$ the next state in queue
8: $queue \leftarrow queue - \{r\}$
9: **for** each symbol a such that $g(r, a) = s \neq fail$ **do**
10: $queue \leftarrow queue \cup \{s\}$
11: $state \leftarrow f(r)$
12: **while** $g(state, a) = fail$ **do**
13: $state \leftarrow fail(state)$
14: **end while**
15: $f(s) \leftarrow g(state, a)$
16: $output(s) \leftarrow output(s) \cup output(f(s))$
17: **end for**
18: **end while**

Algorithm 3, which is adopted from[11], constructs the failure function. It operates in two steps.

1. In the first step, failure values of all the states of depth 1 is set to 0.

```
for all states s of depth 1, f(s) = 0
```

2. In the second step, failure values of the states of depth d are computed using the failure values of the states of depth $d-1$, using the following formula.

```
for all states r such that g(s,a) = r
    if g(f(s)*,a) = t
       then f(r) = t
```

g(f(s)*, a) means that if there is no edge labeled a from the state f(s), then look for f(f(s)), and so on.

To illustrate, the algorithm first sets f(1), and f(3) to zero. When it has to compute the failure value of the state 4, since $g(3,h) = 4$ and $g(fail(3) = 0, h) = 1$, f(4) is set to 1.

The complexity of Algorithm 2 and Algorithm 3 are linearly proportional to the sum of the lengths of the patterns [11]. Thus, their complexity is $O(k)$ where k is the sum of the lengths of the patterns.

3.3.2 Next Move Function

As we stated before, the 3 functions, goto, failure and output, correspond to NFA. In the next phase of the Aho-Corasick algorithm a new function, named next move function is constructed from the other 3 functions. This function actually represents a DFA created from the previous NFA. The algorithm to create the next move function is depicted in Algorithm 4. This algorithm is adopted from [11].

The DFA that is created by Algorithm 4, uses $256 \cdot 4 \cdot (k+1)$ bytes, where k is the sum of the lengths of the patterns. The reason is that the transitions of the DFA are stored in a two dimensional integer (4 byte) array, there are at most $k+1$ states, each state has at most 256 outgoing transitions.

Algorithm 4 Construction of *nextmove* function

Input. Goto function g and failure function f

Output. Next move function δ

1: $queue \leftarrow empty$
2: **for** each symbol a **do**
3: $\quad \delta(0, a) \leftarrow g(0, a)$
4: $\quad$ **if** $g(0, a) \neq 0$ **then**
5: $\quad\quad queue \leftarrow queue \cup \{g(0, a)\}$
6: $\quad$ **end if**
7: **end for**
8: **while** $queue \neq empty$ **do**
9: $\quad r \leftarrow$ the next state in queue
10: $\quad queue \leftarrow queue - \{r\}$
11: $\quad$ **for** each symbol a **do**
12: $\quad\quad$ **if** $g(r, a) = s \neq fail$ **then**
13: $\quad\quad\quad queue \leftarrow queue \cup \{s\}$
14: $\quad\quad\quad \delta(r, a) \leftarrow s$
15: $\quad\quad$ **else**
16: $\quad\quad\quad \delta(r, a) \leftarrow \delta(f(r), a)$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **end while**

Algorithm 4 is quite simple in essence. It adds all the transitions and nodes in the goto function graph to the nextmove function graph. If a state does not have a transition for a character, it looks for all the other states that are reached by the first state through the failure function. If one of these failure states has a transition for that character, that transition is added to the first state in the nextmove function graph. The next move function for the pattern set {his, she} is represented as a DFA in Figure 3.7. 0 is the start state and final states are drawn in squares, all the other states are non final states.

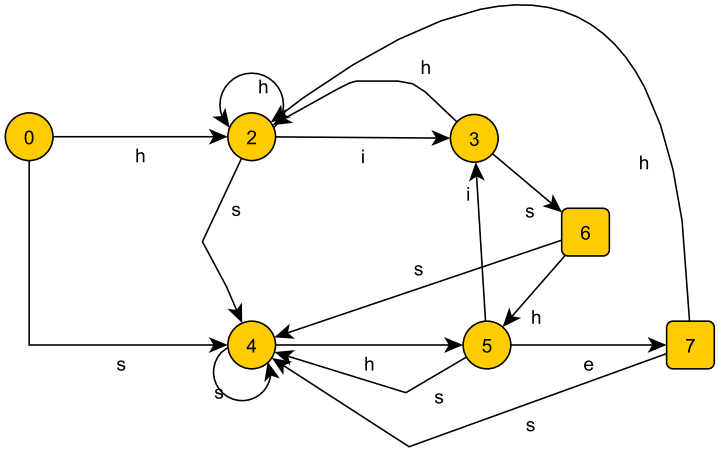


Figure 3.7: DFA for {his, she}

3.3.3 Optimizations to Aho-Corasick Algorithm

In this section we propose two optimizations to reduce the memory consumption of Aho-Corasick algorithm. The two optimizations are listed below.

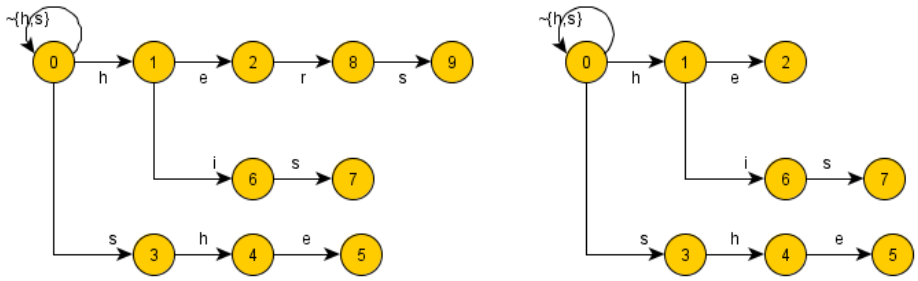
1. Removing the edges of final states and the states connected to these edges
2. Applying DFA minimization

Aho-Corasick algorithm is designed to match a text string to a set of patterns continuously. By continuously we mean that whenever the algorithm matches a pattern in a text it continues to scan the rest of the text for other matches. This behaviour is not needed for networking applications if all match patterns require the same action. If a network packet matches a pattern, the action associated with that pattern is executed immediately, it is not needed to find out how many patterns a network packet matches.

We could make use of this property to reduce the memory consumption of Aho-Corasick algorithm. In order to achieve memory reduction, after goto graph is constructed we prune the graph by removing the outgoing edges of the final states and the other states that are connected to these edges.

For the patterns {he,hers,his,she}, Aho-Corasick algorithm creates the goto graph in Figure 3.8(a). The states number 2,5,7,9 are final states. Our optimization prunes the nodes 8 and 9 which is depicted in Figure 3.8(b).

In the worst case, there is a pattern of length 1 and the algorithm reqires the removal of the all the edges except one edge. Thus it has complexity of $O(v)$ where v is the number of edges. There are atmost k states and $(k-1)$ edges in the goto graph where k is the sum of the lengths of the patterns. Thus, the complexity is $O(k)$



(a) goto graph for {he,hers,his,she}

(b) pruned goto graph

Figure 3.8: Pruning unnecessary nodes

Another optimization we apply is DFA minimization [30][41][44] to the DFA constructed by Aho-Corasick algorithm. Even though Aho-Corasick algorithm tries to merge the common prefix nodes of patterns that have common prefixes, this does not result in a minimized DFA. The reason is that two strings may have common substrings in places other than their prefixes like the strings “alexdesouza” and “deividdesouza”. Applying Aho-Corasick algorithm to these strings will result in a DFA with redundant nodes and edges. In order to overcome this problem we apply DFA minimization using OpenFst library [8].

