Performance analysis of the Linux firewall in a host

A Thesis
Presented to the Faculty of
California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Electrical Engineering

By
Américo J. Melara
June 2002

Authorization for Reproduction of Master’s Thesis

I hereby grant permission for the reproduction of this thesis, in whole or in part, without further authorization from me.

____

Signature (Américo J. Melara)

____

Date

Approval Page

Title: Performance analysis of the Linux firewall in a host

Author: Américo J. Melara

Date Submitted: June 12, 2002

Dr. James Harris ____________________
Advisor Signature

Dr. Hugh Smith ____________________
Committee Member Signature

Dr. Phillip Nico ____________________
Committee Member Signature

Dr. Fred DePiero ____________________
Committee Member Signature

Abstract

Performance analysis of the Linux firewall in a host

Firewalls are one of the most commonly used security systems to protect networks and hosts. Most researchers have focused on analyzing the latency and throughput of router firewalls. Different from this approach, this research focuses on studying the performance impact and the sensitivity of the Linux firewall (`iptables`) for a single host.

In order to be able to measure the performance and the sensitivity of the firewall, we designed and instrumented each layer of the Linux TCP/IP stack. This instrumentation was used to test the host's firewall under two scenarios: In the first scenario, we captured the path and the latency of one single packet; in the second scenario, we captured the latency of multiple packets sent to the host at various transmission rates.

Our measurement results indicate that the firewall is sensitive to the number of rules, the type of filtering, and the transmission rate. The results of our first scenario demonstrate that for each type of filtering, latency increases linearly as the number of rules increase. Furthermore, the second test scenario shows that latency decreases as the packet transmission rate increases.

Our results show that the percentage overhead generated by a firewall when a single packet of 64 bytes of payload travels the TCP/IP stack, for a rule-set of zero and 100 rules, ranges from 6% to up to 75%, respectively.

Acknowledgments

First of all I would like to thank God for the opportunity to be able to come to Cal Poly and finish up my two master's degrees. I want to thank my family for their financial support and unlimited amount of encouragement. I want to thank 3Com for their financial support and commitment to the project. I want to give special thanks to Dr. Jim Harris for giving me the opportunity join the project for even when I did not have the background, he gave me the chance, gave me continuous direction, and encouraged me. I want to thank Dr. Hugh Smith and Dr. Phillip Nico for the unlimited amount of questions and solutions which helped me to set a path for gaining a better understanding of my work, and led to many of the things written in this document. Thanks to James Fischer ("Jim") for his willingness and dedication to helping not only myself but also everyone in the group. To Peter Huang for encouraging me to join the project and helping me with my programming and understanding Linux. To Max Roth, for the countless hours of hearing him sing in the lab; hours that resulted in a huge amount of pain and perturbing me (and "everyone" I may say) from getting any work done! But many thanks for his encouragement, support, discussions and comments about the research, theology, society, and a must add: some "slashdot" useless stuff! Thanks to Jason Hatashita for the useful discussions and comments. To Heather Heiman, Jared Kwek, and Jenny Huang for their friendship and encouragement. And finally, thanks to the rest of the team for their dedication to the project.

Tables of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Network Security is one of the most important fields dealing with the Internet. The ability to access and transfer information in a few seconds allows the government, companies, educational institutions, and individuals to accelerate the decision process or simply be "informed." However, information can be very valuable and there is a need for better and faster security systems to protect information and networks.

Attacks are prevalent on the Internet, and for that reason Firewalls, Intrusion Detection Systems, Virus Scanners, File Protection and Integrity checks software, Buffer overflow protection techniques, and Encryption tools have been developed as security services to protect systems and information. The CiNIC is an independent network device designed to control all of the networking services for the host. Thereby, it can serve as the front line defense mechanism against attacks. The vision for it is to run security services such as the firewall, encryption, authentication, intrusion detection, and other services to secure the host.

Firewalls are the first front line defense mechanism against intruders. There are two different goals for testing them. The first goal is to analyze and test the firewall policies, in other words, to model and test how secure a firewall is in a "real-world" environment. The second goal is to test the performance impact generated by the firewall. Given that our first step to increase the security functions of the CiNIC is to port the firewall to it, we decided to analyze the performance cost of having a firewall in the host. After searching for conference papers that addressed the firewall performance on single hosts, we found that very little research had been done on the topic. In our end-of-the-year meeting with 3Com in December 2001, we were told that a third party vendor discovered that the firewall would degrade the performance of a system tremendously after 30 rules. In view of the lack of research and the uncertainty on what the firewall performance cost might be, we decided to study the performance of the Linux firewall `iptables`. This thesis presents a study of the sensitivity and the performance impact produced by the Linux firewall `iptables` in a host.

We decided to test the performance of the firewall under two scenarios changing various parameters. The first scenario included tracing one single packet in order to measure the sensitivity of the firewall to:

(1) The INPUT policy

(2) The number of rules

(3) The type of filtering

(4) The payload size

(5) The transmission protocol.

The second scenario included a series of tests varying the throughput by sending a stream of packets at 5 and 10 Mbps. The first test results documented in this thesis will show that the performance is only sensitive to the number of rules and the type of filtering. The measurement results obtained in the throughput tests will confirm that the single-packet test measurements are valid, and that may serve as conservative estimates for finding the performance impact generated by the firewall.

The remainder of this document is organized as follows: Chapter 2 gives an overview on the most recent types of security attacks (e.g. denial of service attacks, buffer overflows, operating systems protection) as well as an overview on some security mechanisms (e.g. Intrusion Detection, Authentication, Firewalls). It also explains Firewalls in more detail. Chapter 3 covers the receiving operation of the Linux TCP/IP stack, the `iptables` algorithm, the instrumentation technique used to measure a packet's process throughout the stack, and presents the parameters under test that will be used to determine the sensitivity of the firewall performance. Chapter 4 presents the analysis of the performance of a single packet to determine the sensitivity of the firewall to the parameters mentioned earlier. It also presents the performance of the firewall as a function of the rate of incoming packets. The results are compared in terms of the performance of the host when it runs with and without the firewall. Finally, the summary of the results and future work are presented in Chapter 5.

Chapter 2 Overview of Security

For individuals and enterprises the main purpose of security on a network system is the protection of information. We all use a network system one way or the other, either for sending e-mails, reading the news, making traveling plans, or shopping. In most of our transactions we wish to have one thing – protection of our information. But, what type of protection? Protection from whom? For large firms security includes not only the protection of the company's information from outsiders but also the protection of their entire internal network. For example, top executives do not want their competitors to know their marketing or acquisitions strategies. Nor does a manager want intruders to read, delete, or acquire budgeting information or consumer's information. Thus, the goal of security is to protect information and systems from "malicious intruders."

This chapter's intent is to organize and summarize the area of Security, and explain firewalls in more detail. The chapter is laid out as follows: First, the most relevant vulnerabilities for a host are explained briefly. Following, some of the techniques and tools used to prevent and detect attacks are presented. Finally, the idea of porting a firewall to the CiNIC architecture is presented.

2.1 Network Vulnerabilities – an OSI perspective

Security can be implemented throughout each layer of the network. Using the TCP/IP model we can show how every layer is vulnerable to security breaches and what software is used to protect the systems. The reader should be aware, however, that in spite of the number of security software he may buy for your system it does not make it immune or does not take away the chance of getting a virus, a Trojan horse, or just "get hacked." So, the more knowledgeable the reader is about how someone can break into his system, the more cautious he will be, and the harder it will be to break in.

Figure 2-1 shows the end system's TCP/IP stack model and some of the tools created to provide security for each layer. The figure can be explained as follows. Secure services are available at the *Operating System level* (OS) and at *User space level*. The transport layer is the point where the *OS* and *user space* separate. At the OS level we have a firewall (e.g. `iptables`), intrusion detection systems (e.g. Linux Intrusion Detection System a.k.a. `LIDS`), IP Security or `IpSec`, and Denial of Service prevention. One of the goals of the CiNIC project is to offload these security services from the host to the co-host. For that reason, this research will focus on the firewall for the Linux Operating System.

At the *user level* we have secure standard protocols that use cryptography to secure the transmission of data, such as Secure Socket Layer (SSL) and Transport Secure Layer (TLS) [1][2], which are discussed later in this chapter. Another service used to provide encryption of data is the Pretty Good Privacy (PGP) protocol; this protocol uses keys to encrypt the data sent through e-mails [3]. In addition, virus scanners (e.g. Norton and McAffee) and file integrity software [4] (e.g. Tripwire) are tools commonly used to protect end systems from viruses, worms, and Trojan horses.

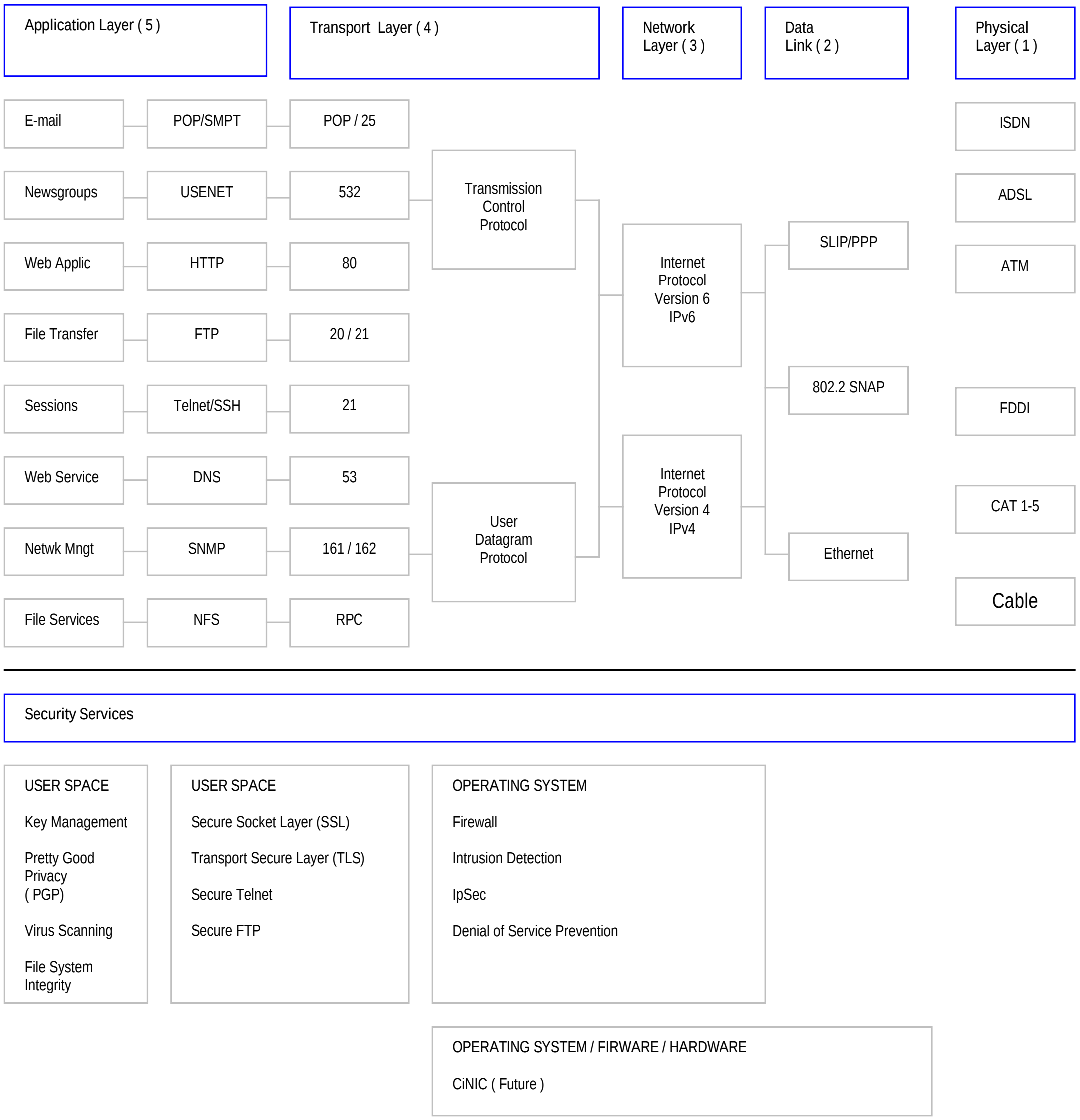
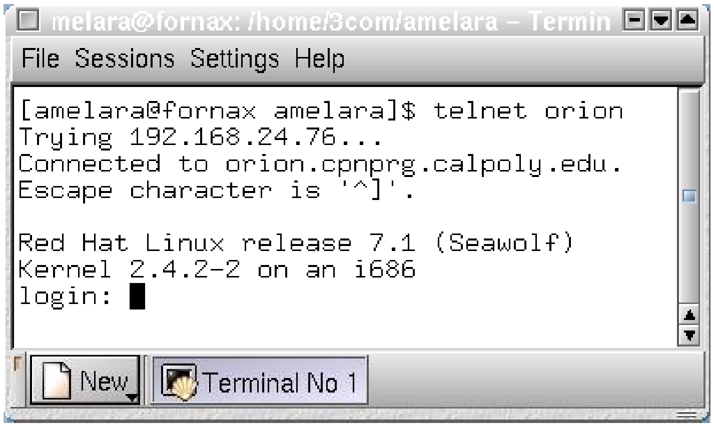


Figure 2-1 Relationship between the TCP/IP Reference Model and various Operating System and User level Security Services

A security issue implied in the network stack is that a weakness or "hole" in one layer could lead to the exploitation of a lower layer, and vice-versa. For example, through a simple `telnet` session anyone can find out the type of operating system running on a particular machine. In Figure 2-2 the host `fornax` has requested a telnet session with the host orion. When the session starts, we observe that the host orion is running Red- Hat Linux 7.1 with a kernel version 2.4.2-2. This may lead the attacker to look for software tools designed to attack the specific vulnerabilities found in the kernel 2.4.2-2.


melara@fornax: /home/3com/amelara - Termin
File Sessions Settings Help
New
Terminal No 1

```
[amelara@fornax amelara]$ telnet orion
Trying 192.168.24.76...
Connected to orion.cpnprg.calpoly.edu.
Escape character is '^]'.

Red Hat Linux release 7.1 (Seawolf)
Kernel 2.4.2-2 on an i686
login: 
```

Figure 2-2 Acquiring the server's Operating System type and version through a Telnet Session

On their search for vulnerable systems, attackers use a technique called *footprinting*. *Footprinting* is defined as the *fine art of gathering information!* [5] Information can be gathered through *scanning* or *enumeration.* S*canning* is a tool used to find open ports and services running on a system, *enumeration* is the "ability to extract valid accounts or exported resource names from systems [5]." Some of the information to be gathered include Domain Names, specific IP addresses of systems reachable through the Internet, TCP and UDP services running on each system, system architecture, routing tables, access control mechanisms, related access control lists, etc. The list of tools available to extract this type of information is large, but some of the most common ones are nmap, the ping of death, tcpdump, rpcinfo, Cheops [5].

The network protocol by nature has its pitfalls. For example, an ICMP (Internet Control Message Protocol) packet, which is normally used to communicate control messages on the Internet between hosts and routers, contains diagnostics about your system. For example, a `ping` contains error detection information (e.g. network/host/port), control messages (e.g. source quench, redirect) or some general information (e.g. timestamp, address mask request.)

Computers inside a local area network (LAN) are usually sitting behind a router and firewall but, even then, the network is ***not*** secure. A report from the FBI Computer Crime Unit says that approximately 80% of network security breaches for an Enterprise happen internal to the network [6].

An intruder can have access to an entire network for days and even weeks without being noticed, because the larger the network, the more complicated it is to design policies to secure that network and the more security holes. Subsequently, responsibility to protect a system (e.g. entire network, server, hosts) cannot be left to the network administrator alone. Therefore, there is indeed a need to make better software and hardware tools to provide greater security for the end systems/hosts.

2.1.1 Operating Systems attacks

Our first security checkpoint is the operating system. The operating system controls every single process, entire network operation, and all the hardware and the software; thus, it is the most delicate and the highest priority point of protection for a system. Operating systems are vulnerable to buffer overflows, worms, and viruses.

2.1.1.1 Buffer overflows

At a recent software engineering conference, Richard Pethia from the Carnegie Mellon Software Engineering Institute (CERT), identified buffer overflow attacks as the single most important security problem [7].

In her research, Nicole Decker [8], explains buffer overflows and how they are used to break into systems. Let's look at the following example: Consider a program that reserves a buffer of 1024 bytes. In such a case, the program's maximum allowable input to that buffer is 1024 bytes. If the size of the input data typed in by the user exceeds the size allocated, and if the input is not checked to reject anything larger than 1024 bytes, it is said that the buffer has been overflowed.

Now, recall the function of the instruction pointer. The function pointer stores the memory address of the next command to be executed by a program. It is through the instruction pointer that the computer knows what should and should not be executed - the computer cannot differentiate between data and instructions.

Assume that the next statement, after reading the user's input just mentioned in the example above, is a `printf` statement. The instruction pointer holds the memory address to the `printf` statement. Let's walk through the process: the computer will read the input from the user, store it into the buffer, check the instruction pointer to find what function should execute next (i.e. the `printf` statement), find the memory address of the `printf` statement, retrieve the contents into a input buffer, and finally, print the data input to the screen.

If the program's buffer is overflowed, those extra bytes (usually allocated on a neighboring region to the original buffer) could overwrite the address of the instruction pointer. If we overwrite this address, instead of pointing to the `printf` statement, we can give to the instruction pointer an address to malicious code.

How does that relate to networks? Well, the most simple buffer overflow attack is called *stack smashing* [8]. Here, the attacker sends a stream of modified packets to overflow the buffers so that the return address of the instruction pointer points to their code - in most cases the function to execute is `/bin/sh`. If a program is running with `root` privileges and the buffer is overflowed, the attacker will gain `root` access and have complete control of your system. Programs written in C are particularly susceptible to buffer overflow attacks because most C code allows direct pointer manipulations without any bound checking [9].

Some solutions to buffer overflows have been proposed. Some of them are: StackGuard [10], Software fault isolation (SFI) [11], LCLint [12], an extension of LCLint [13], among others.

2.1.1.2 Worms and Trojan horses

A Trojan horse is an executable program that "contains hidden functions that can exploit the privileges of a user [running the program], with a resulting security threat. A Trojan horse does things that the user's program did not intend [14]." In other words, a Trojan horse is an executable program that modifies an original file by adding extra functions - malicious code - that the original program was not intended to execute.

A worm is a *self-propagating* malicious code [15]. In other words, it is a malicious code that does not require the user to do something to continue its propagation. "Highly automated nature of worms coupled with the relatively widespread nature of the vulnerabilities they exploit, allows a large number of systems to be compromised within a matter of hours. Code Red infected more than 250,000 systems in 9 hours on July 2001 [15]." Trojan horses and worms can have file extensions like "`exe`", "`vbs`", "`com`", "`bat`", "`pif`", "`scr`", "`lnk`", or "`js`."

2.1.2 Denial of Service Attacks

Denial of Service (DoS) attacks, which are one of the most prevalent attacks on the Internet, will force a machine to stop providing services to a legitimate user. "DoS attacks use multiple systems to attack one or more victim systems with the intent of denying services to the victims [15]." The University of California - San Diego, observed 12,805 denial-of-service attacks on over 5,000 distinct Internet hosts belonging to more than 2,000 distinct organizations during a three-week period [16]. There are two types of Denial of Service (DoS) attacks: *Operating Systems* attacks, which exploit the bugs of a specific operating system (e.g. Windows 98/NT/2000, Linux, Solaris); and *networking* attacks, which exploit inherent limitations of networks.

To protect from *operating system* attacks it is important to continuously check on the patches and updates available for your specific operating system. Network attacks, however, are more complicated. These attacks include *ping flood (a.k.a. ICMP flood)* and *smurf* which are outright floods of data to overwhelm the finite capacity of your connection; and also *spoofed unreach/redirect a.k.a. "click"* which trick your computer into thinking there is a network failure and voluntarily breaking the connection [17]. The latest type of network attack is the *distributed denial of* *service attack* in which the attacker controls one or more “masters” which then control several more “zombies” (compromised systems) to attack one victim [18].

2.2 Network Protection Techniques

Most known protection techniques are used to provide authentication, encryption, identify attacks, and block and filter packets.

2.2.1 Authentication and Encryption

2.2.1.1 Kerberos

Designed in the mid-‘80s at the Massachusetts Institute of Technology (MIT), the Kerberos network protocol is designed to provide secure Authentication between one or several parties. Kerberos [19] uses a cryptographic distributed service system. In Figure 2-3 we show the simplest scenario, which involves three parties: a client or *user*, an application server or *verifier*, and an *Authentication Server (AS)*. In order to establish a connection between the client and the server/*verifier*, the client needs to prove to the *verifier* its identity by means of an encrypted key. Neither the *verifier* nor the *client* hold any encrypted keys. Only the AS provides the keys. So the process is the following: (1) the client connects to the AS to obtain a key. (2) The *verifier* obtains a key (*server key*) from the *AS* which will serve to verify the authenticity of the client. (3) After the client and the server have obtained the keys, the client will forward its key to the *verifier*. The latter will decrypt the key and allow the connection to be established.