DFA minimization takes $O(nslogn)$ lime where n is the number of states and s is the alphabet size [30]. In our case s is always 256, since we use the extended ASCII table. n is equal to the total number of characters in the strings being filtered plus 1. Thus, the complexity becomes $O(256klogk)$, where k is the sum of the lengths of the patterns.

These two optimizations that we put forward assume that all the string patterns which are compiled into a single DFA are associated with the same iptables target (DROP or ACCEPT). In the case that some string patterns have different targets, the resulting DFA cannot be pruned and minimized. This stems from the fact that a longer pattern string might have a higher priority than a shorter one, and our optimizations might prune the nodes which recognize the longer pattern string.

3.4 Extending Iptables with New Match Extensions

Iptables has a fairly extensible architecture. It is possible to add new match and target extensions to iptables, and load them dynamically in case they are needed.

Adding a new extension to iptables involves two parts. The first one is to implement a kernel module which provides the required functionality in kernel space. The second one is to implement a module for the iptables command line tool to manage the kernel module from userspace.

In the following subsections, we dwell upon the issue of implementing an new match extension for iptables. For details on implementing a new target extension please refer to [48].

The userspace module of a match extension can send data to the kernel module. This is made possible by using a shared "struct" whose values are filled by the userspace module. Whenever a new rule which includes the defined match is added to an iptables chain, the fields of the shared struct are filled and copied to the kernel space by the userspace module. When the kernel module is loaded, it can access the filled shared struct. If some of the shared struct's fields are pointers, the kernel module has to copy the content of the pointers from userspace by using the function *copy_from_user*.

3.4.1 Kernel Module for A New Match Extension

Implementing a kernel module for a new match extension for iptables requires the following steps.

1. a struct of type *xt_match* should be defined

2. the member functions of the struct which are entry points called by iptables should be implemented

3. the match should be registered to the iptables system by using the function xt_register_match and the defined struct.

The struct *xt_match* has the following fields

- **name** This field is set to the name of the match function
- **revision** This field is set to the revision number of the match extension
- **checkentry** This field points to function a which is called whenever a new rule containing the match extension is added to iptables. The main duty of the function is to check if the match function is suitable for the table, chain, and the netfilter hook it is attached to. Some match extensions may be developed for some specific chains or tables. It is this function's duty to inspect inconsistencies. This function is also used to allocate some dynamic structures to be used by the match extension.
- **match** This field points to the main match function. This function is handed a network packet in a *sk_buff* structure. It inspects and decides if the packet matches or not.
- **destroy** This field points to a function which is called whenever a new rule containing the match extension is removed from iptables. This function is mainly used to deallocate the dynamic structures allocated by the *checkentry* function.
- **matchsize** This field is set to the size of the shared struct between the userspace and the kernel space.
- **me** This field is set to the macro THIS_MODULE, which returns a pointer to the implemented kernel module.

The kernel module code of a sample match extension could be found in Appendix A.

3.4.2 Userspace Administration Module For a New Match Extension

Implementing a userspace administration module for a new match extension for iptables requires the following steps.

1. a struct of type *xtables_match* should be defined

2. the member functions of the struct, which are entry points called by iptables userspace tool, should be implemented

3. the match should be registered to the iptables userspace part by calling xtables_register_match and the defined struct in the _init function of the userspace module.

The struct xtables_match has the following fields.

- **name** This field is set to the name of the match function
- **revision** This field is set to the revision number of the match extension. It has to match the revision number in kernel space.
- **version** This field is set to the macro XTABLES_VERSION. This field ensures that the module binary is compatible with the iptables binaries in use
- **size** This field is set to the size of the shared struct between the userspace and the kernel space.
- **help** This field points to a function which prints information about how to administer the match extension from userspace
- **init** This field points to a function which is called first when a new rule using the match extension is requested to be added to iptables. This function is usually used to fill in the shared struct.
- **parse** This field points to a function which is used to parse the options of the match extension. This function is called right after init function
- **final_check** This field points to a function which is used to check if the obligatory options of the match extension are specified. For example, specifying *–algo* option is compulsory for string extension to be initialized.
- **print** This field points to a function which prints information about the match. This information usually consists of the options specified.

- **save** This field points to function which does the reverse of what parse function does, it prints the parsed options that are used by this rule. This function is used by iptables-save command [48].

- **extra_opts** This field points to a struct which defines the options of the match extension. For example string extension has the options *–algo*, *–string*, *–hex-string* and so on.

The userspace administration module code of a sample match extension could be found in Appendix B.

3.5 Packet Handling in Userspace

It is possible to process network packets and decide which ones to drop or allow in userspace. Initially, the network packets must be sent to userspace using one of the targets QUEUE or NFQUEUE. NFQUEUE is merely an extension of QUEUE in that the packets could be sent to a specific queue with a 16-bit queue number. If QUEUE is used instead, all the packets are sent to the 0 numbered queue. In order to capture and process the packets in userspace using QUEUE target, one has to use libipq library if he uses a linux kernel older than version 2.6.14. After the development of NFQUEUE target another library has been developed in order to process network packets in userspace, namely libnetfilter_queue library [5], and libipq library became deprecated in favor of the new library for newer linux kernels. libnetfilter_queue library can handle the network packets that are sent to QUEUE or NFQUEUE targets.

The following iptables command sends all the incoming packets to userspace.

```
iptables -A INPUT -j QUEUE
```

Next, an application to process the network packets must be run in userspace. If there is no application running to process the network packets, all packets are dropped. An example application code using lib_netfilterqueue can be seen in Appendix C.

This approach has several advantage.

1. Programming in kernel is restrictive

2. Any programming language can be used

3. Debugging, development and upgrading of modules is easier.

However, depending on amount of information you need in userspace (headers and packet payload) some extra CPU time is spent to copy content from kernel memory space to user memory space and for context switching. Thus implementations based on libipq and libnetfilter_queue provide lower bandwidth values. This deficit could be compensated by a system which can filter network packets in userspace using multiple threads to utilize multi-core processors.

3.6 FSA Based Packet Filtering in Kernel

This section presents an architecture which can filter network packets in kernel space against a set of string signatures. Moreover, the proposed architecture inspects each network packet only once regardless of the number of strings to be matched.

The architecture is depicted in 3.9. The squares with thick borders represent applications in the architecture, whereas the ovals represent files. Furthermore, the thick arrows in the figure represent the data flow among modules, whereas the thin arrows represent the paths the network packets traverse. The numbers in the thick arrows depict in which order the data flow among the modules take place.

Our architecture consists of mainly two components: DFA creator and the fsa match extension for iptables. fsa match extension is implemented as two modules, the kernel module and the userspace administration module.

Initialization of our architecture takes the following steps.

1. The strings to be filtered are written to the signatures file. Characters in the strings could also be given as hexadecimal numbers, such as \xAB.

2. DFA creator is executed. This application reads all the strings in the signatures file and creates a DFA using Aho-Corasick algorithm and the two optimizations described in Section 3.3.3, which is able to recognize all strings in the signature set. Afterwards

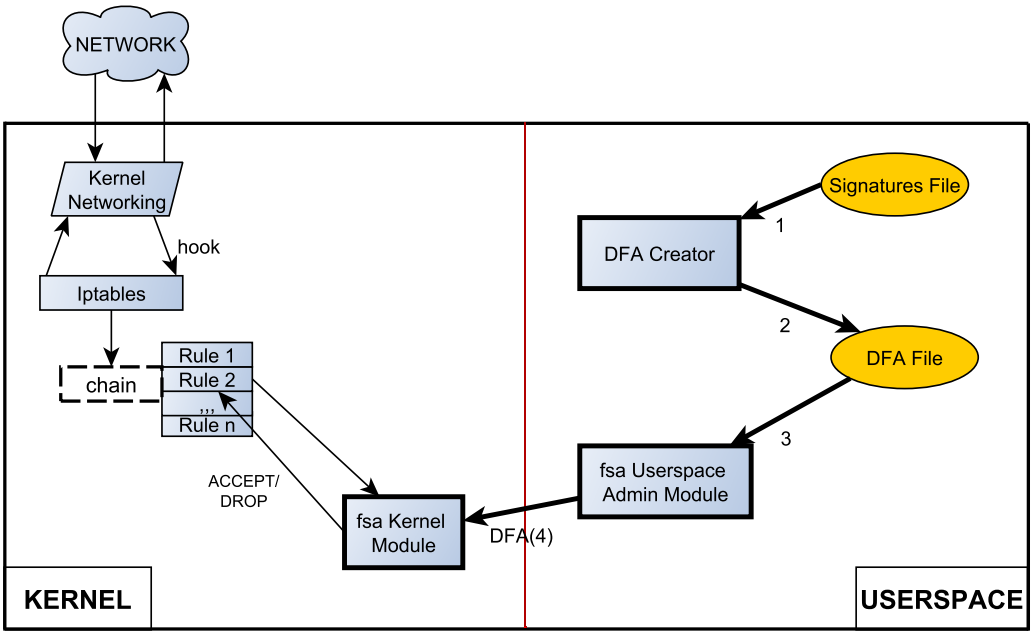


Figure 3.9: DFA-based multiple string matching architecture in kernel. The thick arrows represents the data flow among modules, whereas the thin arrows represent the paths the network packets traverse

DFA creator writes the result DFA into the DFA file which resides in a well defined path in the file system.

3. fsa extension's userspace administration module is started. The following command is executed to achieve this.

```
iptables -A CHAINNAME -m fsa -j DROP
```

Upon the execution of the command, iptables fsa extension's userspace module is loaded. This module reads the DFA file and and saves the DFA in a shared struct. This results in the transfer of the shared struct to kernel space. An important detail is that this module runs only once. It tells iptables framework to load the fsa kernel module and transfer the shared struct to kernel space and its entry functions in the *xtables_match* struct are not called afterwards. However, its address space is not deallocated because the shared struct's fields could include pointers, and the kernel module could request the copying of data which are pointed by these pointers from userspace.

4. kernel fsa module gets loaded. It reads the DFA in the shared struct and saves it into its address space. Next, the rule which is presented by the iptables command, which drops the network packets that are matched by fsa match extension is added to the iptables

chain of name CHAINNAME.

The flow of the network packets after the initialization is completed takes place as follows.

1. A network packet arrives at the chain that the rule including fsa match extension is added.
2. fsa match extension's kernel module tries to match the network packet using the constructed DFA. The algorithm to match a network packet with a DFA is presented in Algorithm 5.
3. If the match takes place, the packet is dropped, otherwise it continues in its traversal path.

Algorithm 5 Algorithm to match a network packet payload with a DFA

Input. A packet payload *payload* and a DFA represented by three functions

startstate() returns the start state of the DFA

$isFinalState(st)$ returns *true* if st is a final state

$getNextState(st, c)$ returns the state connected to *st* with edge *c*

```
1: state ← startState()
2: for i ← 0 until payload length do
3:     if isFinalState(state) then
4:         return true
5:     else
6:         state ← getNextState(state, payload[i])
7:     end if
8: end for
9: return false
```

3.7 FSA Based Packet Filtering in Userspace

This section presents an architecture which can filter network packets in userspace against a set of string signatures. We designed and implemented this architecture to assess the the effectiveness of packet filtering in kernel space with regards to userspace filtering.

The architecture is depicted in 3.10. The squares with thick borders represent applications in the architecture, whereas the ovals represent files. Furthermore, the thick arrows in the figure represent the data flow among modules, whereas the thin arrows represent the paths the network packets traverse. The numbers in the thick arrows depict in which order the data flow among the modules take place.

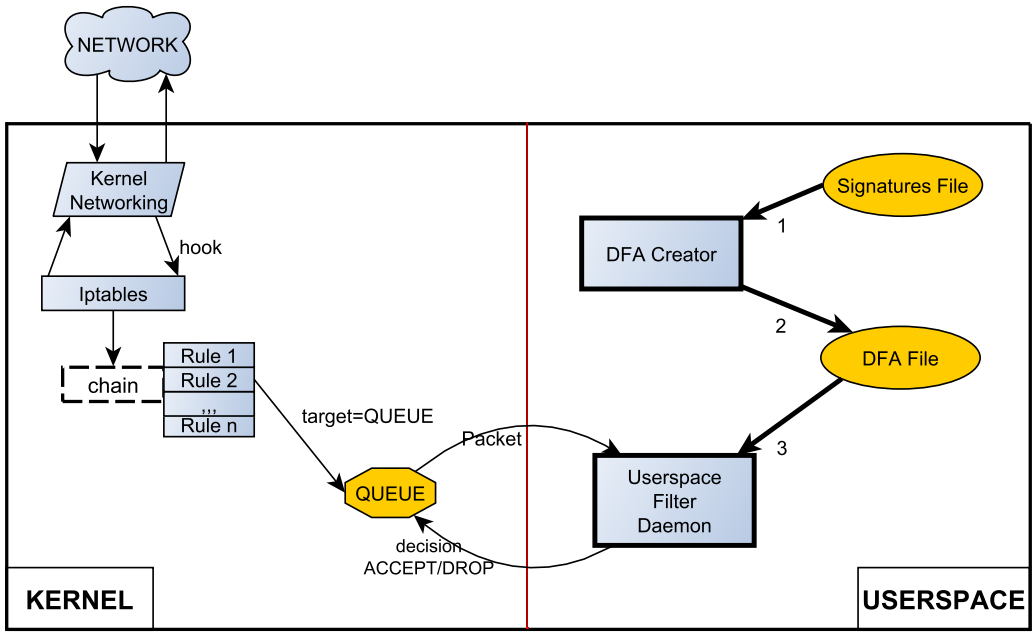


Figure 3.10: DFA-based multiple string matching architecture in userspace The thick arrows represents the data flow among modules, whereas the thin arrows represent the paths the network packets traverse

Our architecture consists of mainly two components: DFA creator and the Userspace Filter Daemon.

Initialization of the userspace filtering architecture follows the following steps.

1. The strings to be filtered are written to the signatures file. Characters in the strings could also be given as hexadecimal numbers, such as \xAB.

2. DFA creator is executed. This application reads all the strings in the signatures file and creates a DFA using Aho-Corasick algorithm and the two optimizations described in Section 3.3.3, which is able to recognize all strings in the signature set. Afterwards DFA creator writes the result DFA into the DFA file which resides in a well defined path in the file system.

3. This step differs from the previous architecture. There is no iptables extension. Instead network packets must be directed to and filtered in userspace. The following iptables command is executed in order to direct the network packets that arrive at the chain with name CHAIN_NAME to userspace.