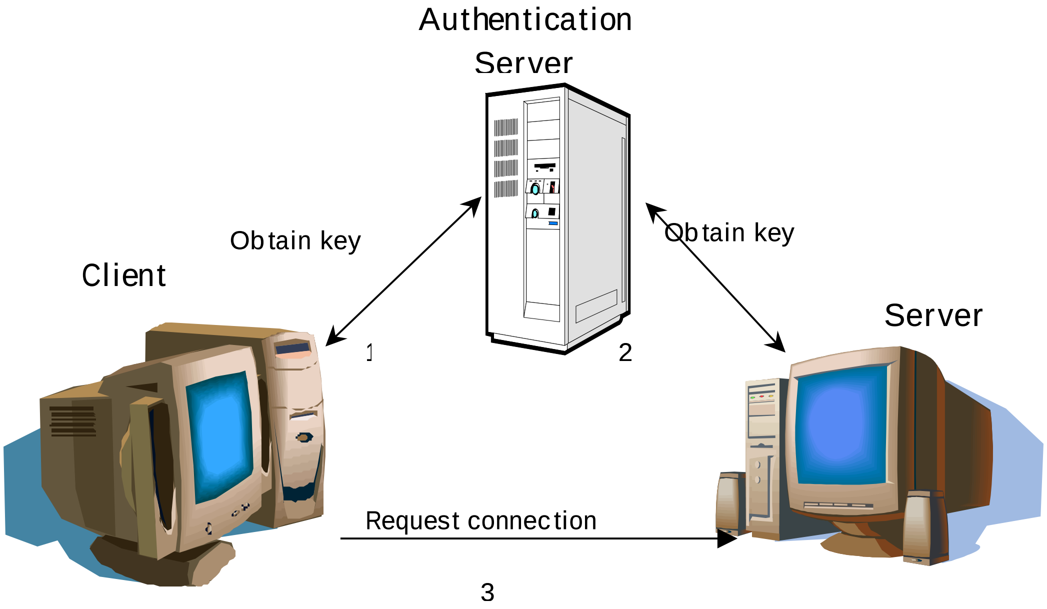


Figure 2-3 Authentication process to establish an encrypted communication between a client and a server using Kerberos

2.2.1.2 SSL/TLS

The Secure Socket Layer [1] (SSL) and Transport Layer Security [2] (TLS) Protocols are security protocols that use cryptography to provide privacy. These protocols provide "integrity between two communicating applications" by means of (1) a private connection – "using data encryption and transaction of keys" and (2) a reliable connection – "the message includes a message integrity check using a keyed MAC." More information, libraries, and software toolkits can be found in the OpenSSL Project [20] website.

The difference between *SSL/TLS* protocols and the *Kerberos* protocol is that the latter needs an Authentication Server to transfer keys while the first two do not.

Therefore, the communication and transaction of keys is only performed between a client and a server, there is no need of a third party to hold keys.

2.2.2 Intrusion Detection

The purpose of Intrusion-Detection Expert Systems (IDES) is to detect suspicious or abnormal use of a system. An IDES works as a system monitor of all the activities performed in the targeted system.

There are two types of detection techniques: *anomaly detection* and *misuse detection.* The former "uses models of the intended behavior or users and applications, interpreting deviations from this 'normal' behavior as a problem [21]." In other words, it keeps an activity log of either the users or the applications used of a system. When it finds an activity different than what is normally used for, it will flag the activity as suspicious.

Misuse detection systems "contain attack descriptions (or 'signatures') and match them against the audit data stream, looking for evidence of known attacks [21]." The intrusion is detected by a "rule-based pattern matching [22]." When a given action is generated, the action is matched against the profiles or the rule-set and the IDES fires an alarm.

2.2.3 Firewalls

Our research focuses on the free available Linux firewall `iptables`. In this section we describe what a firewall is, the ways to implement it, the types, and the architectures.

A *firewall* is the front line defense mechanism against intruders. "It is a system designed to prevent unauthorized access to or from a private network. Firewalls can be implemented in both hardware and software, or a combination of both [23]."

Firewalls can be applied in different ways [24]: *Packet filtering firewalls* are those designed to filter IP addresses, MAC addresses, TCP or UDP ports, and subnets, among others. *Proxy firewall* is a proxy that separates internal networks from the external networks (e.g. the Internet), so that, for outsiders the proxy operates as a server, and for the insiders the proxy operates as the client. A s*tateful-inspection firewall* has the capability of tracking connections and to make decisions based on the dynamic connection state of packets [25]. For example, if an internal client establishes a connection to the Internet through a specific port, the firewall will maintain state information about the connection pertaining to that specific port. Thus, an ICMP packet is checked if it is *related* to that TCP/UDP connection. Any TCP/UDP packets are checked against the state table to find if the packet matches with the *established* port of that connection. An a*pplication firewall* is a software-based firewall (e.g. McAffee personal firewall) in which a user can control (in real-time) to allow or deny connections to it [26]. These different firewall implementations can be used alone or as a combination of several of them.

Firewalls can be implemented in two different architectures [24]: single box (Figure 2-4) and as stand-alone edge device (Figure 2-5). Our research focuses on firewalls in a single-box. The firewall in a s*ingle box* is designed to protect only that single machine. Usually, only outgoing connections are allowed and all incoming connection requests are rejected. On the other hand, a stand-alone edge device can be a *router* or a *dual-homed host*. A *router* is a device that forwards packets between different subnets. A *router firewall* is a router that can filter packets, block ports, maintain stateful-inspection, or do some other type of filtering. A dual-home host is a single computer, with at least two network interface cards, serving the function of a firewall router.

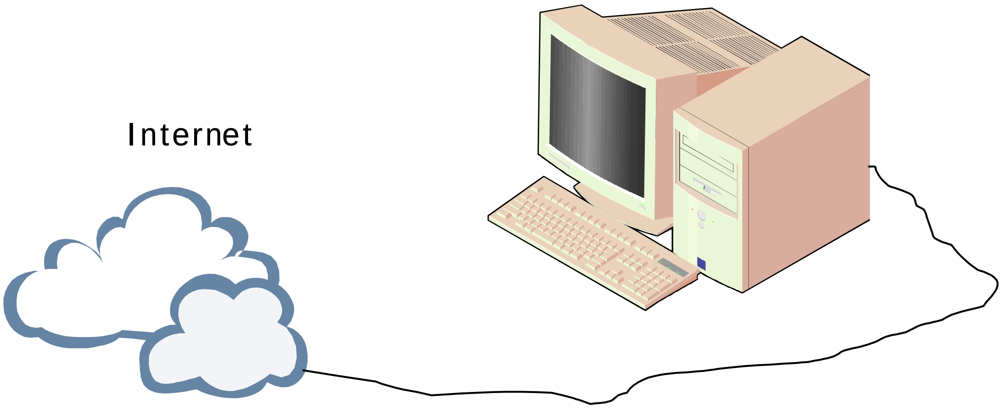


Figure 2-4 A Single-host Firewall protects only one computer

In general, firewalls can be of two types: packet filtering gateways and application proxys. *Packet filtering gateways* look at each packet header entering or leaving the network and accept or reject a particular packet based on specific rules defined by the user/network administrator. Packet filtering is fairly effective and transparent to users. They, however, are difficult to configure and are also susceptible to IP spoofing – a technique used to gain unauthorized access to computers, whereby the intruder sends messages to a computer with an IP address indicating that the message is coming from a trusted host. *Proxy servers*, on the other hand, intercept all the messages entering and leaving the network but it differs in that the proxy hides IP addresses of the clients in the internal network.

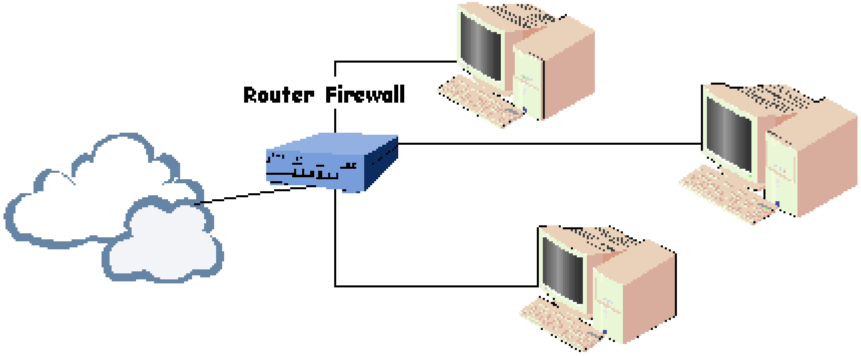


Figure 2-5 Example of a Router Firewall protecting multiple computers inside a network

Firewalls can be commercial or freely available (i.e. open-source such as iptables). But which one is more secure? An expert comments, “Open source follows the ‘many eyes’ principle – the more developers work on the code the fewer secrets and the harder to compromise. Security-by-obscurity argues for hiding the code as a deterrent to breaking the code. Which approach is better is not a simple question [27].” The fact is that a firewall is an extremely important tool that can protect systems from malicious traffic; not having one only means that you want other people to have fun with your systems!

2.2.3.1 Future of firewalls

In 1997, Scuba and Spafford from COAST Labs submitted a paper describing a model or framework for the design of firewalls [28]. According to them, firewalls should provide *authentication* –provide assurance of the integrity of the connecting host or server, *integrity* – “shielding communication traffic from unnoticed and unauthorized modifications such as insertion, replacement or deletion of data,” *access control* – to provide a dynamic mechanism that generates questions about a particular traffic (e.g. IP x.x.x.x wants to establish connection on port 21, do you want to allow this connection?) *Audit* –keeping track of connections/traffic flowing through the firewall, also referred as “log files.”

Some of these functions (e.g. authentication and audit) are built in CISCO’s IOS [27]. However, personal firewalls do not provide authentication but some of them, such as McAffee’s personal firewall [26], provide dynamic access control where the user is notified “on the fly” if a certain IP address desiring to establish a connection should be allowed or not. So, personal or *end-client* firewalls are still under development.

Orman said, “We should look to a future in which every machine is its own firewall [1].” The CiNIC [29] is an independent network device that provides control of all networking services for the host. Thereby, it can serve as the front line defense mechanism against attacks. The vision is that it may not only run a firewall but also provide encryption, authentication, intrusion detection, and other services to secure the host. The Cal Poly Network Performance Research Group is working towards making this future a reality.

Chapter 3 Firewall Performance Study

A firewall is “a device that enforces an access control policy between networks [30].” Firewalls can be used in two ways, as a *stand-alone edge* device that protects and forwards packets to a local area network or as an *operating system component* for protecting a single host. As we will see, researchers have focused on studying the *latency, throughput,* and *total transaction time* of a firewall as a *stand-alone edge device,* but we were unable to find peer-reviewed research papers that specifically addressed the performance of the firewall on a single host. For this reason, our investigation focuses on studying the performance of single-host firewalls, and specifically on the Linux firewall `iptables`.

By the time our research began, we were unable to find documentation that depicted the exact path that a packet follows as it traverses the network stack. Thus, our first efforts focused on capturing this path. Once the path was captured, we performed our first tests. The purpose of the first tests was to understand the firewall sensitivities and the performance impact on the host by varying the transmission speed, payload size, INPUT policy, number of rules, and packet transmission protocol.

For our second test analysis, we sent multiple frames at various transmission rates to a single host. Here, we tried to overload the host using the SmartBits network testing system and measure the host’s latency depending on the packet transmission rate, or throughput. The results obtained from these measurements will confirm if the single-packet test measurements are valid.

3.1 Previous Research

There are two different goals for testing firewalls. The first goal is to analyze and test the firewall policies, in other words, modeling and testing how secure a firewall is in a "real-world" environment. The second goal to test the firewall performance.

3.1.1 Analyzing and Testing Firewall Policies

Most experts would agree that the most difficult part in the design of a firewall is the process of defining the security policy and the configuration of the firewall [31]. The configuration is the process of deploying the policy. To define the policy means to understand the network topology of the LAN, decide what services will be allowed, and who will have access to what information.

Various research papers have presented methods that could serve as a basis for testing firewalls that protect internal networks. Vigna proposes a mathematical model for firewall "field-testing" taking into account the topology and operational environment and not the internal architecture of the firewall [32]. Another method presented is the Firewall ANalysis enGine (*Fang*) [33]. *Fang* is a tool that "reads relevant configuration files, and builds an internal representation of the implied policy and network topology" to simulate spoofing attacks and the behavior of the firewall in response to those attacks. Hazelhurs, Attar, and Sinnapan [34] present a "binary decision diagram" to test the rules of firewalls. All of the above are similar in that they all target to model LANs and not personal firewalls.

Experts may use the mathematical models above or some hacking tools in order to test firewalls. But in reality, there are no standard procedures to test firewalls. Vigna says that the current methodologies to test firewalls are mainly based on expertise and individual skill [35]. The reason behind this is because, in business terms, every customer wants a different specification for the network and for its security (e.g. topology, services running). So, experts use hacking tools such as SATAN, Neus, COPS, Internet Security Scanner (ISS), and BSD Monitor to test if the firewall is secure enough to protect a LAN and satisfy the customer's need.

3.1.2 Testing the Performance of the Firewall

3.1.2.1 Router firewalls

A firewall router reads header information of a packet, checks the header with a number of rules, and decides to forward the packet or not. Various studies have been made on router firewalls. In [27], Patton, Doss, and Yurcik compared the performance of open source versus commercial firewalls. So, they compared the old Linux `ipchains` included in RedHat version 6.0 against CISCO's IOS firewall, the latter consisting of hardware and software. At the time, the older Linux netfilter (`ipchains`) had the disadvantage of lacking functionality; it was not a *stateful* firewall while IOS was.

"The results show that the Linux firewall has consistently higher transaction throughput rates than the Cisco's stateful firewall for rule sets varying from 0 to 200 rules and for packet sizes of 1 and 128 bytes [27]." No specifics were given on the rule set used.

Other studies measured and compared the *latency* and *total transaction time* to download small and large HTTP and FTP files [35]. The tests setup included several clients inside a LAN connecting to a server outside the LAN and a router firewall sitting in between the networks. The firewall would be configured to 7 different policies, one for each HTTP and FTP test. The clients would run a script to establish the connections. The tests for HTTP and FTP were performed independent from one another. For HTTP tests, the clients made connections to download small sizes of data. On the other hand, for FTP tests the client would make small or large number of connections and download files of either 1MB files during one test or 5MB files in another. Those tests were also independent from one another.

The results implied that "the performance difference among security levels due to the overhead of packet filtering for more security is negligible when compared with the outside traffic interface [35]." In other words, performance decreases as the number of connections increase, and is not affected by the security policy. Unfortunately, no specifics were given on the rule-set.

Other tests, such as [36] [37] and [38], have been performed to compare commercial router firewalls, but the results are not presented in this document because they are out of the scope of our research.

3.1.2.2 Single-host firewalls

Different from edge firewall routers, there has not been much research done to analyze the performance or the processing overhead produced by single-host firewalls. One paper presented the results on the throughput and CPU utilization of two machines connected through a 10Mb hub [39]. The purpose of the tests was to measure `iptables` on a single-host. The CPU utilization was measured using "vmstat 3." The sending box sent a byte stream of 187,000,000 bytes. The payload size per packet was 3,700 bytes. The throughput was measured by dividing the size of the bit stream by the time (in seconds) to receive the stream. Finally, the input policies (i.e. INPUT/ OUTPUT/ FORWARD) were set to ACCEPT. The results of the four tests are described below.

On the first test, without a firewall and with one single connection, the throughput was 9.09 Mbits/sec. The CPU utilization was not provided. Another test running "real-world" `iptables` rules and one single connection showed a 9.10 Mbps and a CPU utilization of 19-23% on the sender and 16-20% on the receiver. Another test included establishing five TCP connections and no rules, in order to measure the CPU impact by TCP/IP traffic. The sum of the throughput was 9.13 Mbps, and the CPU utilization varied from 19-20% on the sender and 15-18% in the receiver.

For the last test, the intention was to "measure real-world stress on the iptables rule-set. Five connections were used: two open TCP ports, a TCP port rejected with a TCP reset, a closed TCP port, and an open UDP port." The CPU utilization on the receiver was 15-20% and 23-30% on the sender. For the UDP component the throughput yielded 10.57 Mbps. For the two non-blocked TCP connections the throughput yielded 8.14 Mbps. For TCP in the latter test, it is understandable that as the amount of filtering and connections increase the throughput might decrease. However, for UDP, having a 10.57Mbps throughput on a 10 Mbps hub is suspect.

3.2 Terminology

As we have seen, firewall performance has been studied in terms of the latency, throughput, and total transaction time. In those tests the parameters used have been the number of rules, the number of connections, and the number of bytes per packet or per file, the type of download (e.g. HTTP and FTP).

Some of the researchers have used the firewall-benchmarking terminology defined in the RFC 2647 [30]. However, we had to redefine some terms to make them applicable to our investigation.

Earlier studies made by the Cal Poly Network Project Research Group (CPNPRG) on the performance of the Linux and Windows [29] have been made on the *sending* operation, that is, a study of the latency and throughput when a packet is sent from the application layer until the data is sent out to the wire. Our study focuses on the *receiving* operation, and specifically, in studying of the performance impact produced by a firewall when a packet is traveling up the stack.

3.2.1 Performance Metrics

The first portion of our study focuses on finding the *start latency*. Now, *latency* is the period of time that a packet takes to be transmitted from one end (e.g. a host) to another. *Protocol latency* as the period of time that a network sub-layer holds a payload before it forwards it to the next sub-layer [40] and is divided into: start and stop latency. See Figure 3-1 shows a picture of a latency model.

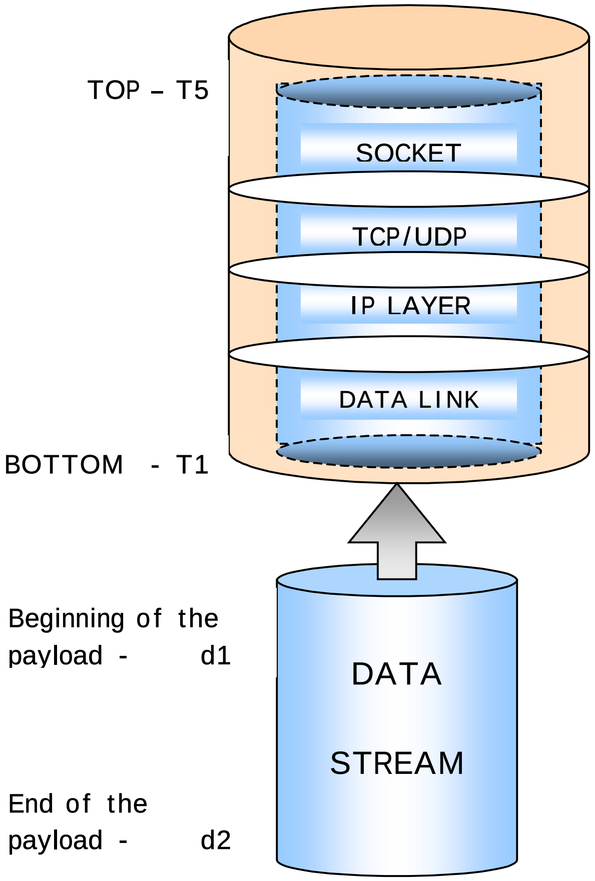


Figure 3-1 Example of a packet in order to measure its latency to traverse the stack

Start latency = T5 – T1 of each packet [seconds]
Stop latency = T5 of the last packet – T1 of the first packet [seconds]
Payload throughput = (payload size) / (stop latency) [bps]

From Figure 3-1 the *start latency* can be defined as the period of time the *beginning of the packet's payload* (d1) to reach the bottom of the stack (T1) until the *beginning of the payload* (d1) reaches the top of the stack (T5). Start latency can be used to determine the efficiency of the Device Under Test (DUT) because it provides "per payload" processing information. *Stop latency* is the amount of time that it takes for the *beginning of the payload* to pass from the bottom of the stack until the *last-bit* *of the payload* reaches the top of the stack. The *stop latency* can be affected by packets that are dropped by either the network or by the host's TCP/IP stack because, if a packet is dropped, the *stop latency* will include the start latency plus the time that it takes for TCP to ask for retransmission and the packet to be retransmitted.

Start and *stop latency* are equal to each other when the payload is less than or equal to the maximum transfer unit (MTU) minus the Ethernet headers. *Firewall overhead* in the *protocol latency,* or just *overhead,* is the impact in the processing time caused by the firewall as it processes every packet header.

The following terms will also be used in this document: *Packet,* used interchangeably with *Ethernet frame,* includes all the headers plus the payload. *Payload* is the information data encapsulated inside the Ethernet frame *excluding all* headers. The *throughput* is the "measure of the rate at which data can be sent through the network, and is usually specified in *bits per second* [40]". The *protocol throughput* is the amount of data that a protocol stack can process per unit of time (Kbps or Mbps). The *payload throughput* is the amount of payload that the DUT can process per unit time. It is calculated as follows[1]:

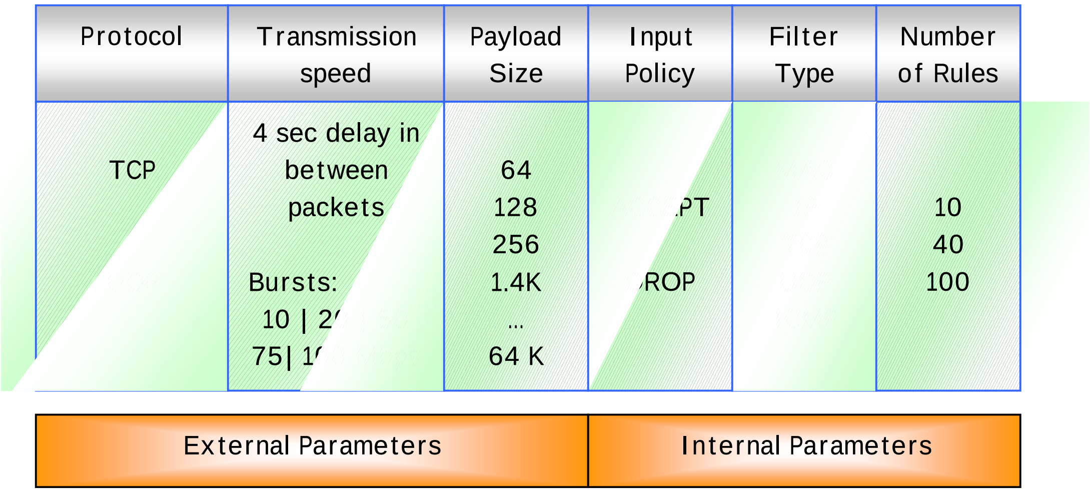
$$\text{Payload throughput} = (\text{size of the payload}) \,/\, (\text{stop latency})$$

[1] More in depth explanation about latency, throughput, and CPU utilization can be found in Peter Xie's master's thesis [37]

3.2.2 Parameters to determine the firewall sensitivity

As mentioned earlier, firewalls have been tested by modifying a set of parameters such as the number of rules, the number of connections, the number of bytes and the transmission rate.

Our investigation focuses on analyzing and testing the sensitivity of the firewall, and the performance impact generated by it, by varying a set of *external* and *internal* parameters presented in Figure 3-2. External parameters are those that cannot be controlled by the firewall such as *transmission protocol*, *transmission speed*, and *payload size.* Internal parameters are those that can be controlled by the firewall such as *Input policy*, *filtering type*, and *number of rules*.



Protocol	Transmission speed	Payload Size	Input Policy	Filter Type	Number of Rules
TCP	4 sec delay in between packets Bursts: 10 | 2 75| 1	64 128 256 1.4K … 64 K	PT ROP		10 40 100

External Parameters

Internal Parameters

Figure 3-2 Parameters to determine the sensitivity of the firewall

A series of tests will involve varying the parameters presented above in two different scenarios. The first scenario will consist on capturing a single packet and analyzing the performance impact as it traverses the stack. The second test consists on tracing a stream of packets at various transmission rates. We explain the two scenarios below.

3.3 Tests definitions

There are two main issues to resolve in the two scenarios just mentioned above, and they can be summarized in two questions: (1) Does a single packet carry enough information to explain the sensitivities of the firewall? (2) Will the measurements obtained for single-packet tests be sufficient enough to measure the performance on the host?

The data collected from tracing a single packet in the stack should provide enough information to find a time approximation of the sensitivities of the firewall and the total processing time to the parameters already presented. On the other hand, multiple packets will provide more "accurate" results. This can be explained with the following example: think of the operating system to be analog to the plumbing system of a kitchen sink. Imagine that you desire to know how long would 100 liters of water take to pass through the plumbing. There are two ways to measure the time: the first way is by pouring one liter of water and multiplying it by 100; the second way is to drain the 100 liters.

In the first scenario, you pour 1 liter of water, let it go down the drain, and measure how long it took for that liter to enter and exit the system. An average can be calculated after doing this several times. The average can be multiplied by 100 times to find an approximate to pouring the 100 liters. In the second scenario, you can open up the faucet and measure the time that it takes to drain the 100 liters. The former is a clean and fast way to find an approximation of the total time to complete the system because, whether we pour 1 or 100 liters, the water will flow through the same path. On the other hand, the latter will provide more “accurate” results because they include the rate at which the water was expelled from the faucet and the pressure exerted by the mass of water pushing down the pipe.

Just as the water flows through the same path, in the same way, packets follow the same path when they traverse the TCP/IP stack. Consequently, single-packet tests are analogous to pouring only one liter at a time. These tests will provide an approximation of the time that a packet is held at each point in the stack. Furthermore, throughput or multiple packets tests at various rates are analogous to pouring 100 liters at one time because they take into account the queuing of packets by the OS, the processor speed, and the rate of transmission.

In this thesis, only single packet tests are performed to understand and measure the sensitivities of the firewall. They also provide a conservative approximation to the actual latency for multiple packet tests.

3.3.1 Single-packet tests

Single-packet tests are performed using two PCs. Volans, our Device Under Test (DUT), is a dual 450MHz Intel Pentium processor with a modified 2.4.7 kernel running the server application. One of the CPUs is turned off in the SMP option of the kernel configuration – the kernel configuration file is included in the CD attached with this document. The `iptables-1.2.4` version was installed to the kernel. The modified kernel has 5 different points to store timing measurements as the packet traverses throughout the stack. The files modified are: *dev.c*, *ip_input.c*, *tcp_input.c*, *udp.c*, and *socket.c*. At boot time, both machines will start in run-level 3. During the tests, no services will run in the background, see Appendix B and C for details on how to run the tests and to see the scripts. Libra, the client, is a 233MHz Pentium II processor. Both machines are isolated from any outside traffic and connected through a 100 Mbps 3Com switch. Refer to Figure 3-3 to see the test bed.

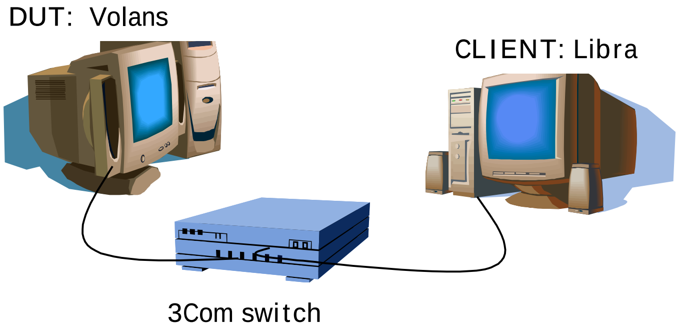


Figure 3-3 Test setup to measure the latency when a single packet is sent every 4 seconds

Single packet tests procedures are included in Chapter 4. See Appendix C for the source code used to generate the rules.

3.3.2 Throughput tests

Throughput tests are performed using the Spirent's Network Tester "SmartBits". These tests will show the latency of the network stack when multiple packets are sent at different transmission rates.

The tests are performed in the Cal Poly Cisco lab. A Windows 95 PC is connected via a Patch panel to control the SmartBits 2000. The SmartBits cards, model ML-7710, are connected via the patch panel to a Cisco 2900 XL switch to communicate with our DUT, which is Volans. The test bed is shown in Figure 3-4.

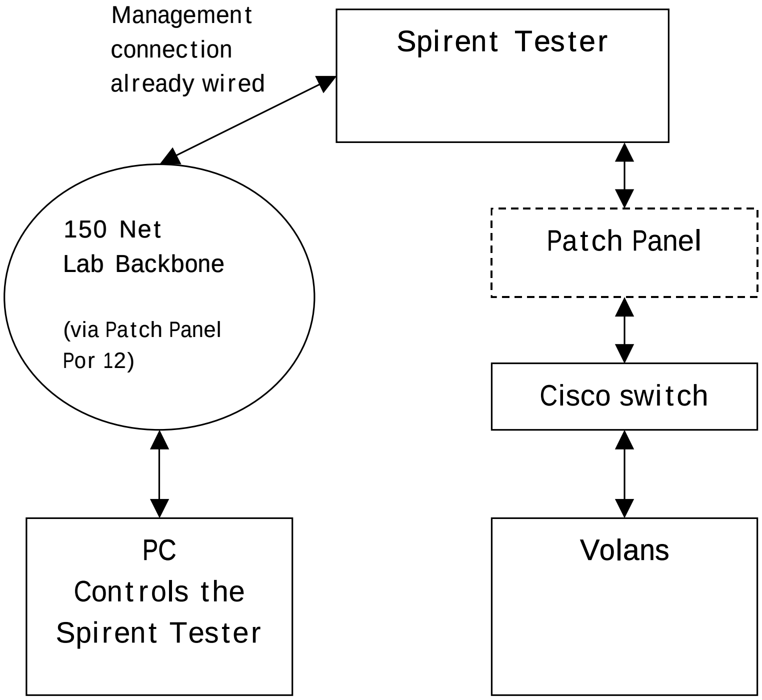


Figure 3-4 Test setup using the Spirent's network tester to vary the throughput

Two SmartBit cards were connected to the switch, one is to send the stream of test packets at different rates, and the other is used to send 2 packets to port 6789 which serves to reset the memory buffers where the measurements are stored. During the tests, the Smartbits would run for one minute before the timestamp measurements were taken. This is because we considered that one-minute would be enough to reach steady-state for the transmission rate performance testing.

Given that the number of packets increase as the transmission rate increases, the timestamping instrumentation inside the kernel was modified for each test. A new counter was added to the code so that the timestamps would be taken after one minute. The number of packets sent in a minute is automatically obtained with the SmartWindows application. The second modification was to increase the memory buffers in order to store 4000 timestamps instead of 50 as it was before. A third modification was to match incoming packets on the port number instead of reading the payload. One last change had to be made to the instrumentation code inside the `netif_rx` function. Since `netif_rx` executes with interrupts disabled and I/O operations are costly, we removed the only `memcpy` from our instrumentation code.

3.3.3 Packet specifications

The test's packets must be less than the MTU because of fragmentation. If the payload is larger than the MTU, by nature, the protocol stack will fragment the payload into packets, one(s) that will have the size of an Ethernet frame with the last one possibly having a payload less than an Ethernet frame. Having to deal with different payload sizes in a test may cause a discrepancy and could ruin the results, or at least make the results difficult to interpret.

3.4 The Linux TCP/IP stack

3.4.1 Understanding the packet data flow

It is critical to understand the packet data flow in order to be able to add the timestamps and perform the measurements. Unfortunately, by the time this analysis was made, there was no detailed documentation on the receiving operation or the netfilter/firewall hooks *specific* to the Linux kernel 2.4 other than the source code. Thus, our first efforts focused on capturing the data flow from the data link layer to the application layer and finding the netfilter hooks. On the other hand, by the time this document was written we found documentation (sections 3.6.2 and 3.6.3) that confirmed our findings.

3.4.1.1 The receiving operation

From the basics of networking we understand that in order to establish a TCP connection a server must be listening to an open port. A client wanting to establish a connection sends a `SYN` packet to the server. The server responds by sending a `SYN/ACK` to finish the *handshake* and the client sends an `ACK` plus the `PACKET`. Followed by the *handshake* only `PACKETS` are sent to the server until a `FIN` packet is received in order to close the connection [41]. Refer to Figure 3-5.

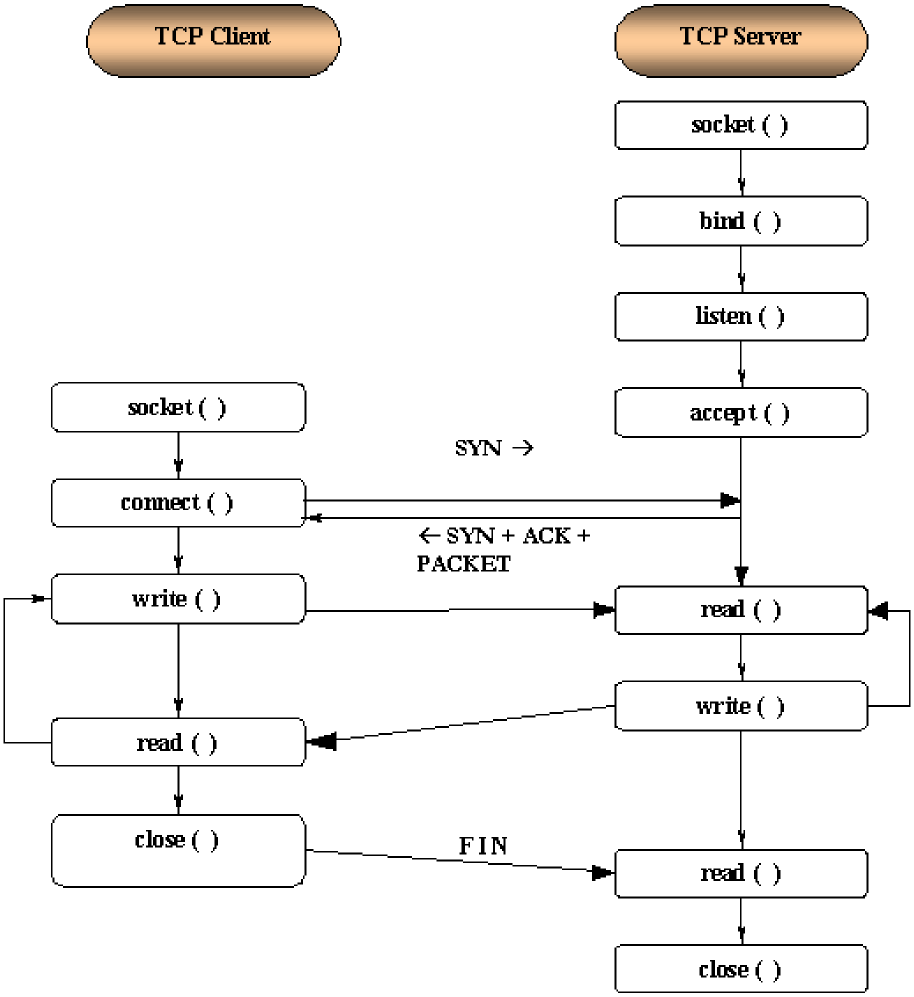


Figure 3-5 Basic TCP client-server connection

The Linux operating system separates the receiving operation in two parts. The first is when the server holds listening to a port, which is from the application layer down to the Linux socket layer [42]. The other happens when a packet is coming in from the network, or from the physical layer up. These two operations are explained in detail in the next sections.

3.4.1.2 Analysis from the application down

When a server application opens a connection and is ready to receive a packet, it will make a call to *read( )* or *recv( )* on a socket. Then, *read( )* makes a system call to *sock_read( ).* The latter will call *sock_recvmsg( )*, which will then call *sock->ops->recvmsg( ).* For a TCP connection the *"ops"* corresponds to a pointer to *inet*, where *inet* calls the *recvmsg( )* function. Finally, the *inet_recvmsg( )* calls *sk->proto[tcp|udp]->recvmsg( )* and the application sleeps. The latter is put into the run queue or is woken up after the TCP layer has processed any incoming packets. Figure 3-6 shows this process.

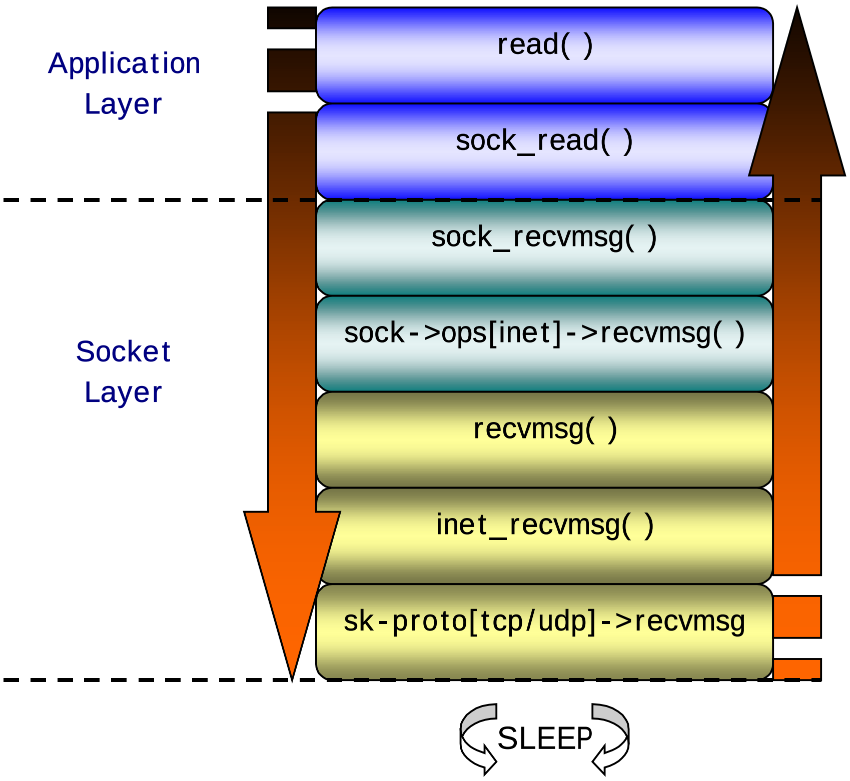


Figure 3-6 Receiving operation from Application to Socket layer

3.4.1.3 Analysis from Datalink layer to Socket layer

Initially, as a packet comes in from the physical layer it causes the Ethernet device to "fire up" an interrupt. Interrupts are handled by top-halves and bottom-halves [43]. The top-half is handled by the network adapter's device driver (e.g. 3c59x.c). The device driver calls the `eth_type_trans()` function located in the *eth.c* file. This function organizes the first part of the packet header (i.e. MAC header) inside an `sk_buff` structure.

All the information contained inside a packet is carried out through the stack in the form of an `skbuff` structure until we get to the socket layer. In the Linux source code we always find a structure `skb` of type `skbuff`. So, for example, when a packet enters from the network, `skb->data` points to beginning of the entire information of that incoming packet. The data is not organized in the `skbuff` structure all at once but, as the packet passes through the stack, each layer will reorganize the packet's information inside that `skb` structure. After the TCP/UDP layer has been processed, it will pass the pointer to `skb` structure to the socket layer. The socket layer will extract information inside the `skb` structure and create a new structure of type `sock.` Thus, the `sock` structure will contain information such as source and destination port, the pointer to the payload, and more. More details on the information inside these structures can be found in the `sock.h` and `skbuff.h` files of the Linux source code.

Going back to the execution of the top-half, when the `eth_type_trans` function returns the device driver calls the device controller (i.e. `netif_rx)` located inside the *dev.c* file. This file controls all the network device drivers and it is located in the `usr/src/linux/net/core/` directory. Two main functions separate the top-half from the bottom-half: `netif_rx()` and `net_rx_action()`, respectively.

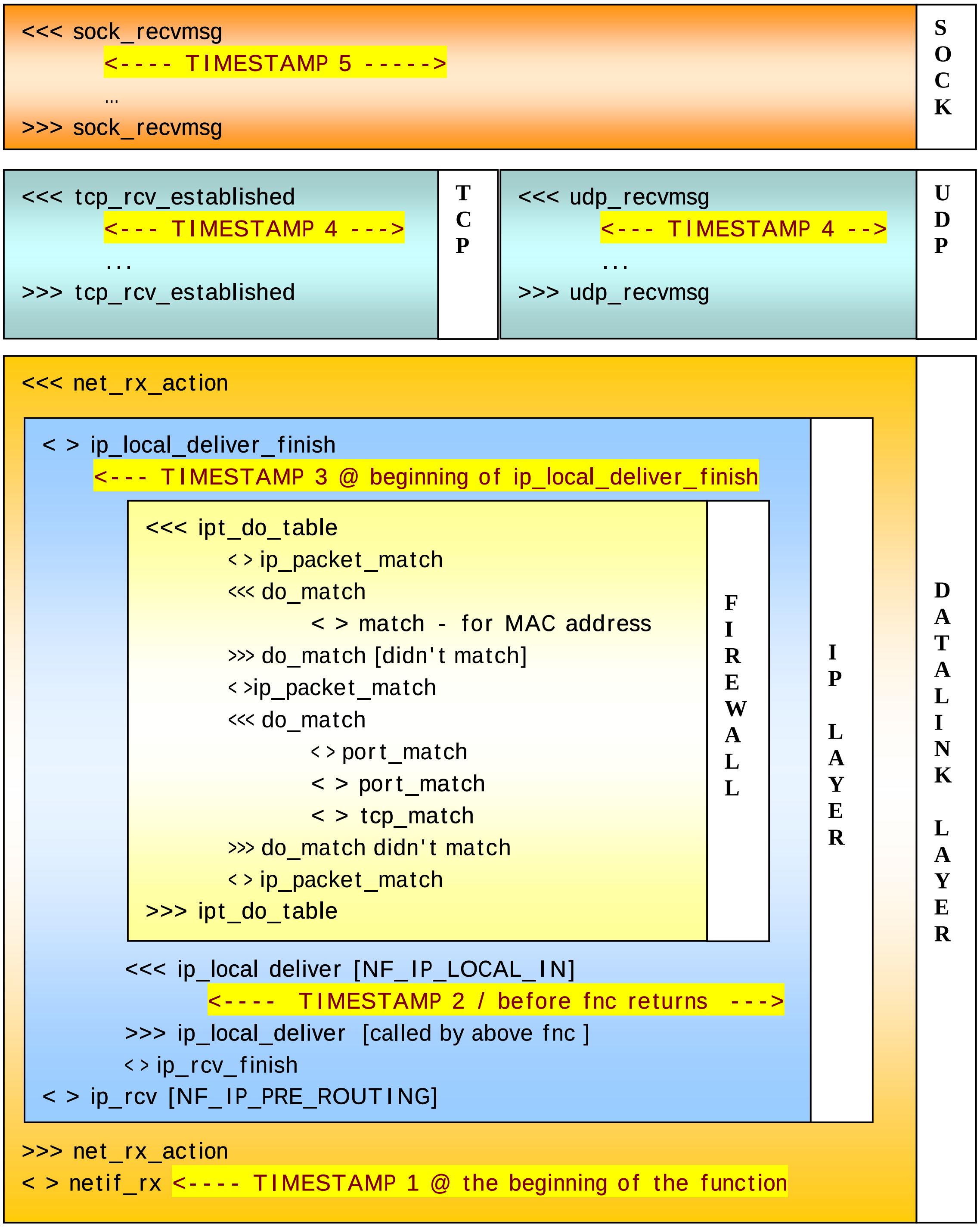
After the top-half executes, the *swapper* will be in charge of running the bottom-half. Note that it is the *swapper* and not the *scheduler* who handles this operation. The difference between the *swapper* and the *scheduler* is that the swapper is in charge of completing the execution of the pending bottom-halves [43] and the latter is in charge of handling processes.

The `netif_rx()` function takes a timestamp by calling the `get_fast_time(&skb->stamp)` function. This timestamp serves as a unique ID for each packet. This packet ID is transferred throughout the entire stack inside the `skb` structure, serving as a mean to match/differentiate the measurements for a specific packet at each layer. After the top-half executes, the swapper schedules to execute the bottom-half which starts with `net_rx_action()`.

Figure 3-7 presents the example of a single packet traversing the TCP/IP stack with a firewall of two rules, matching a MAC address and a TCP port. The packet is traced through all the layers of the stack until the socket layer hands the data to the application. The symbols in Figure 3-7 represent the following:

< >	enter and exit function ( )
>>>	enter function ( )
<<<	exit function ( )
. . .	several functions

Figure 3-7 will serve as the basis for the instrumentation and analysis because every TCP and UDP packet destined for the host will follow the path outlined in this figure. Notice that the IP layer and the firewall are inside the Data Link layer. This is because the MAC header is “stripped off” along with the IP header right before the `net_rx_action()` function exits.



```
<<< sock_recvmsg
        <---- TIMESTAMP 5 ----->
        …
>>> sock_recvmsg

<<< tcp_rcv_established
        <--- TIMESTAMP 4 --->
        ...
>>> tcp_rcv_established

<<< udp_recvmsg
        <--- TIMESTAMP 4 -->
        ...
>>> udp_recvmsg

<<< net_rx_action
  < > ip_local_deliver_finish
      <--- TIMESTAMP 3 @ beginning of ip_local_deliver_finish
        <<< ipt_do_table
                < > ip_packet_match
                <<< do_match
                        < > match - for MAC address
                >>> do_match [didn't match]
                < >ip_packet_match
                <<< do_match
                        < > port_match
                        < > port_match
                        < > tcp_match
                >>> do_match didn't match
                < > ip_packet_match
        >>> ipt_do_table
      <<< ip_local deliver [NF_IP_LOCAL_IN]
              <----  TIMESTAMP 2 / before fnc returns  --->
      >>> ip_local_deliver  [called by above fnc ]
      < > ip_rcv_finish
  < > ip_rcv [NF_IP_PRE_ROUTING]
>>> net_rx_action
< > netif_rx <---- TIMESTAMP 1 @ the beginning of the function
```

Figure 3-7 Traversing the Network Stack – from bottom up

Notice the timestamps placed throughout the stack in Figure 3-7; these are placed at critical points in order to take timing measurements in the stack. Timestamp 1 is our reference point ("T1" in Figure 3-1). Timestamp 2 is placed before the firewall starts its execution. Note that this is not the point where the IP layer begins, but the point where the netfilter/firewall begins. Timestamp 3 is placed after the firewall has processed the packet and has finished its execution. The difference between the measured values of Timestamp 3 and Timestamp 2 tell us the cost of having a firewall.