   ```
   iptables -A CHAIN_NAME -j QUEUE
   ```

4. Userspace Filter Daemon is executed. The application reads FSA file and starts to listen for the packets that are directed to user. This application works as a daemon and continuously listens for directed packets unless it is terminated by external interruption.

The flow of the network packets after the initialization is completed takes place as follows.

1. A network packet arrives at the chain that the rule directing the packets to QUEUE is added. This causes the network packet to jump to queue numbered 0.

2. Userspace Filter Daemon grabs the directed packet from the queue and tries to match it using the constructed DFA. The algorithm to match a network packet with a DFA is presented in Algorithm 5.

3. If the match takes place, the packet is sent back to kernel space with the verdict NF_DROP. This causes the packet to be dropped. If no match takes place, then the packet is sent back to kernel space with verdict NF_ACCEPT. The packet continues in its traversal path from the next rule in the kernel space.

CHAPTER 4

EVALUATION

4.1 Optimization Evaluation

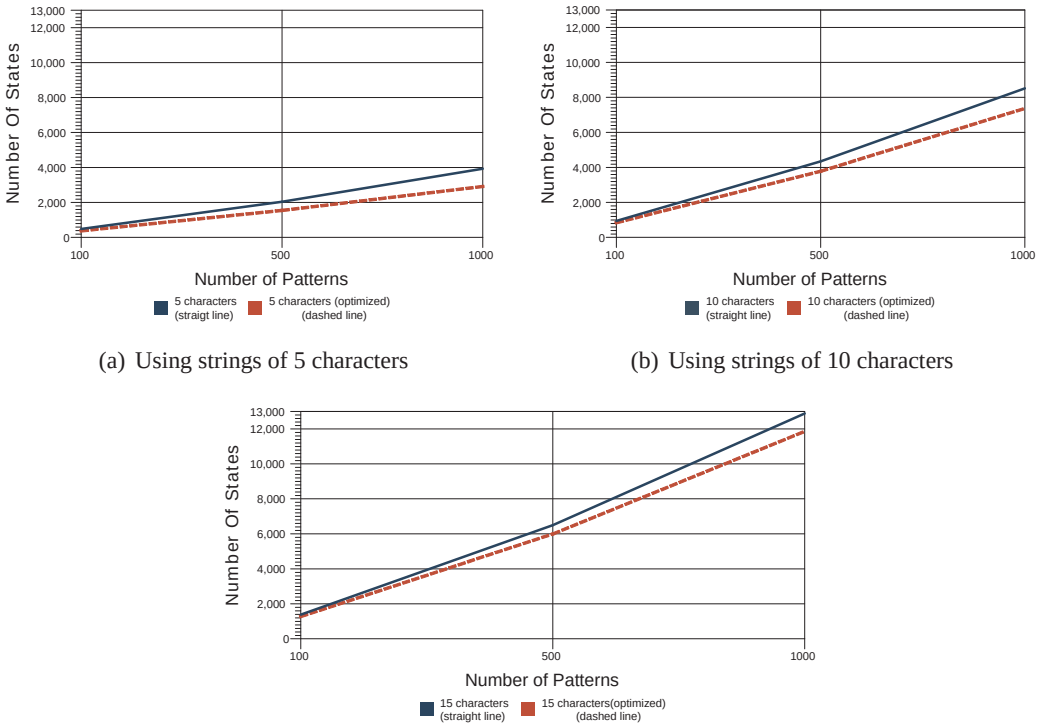
In this section we present the experiments we employed in order to assess the efficiency of our optimizations to Aho-Corasick and the results of the experiments.

We employed the following two experiments to assess the efficiency of our optimizations.

4.1.1 Optimization Experiment 1

In this experiment, we measure the number of states of the final deterministic state automata that are produced by the Aho-Corasick algorithm with and without using our optimizations. We measure the number of states of the final automata that are produced according to 3 distinct parameters. The first parameter is the length of the string patterns that are used in the Aho-Corasick algorithm. This parameter takes one of the values 5, 10 and 15. The string patterns are randomly composed from the extended ASCII alphabet. The second parameter is the number of string patterns that are compiled to produce the final deterministic state automata. This parameter takes one of the values 100, 500 and 1000. The third parameter is whether our optimizations are utilized in creating the final automata or not. The number of states of different DFA that are created according to these 3 parameters are presented in Figure 4.1. The 3 different charts correspond to the automata that are compiled using different lengths of strings patterns. In each chart, the y-axis depicts the number of states in the final DFA and the x-axis depicts the number of string patterns compiled. Each chart depicts the number of states in the two automata that are created using and without using the optimizations,

respectively. Straight lines depict the automata without optimization, whereas the dashed depict the automata with optimization.



(a) Using strings of 5 characters

(b) Using strings of 10 characters

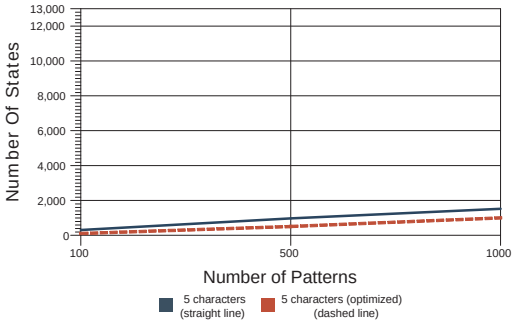
(c) Using strings of 15 characters

Figure 4.1: Optimization Experiment 1 Results

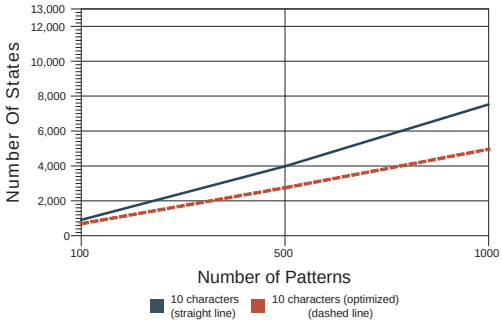
These results show that the number of states of the final DFA produced by the Aho-Corasick algorithm go up with increasing number of patterns and increasing number of characters per pattern. Moreover, our optimizations succeed in decreasing the number of states of the final DFA. For example, a deterministic state automaton that is produced by Aho-Corasick algorithm using 1000 string patterns of 10 characters has 8515 states. Our optimizations reduce this number to 7366 (86.5%).

4.1.2 Optimization Experiment 2

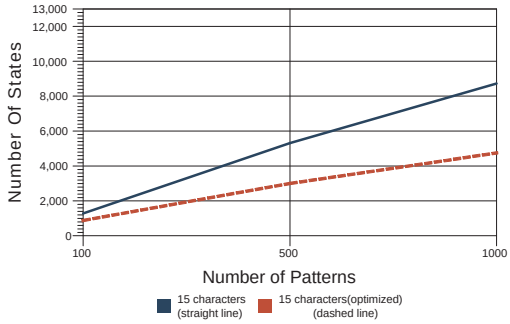
In this experiment, we measure the number of states of the final DFA that are produced by the Aho-Corasick algorithm with and without using our optimizations. The difference of this experiment from the previous one is that the string patterns are selected from the words in an English dictionary instead of being randomly composed from the extended ASCII alphabet. The results are depicted in Figure 4.2.