Timestamp 4 is placed after the TCP or UDP layers have been processed. At first, we speculated that if we block on TCP ports or MAC addresses, filtering should happen at the TCP layer or at the Data Link layer respectively, but Figure 3-7 proved us wrong. Finally, Timestamp 5 (point "T5"of Figure 3-1) is placed before the socket layer passes the payload to the application.

Once the path followed by a packet in the stack was studied, we performed various tests. To understand the results we found it necessary to study the source code and the `iptables` algorithm, which is explained in the last section of this chapter.

3.5 Software instrumentation

3.5.1 Software design and issues

The design and implementation of the timestamps involved some challenges. When tracing a packet through the stack, an important factor to take into consideration was the uniqueness of a packet. For example, after performing our first tests the measurement results showed that some of the packets were missing at the TCP and socket layers. It was not until we ran the network sniffer that we found the problem to be in the client application and in the timestamps implementation. Timestamps were taken for every single packet coming in from the network. For the client application, every time that the client would send a packet it would close the connection, re-negotiate with the server, and send the packet. So, when packets were "lost" it was because our instrumentation was taking measurements for ARPs, SYN, ACKs and other packets which do not traverse all the way up to the socket layer!

Another problem encountered was running other services, such as Xwindows, system logger, and NIS. Since we had a timestamp at this layer, the timestamp code would be called constantly, thus, taking measurements that did not belong to our test packets. NIS was the worst of them because both PCs would constantly send ARPs to find the NIS server and undesired packets kept coming in. The best solution was to shut down all the services (which we did later in our tests) but, if we wanted to have different traffic coming in, how do we identify our test packets? Well, we marked the payload. So we changed the hub for a switch and marked the payload with A's at the beginning of the data and E's at the end, just like this:

MAC header | IP header | TCP header | AAAAA******** *********************EEEEE

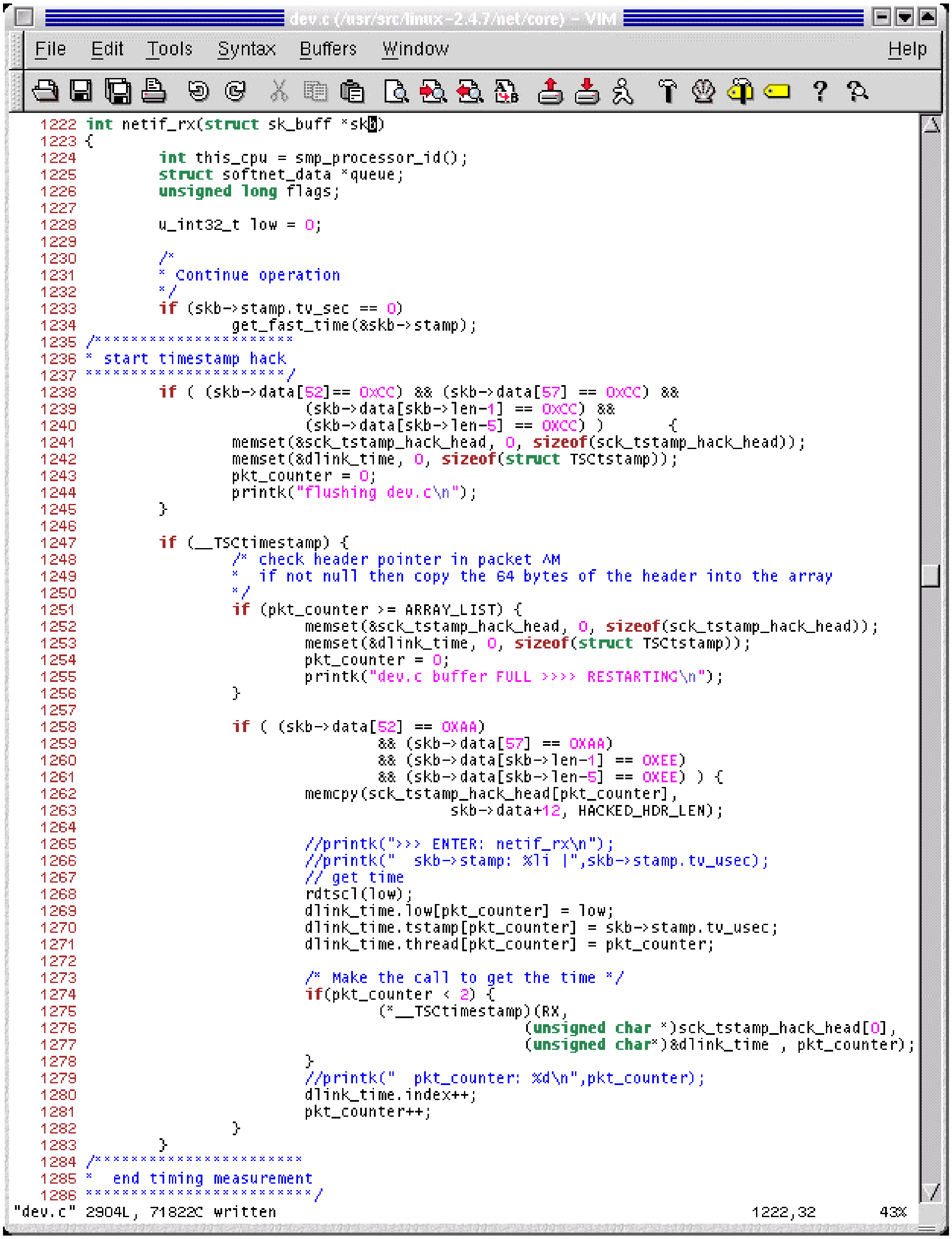
Marking the packet lead to finding out a way to read the payload as it traversed the stack. Well, recall the `skb` structure of type `skbuff` discussed in previous sections. This structure is modified in every layer of the stack. For an incoming packet, the `data` element inside the structure points to the beginning of the entire packet. So, when a packet comes in through the network, skb->data[0] points to the beginning of the MAC header. Then, in order to read the A's and E's of the payload, we have to offset skb->data to the beginning of the payload. For example, for a TCP packet the first A is at offset 52, (i.e. skb->data[52] – see Figure 3-8) and for a UDP packet the offset is at 29 (i.e. skb->data[29]). As the packet traverses the stack, the offset decreases because the layers modify the structure and strip off some parts of the header.

3.5.2 Instrumentation of the Timestamps

The performance measurement timestamps are taken by using the rtdscl macro. This macro reads the lower 32 bits of the Time Stamp Counter (TSC) using assembly instructions thus, giving a more precise time [43].

3.5.2.1 Timing measurements at the Data Link layer

As it has been explained, we place the first timing measurement inside the netif_rx() because this is the starting point of the stack. Only the packets marked with A's and E's will be measured. The skb->data is the pointer to the beginning of all the data. At this point in the stack skb->data[0] points to the beginning of the IP header. The payload starts at skb->data[52] for TCP and for UDP is at skb->data[29]. The packet information (i.e. timestamp, ID, count, and TCP header) is stored in arrays declared as *global* in order to reserve the memory space at boot time. The pointer to the data structure where the information is stored is passed to the /proc file system using __TSCtimestamp function.



```c
int netif_rx(struct sk_buff *skb)
{
        int this_cpu = smp_processor_id();
        struct softnet_data *queue;
        unsigned long flags;

        u_int32_t low = 0;

        /*
        * Continue operation
        */
        if (skb->stamp.tv_sec == 0)
                get_fast_time(&skb->stamp);
/**********************
* start timestamp hack
**********************/
        if ( (skb->data[52]== 0xCC) && (skb->data[57] == 0xCC) &&
                        (skb->data[skb->len-1] == 0xCC) &&
                        (skb->data[skb->len-5] == 0XCC) )       {
                memset(&sck_tstamp_hack_head, 0, sizeof(sck_tstamp_hack_head));
                memset(&dlink_time, 0, sizeof(struct TSCtstamp));
                pkt_counter = 0;
                printk("flushing dev.c\n");
        }

        if (__TSCtimestamp) {
                /* check header pointer in packet AM
                *  if not null then copy the 64 bytes of the header into the array
                */
                if (pkt_counter >= ARRAY_LIST) {
                        memset(&sck_tstamp_hack_head, 0, sizeof(sck_tstamp_hack_head));
                        memset(&dlink_time, 0, sizeof(struct TSCtstamp));
                        pkt_counter = 0;
                        printk("dev.c buffer FULL >>>> RESTARTING\n");
                }

                if ( (skb->data[52] == 0XAA)
                                && (skb->data[57] == 0XAA)
                                && (skb->data[skb->len-1] == 0XEE)
                                && (skb->data[skb->len-5] == 0XEE) ) {
                        memcpy(sck_tstamp_hack_head[pkt_counter],
                                        skb->data+12, HACKED_HDR_LEN);

                        //printk(">>> ENTER: netif_rx\n");
                        //printk("  skb->stamp: %li |",skb->stamp.tv_usec);
                        // get time
                        rdtscl(low);
                        dlink_time.low[pkt_counter] = low;
                        dlink_time.tstamp[pkt_counter] = skb->stamp.tv_usec;
                        dlink_time.thread[pkt_counter] = pkt_counter;

                        /* Make the call to get the time */
                        if(pkt_counter < 2) {
                                (*__TSCtimestamp)(RX,
                                                (unsigned char *)sck_tstamp_hack_head[0],
                                                (unsigned char*)&dlink_time , pkt_counter);
                        }
                        //printk("  pkt_counter: %d\n",pkt_counter);
                        dlink_time.index++;
                        pkt_counter++;
                }
        }
/************************
*  end timing measurement
*************************/
"dev.c" 2904L, 71822C written                                   1222,32       43%
```

Figure 3-8 **Timestamp hack for the device driver (Timestamp 1)**

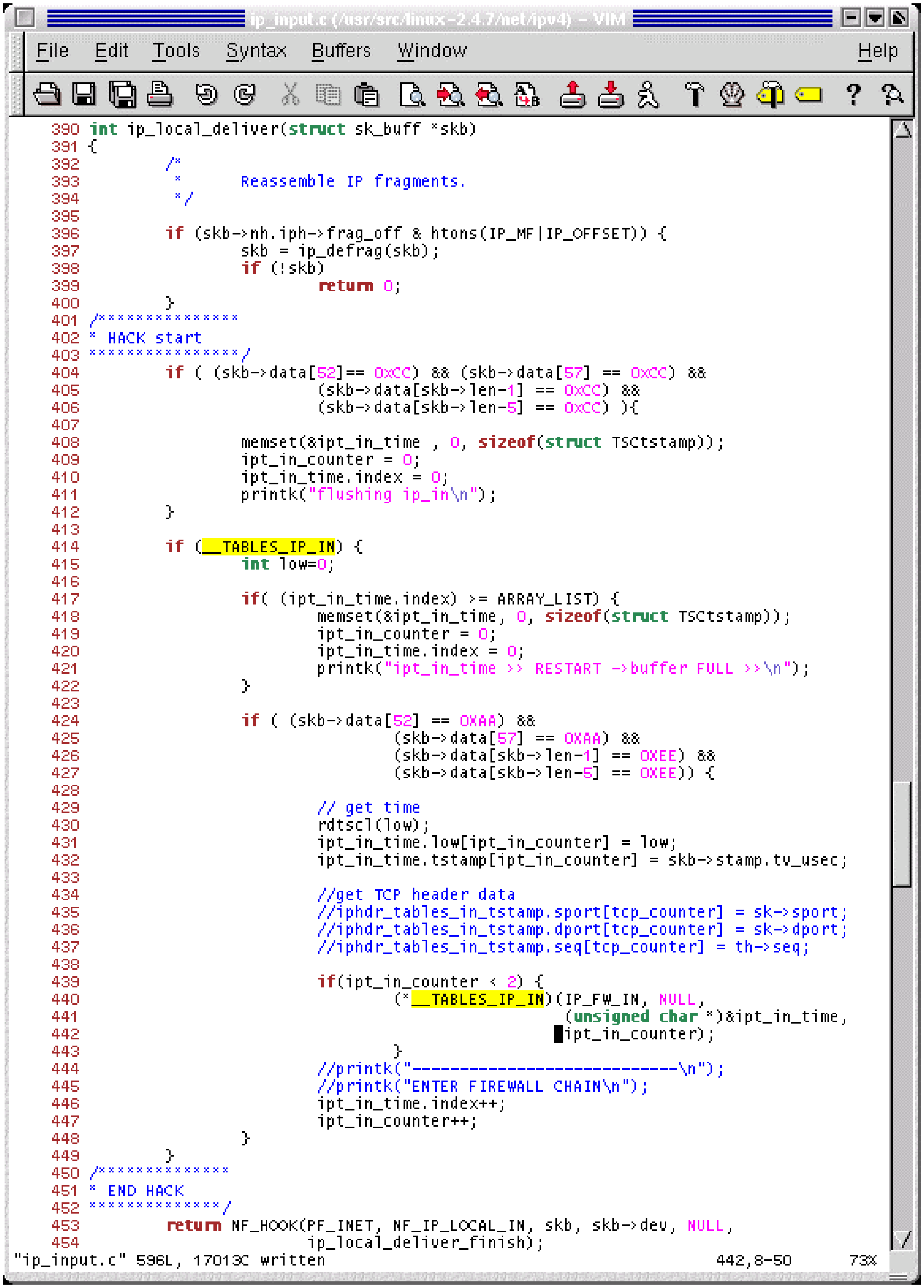
3.5.2.2 IP Layer

Theoretically, when a packet traverses throughout the stack the operating system should strip-off the header of each layer as it moves up. As mentioned earlier, in the Linux kernel, the MAC and the IP headers are not stripped-off until the firewall has processed the packet.

Timestamps 2 and 3 are taken at this layer to measure the firewall. Timestamp 2 is located right before the `ip_local_deliver()` returns. This is because when this function returns it makes a call to `NF_IP_LOCAL_IN`. Timestamp 3 is placed at the beginning of `ip_local_deliver_finish.` Notice in the Figures 3-9 and 3-10 that the calls to `__TABLES_IP_IN` and `__TABLES_IP_OUT` are the entry points to take the timestamps. When our instrumentation is not loaded, the addresses of `__TABLES_IP_IN` and `__TABLES_IP_OUT` point to NULL. Again, the timestamp is taken using the `rdtscl` macro. In order to reset the memory buffers, we send a packet marked with C's at the beginning and at the end of the payload. Thus the 'if' statement:

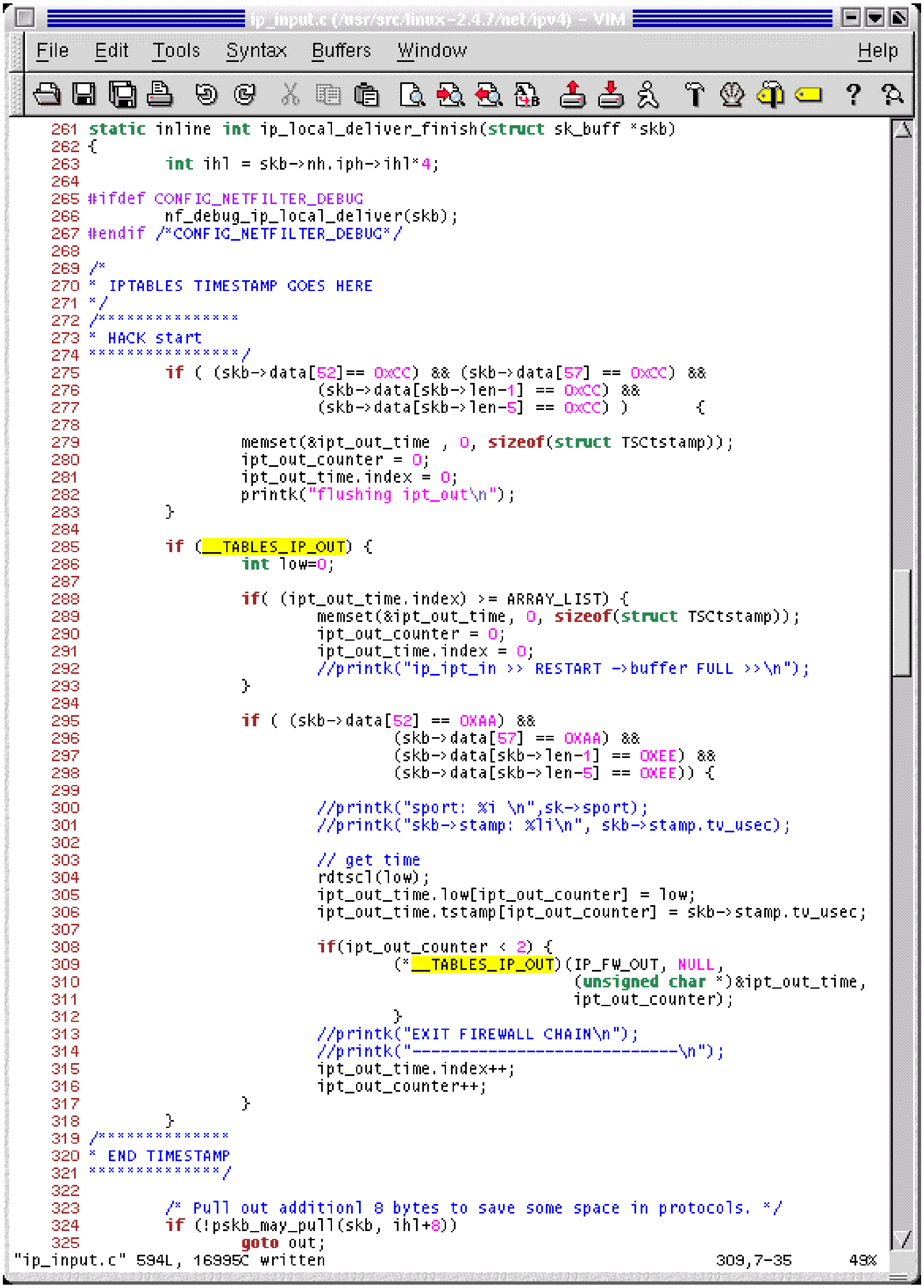
```
if ((skb->data[52] == 0xCC) && (skb->data[57] == 0xCC) &&
              (skb->data[skb->len-1] == 0xCC) &&
              (skb->data[skb->len-5] == 0xCC) )
```

reads the C's inside the payload and resets the buffers using and the counters. At the bottom of Figure 3-9 we observe the call to the `NF_HOOK` (i.e. netfilter hook), which is the entry point to the firewall.



```
int ip_local_deliver(struct sk_buff *skb)
{
        /*
         *      Reassemble IP fragments.
         */

        if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
                skb = ip_defrag(skb);
                if (!skb)
                        return 0;
        }
/***************
 * HACK start
****************/
        if ( (skb->data[52]== 0xCC) && (skb->data[57] == 0xCC) &&
                        (skb->data[skb->len-1] == 0xCC) &&
                        (skb->data[skb->len-5] == 0xCC) ){

                memset(&ipt_in_time , 0, sizeof(struct TSCtstamp));
                ipt_in_counter = 0;
                ipt_in_time.index = 0;
                printk("flushing ip_in\n");
        }

        if (__TABLES_IP_IN) {
                int low=0;

                if( (ipt_in_time.index) >= ARRAY_LIST) {
                        memset(&ipt_in_time, 0, sizeof(struct TSCtstamp));
                        ipt_in_counter = 0;
                        ipt_in_time.index = 0;
                        printk("ipt_in_time >> RESTART ->buffer FULL >>\n");
                }

                if ( (skb->data[52] == 0XAA) &&
                                (skb->data[57] == 0XAA) &&
                                (skb->data[skb->len-1] == 0XEE) &&
                                (skb->data[skb->len-5] == 0XEE)) {

                        // get time
                        rdtscl(low);
                        ipt_in_time.low[ipt_in_counter] = low;
                        ipt_in_time.tstamp[ipt_in_counter] = skb->stamp.tv_usec;

                        //get TCP header data
                        //iphdr_tables_in_tstamp.sport[tcp_counter] = sk->sport;
                        //iphdr_tables_in_tstamp.dport[tcp_counter] = sk->dport;
                        //iphdr_tables_in_tstamp.seq[tcp_counter] = th->seq;

                        if(ipt_in_counter < 2) {
                                (*__TABLES_IP_IN)(IP_FW_IN, NULL,
                                                  (unsigned char *)&ipt_in_time,
                                                  ipt_in_counter);
                                }
                        //printk("----------------------------\n");
                        //printk("ENTER FIREWALL CHAIN\n");
                        ipt_in_time.index++;
                        ipt_in_counter++;
                }
        }
/**************
 * END HACK
**************/
        return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                       ip_local_deliver_finish);
"ip_input.c" 596L, 17013C written                                    442,8-50      73%
```

Figure 3-9 IP layer – Beginning of the firewall (Timestamp 2)