(a) Using strings of 5 characters



(b) Using strings of 10 characters



(c) Using strings of 15 characters

Figure 4.2: Optimization Experiment 2 Results

The first different result of this experiment from the previous one is the reduced number of states in the resulting automata. The number of states of the deterministic state automaton produced by Aho-Corasick algorithm using 1000 different string patterns of 10 characters without optimization drops from 8515 to 7526. The second result is the fact that our optimizations reduce the number of states more effectively compared to the previous experiment. The number of states of a deterministic state automaton that is produced by Aho-Corasick algorithm using 1000 string patterns of 10 characters drops from 7526 to 4950 (65.7%). This rate was 86.5% in the previous experiment. Moreover, the number of states of the automaton produced using string patterns of 10 characters with optimization becomes less then the number of states of the automaton produced using string patterns of 15 characters without optimization. These results are due to the fact that the words in a dictionary have higher chance to have overlapping parts, compared to randomly composed strings. This fact enables Aho-Corasick algorithm to create DFA with smaller number of states and enables our optimizations to reduce the number of states even more.

4.2 Packet Filtering Evaluation

In this section we present the experiments we employed in order to assess the efficiency of our packet filtering architecture and the results of the experiments.

In order to employ the experiments we made used of iperf [3] tool. Iperf is a software application that can measure the bandwidth between 2 computers using TCP packets. Iperf has to be installed on both of the computers. It should be executed in one of the computers in server mode and in the other one in client mode. Iperf measures the bandwidth between the two computers by sending TCP packets from client to server. In other words, it actually measures the bandwidth of the incoming traffic to the server machine.

We employed the following 3 different experiments to assess the efficiency of our packet filtering architecture.

4.2.1 Experiment 1

In this experiment, two computers are connected to each other through a 1 Gigabit ethernet switch. Each computer has a 2.80 Ghz Intel Pentium Dual-Core processor, 2 GB memory and 1 Gigabit network interface card. We use one computer as the iperf server and the other one as the iperf client. We measure the bandwidth of the incoming traffic to the server when one of the three filtering cases configured to filter incoming network traffic. The first one is iptables string extension, the second is the architecture we developed to filter network packets in kernel space and the third is the architecture we developed to filter network packets in userspace.

To evaluate the packet filtering efficiency of these applications, we configure each of them to filter a specified number of string patterns and measure the bandwidth of the traffic from the client to the server machine using iperf. The pattern strings that we use are all made up of 10 characters. The strings are randomly composed from the extended ASCII alphabet.

The results of the first experiment can be seen in Figure 4.3 and Table 4.1. The first column in the table shows the number of string patterns used, and the applications used for filtering are written in the second row. **string** denotes iptables string extension, **fsa_kernel** denotes the architecture we developed to filter network packets in kernel space, and **userspace filter**

Table 4.1: Experiment 1 Results

	Bandwidth (Mbits/sec)		
number of patterns	string	fsa_kernel	userspace filter
0	923	923	923
1	693	651	296
5	346	651	296
10	210	651	296
50	49.7	651	296
100	25.4	651	296
500	5.09	651	296
1000	2.6	651	296

denotes the architecture that we developed to filter packets in userspace. The values in the cells in second to fourth column shows the measured bandwidth in Mbits/sec.

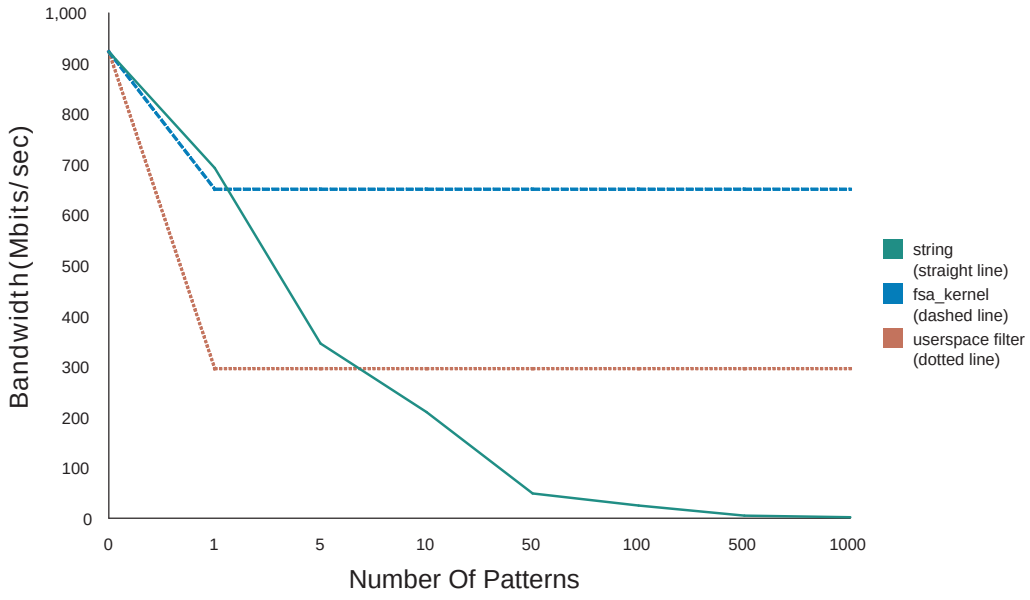


Figure 4.3: Experiment 1 Results

When there are no patterns to filter, the bandwidth is measured as 923 Mbits/sec. When only one string pattern is used to filter incoming traffic, string extension causes the bandwidth to decline to 693 Mbits/sec, fsa_kernel to 651 Mbits/sec and userspace filter to 296 Mbits/sec. When more patterns are added to be filtered, string extension causes the bandwidth to drop rapidly, whereas fsa_kernel and userspace filter consume the same amount of bandwidth regardless of the number of string patterns. For example, when 1000 string patterns are used for filtering, the bandwidth drops to 2.6 Mbits/sec when using iptables string extension. On the other hand the bandwidth that is measured when fsa_kernel or userspace filter is used does

not change, 651 Mbits/sec and 296 Mbits/sec.

This result is due to the fact that the string extension of iptables inspect a network packet as many times as the number of string patterns using Knuth-Morris-Pratt algorithm. In other words, iptables string extension has $O(m \cdot n)$ complexity where n is the packet payload size and m is the number of patterns. On the other hand the applications we developed inspect a network packet and each character in the packet only once and match the network packet against multiple string patterns simultaneously, owing to Aho-Corasick algorithm. fsa_kernel and userspace filter both have $O(n)$ complexity where n is the packet payload size.

Another result that we could draw from the measurements is that filtering network packets in userspace consumes more bandwidth than filtering them in kernel space, even though they use the same algorithm. This result is due to the fact that when network packets are filtered in userspace, they have to be copied across kernel and userspace. This copying of packets generates an overhead which could be seen in the results. In this experiment, this overhead consumes 54% $((651 - 296)/651)$ of the bandwidth provided by fsa_kernel.

4.2.2 Experiment 2

In this experiment, we make use of three computers. Hardware specifications of the computers are the same as the ones in the first experiment. One computer is used as the iperf client and another one is used as the iperf server as in the previous experiment. However these two computers are not connected to each other through the same network. The third computer is connected to 2 different networks such that each network is composed of the the third computer and the server or the client computer. The third computer has packet forwarding enabled and acts like a router between the other 2 computers. Each network packet that is sent to the server computer from the client computer passes through the third computer, and vice versa. Each network uses a 1 Gigabit ethernet switch.

In this experiment, we filter the network packets in the router computer using the FORWARD chain of iptables. We measure the network bandwidth between the client and the server computer as in the previous experiment. The results of the second experiment can be seen in Figure 4.4 and Table 4.2.