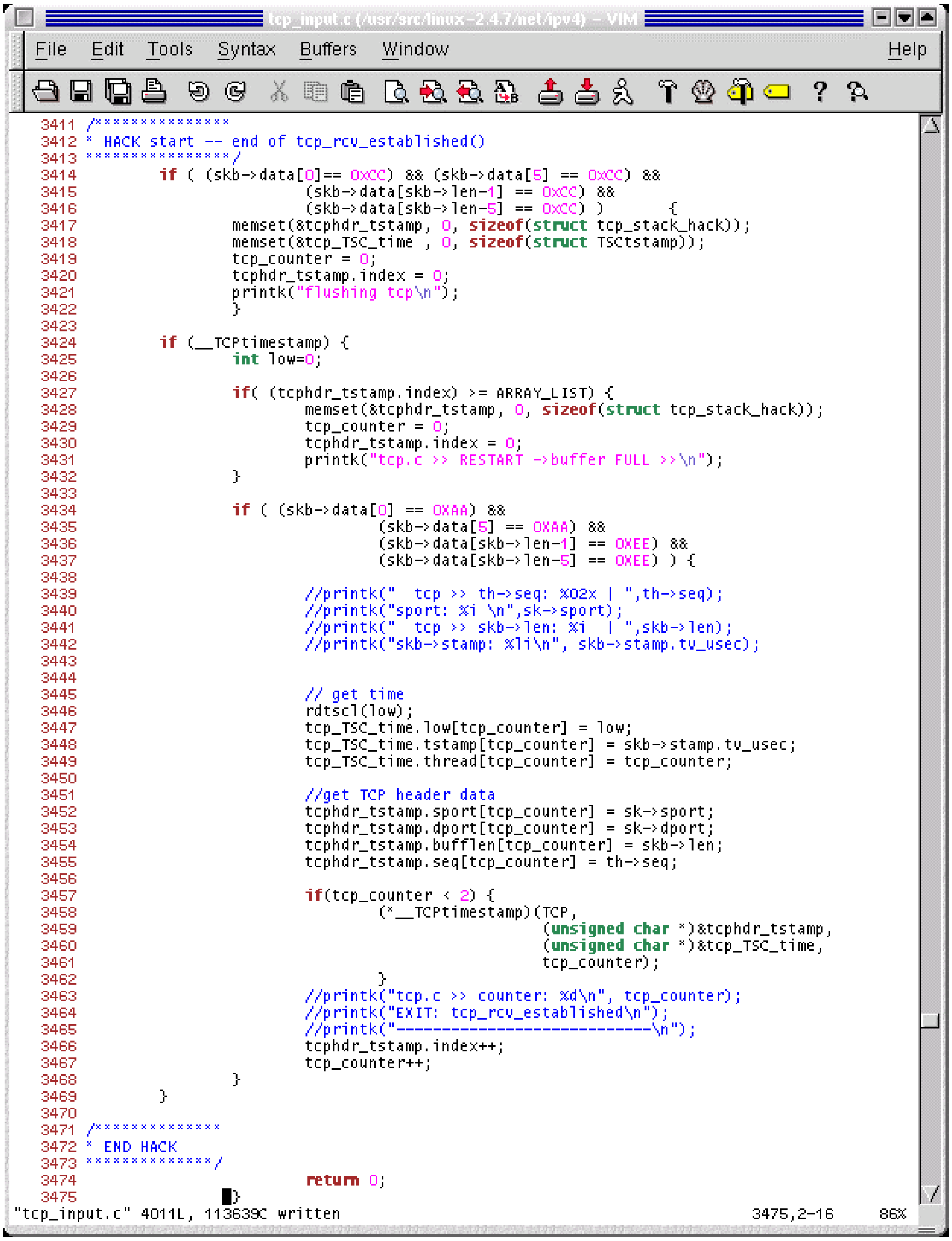


```
static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
        int ihl = skb->nh.iph->ihl*4;

#ifdef CONFIG_NETFILTER_DEBUG
        nf_debug_ip_local_deliver(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/

/*
 * IPTABLES TIMESTAMP GOES HERE
 */
/***************
 * HACK start
 ***************/
        if ( (skb->data[52]== 0xCC) && (skb->data[57] == 0xCC) &&
                        (skb->data[skb->len-1] == 0xCC) &&
                        (skb->data[skb->len-5] == 0xCC) )       {

                memset(&ipt_out_time , 0, sizeof(struct TSCtstamp));
                ipt_out_counter = 0;
                ipt_out_time.index = 0;
                printk("flushing ipt_out\n");
        }

        if (__TABLES_IP_OUT) {
                int low=0;

                if( (ipt_out_time.index) >= ARRAY_LIST) {
                        memset(&ipt_out_time, 0, sizeof(struct TSCtstamp));
                        ipt_out_counter = 0;
                        ipt_out_time.index = 0;
                        //printk("ip_ipt_in >> RESTART ->buffer FULL >>\n");
                }

                if ( (skb->data[52] == OXAA) &&
                                (skb->data[57] == OXAA) &&
                                (skb->data[skb->len-1] == OXEE) &&
                                (skb->data[skb->len-5] == OXEE)) {

                        //printk("sport: %i \n",sk->sport);
                        //printk("skb->stamp: %li\n", skb->stamp.tv_usec);

                        // get time
                        rdtscl(low);
                        ipt_out_time.low[ipt_out_counter] = low;
                        ipt_out_time.tstamp[ipt_out_counter] = skb->stamp.tv_usec;

                        if(ipt_out_counter < 2) {
                                (*__TABLES_IP_OUT)(IP_FW_OUT, NULL,
                                                   (unsigned char *)&ipt_out_time,
                                                   ipt_out_counter);
                                }
                        //printk("EXIT FIREWALL CHAIN\n");
                        //printk("----------------------------\n");
                        ipt_out_time.index++;
                        ipt_out_counter++;
                }
        }
/**************
 * END TIMESTAMP
 **************/

        /* Pull out additionl 8 bytes to save some space in protocols. */
        if (!pskb_may_pull(skb, ihl+8))
                goto out;
"ip_input.c" 594L, 16995C written                                  309,7-35      49%
```

Figure 3-10 IP layer – End of the firewall (Timestamp 3)

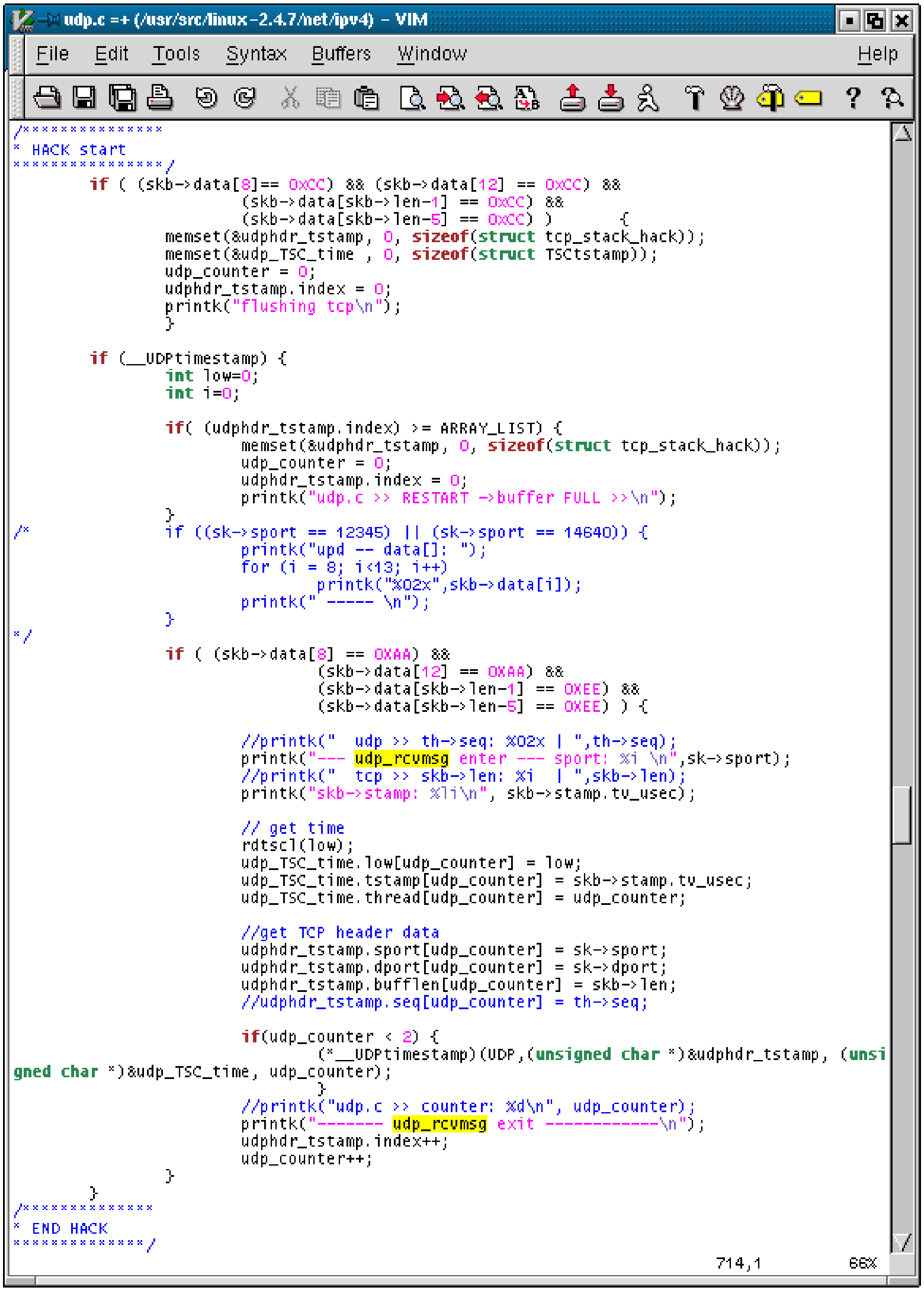
3.5.2.3 TCP and UDP layers

The timestamp is taken at the top of the TCP and UDP layers. For TCP the pointer to the payload is `skb->data[0]`, but for UDP the pointer to the payload is at `skb->data[8]`, See Figure 3-11 and 3-12.



```
/***************
* HACK start -- end of tcp_rcv_established()
****************/
        if ( (skb->data[0]== 0xCC) && (skb->data[5] == 0xCC) &&
                        (skb->data[skb->len-1] == 0xCC) &&
                        (skb->data[skb->len-5] == 0xCC) )       {
                memset(&tcphdr_tstamp, 0, sizeof(struct tcp_stack_hack));
                memset(&tcp_TSC_time , 0, sizeof(struct TSCtstamp));
                tcp_counter = 0;
                tcphdr_tstamp.index = 0;
                printk("flushing tcp\n");
                }

        if (__TCPtimestamp) {
                int low=0;

                if( (tcphdr_tstamp.index) >= ARRAY_LIST) {
                        memset(&tcphdr_tstamp, 0, sizeof(struct tcp_stack_hack));
                        tcp_counter = 0;
                        tcphdr_tstamp.index = 0;
                        printk("tcp.c >> RESTART ->buffer FULL >>\n");
                }

                if ( (skb->data[0] == 0XAA) &&
                                (skb->data[5] == 0XAA) &&
                                (skb->data[skb->len-1] == 0XEE) &&
                                (skb->data[skb->len-5] == 0XEE) ) {

                        //printk("  tcp >> th->seq: %02x | ",th->seq);
                        //printk("sport: %i \n",sk->sport);
                        //printk("  tcp >> skb->len: %i   | ",skb->len);
                        //printk("skb->stamp: %li\n", skb->stamp.tv_usec);


                        // get time
                        rdtscl(low);
                        tcp_TSC_time.low[tcp_counter] = low;
                        tcp_TSC_time.tstamp[tcp_counter] = skb->stamp.tv_usec;
                        tcp_TSC_time.thread[tcp_counter] = tcp_counter;

                        //get TCP header data
                        tcphdr_tstamp.sport[tcp_counter] = sk->sport;
                        tcphdr_tstamp.dport[tcp_counter] = sk->dport;
                        tcphdr_tstamp.bufflen[tcp_counter] = skb->len;
                        tcphdr_tstamp.seq[tcp_counter] = th->seq;

                        if(tcp_counter < 2) {
                                (*__TCPtimestamp)(TCP,
                                                  (unsigned char *)&tcphdr_tstamp,
                                                  (unsigned char *)&tcp_TSC_time,
                                                  tcp_counter);
                                }
                        //printk("tcp.c >> counter: %d\n", tcp_counter);
                        //printk("EXIT: tcp_rcv_established\n");
                        //printk("---------------------------\n");
                        tcphdr_tstamp.index++;
                        tcp_counter++;
                }
        }
/**************
* END HACK
**************/
                        return 0;
                }
"tcp_input.c" 4011L, 113639C written                          3475,2-16      86%
```

Figure 3-11 TCP layer – Timing the end of the TCP/IP stack

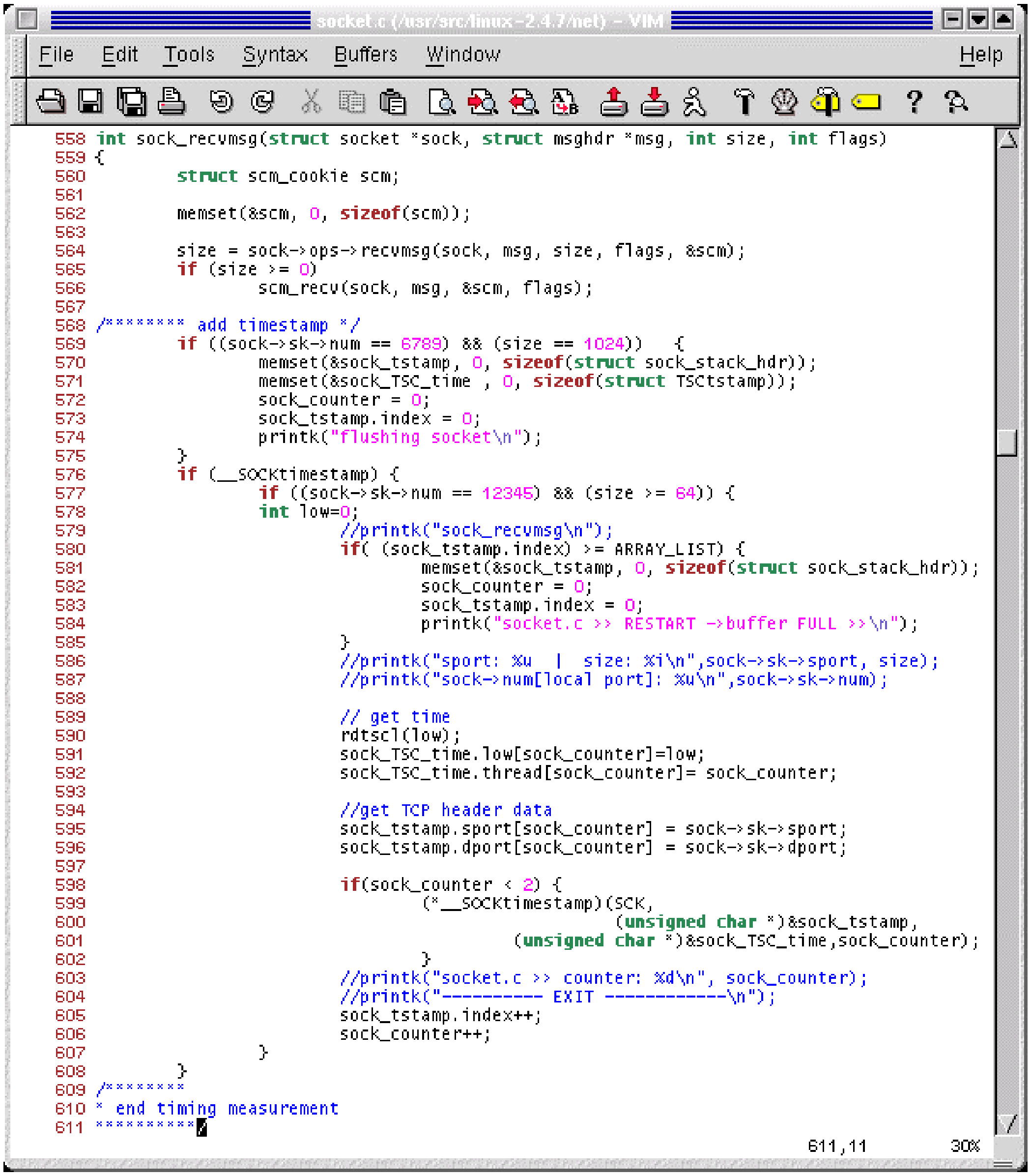


```
/***************
* HACK start
****************/
        if ( (skb->data[8]== OxCC) && (skb->data[12] == OxCC) &&
                        (skb->data[skb->len-1] == OxCC) &&
                        (skb->data[skb->len-5] == OxCC) )       {
                memset(&udphdr_tstamp, 0, sizeof(struct tcp_stack_hack));
                memset(&udp_TSC_time , 0, sizeof(struct TSCtstamp));
                udp_counter = 0;
                udphdr_tstamp.index = 0;
                printk("flushing tcp\n");
                }

        if (__UDPtimestamp) {
                int low=0;
                int i=0;

                if( (udphdr_tstamp.index) >= ARRAY_LIST) {
                        memset(&udphdr_tstamp, 0, sizeof(struct tcp_stack_hack));
                        udp_counter = 0;
                        udphdr_tstamp.index = 0;
                        printk("udp.c >> RESTART ->buffer FULL >>\n");
                }
/*              if ((sk->sport == 12345) || (sk->sport == 14640)) {
                        printk("upd -- data[]: ");
                        for (i = 8; i<13; i++)
                                printk("%02x",skb->data[i]);
                        printk(" ----- \n");
                }
*/
                if ( (skb->data[8] == OXAA) &&
                                (skb->data[12] == OXAA) &&
                                (skb->data[skb->len-1] == OXEE) &&
                                (skb->data[skb->len-5] == OXEE) ) {

                        //printk("  udp >> th->seq: %02x | ",th->seq);
                        printk("--- udp_rcvmsg enter --- sport: %i \n",sk->sport);
                        //printk("  tcp >> skb->len: %i   | ",skb->len);
                        printk("skb->stamp: %li\n", skb->stamp.tv_usec);

                        // get time
                        rdtscl(low);
                        udp_TSC_time.low[udp_counter] = low;
                        udp_TSC_time.tstamp[udp_counter] = skb->stamp.tv_usec;
                        udp_TSC_time.thread[udp_counter] = udp_counter;

                        //get TCP header data
                        udphdr_tstamp.sport[udp_counter] = sk->sport;
                        udphdr_tstamp.dport[udp_counter] = sk->dport;
                        udphdr_tstamp.bufflen[udp_counter] = skb->len;
                        //udphdr_tstamp.seq[udp_counter] = th->seq;

                        if(udp_counter < 2) {
                                (*__UDPtimestamp)(UDP,(unsigned char *)&udphdr_tstamp, (unsi
gned char *)&udp_TSC_time, udp_counter);
                                }
                        //printk("udp.c >> counter: %d\n", udp_counter);
                        printk("------- udp_rcvmsg exit ------------\n");
                        udphdr_tstamp.index++;
                        udp_counter++;
                }
        }
/**************
* END HACK
**************/
                                                                714,1         66%
```

Figure 3-12 UDP layer – Timestamp 4

3.5.2.4 *SOCKET layer*

The socket layer is the portion of the stack that forwards the data to the application layer. At this point we can no longer match the A's and E's of the payload, so we trace the packet using the port that it is destined for (i.e. 12345). See Figure 3-13.



```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
{
        struct scm_cookie scm;

        memset(&scm, 0, sizeof(scm));

        size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
        if (size >= 0)
                scm_recv(sock, msg, &scm, flags);

/******** add timestamp */
        if ((sock->sk->num == 6789) && (size == 1024))   {
                memset(&sock_tstamp, 0, sizeof(struct sock_stack_hdr));
                memset(&sock_TSC_time , 0, sizeof(struct TSCtstamp));
                sock_counter = 0;
                sock_tstamp.index = 0;
                printk("flushing socket\n");
        }
        if (__SOCKtimestamp) {
                if ((sock->sk->num == 12345) && (size >= 64)) {
                int low=0;
                        //printk("sock_recvmsg\n");
                        if( (sock_tstamp.index) >= ARRAY_LIST) {
                                memset(&sock_tstamp, 0, sizeof(struct sock_stack_hdr));
                                sock_counter = 0;
                                sock_tstamp.index = 0;
                                printk("socket.c >> RESTART ->buffer FULL >>\n");
                        }
                        //printk("sport: %u  |  size: %i\n",sock->sk->sport, size);
                        //printk("sock->num[local port]: %u\n",sock->sk->num);

                        // get time
                        rdtscl(low);
                        sock_TSC_time.low[sock_counter]=low;
                        sock_TSC_time.thread[sock_counter]= sock_counter;

                        //get TCP header data
                        sock_tstamp.sport[sock_counter] = sock->sk->sport;
                        sock_tstamp.dport[sock_counter] = sock->sk->dport;

                        if(sock_counter < 2) {
                                (*__SOCKtimestamp)(SCK,
                                                   (unsigned char *)&sock_tstamp,
                                         (unsigned char *)&sock_TSC_time,sock_counter);
                                }
                        //printk("socket.c >> counter: %d\n", sock_counter);
                        //printk("---------- EXIT ------------\n");
                        sock_tstamp.index++;
                        sock_counter++;
                }
        }
/********
* end timing measurement
**********
```

Figure 3-13 SOCKET layer – Before data is sent to the application (Timestamp 5)

3.6 The Linux Firewall – IPTABLES

3.6.1 Iptables Application

The Linux `iptables` was introduced with the 2.4.0 kernel to replace `ipchains`. With `iptables` the user can create and delete chains and matching rules to filter packets. There are 3 default policies: INPUT – to check the headers of incoming packets, OUTPUT – for outgoing packets/connections, and FORWARD – if the machine is used as a router (e.g. as a Network Address Translator.) Each policy has its own set of rules.

Basically, rules are instructions with pre-defined characteristics to match on a packet. When a match is found the firewall makes a decision to handle that packet. Each rule is executed in order until a match is found. A rule can be set like this:

iptables [table] <command> <match> <target/jump>

See the example below:

```
#iptables –P INPUT ACCEPT
#iptables –A INPUT –p tcp –dport 23 –j DROP
#iptables –A INPUT –p udp –dport 80 –j DROP
#iptables –A INPUT –p icmp –j DROP
```

Where: `–P`: policy; `–A`: append; `–p`: protocol; `–dport`: destination port; `–j`: jump

In the example, the first rule says that we accept any incoming connections from anywhere from the network. The next rule checks if the packet is a TCP, UDP or ICMP packet, respectively. If the incoming packet is TCP and if it is trying to establish a connection to port 23 (i.e. telnet), the packet is DROPed. The next rule drops any UDP packets trying to connect to port 80. The last rule drops all ICMP packets.

Iptables and ipchains, ironically enough, have the benefit of "chains"!! Chains are basically a sublayer of rules so that, if we want to capture a packet with specific characteristics, it is efficient not to make it go through the rest of rules that might never match that specific packet.

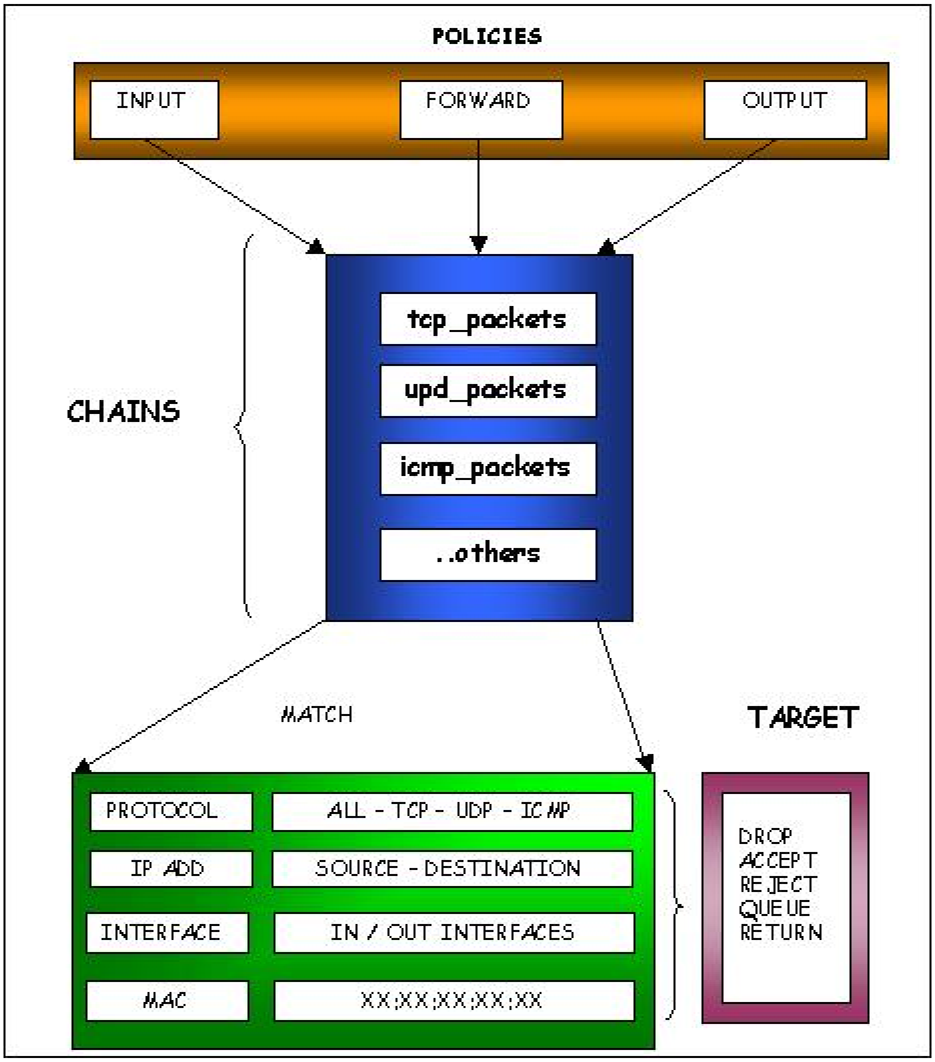


Figure 3-14 Iptables configuration Process

In Figure 3-14 we see how, under each POLICY, we can create chains. For example, imagine that the user wants to accept only those packets coming from the subnet 192.168.X.X. Also, for those specific packets belonging to that subnet, the user wants to accept TCP packets destined to ports 21 and 80, and UDP packets destined for ports 81 and 12345. In that case, the configuration of the firewall looks like this:

```
#iptables –P INPUT DROP
#iptables –A INPUT –s 192.168.0.0/16 ACCEPT
#iptables –N tcp_packets
#iptables –N udp_packets
#iptables –A INPUT –p tcp –j tcp_packets
#iptables –A INPUT –p udp –j udp_packets
#iptables –A tcp_packets –dport 21 –j ACCEPT
#iptables –A tcp_packets –dport 80 –j ACCEPT
#iptables –A udp_packets –dport 81 –j ACCEPT
#iptables –A udp_packets –dport 12345 –j ACCEPT
```

In the example, we have specified to drop every packet except those packets coming from the subnet 192.168.X.X, and they should be checked under the rule set. We create two chains, `tcp_packets` and `udp_packets`. Under each chain we create a set of rules to match the packet and with the rule we specify a target (e.g. ACCEPT / DROP / REJECT / QUEUE / RETURN). A TCP packet coming from the trusted IP will be checked under the `tcp_packets` chain. Inside that chain we check if the packet is destined for ports 21 or 80. If it is not destined for any of the two ports the packet is dropped. Only TCP packets will be checked under the `tcp_packets` chain. The same happens for UDP, anything destined for port 81 and 12345 is accepted, otherwise the packets will be dropped.

3.6.2 Architecture of the Netfilter[2]

The Linux Netfilter consists in a series of “hooks” placed in several points in the network stack – so far IPv4, IPv6 and DECnet.

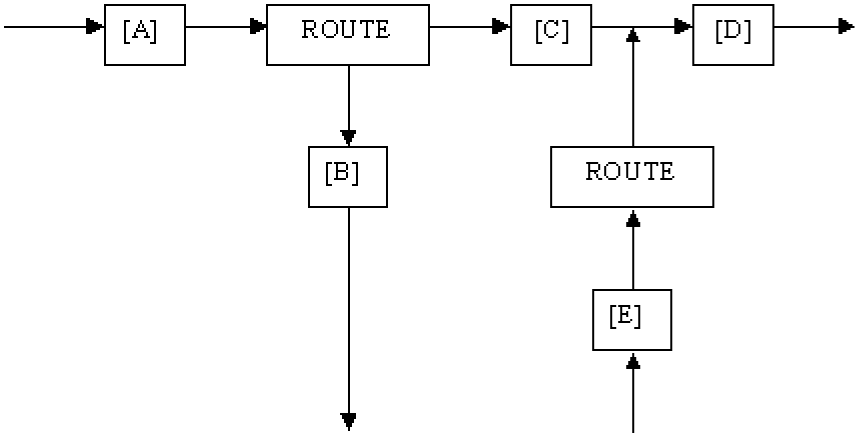


Figure 3-15 Packet traverses the netfilter

In Figure 3-15 a packet comes in from the left hand side of the picture. The first check point to the netfilter’s framework is the NF_IP_PRE_ROUTING [A] hook; this is after the packet has passed simple sanity checks, such as not truncated, IP checksum OK, not a promiscuous receive. The routing code will decide whether the packet is destined for another interface, or for a local process. Packets that are unroutable may be dropped.

[2] The information provided in sections 3.6.2 and 3.6..3 have been taken, and some parts even copi ed, “as is” from the “Netfilter Hacking HOWTO: Netfilter Architecture” Document [22]. I want to take no credit for the information presented in these two sections because the document is short, simple to understand, and to the point. Some things have be en reworded but the author, and maintainer of iptables Rusty Russell, did an excellent job presenting this in a very simple way.

If the packet is destined for the box itself, the netfilter's framework NF_IP_LOCAL_IN [B] hook is called. The analysis of the performance of the firewall starts at this point.

However, if the packet is destined to another interface, the netfilter framework is called for the NF_IP_FORWARD [C] hook. Finally, the packet is passed to the NF_IP_POST_ROUTING [D] hook it goes out to the outside.

When packets are created locally and the netfilter has been configured to filter outgoing traffic, the NF_IP_LOCAL_OUT [E] hook is called. Here, "routing occurs after this hook is called - in fact, the routing code is called first (to figure out the source IP address and some IP options) - if you want to alter the routing, you must alter the '`skb->dst`' field yourself, as is done in the NAT code."

3.6.3 Netfilter Base

The firewall is modular; this means that the network hooks will only be called when a rule has registered that hook. Rusty Russell explains, "Kernel modules can be registered to listen at any of these hooks. A module that registers a function must specify the priority of the function within the hook." In other words, when creating a module, the module should specify what netfilter hook(s) will be used so that "when a netfilter hook is called from the networking code, each module registered at that point is called in the order of priorities, and is free to manipulate the packet." The module can then tell netfilter to do one of five things:

1. NF_ACCEPT: continue traversal as normal
2. NF_DROP: drop the packet; don't continue traversal

3. NF_STOLEN: I’ve taken over the packet; don’t continue traversal

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

5. NF_REPEAT: call this hook again

For example, when using Network Address Translation (NAT), “for non-local packets, the NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks are perfect for destination and source alterations respectively;” this is because pre-routing checks on the destination address of the packet and makes a decision to forward it or pass it to the host itself. Post-routing checks if the packet is allowed to be forwarded or not. More detailed information can be found in [22].

3.6.4 Iptables Algorithm

The `iptables` algorithm will explain the results in Chapter 4. Notice what happens when the netfilter’s framework `NF_IP_LOCAL_IN` hook is called in Figure 3-16. `Iptables` executes `ipt_do_table,` which then executes `ipt_packet_match`, Figure 3-17.

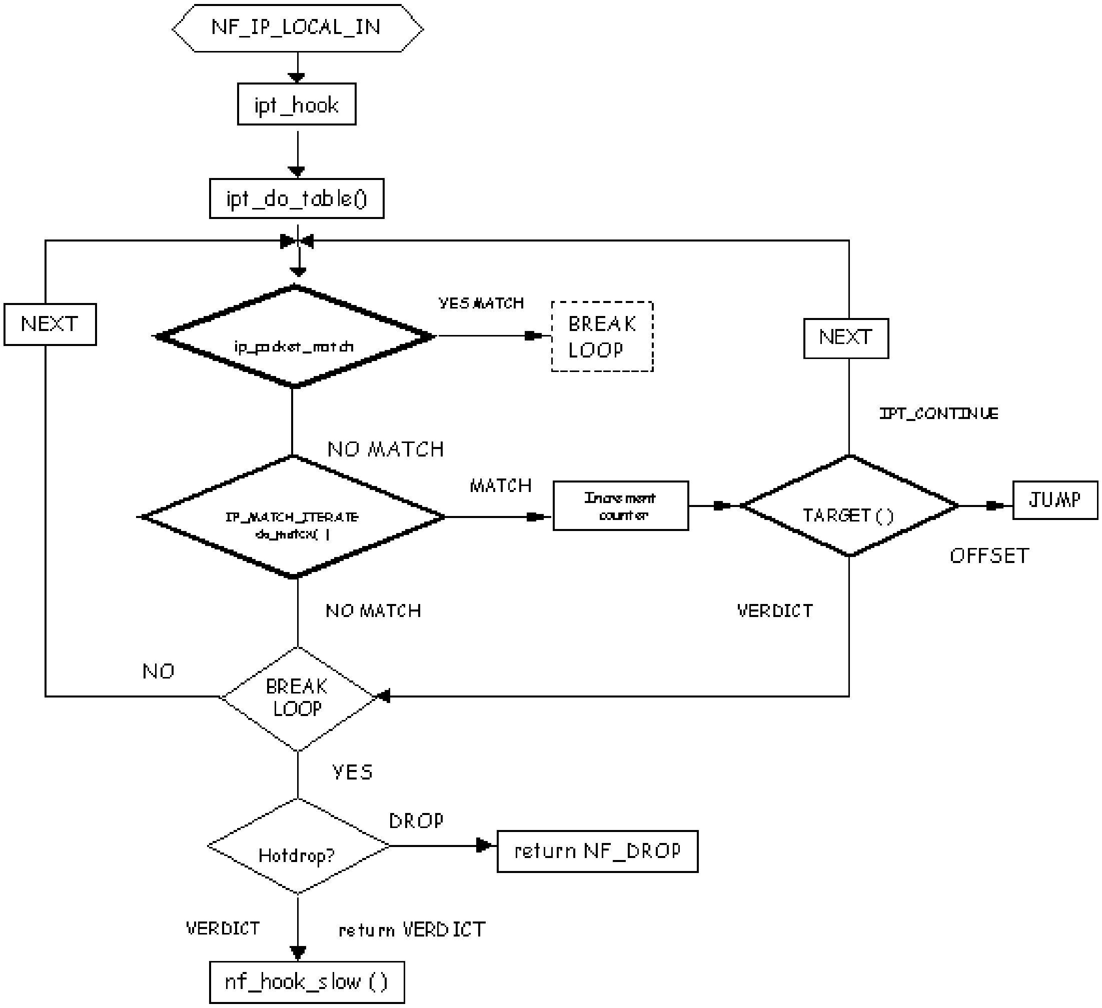


Figure 3-16 IPTABLES Algorithm – ipt_do_table( ) checks for matches in the rule-set

Observe in Figure 3-17 that the firewall checks the source and destination IP address first. If no match is found, it tries to find a match for the input device, then check for the output interface device, then the protocol, and finally it checks if the packet is a fragment. If a match is not found, `ip_packet_match` will return 0, continue to the next rule, or break out of the loop.

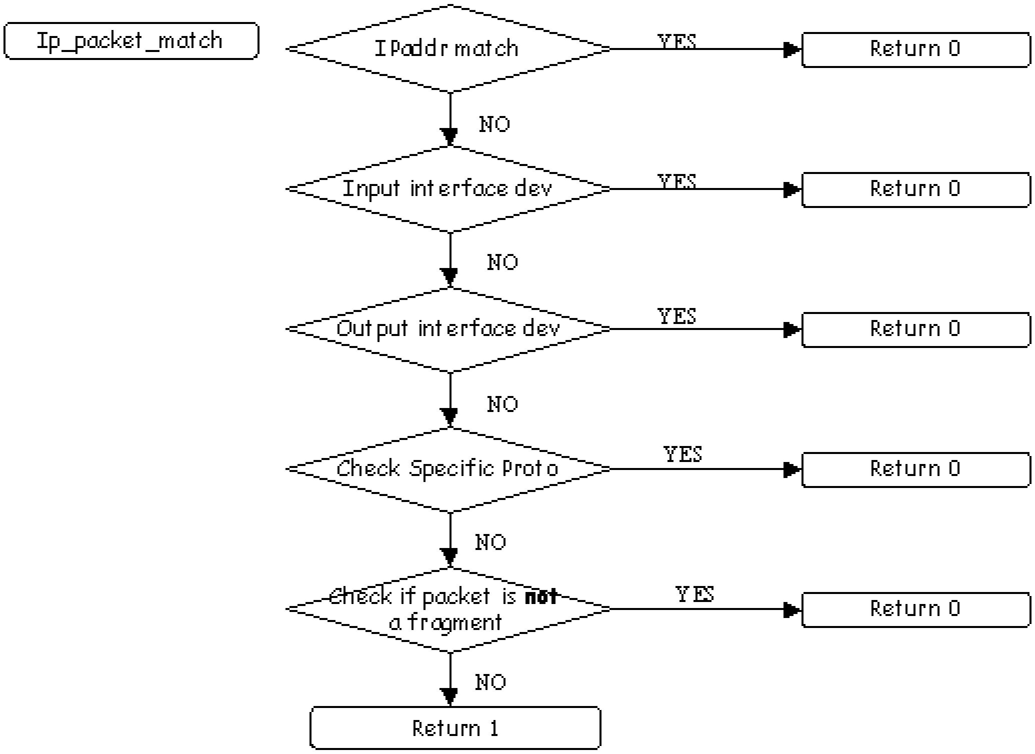


Figure 3-17 ip_packet_match function–IP address are always checked regardless of the type of filtering in the rule-set

After passing through the `ip_packet_match` and finding no matches, the next step is to execute the `IP_MATCH_ITERATE`, in Figure 3-16. Here, the firewall calls the `do_match` function pertaining to the specific rule. Every module has a specific do match function. In other words, if we are filtering/matching a MAC address the `iptables` algorithm will call the `do_match` function specific to MAC addresses. If a match is found, the chain breaks to perform a TARGET check. Targets can be ACCEPT, DROP, QUEUE, STOLEN, REPEAT or "JUMP" to another chain when a chain has been added to the rule-set. If `IP_MATCH_ITERATE` does not find a match it will either continue to the next rule or exit the loop.

Iptables breaks out of the loop when it finds a match, when the packet is a fragment, or when all the rules have been checked. Hotdrop is a variable initialized to zero; when a packet is a fragment the hotdrop variable is changed to a 1, which indicates that the packet should be dropped. A fragment is a malicious packet (e.g. a packet with a TCP header larger or smaller than the standard) and will always be dropped. The VERDICT is a variable that tells the algorithm what to do with the packet (e.g. NF_ACCEPT, NF_DROP).

In summary, the firewall will always go through the `ip_packet_match` function regardless of the type of matching. For example, every rule that filters TCP ports includes checking for IPs, interfaces, protocol, fragments, and at last matching the TCP port.

Chapter 4 Firewall Performance Results

In this chapter we will compare the host's performance with and without the firewall. We will analyze the data in terms of the host's latency for a single packet and also for a stream of packets when they traverse the stack. The analysis of the latency for a single packet will show us the firewall sensitivity to the number of rules, the type of filtering (also referred as the type of matching), and the payload size. For both scenarios, single packets and throughput tests, the latency will show how the total processing time is impacted by the transmission rate. The chapter is divided in two parts:

(1) Results from single packet tests

(2) Results from throughput tests

4.1 Back-to-back timing for the single-packet tests instrumentation

The back-to-back timestamps were placed at the beginning and at the end of the timestamping instrumentation in order to measure the overhead created by it. The back-to-back timestamps were taken using the `rdtscl()` macro. Two tests were performed in which 40 UDP packets were sent to the host, for a total of 80 samples.

The difference between the end time and the beginning time is the instrumentation overhead. This overhead is subtracted from the test results to obtain better estimates. In other words, for example, the total time to process one TCP packet of 64 bytes of payload (i.e. T5-T1) with the instrumentation is 31.86 µs as shown in Table 4-1. This time included the overhead generated by T1, T2, T3, T4, and T5. Subtracting the overhead from the measurements will give a better estimate.

Table 4-1 Packet's latency (including the instrumentation overhead) as it travels the TCP/IP stack

	NOFIREWALL [units: µs]			
TCP	T2 – T1	T3 - T1	T4 - T1	T5 - T1
64 bytes	11.89	14.04	28.35	31.86
1400 bytes	13.49	15.61	39.71	42.92
UDP	T2 – T1	T3 - T1	T4 - T1	T5 - T1
64 bytes	12.11	14.46	24.44	27.74
1400 bytes	13.97	16.35	36.38	40.18

Table 4-1 shows the time that it takes for a packet to travel from the bottom of the stack to any other point in the stack, for example T2-T1 is the time that it takes for a packet to travel from the device driver to the beginning of the firewall. These times include the instrumentation overhead. Now, Table 4-2 shows the results of the back-to-back tests. Here, the Datalink layer's overhead is about 1 µs and the rest of them add a little more than half of a microsecond each. The reason why the overhead of the Datalink is greater than the rest of them is because of a `memcpy()`. This `memcpy()` served to copy the header of the packet into an array that was passed to the `/proc` file system in order compare the headers of each packet and the sequence number. Later, for the throughput tests, we found the `memcpy()` to be unnecessary and it was removed from the instrumentation code.

Table 4-2 Overhead of the single-packet tests instrumentation

	Test 1	Test 2	Average	
Overhead of T1 (OT1) [us]	1.0725	1.0775	1.075	µs
Overhead of T2 (OT2) [us]	0.555	0.555	0.555	µs
Overhead of T3 (OT3) [us]	0.515	0.515	0.515	µs
Overhead of T4 (OT4) [us]	0.5675	0.5675	0.5675	µs
Overhead of T5 (OT5) [us]	---	---	0.5675	µs

Notice that in the Table 4-2, T5 was not included in the back-to-back tests because of an error in our back-to-back instrumentation. This error was found and fixed for the throughput back-to-back tests. However, given that the code for T5 is very similar to that of T2, T3, and T4, we inferred the back-to-back time for T5 is approximately the same as the others. For the purpose of our analysis, we selected 0.5675 (the same value as T4) as a conservative estimate for T5.

The results in Table 4-2 were subtracted from the results in Table 4-1. So, T2-T1 without the instrumentation overhead is equal to: T2-T1 with overhead minus the overhead of T1 (OT1) + the overhead of T2 (OT2). Then, T3-T1 without the overhead is equal to: T3-T1 with overhead minus OT1 + OT2 + OT3, and so on. Thus, the time to process the stack without the instrumentation overhead is shown in Table 4-3.

Table 4-3 Packet's latency (excluding the instrumentation overhead) as it travels the TCP/IP stack

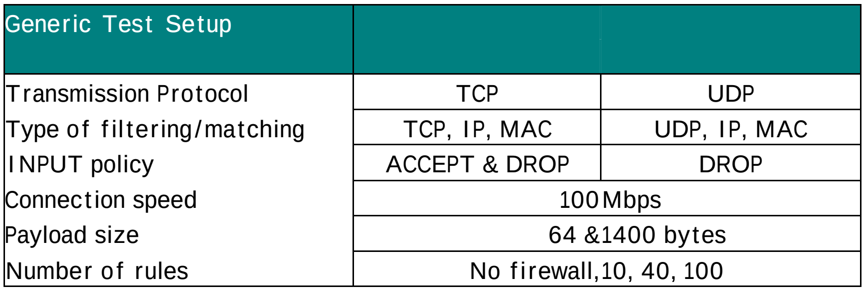
NOFIREWALL [µs]				
TCP	T2 – T1	T3 - T1	T4 - T1	T5 - T1
64 bytes	10.26	11.90	25.63	28.58
1400 bytes	11.86	13.46	37.00	39.64
UDP	T2 - T1	T3 - T1	T4 - T1	T5 - T1
64 bytes	10.48	12.32	21.73	24.46
1400 bytes	12.34	14.21	33.67	36.90

From this point on, all the results shown in the tables exclude the instrumentation overhead.

4.2 Procedures for single-packet tests

The parameters under test, shown in Table 4-4, included the transmission protocol, connection speed, payload size, number of rules, type of filtering, and the INPUT policy.