The results of this experiment are the same as the previous experiment. When more patterns

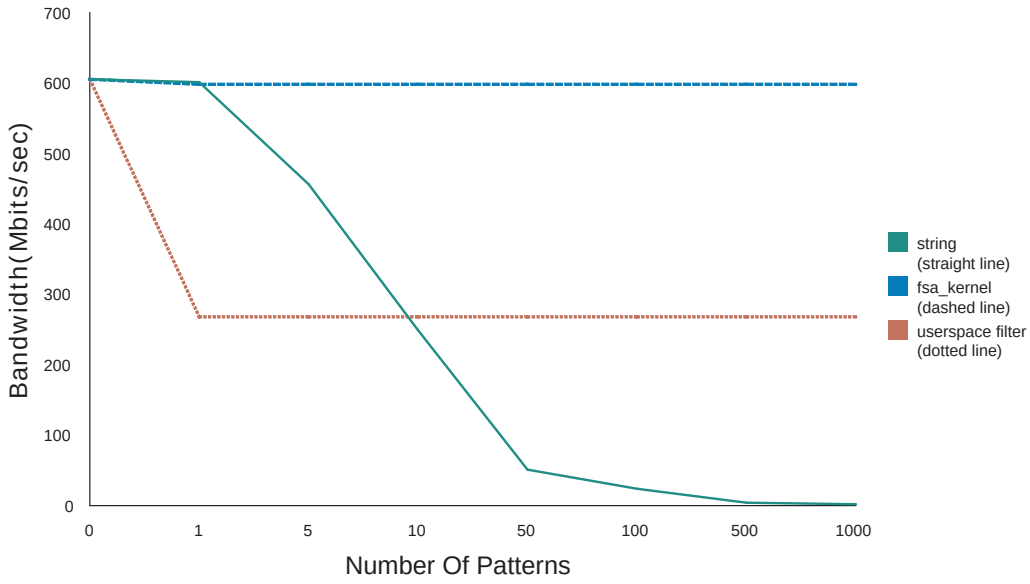


Figure 4.4: Experiment 2 Results

Table 4.2: Experiment 2 Results

	Bandwidth (Mbits/sec)		
number of patterns	string	fsa_kernel	userspace filter
0	605	605	605
1	601	598	268
5	456	598	268
10	249	598	268
50	51.4	598	268
100	24.5	598	268
500	4.28	598	268
1000	2.35	598	268

are added to be filtered, string extension causes the bandwidth to drop rapidly, whereas fsa_kernel and userspace filter consume the same amount of bandwidth regardless of the number of string patterns and filtering packets in userspace reduces more bandwidth. The only difference in this experiment is the bandwidth value when network packets are not filtered. The bandwidth drops to 605 Mbits/sec from 923 Mbits/sec. This stems from the fact that the second ethernet card that we use provides a maximum bandwidth value of 607 Mbits/sec.

4.2.3 Experiment 3

This experiment measures the bandwidth of the traffic between two computers like the first experiment. We make use of a cluster machine with an infiniband network connection instead

Table 4.3: Experiment 3 Results

	Bandwidth (Mbits/sec)	
number of patterns	string	fsa_kernel
0	8290	8290
1	1980	1860
5	461	1860
10	236	1860
50	48.8	1860
100	25.5	1860
500	6.41	1860
1000	3.94	1860

of using 2 computers connected through a 1 Gigabit ethernet switch.The cluster is comprised of 46 computational nodes. Each node has 2 quad-core CPUs and 16 GB memory. All the nodes are connected through an infiniband switch. Each CPU is an Intel Xeon E5430 Quad-Core CPU (2.66 GHz, 12 MP L2 Cache, 1333 MHz FSB). One computational node of the cluster is used as the server machine and another one as the client machine. The results of this experiment are depicted in Figure 4.5 and Table 4.3.

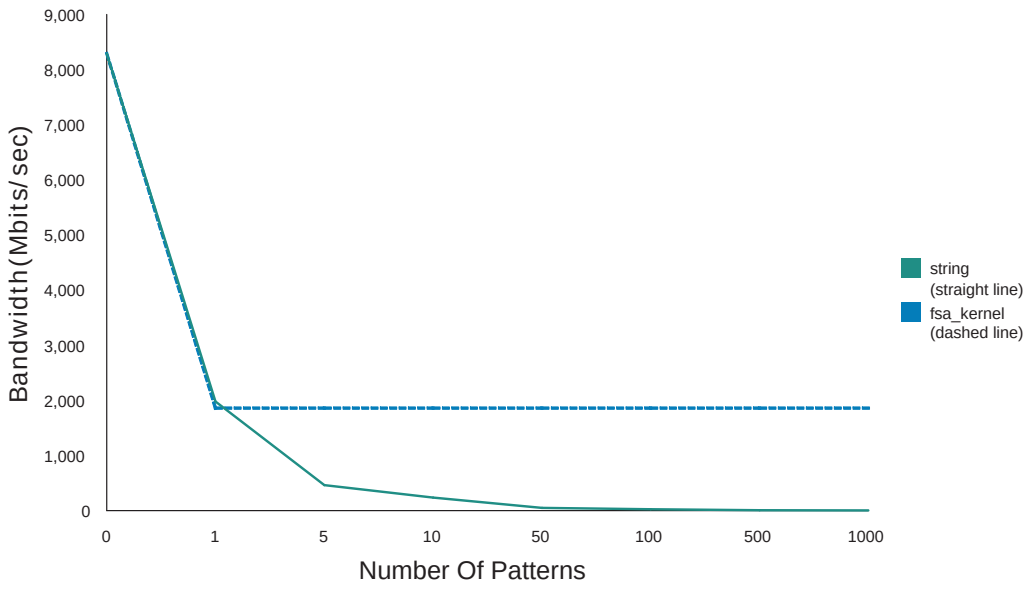


Figure 4.5: Experiment 3 Results

We made this experiment to demonstrate that we could filter network packets by scanning their payloads in multi-gigabit rates. The results of the experiment confirm our predictions. We are able to filter network packets with a constant bandwidth rate of 1860 Mbits/sec regardless of the number of pattern strings that are being filtered. The results of the userspace filter lacks

in this experiment, because of incompatibility of kernel and library versions in the cluster machine. A negative result of the experiment is that filtering network traffic by scanning packet payloads using either iptables string extension or our own architecture consumes a high portion of the bandwidth. Our architecture drops the bandwidth from 8290 Mbits/sec to 1860 Mbits/sec (22.4%), on the other hand when filtering network traffic using a single pattern and iptables string extension it drops to 1980 Mbits/sec (23.8%). These results are due to the fact that Linux operating system cannot utilize multiple processors to increase network bandwidth [13]. All the interrupts created by the same network interface card is processed by the same CPU [13], this causes all the network traffic to be handled by the same CPU even if there are other idle CPUs. We expect our results to be better in a future Linux kernel which can utilize SMP (symmetric multiprocessing) systems to process network traffic.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we have presented an architecture for content based packet filtering using Deterministic Finite Automata, implemented on top of iptables framework. Our architecture can filter network packets in Linux kernel based on their payloads by matching them against several string patterns at once. We have shown that our architecture works more efficiently then the existing payload filtering engine in iptables, namely string extension, owing to Aho-Corasick algorithm. Additionally, our tests confirmed that packet filtering in kernel space is more efficient than performing it in userspace. In one of our experiments, we have shown that we can filter network packets against 1000 string patterns at a constant bandwidth rate of 651 Mbits/sec on a 1 Gbit network. This bandwidth rate drops to 296 Mbits/sec from 651 Mbits/sec (45.4%) if we filter network packets in userspace instead of filtering them in kernel space. Moreover our architecture to filter network packets based on their payloads reduces system bandwidth by a constant amount, regardless of the number of string patterns that we use to match the network packets. We have also shown that our payload packet filtering architecture could work in multi-gigabit bandwidth rates if it is run on on a network with infiniband connection. In our experiments, we have reached a maximum bandwidth value of 1980 Mbits/sec while filtering network traffic against 1000 string patterns.