Table 4-4 Parameters under test



Generic Test Setup		
Transmission Protocol	TCP	UDP
Type of filtering/matching	TCP, IP, MAC	UDP, IP, MAC
INPUT policy	ACCEPT & DROP	DROP
Connection speed	100 Mbps	
Payload size	64 &1400 bytes	
Number of rules	No firewall,10, 40, 100	

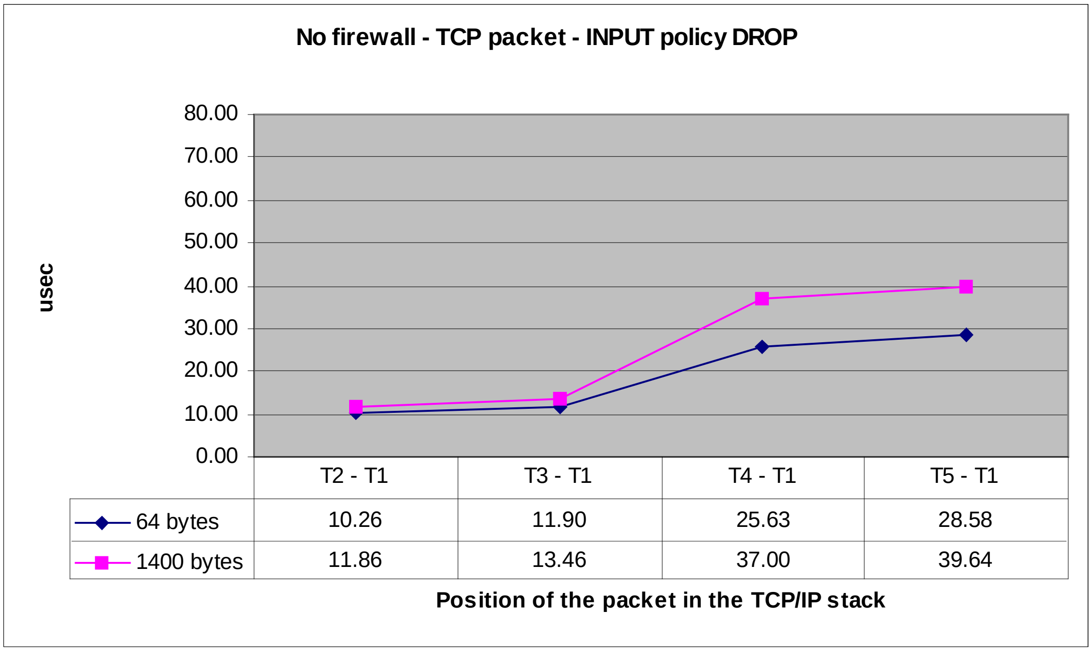
Table 4-4 shows a generic table for the test setup. During the tests neither the server nor the client ran any services. Both machines used a 10/100 Mbps 3Com NIC, model 3C905C. The connection speed was 100 Mbps in an isolated system, sending one packet every 4 seconds. The payload size varied between 64 and 1400 bytes. The type of filtering was IP and MAC addresses for both protocols, and TCP and UDP for each respective transmission protocol. The number of rules under each type of filtering was zero (or No Firewall), 10, 40, and 100 rules. Both INPUT policies ACCEPT and DROP were tested for TCP, but only the INPUT policy DROP was tested for UDP. A total of 40 packets or samples on one single test were sent to the host. The results were accessed via the `/proc/` file system. Three tests were performed for each type of filtering, from which we took the median of the total samples to exclude any outliers. The medians were averaged for a final result.

4.3 Single-packet test results

The results obtained from single-packet tests provided the following information: (a) that the payload size impacts the performance before and after the firewall but not the firewall itself, (b) that the INPUT policy does not affect the performance of the firewall, (c) that the firewall (T3 – T2) is affected only by type of filtering/matching and the number of rules, and (d) that the time to process a packet from T1 to T5 is affected by the parameters in (c) and also by the payload size.

4.3.1 Timing the network stack

The first analysis involved plotting all the measurement data obtained at each layer. Figure 4-1 shows the timestamps of T2 to T5 with respect to T1. The line connecting T2-T1 and T3-T1 represents the time that it takes for the firewall to execute.



	T2 - T1	T3 - T1	T4 - T1	T5 - T1
◆ 64 bytes	10.26	11.90	25.63	28.58
■ 1400 bytes	11.86	13.46	37.00	39.64

Figure 4-1 Latency increases as the payload size increases

From Figure 4-1 and 3-7 we have:

- Start time = T2 – T1
- Firewall = (T3 – T1) – (T2 – T1) = T3 – T2
- TCP layer = (T4 – T1) – (T3 – T1) = T4 – T3
- Socket layer = (T5 – T1) – (T4 – T1) = T5 – T4
- Total processing time = T5 - T1

4.3.2 T2 – T1

The results for TCP and UDP in Table 4-5 show that the difference between T2 – T1 increases as the payload size increases. For example, compare the averages for 64 bytes with the averages for 1400 bytes. The reason for this is because the packet is copied from the network into kernel space.

Table 4-5 Payload impact in T2-T1 – time increases as the payload size increases

T2 - T1 [units: us]							
	TCP				UDP		
	IP	MAC	TCP		IP	MAC	UDP
64 bytes	T2 - T1	T2 - T1	T2 - T1	64 bytes	T2 - T1	T2 - T1	T2 - T1
No firewall	10.26	10.26	10.26	No firewall	10.48	10.48	10.48
10 rules	10.46	10.23	10.54	10 rules	10.45	10.50	10.45
40 rules	10.58	10.59	10.26	40 rules	10.51	10.60	10.61
100 rules	10.72	10.64	10.56	100 rules	10.66	10.55	10.68
Average	10.51	10.43	10.40	Average	10.52	10.53	10.55
1400 bytes				1400 bytes			
No firewall	11.86	11.86	11.86	No firewall	12.34	12.34	12.34
10 rules	12.02	11.90	11.94	10 rules	12.38	12.27	12.36
40 rules	12.05	12.10	11.92	40 rules	12.39	12.43	12.52
100 rules	12.31	12.15	12.30	100 rules	12.42	12.48	12.54
Average	12.06	12.00	12.00	Average	12.38	12.38	12.44

4.3.3 T4 – T3

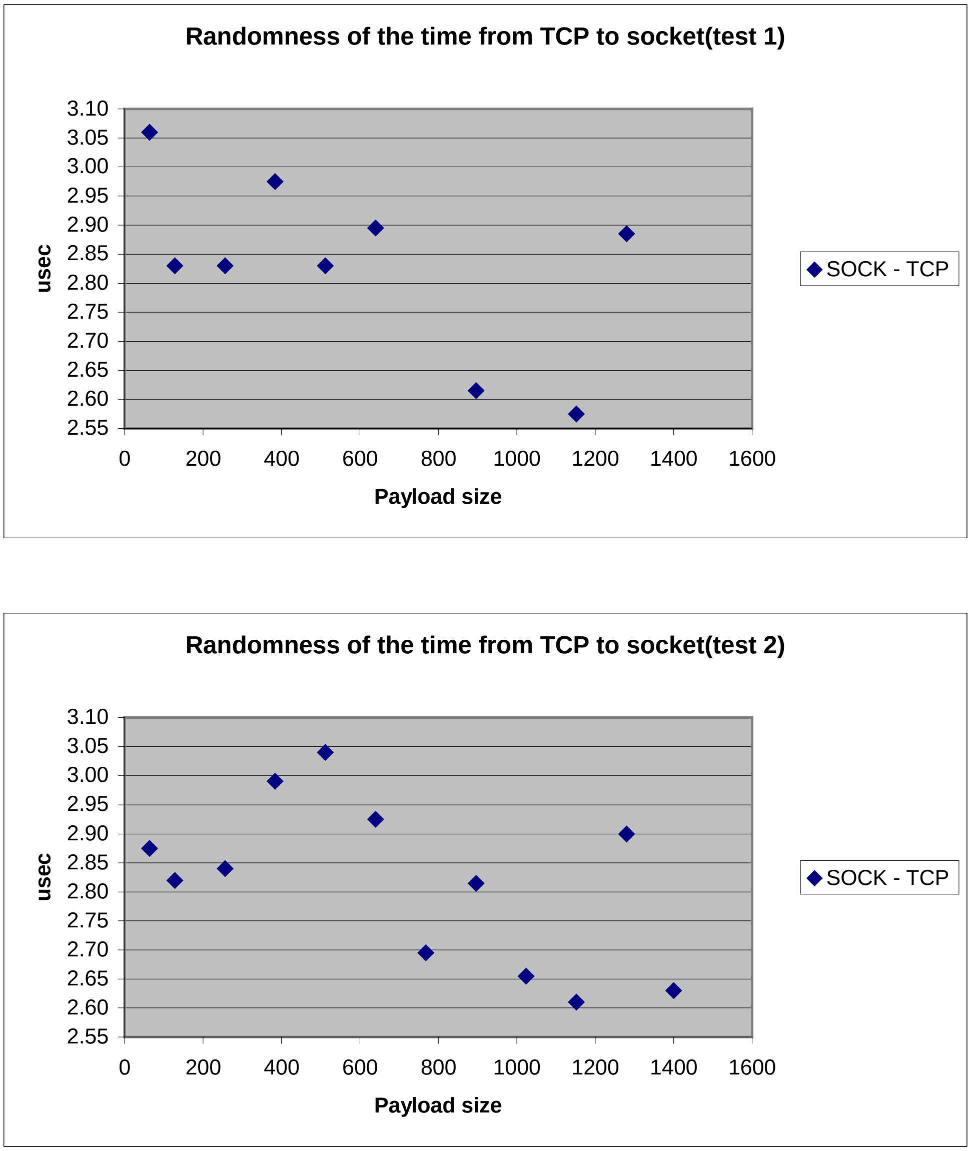
At the TCP and UDP layers, the latter is processed faster than the TCP layer because of the nature of the complexity of their algorithm. However, the time to process the layers is influenced by the payload size because the data is copied from kernel space to user space. For example, the results in Table 4-6 demonstrate that the average time to process 64 bytes of payload is shorter than 1400 bytes of payload.

Table 4-6 Impact of the payload size in T4 – T3 – time increases as the payload size increases

T4 - T3 [units: µs]							
	TCP				UDP		
64 bytes	IP	MAC	UDP	64 bytes	IP	MAC	UDP
No firewall	13.74	13.74	13.74	No firewall	9.42	9.42	9.42
10	13.89	14.14	14.20	10	9.48	9.40	9.42
40	14.32	14.68	14.72	40	9.63	9.95	9.89
100	14.57	15.02	14.81	100	10.09	10.25	10.18
Average	14.13	14.39	14.37	Average	9.65	9.75	9.72
	TCP				UDP		
1400 bytes	IP	MAC	UDP	1400 bytes	IP	MAC	UDP
No firewall	23.54	23.54	23.54	No firewall	19.46	19.46	19.46
10	23.65	24.07	24.22	10	19.46	19.35	19.23
40	24.00	24.86	24.67	40	19.48	19.83	19.75
100	24.38	25.07	24.62	100	19.90	20.02	20.03
Average	23.89	24.38	24.26	Average	19.57	19.66	19.62

4.3.4 T5 – T4

Different from T2 – T1 and T4 – T3, the socket layer is processed at random times. Two tests were performed in order to study the time to process the socket layer with respect to the size of the payload. The tests did not include a firewall. Figure 4-2 shows that the time to process this layer is not dependent on the payload size but given that it is a process controlled by the scheduler, it is executed at a random time.


Randomness of the time from TCP to socket(test 1)

Randomness of the time from TCP to socket(test 2)

Figure 4-2 Randomness at the socket layer –socket layer is called randomly regardless of the number of rules

4.4 INPUT policy ACCEPT vs. DROP

After having analyzed the other layers, we study the sensitivities of the firewall. The analysis of the INPUT policies is done first. Table 4-7 shows the time difference between T5 - T1 for each policy using various types of matching, number of rules and payload size. It becomes evident that the time differences between the T5 – T1 for both policies (i.e. Accept – Drop) are insignificant. Consequently, we believe that the policy has no effect in the performance.

Table 4-7 Difference between INPUT policy ACCEPT and DROP – firewall is not sensitive to the INPUT policy

Number of rules –	(payload size)		[units: µs]
	INPUT policy - [T5 – T1]		
IP matching	Accept	Drop	Acc - Drop
10 rules - (64 bytes)	29.94	29.92	0.02
10 rules - (1400)	41.14	40.77	0.38
40 rules - (64)	34.12	34.00	0.12
40 rules - (1400)	45.00	44.90	0.10
	INPUT policy - [T5 – T1]		
MAC matching	Accept	Drop	Acc - Drop
10 rules - (64)	35.66	35.23	0.43
10 rules - (1400)	47.00	46.57	0.43
40 rules - (64)	57.13	55.41	1.72
40 rules - (1400)	68.67	66.84	1.83
	INPUT policy - [T5 – T1]		
TCP matching	Accept	Drop	Acc - Drop
10 rules - (64)	35.93	36.00	0.07
10 rules - (1400)	47.02	47.30	0.28
40 rules - (64)	54.48	54.73	0.25
40 rules - (1400)	65.66	65.92	0.26

4.5 TCP and UDP Firewall Performance [T3 – T2]

As mentioned at the beginning of the chapter, the type of filtering and the number of rules have a performance impact in the firewall (T3 - T2) but the payload size does not. The tables presented in this section show our findings.

The results are organized as follows:

1. Study of the impact generated by the payload size
2. Study of the impact generated by the number of rules

Note: The results in the tables are from tests that used INPUT policy DROP.

4.5.1 Payload size effect

The tables below (Table 4-8 through Table 4-10) present the results for different types of matching. Inside each table and under *N* rules, there are two different payload sizes, 64 and 1400 bytes. It is clear that the time difference [T3 – T2] between the payload sizes belonging to a specific number of rules is very small. This demonstrates that the payload size does not affect the performance between T2 and T3. Also notice that, as expected, IP matching took less processing time than any other type of matching – refer to the `iptables` algorithm in Chapter 3.

Table 4-8 IP matching for TCP and UDP packets – firewall is not sensitive to the payload size

TCP PACKETS		IP [units: µs]		UDP PACKETS		IP [units: µs]	
10 RULES	T2 - T1	T3 - T1	T3 - T2	10 RULES	T2 - T1	T3 - T1	T3 - T2
64 bytes	10.46	13.10	2.64	64 bytes	10.45	13.59	3.13
1400 bytes	12.02	14.66	2.63	1400 bytes	12.38	15.57	3.18
40 RULES			T3 - T2	40 RULES			T3 - T2
64 bytes	10.58	16.81	6.22	64 bytes	10.51	17.01	6.50
1400 bytes	12.05	18.44	6.38	1400 bytes	12.39	19.12	6.73
100 RULES			T3 - T2	100 RULES			T3 - T2
64 bytes	10.72	24.66	13.94	64 bytes	10.66	24.72	14.06
1400 bytes	12.31	26.27	13.96	1400 bytes	12.42	27.10	14.68

Table 4-9 MAC matching for TCP and UDP packets – firewall is not sensitive to the payload size

TCP PACKETS		MAC [units: µs]		UDP PACKETS		MAC [units: µs]	
10 RULES	T2 - T1	T3 - T1	T3 - T2	10 RULES	T2 - T1	T3 - T1	T3 - T2
64 bytes	10.23	18.19	7.96	64 bytes	10.50	19.59	9.09
1400 bytes	11.90	19.95	8.06	1400 bytes	12.27	21.65	9.38
40 RULES			T3 - T2	40 RULES			T3 - T2
64 bytes	10.59	37.94	27.35	64 bytes	10.60	39.41	28.81
1400 bytes	12.10	39.49	27.40	1400 bytes	12.43	41.75	29.32
100 RULES			T3 - T2	100 RULES			T3 - T2
64 bytes	10.64	80.56	69.92	64 bytes	10.55	80.65	70.09
1400 bytes	12.15	82.08	69.93	1400 bytes	12.48	82.86	70.38

Table 4-10 TCP/UDP ports matching for TCP and UDP packets – firewall is not sensitive to the payload size

TCP PACKETS		TCP [units: µs]		UDP PACKETS		UDP [units: µs]	
10 RULES	T2 - T1	T3 - T1	T3 - T2	10 RULES	T2 - T1	T3 - T1	T3 - T2
64 bytes	10.54	19.04	8.51	64 bytes	10.45	19.16	8.71
1400 bytes	11.94	20.53	8.59	1400 bytes	12.36	21.33	8.97
40 RULES			T3 - T2	40 RULES			T3 - T2
64 bytes	10.26	37.19	26.93	64 bytes	10.61	38.67	28.06
1400 bytes	11.92	38.81	26.89	1400 bytes	12.52	41.30	28.78
100 RULES				100 RULES			T3 - T2
64 bytes	10.56	78.46	67.90	64 bytes	10.68	78.56	67.89
1400 bytes	12.30	80.28	67.98	1400 bytes	12.54	80.69	68.16

Tables 4-8 through 4-10 show matching for 10, 40, and 100 rules for TCP and UDP packets. It is evident that the payload size can be considered negligible for the performance given by T3 – T2.

4.5.2 Number of rules effect

To demonstrate that the number of rules and the type of matching have an effect in the performance of the firewall, the tables presented above are reorganized. Notice, in the Tables 4-11 through 4-13, that as the number of rules increase the difference between T3 – T2 also increases. Subsequently, the number of rules impacts the performance of the firewall.

Notice in the tables that for "No firewall," the results for T3 – T2 is non-zero. This can be explained with the `iptables` algorithm because, as depicted in Figure 3-16, when the `ip_local_deliver` function returns it makes a call to the netfilter hook NF_IP_LOCAL_IN. When a netfilter hook is called, the function `ipt_hook()` is executed. This latter returns a call to `ipt_do_table`. This latter will check the iptables rule-set and if no rules are found, the function will exit `ipt_do_table` and `ipt_hook` and finally make a call to `ip_local_deliver_finish.` This process will take between 1.60 to 1.90 µs.

Table 4-11 Matching IP – time increases as the rules increase

TCP packets		IP matching		UDP packets		IP matching	
		[units: µs]				[units: µs]	
64 bytes	T2 - T1	T3 – T1	T3 - T2	64 bytes	T2 - T1	T3 - T1	T3 – T2
No firewall	10.26	11.90	1.64	No firewall	10.48	12.32	1.84
10 rules	10.46	13.10	2.64	10 rules	10.45	13.59	3.13
40 rules	10.58	16.81	6.22	40 rules	10.51	17.01	6.50
100 rules	10.72	24.66	13.94	100 rules	10.66	24.72	14.06
1400 bytes	T2 - T1	T3 – T1	T3 - T2	1400 bytes	T2 - T1	T3 - T1	T3 - T2
No firewall	11.86	13.46	1.60	No firewall	12.34	14.21	1.87
10 rules	12.02	14.66	2.63	10 rules	12.38	15.57	3.18
40 rules	12.05	18.44	6.38	40 rules	12.39	19.12	6.73
100 rules	12.31	26.27	13.96	100 rules	12.42	27.10	14.68

Table 4-12 Matching MAC addresses – time increases as rules increase

TCP packets		MAC matching [units: µs]		UDP packets		MAC matching [units: µs]	
64 bytes	T2 - T1	T3 – T1	T3 - T2	64 bytes	T2 - T1	T3 - T1	T3 - T2
No firewall	10.26	11.90	1.64	No firewall	10.48	12.32	1.84
10 rules	10.23	18.19	7.96	10 rules	10.50	19.59	9.09
40 rules	10.59	37.94	27.35	40 rules	10.60	39.41	28.81
100 rules	10.64	80.56	69.92	100 rules	10.55	80.65	70.09
1400 bytes	T2 - T1	T3 – T1	T3 - T2	1400 bytes	T2 - T1	T3 - T1	T3 - T2
No firewall	11.86	13.46	1.60	No firewall	12.34	14.21	1.87
10 rules	11.90	19.95	8.06	10 rules	12.27	21.65	9.38
40 rules	12.10	39.49	27.40	40 rules	12.43	41.75	29.32
100 rules	12.15	82.08	69.93	100 rules	12.48	82.86	70.38

Table 4-13 Matching TCP ports – time increases as the rules increase

TCP packets		TCP matching [units: µs]		UDP packets		UDP matching [units: µs]	
64 bytes	T2 - T1	T3 - T1	T3 - T2	64 bytes	T2 - T1	T3 - T1	T3 - T2
No firewall	10.26	11.90	1.64	No firewall	10.48	12.32	1.84
10 rules	10.54	19.04	8.51	10 rules	10.45	19.16	8.71
40 rules	10.26	37.19	26.93	40 rules	10.61	38.67	28.06
100 rules	10.56	78.46	67.90	100 rules	10.68	78.56	67.89
1400 bytes	T2 - T1	T3 - T1	T3 - T2	1400 bytes	T2 - T1	T3 - T1	T3 - T2
No firewall	11.86	13.46	1.60	No firewall	12.34	14.21	1.87
10 rules	11.94	20.53	8.59	10 rules	12.36	21.33	8.97
40 rules	11.92	38.81	26.89	40 rules	12.52	41.30	28.78
100 rules	12.30	80.28	67.98	100 rules	12.54	80.69	68.16

4.5.3 Linear relationship of [T3 – T2]

The plot of the data just presented shows a linear relationship between the performance impact and the number of rules. Figures 4-3 and 4-4 present the T3 – T2 trendlines for TCP and UDP using the data obtained for packets of 64 bytes of payload. They also present a set of equations that may serve to estimate the time to process T3 – T2 up to 100 rules.