For future work, our architecture could be extended with several improvements and functionalities.

- We could assess the efficiency of our packet filtering architecture in a multi-core system which can utilize symmetric multiprocessing.
- Our signatures for packet filtering consists of string patterns that matches only the payload portions of network packets. We could add support to our architecture for signa-

tures that are able to match header fields of network packets, along with the payload portion.

- We could add range information to our signatures so that our architecture scans only a predefined portion of a network packet against that signature, instead of scanning the complete payload.
- We could add support to our architecture for processing signatures with regular expressions instead of just plain strings.
- Another improvement could be to extend the functionalities of the architecture such that it does not perform only packet filtering, but also other networking tasks that could be done in the kernel space, such as packet classification. While implementing packet classification we could make use of user defined chains such that the rules for packet filtering are added to a predefined user defined chain, and the rules for packet classification are added to another one.

REFERENCES

[1] EdenWall Security Appliance, http://www.edenwall.com/en/products/edenwall-security-appliances, 24 September 2011.

[2] fwsnort: Application Layer IDS/IPS with iptables, http://cipherdyne.org/fwsnort/, 24 September 2011.

[3] iperf, http://sourceforge.net/projects/iperf/, 24 September 2011.

[4] Netfilter, http://www.netfilter.org/, 24 September 2011.

[5] The netfilter.org libnetfilter-queue project, http://www.netfilter.org/projects/libnetfilter-queue/index.html, 24 September 2011.

[6] Nfr Security, http://www.checkpoint.com/corporate/nfr/index.html, 24 September 2011.

[7] opendpi, http://code.google.com/p/opendpi/, 24 September 2011.

[8] OpenFst Library, http://www.openfst.org/, 24 September 2011.

[9] Pace: Network analysis with layer-7 deep packet inspection, http://www.ipoque.com/en/products/pace-network-analysis-with-layer-7-deep-packet-inspection, 24 September 2011.

[10] What Is Deep Inspection, http://www.ranum.com/security/computer-security/editorials/deepinspect/, 24 September 2011.

[11] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[12] Oskar Andreasson. Iptables Tutorial 1.2.2, http://www.frozentux.net/iptables-tutorial/iptables-tutorial.html, 24 September 2011.

[13] Annie Foong, Jason Fung, and Don Newell. Improved Linux* SMP Scaling: User-directed Processor Affinity. *Intel*, October 2008.

[14] Florin Baboescu and George Varghese. Scalable packet classification. *SIGCOMM Comput. Commun. Rev.*, 31:199–210, August 2001.

[15] Roni Bar-Yanai, Michael Langberg, David Peleg, and Liam Roditty. Realtime classification for encrypted traffic. In *SEA'10*, pages 373–385, 2010.

[16] Andrew Begel, Steven Mccanne, and Susan L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *In SIGCOMM*, pages 123–134, 1999.

[17] Herbert Bos, Willem De Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. Ffpf: Fairly fast packet filters. In *In Proceedings of OSDI 04*, pages 347–363, 2004.

[18] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20:762–772, October 1977.

[19] Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high speed networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Richard Cole. Tight bounds on the complexity of the boyer-moore string matching algorithm. In *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, SODA '91, pages 224–233, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[21] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, London, UK, 1979. Springer-Verlag.

[22] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[23] Deep packet inspection, http://en.wikipedia.org/wiki/deep_packet_inspection, 24 September 2011.

[24] Patrick Th. Eugster. Kernel korner: Linux socket filter: sniffing bytes over the network. *Linux J.*, June 2001.

[25] Firewall(computing), http://en.wikipedia.org/wiki/Firewall_(computing), 24 September 2011.

[26] Mike Fisk and George Varghese. Fast content-based packet handling for intrusion detection. Technical report, 2001.

[27] Mike Fisk and George Varghese. Applying fast string matching to intrusion detection. Technical report, Los Alamos National Laboratory, University of California San Diego, 2004.

[28] Masanobu Yuhara Fujitsu, Masanobu Yuhara, Brian N. Bershad, Chris Maeda, J. Eliot, and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *In Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, 1994.

[29] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[30] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[31] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[32] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 111–, Washington, DC, USA, 2002. IEEE Computer Society.

[33] Intrusion detection system, http://en.wikipedia.org/wiki/intrusion_detection_systems, 24 September 2011.

[34] J. Ioannidis, K. G. Anagnostakis, and A. D. Keromytis. xpf: Packet filtering for low-cost network monitoring, 2002.

[35] Sun Kim and Yanggon Kim. A fast multiple string-pattern matching algorithm. In *In Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

[36] Donald E. Knuth and Vaughan R. Pratt. Fast pattern matching in strings. *Siam Journal on Computing*, 6:323–350, 1977.

[37] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.*, 36:339–350, August 2006.

[38] Po-Ching Lin, Ying-Dar Lin, Tsern-Huei Lee, and Yuan-Cheng Lai. Using string matching for deep packet inspection. *Computer*, 41(4):23 –28, April 2008.

[39] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[40] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. *SIGOPS Oper. Syst. Rev.*, 21:39–51, November 1987.

[41] Mehryar Mohri. Minimization algorithms for sequential transducers. *Theor. Comput. Sci.*, 234:177–201, March 2000.

[42] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 31–, Washington, DC, USA, 2003. IEEE Computer Society.

[43] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[44] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theor. Comput. Sci.*, 92:181–189, January 1992.

[45] Martin Roesch and Stanford Telecommunications. Snort - lightweight intrusion detection for networks. pages 229–238, 1999.

[46] P. Rolando, R. Sisto, and F. Risso. Spaf: Stateless fsa-based packet filters. *Networking, IEEE/ACM Transactions on*, 19(1):14 –27, February 2011.

[47] Rusty Russell. Linux 2.4 Packet Filtering HOWTO, http://www.netfilter.org/documentation/howto/packet-filtering-howto.html, 24 September 2011.

[48] Rusty Russell and Harald Welte. Linux netfilter Hacking HOWTO, http://www.netfilter.org/documentation/howto/netfilter-hacking-howto.html, 24 September 2011.

[49] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *in IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[50] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 262–271, New York, NY, USA, 2003. ACM.

[51] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. *SIGARCH Comput. Archit. News*, 33:112–122, May 2005.

[52] Thomas Heinz. *High Performance Packet Classification for Netfilter*. PhD thesis, Fachbereich Informatik der Universitat des Saarlandes, 28 February 2004.

[53] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *In IEEE Infocom, Hong Kong*, pages 333–340, 2004.

[54] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical report, 1994.

[55] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: a fast dynamic packet filter. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 279–292, Berkeley, CA, USA, 2008. USENIX Association.

[56] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 93–102, New York, NY, USA, 2006. ACM.

APPENDIX A

KERNEL MODULE CODE OF AN IPTABLES MATCH EXTENSION

Here is the skeleton code of the kernel module of a sample iptables match extension named foo that uses the shared struct xt_foo_info which resides in linux/netfilter/xt_foo.h.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/netfilter/x_tables.h>
#include <linux/netfilter/xt_foo.h>

MODULE_AUTHOR("Foo Author <foo@author.net>");
MODULE_DESCRIPTION("Foo match extension for iptables");
MODULE_LICENSE("GPL");
MODULE_ALIAS("ipt_foo");
MODULE_ALIAS("ip6t_foo");

// called when a packet arrives to the hook this match is attached at
static bool
foo_mt(const struct sk_buff *skb, struct xt_action_param *par)
{
    // get the shared struct from the parameter par
    const struct xt_foo_info *conf = par->matchinfo;

    // try to match the network packet in the buffer skb
```

```
    if ( network packet matches )
        return true;
    else
        return false;
}

// called when a new rule containing this match is added to iptables
static int foo_mt_check(const struct xt_mtchk_param *par)
{
    // get the shared struct from the parameter par
    struct xt_foo_info *conf = par->matchinfo;

    // check if the match module compiles with the hook,
    // chain and the table it is attacted to

    // allocate dynamic structures if necessary

    return 0;
}

// called when the rule containing this match is erased
static void foo_mt_destroy(const struct xt_mtdtor_param *par)
{
    // destroy any initialized data by foo_mt_check
}

// xt_match structure
static struct xt_match xt_foo_mt_reg = {
    .name       = "foo",
    .revision   = 1,
    .family     = NFPROTO_UNSPEC,
    .checkentry = foo_mt_check,
    .match      = foo_mt,
    .destroy    = foo_mt_destroy,
    .matchsize  = sizeof(struct xt_foo_info),
    .me         = THIS_MODULE,
};
```

```
static int __init foo_mt_init(void)
{
    // register match
    return xt_register_match(&xt_foo_mt_reg);
}

static void __exit foo_mt_exit(void)
{
    // unregister match
    xt_unregister_match(&xt_foo_mt_reg);
}

module_init(foo_mt_init);
module_exit(foo_mt_exit);
```

APPENDIX B

USERSPACE MODULE CODE OF AN IPTABLES MATCH EXTENSION

Here is the skeleton code of the userspace administration module of a sample iptables match extension named foo that uses the shared struct xt_foo_info which resides in linux/netfilter/xt_foo.h.

```
#include <xtables.h>
#include <linux/netfilter/xt_foo.h>

static void foo_help(void)
{
    printf(
       "foo match options:\n"
       "--option1          Foo Option 1\n"
    );
}

static const struct option foo_opts[] = {
    { "option1", 1, NULL, '1' },
    { .name = NULL }
};

// called when a rule using this match is added to iptables
static void foo_init(struct xt_entry_match *m)
{
    // init the match
```

```
    // get the shared struct
    struct xt_foo_info *fooinfo =
        (struct xt_foo_info *)(*m)->data;
    // fill in the shared struct
}

// called after foo_init
static int
foo_parse(int c, char **argv, int invert, unsigned int *flags,
              const void *entry, struct xt_entry_match **match)
{
    // parse foo options
    return 1;
}

// called after foo_parse
static void foo_check(unsigned int flags)
{
    // check if the obligatory options are specified
}

static void
foo_print(const void *ip, const struct xt_entry_match *match, int numeric)
{
    // print information about the match
}

static void foo_save(const void *ip, const struct xt_entry_match *match)
{
    // print the used options
}


// xtables_match structure
static struct xtables_match foo_mt_reg = {
    .name          = "foo",
    .revision      = 1,
```

```
    .version       = XTABLES_VERSION,
    .size          = sizeof(struct xt_foo_info),
    .help          = foo_help,
    .init          = foo_init,
    .parse         = foo_parse,
    .final_check   = foo_check,
    .print         = foo_print,
    .save          = foo_save,
    .extra_opts    = foo_opts,
};

void _init(void)
{
    // register the match
    xtables_register_match(foo_mt_reg);
}
```

APPENDIX C

AN EXAMPLE APPLICATION USING LIBNETFILTER_QUEUE

The following code is taken from
http://svn.netfilter.org/netfilter/trunk/libnetfilter_queue/utils/nfqnl_test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <linux/types.h>
#include <linux/netfilter.h>/* for NF_ACCEPT */

#include <libnetfilter_queue/libnetfilter_queue.h>

/* returns packet id */
static u_int32_t print_pkt (struct nfq_data *tb)
{
    int id = 0;
    struct nfqnl_msg_packet_hdr *ph;
    u_int32_t mark,ifi;
    int ret;
    char *data;

    ph = nfq_get_msg_packet_hdr(tb);
    if (ph){
        id = ntohl(ph->packet_id);
```

```
        printf("hw_protocol=0x%04x hook=%u id=%u ",
        ntohs(ph->hw_protocol), ph->hook, id);
    }

    mark = nfq_get_nfmark(tb);
    if (mark)
        printf("mark=%u ", mark);

    ifi = nfq_get_indev(tb);
    if (ifi)
        printf("indev=%u ", ifi);

    ifi = nfq_get_outdev(tb);
    if (ifi)
        printf("outdev=%u ", ifi);

    ret = nfq_get_payload(tb, &data);
    if (ret >= 0)
        printf("payload_len=%d ", ret);

    fputc('\n', stdout);

    return id;
}


static int cb(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
      struct nfq_data *nfa, void *data)
{
    u_int32_t id = print_pkt(nfa);
    printf("entering callback\n");
    return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
}

int main(int argc, char **argv)
{
    struct nfq_handle *h;
```

```
struct nfq_q_handle *qh;
struct nfnl_handle *nh;
int fd;
int rv;
char buf[4096];

printf("opening library handle\n");
h = nfq_open();
if (!h) {
    fprintf(stderr, "error during nfq_open()\n");
    exit(1);
}

printf("unbinding existing nf_queue handler for AF_INET (if any)\n");
if (nfq_unbind_pf(h, AF_INET) < 0) {
    fprintf(stderr, "error during nfq_unbind_pf()\n");
    exit(1);
}

printf("binding nfnetlink_queue as nf_queue handler for AF_INET\n");
if (nfq_bind_pf(h, AF_INET) < 0) {
    fprintf(stderr, "error during nfq_bind_pf()\n");
    exit(1);
}

printf("binding this socket to queue '0'\n");
qh = nfq_create_queue(h,  0, &cb, NULL);
if (!qh) {
    fprintf(stderr, "error during nfq_create_queue()\n");
    exit(1);
}

printf("setting copy_packet mode\n");
if (nfq_set_mode(qh, NFQNL_COPY_PACKET, 0xffff) < 0) {
    fprintf(stderr, "can't set packet_copy mode\n");
    exit(1);
}
```

```
    nh = nfq_nfnlh(h);
    fd = nfnl_fd(nh);

    while ((rv = recv(fd, buf, sizeof(buf), 0)) && rv >= 0) {
        printf("pkt received\n");
        nfq_handle_packet(h, buf, rv);
    }

    printf("unbinding from queue 0\n");
    nfq_destroy_queue(qh);

#ifdef INSANE
    /* normally, applications SHOULD NOT issue this command, since
     * it detaches other programs/sockets from AF_INET, too ! */
    printf("unbinding from AF_INET\n");
    nfq_unbind_pf(h, AF_INET);
#endif

    printf("closing library handle\n");
    nfq_close(h);

    exit(0);
}
```