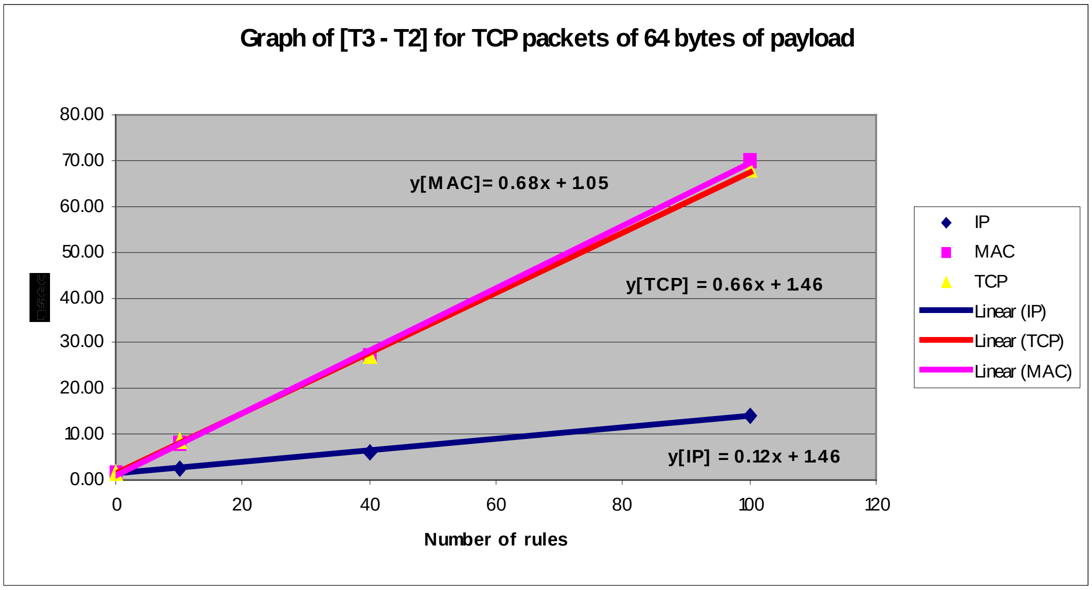


Figure 4-3 TCP connection – [T3 – T2] – linear relationship between the number of rules and the time to process the firewall

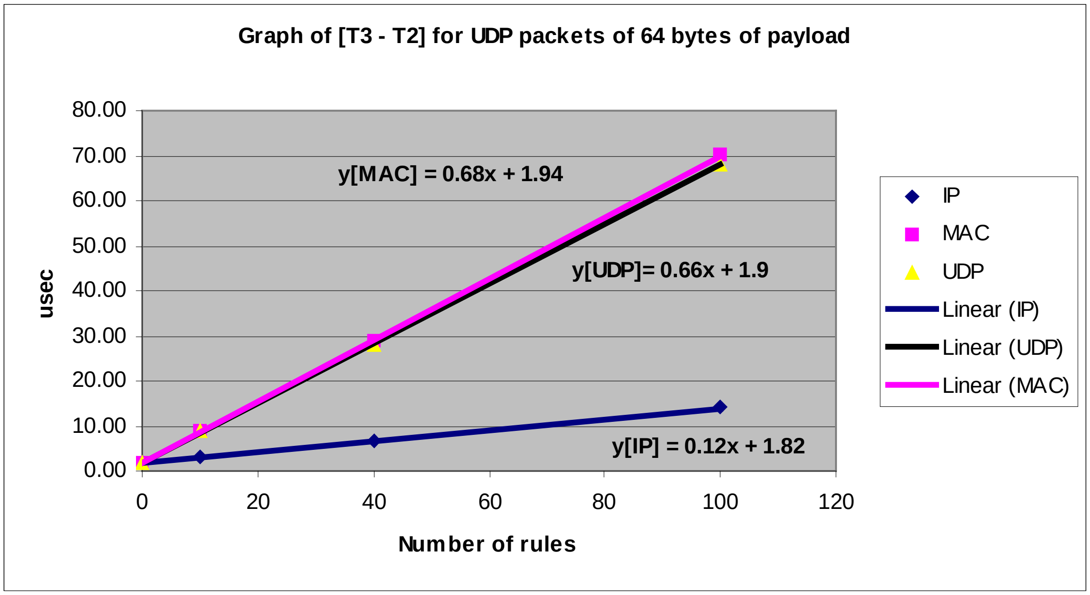


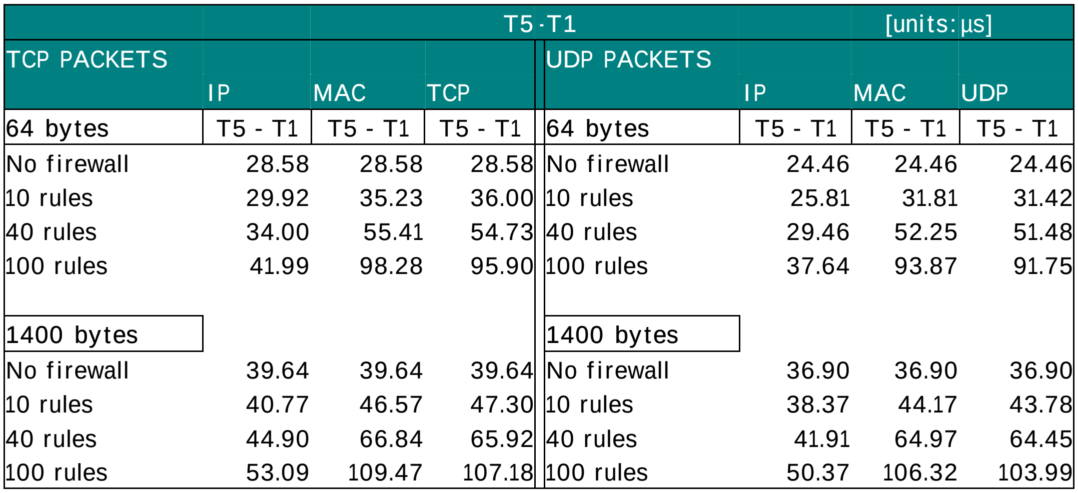
Figure 4-4 UDP connection – [T3 – T2] - linear relationship between the number of rules and the time to process the firewall

Evidently, there is a linear relationship between the number of rules and time to process T3 –T2.

4.6 Total processing time [T5 – T1] with respect to [T3 – T2]

The total processing time (T5 –T1) is expected to be slower for TCP packets than for UDP packets; refer to the T4 – T3 section presented earlier. The results in the Table 4-14 confirm that the number of rules directly affects the total processing time. In addition, it shows that the payload size also impacts the total processing time (e.g. compare “No firewall” for 64 and 1400 bytes.)

Table 4-14 TCP and UDP – Difference in Total processing time [T5 – T1] for three types of filtering rules



			T5 - T1			[units: µs]	
TCP PACKETS	IP	MAC	TCP	UDP PACKETS	IP	MAC	UDP
64 bytes	T5 - T1	T5 - T1	T5 - T1	64 bytes	T5 - T1	T5 - T1	T5 - T1
No firewall	28.58	28.58	28.58	No firewall	24.46	24.46	24.46
10 rules	29.92	35.23	36.00	10 rules	25.81	31.81	31.42
40 rules	34.00	55.41	54.73	40 rules	29.46	52.25	51.48
100 rules	41.99	98.28	95.90	100 rules	37.64	93.87	91.75
1400 bytes				1400 bytes			
No firewall	39.64	39.64	39.64	No firewall	36.90	36.90	36.90
10 rules	40.77	46.57	47.30	10 rules	38.37	44.17	43.78
40 rules	44.90	66.84	65.92	40 rules	41.91	64.97	64.45
100 rules	53.09	109.47	107.18	100 rules	50.37	106.32	103.99

4.6.1 Firewall % overhead with respect to T5 – T1

The impact of T3 – T2 can be expressed in terms of a percentage overhead generated by the firewall over the total processing time (T5 – T1). The results in Tables 4-15 through 4-17 show that: (1) as the number of rules increases the percentage overhead increases up to a 75% for UDP and up to 71% for TCP; on the other hand, (2) as the payload size increases for a specific number of rules, the percentage overhead decreases – this is because the firewall is not sensitive to the payload size, consequently, an increase in payload will increase T5 – T1 while T3 – T2 will remain the same. The percentage overhead was calculated as follows:

Firewall's % overhead = (T3 - T2)/(T5 – T1) * 100

Table 4-15 Percentage overhead of IP matching over the T5 – T1 – overhead increases as the number of rules increase

TCP PACKETS				UDP PACKETS			
IP matching				IP matching			
64 bytes	T5 - T1	T3 - T2	% overhead	64 bytes	T5 - T1	T3 – T2	% overhead
No firewall	28.58	1.64	6%	No firewall	24.46	1.84	8%
10 rules	29.92	2.64	9%	10 rules	25.81	3.13	12%
40 rules	34.00	6.22	18%	40 rules	29.46	6.50	22%
100 rules	41.99	13.94	33%	100 rules	37.64	14.06	37%
1400 bytes				1400 bytes			
No firewall	39.64	1.60	4%	No firewall	36.90	1.87	5%
10 rules	40.77	2.63	6%	10 rules	38.37	3.18	8%
40 rules	44.90	6.38	14%	40 rules	41.91	6.73	16%
100 rules	53.09	13.96	26%	100 rules	50.37	14.68	29%

Table 4-16 Percentage overhead of MAC matching over the T5 – T1 - overhead increases as the number of rules increase

TCP PACKETS				UDP PACKETS			
MAC matching				MAC matching			
64 bytes	T5 - T1	T3 - T2	% overhead	64 bytes	T5 - T1	T3 - T2	% overhead
No firewall	28.58	1.64	6%	No firewall	24.46	1.84	8%
10 rules	35.23	7.96	23%	10 rules	31.81	9.09	29%
40 rules	55.41	27.35	49%	40 rules	52.25	28.81	55%
100 rules	98.28	69.92	71%	100 rules	93.87	70.09	75%
1400 bytes			% overhead	1400 bytes			% overhead
No firewall	39.64	1.60	4%	No firewall	36.90	1.87	5%
10 rules	46.57	8.06	17%	10 rules	44.17	9.38	21%
40 rules	66.84	27.40	41%	40 rules	64.97	29.32	45%
100 rules	109.47	69.93	64%	100 rules	106.32	70.38	66%

Table 4-17 Percentage overhead of TCP matching over T5 – T1 - overhead increases as the number of rules increase

TCP PACKETS				UDP PACKETS			
TCP matching				UDP matching			
64 bytes	T5 - T1	T3 - T2	% overhead	64 bytes	T5 - T1	T3 - T2	% overhead
No firewall	28.58	1.64	6%	No firewall	24.46	1.84	8%
10 rules	36.00	8.51	24%	10 rules	31.81	8.71	27%
40 rules	54.73	26.93	49%	40 rules	52.25	28.06	54%
100 rules	95.90	67.90	71%	100 rules	93.87	67.89	72%
1400 bytes			% overhead	1400 bytes			% overhead
No firewall	39.64	1.60	4%	No firewall	36.90	1.87	5%
10 rules	47.30	8.59	18%	10 rules	44.17	8.97	20%
40 rules	65.92	26.89	41%	40 rules	64.97	28.78	44%
100 rules	107.18	67.98	63%	100 rules	106.32	68.16	64%

4.7 Latency results for various throughputs

4.7.1 Test procedures

The tests were performed using the SmartBits network tester. Because of the limitation of the system, we only tested UDP packets. The SmartBits was configured to transmit packets to the DUT for one minute before the timestamps were taken. The SmartBits' "Smart Window" application automatically showed how many packets would be transmitted in one minute. The DUT kept the count of the number of incoming packets until a minute had elapsed. A total of 4000 timestamps were stored in the memory buffers. The median of 3 tests with 4000 measurements each was calculated for a final result. The parameters for each test are shown in the Table 4-18.

Table 4-18 Parameters under test

Generic Test Setup	
Transmission Protocol	UDP
Type of filtering/matching	IP, MAC
INPUT policy	DROP
Throughput / transmission rates	5 & 10 Mbps
Payload size	64 bytes
Number of rules	No firewall & 100

The tests were performed only for two types of filtering, IP addresses and MAC addresses. The number of rules used was zero (or No firewall) and 100. Even though the hardware supported a 100 Mbps throughput, only tests for 5 and 10 Mbps were performed because the timestamping instrumentation made the DUT loose interrupts. When our instrumentation was loaded and 100 rules filtering MAC addresses were added to the rule-set, a link utilization higher than 12% (i.e. 12 Mbps) would cause loss of interrupts. Tests were performed without the instrumentation and 100% link utilization (i.e. 100 Mbps) could be reached without any loss of interrupts even when filtering 100 MAC addresses. This latter test is very important because it eliminates the possibility that the firewall is the cause of the interrupt loss but that the problem is caused by the instrumentation.

4.7.2 Back-to-back timing measurements for throughput tests

As described in detail in Chapter 3 some changes were made to the instrumentation. So, as it was done for the single-packet tests, new back-to-back measurements were taken and subtracted from the measurement results to obtain better estimates. Notice in the results in the Table 4-19 that the overhead produced by T1 is 0.36 µs compared to the 1 µs overhead obtained for the single-packet back-to-back tests shown in the Table 4-2; this is because of the changes made to the `dev.c` file.

Table 4-19 Instrumentation overhead

	Test 1	Test 2	Average
Overhead of T1 (OT1) [us]	0.37	0.3625	0.3625
Overhead of T2 (OT2) [us]	0.4625	0.4625	0.4625
Overhead of T3 (OT3) [us]	0.4675	0.4775	0.4775
Overhead of T4 (OT4) [us]	0.6075	0.6025	0.6025
Overhead of T5 (OT5) [us]	0.545	0.545	0.545

Notice that this time we were able to measure the overhead of T5. As mentioned earlier, the error found for the single-packet back-to-back tests for T5 was fixed; this allowed us to measure the overhead of T5 for the instrumentation used in the throughput tests.

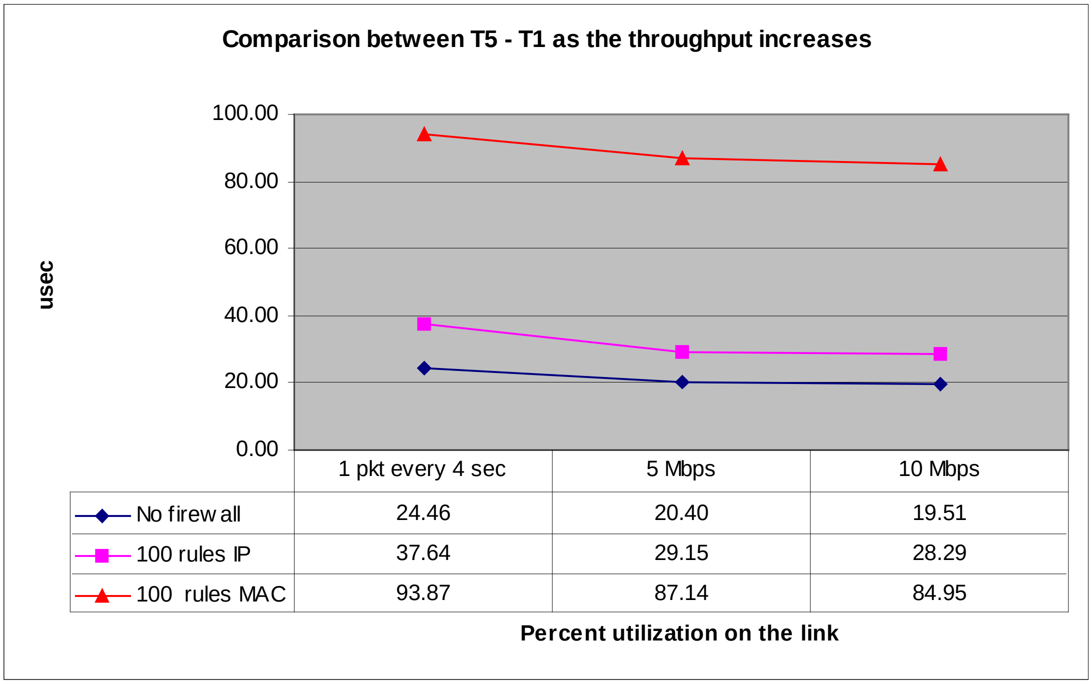
4.7.3 Test results

The results obtained using the SmartBits tool show in the Table 4-20 that as the throughput increases, the latency decreases. In other words, the faster the transmission rate, the faster the packet is processed in the stack.

Table 4-20 Difference in the latency for various throughput – latency decreases as the throughput increases

SINGLE-PACKET every 4 seconds [units: us]				
64 BYTES	T2 – T1	T3 - T1	T4 - T1	T5 - T1
No firewall	10.48	12.32	21.73	24.46
100 rules IP	10.66	24.72	34.81	37.64
100 rules MAC	10.55	80.65	90.90	93.87
MULTIPLE PACKETS – 5 Mbps				
64 BYTES	T2 – T1	T3 - T1	T4 - T1	T5 - T1
No firewall	11.26	12.46	18.52	20.40
100 rules IP	11.61	20.27	27.07	29.15
100 rules MAC	12.35	77.08	84.76	87.14
MULTIPLE PACKETS – 10 Mbps				
64 BYTES	T2 – T1	T3 - T1	T4 - T1	T5 - T1
No firewall	11.06	12.15	17.67	19.51
100 rules IP	11.84	20.03	26.27	28.29
100 rules MAC	12.03	76.30	82.66	84.95

Notice in the Figure 4-5 that the single-packet tests show the highest latency between T5-T1. On the other hand, the smallest latency (i.e. faster processing time) is when the throughput is 10 Mbps. Consequently, this data shows that the single-packet measurement results may serve as a conservative upper bound to estimate the time to process the packets by the stack.



	1 pkt every 4 sec	5 Mbps	10 Mbps
◆ No firewall	24.46	20.40	19.51
■ 100 rules IP	37.64	29.15	28.29
▲ 100 rules MAC	93.87	87.14	84.95

Figure 4-5 Comparison between T5-T1 for different transmission rates – latency decreases as the throughput increases

By breaking up the stack into layers, the results in Table 4-21 show the time that a packet is held by the firewall, the UDP layer, and the Socket layer. Notice in the table that between T3-T2 (i.e. the firewall) and T4-T3 (i.e. the UDP layer) the packet is processed faster as the throughput increases. On the other hand, this is not the case for T5-T4 (i.e. the socket layer) where the time to process this layer is random, lying between 2 and 3 μs regardless of the throughput.

Table 4-21 Time that a packet is held on each layer

SINGLE-PACKET every 4 seconds	[units: us]		
64 BYTES	T3 – T2	T4 - T3	T5 - T4
No firewall	1.84	9.42	2.73
100 rules IP	14.06	10.09	2.83
100 rules MAC	70.09	10.25	2.97
MULTIPLE PACKETS	5 Mbps		
64 BYTES	T3 - T2	T4 - T3	T5 - T4
No firewall	1.20	6.06	1.88
100 rules IP	8.67	6.80	2.08
100 rules MAC	64.73	7.69	2.37
MULTIPLE PACKETS	10 Mbps		
64 BYTES	T3 - T2	T4 - T3	T5 - T4
No firewall	1.09	5.52	1.85
100 rules IP	8.19	6.24	2.02
100 rules MAC	64.28	6.35	2.30

Chapter 5 Conclusion and Future Work

5.1 Summary

The goal of this research was to study the sensitivities and the performance impact of the Linux firewall `iptables` in a host. We placed timestamps throughout the TCP/IP stack of a host PC running Linux version 2.4.7. With each timestamp, we looked at the latency of a packet as it traversed the entire network stack. To collect accurate data from our instrumentation, we analyzed the path that an incoming packet follows in the stack.

The purpose of the single-packet tests was to find the sensitivities of the firewall. The results obtained showed the following:

(1) That the firewall is not sensitive to the transmission protocol (i.e. TCP or UDP), the INPUT policy, or the payload size. However, we found that the transmission protocol and the payload size impact the host's network stack.

(2) We found the firewall to be sensitive to the type of filtering and the number of rules. When filtering IP addresses, TCP/UDP ports, and MAC addresses the cost per rule increases linearly and its cost is approximately 0.12, 0.66, and 0.68 μs/rule, respectively. We were able to explain the difference in the performance cost between IP and the other types of filtering through the `iptables` algorithm. Also, our results showed that the percentage overhead generated by a firewall when a single packet of 64 bytes of payload travels the TCP/IP stack, and for a rule-set of zero and 100 rules, ranges from 6% to up to 75%, respectively.

We performed throughput tests for 5 and 10 Mbps with the instrumentation, and for 100 Mbps without the instrumentation. The results were surprising because we did not expect to see a decrease in the latency for higher throughput, neither did we expected to be able to perform a 100% link utilization (i.e. 100 Mbps throughput) without any interrupt loss. The performance measurements obtained in the 5 and 10 Mbps tests demonstrated that the single-packet test results hold to be valid conservative estimates, and that they can serve as an upper bound to estimate the overhead generated by the firewall. The 100 Mbps tests showed that there is no interrupt loss for a firewall with 100 rules filtering MAC addresses. From this, we infer that the firewall does not affect the protocol throughput.

Finally, as mentioned in the Introduction, according to 3Com, a third party vendor discovered that after 30 rules a firewall degraded the performance of a system tremendously. Our results have proved a steady increase in performance overhead as the number of rules increase, proving that their data does not pertain to the `iptables` netfilter.

5.2 Possible future work

- Our studies have focused only on 4 types of `iptables` matches (i.e. IP, MAC, TCP and UDP.) Future work could expand on this by testing the performance of other types of matches. Also, it would be interesting to compare the performance of commercial firewalls versus the open source firewall `iptables.`

- Some problems were found in the instrumentation. When loaded, as the throughput and number of rules in the iptables rule-set increased, the timestamp instrumentation caused the kernel network device driver to lose interrupts. On the other hand, without the timestamp instrumentation and with 100 rules filtering MAC addresses a 100% utilization in the link (i.e. 100 Mbps) could be reached without any packet loss. Therefore, the instrumentation must be debugged to support higher throughputs.
- Test the performance of TCP packets. We were not able to perform this tests because of our timestamp instrumentation and the SmartBits tester only allows us to control the flow of UDP packets.
- Analyze and compare the performance of the netfilter when it is used as a firewall router.
- The `iptables` netfilter has been ported to the CiNIC architecture for the kernel 2.4.3. In order to be able to compare the results presented in this document, the CiNIC should be upgraded to the 2.4.7 version of the Linux kernel, and then compare the tests results.
- A feature not yet supported by firewalls but mentioned by some experts [31], is to design a firewall that filters the payload data inside packets. Filtering the data inside the packet could serve to prevent packets carrying worms or viruses. Future research can be done to study this matter.

Bibliography

[1] Alan O. Freir, Philip Karlton, Paul C. Kocher. November 1996. "The SSL Protocol version 3.0" Internet-draft. <http://wp.netscape.com/eng/ssl3/draft302.txt>. Accessed February 2002.

[2] T. Dierks, C. Allen. January 1999. RFC 2246: "The TLS Protocol version 1.0."

[3] M. Elkins. October 1996. RFC 2015: "MIME Security with Pretty Good Privacy (PGP)."

[4] Charles Kolodgy, Roseann Day, Christian A. Christiansen, and John Daly. May 2001. "Data and Network Integrity (DNI) Technology to Invoke Trust in IT – The Tripwire Solution." <http://www.tripwire.com>

[5] Scambray, McClure, Kurtz. 2001. Hacking Exposed 2nd Edition. Berkeley, California.

[6] Computer Security Institute and Federal Bureau of Investigation. March 2000. "2000 CSI/FBI Computer Crime and Security Survey." Computer Security Institute publication.

[7] Richard D. Pethia. November 2000. "Bugs in Programs." SIGSOFT Foundations of Software Engineering.

[8] Nicole LaRock Decker. November 2000. "Buffer Overflows: Why, How and Prevention." Information Security Reading Room, SANS Institute. <http://rr.sans.org/threats/buffer_overflow.php>

[9] David Larochelle and David Evans. August 2001. "Statically Detecting Likely Buffer Overflow Vulnerabilities." Proceedings of the 10th USENIX Security Symposium.

[10] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. January 1998. "Automatic Detection and Prevention of Buffer-Overflow Attacks." 7th USENIX Security Symposium.

[11] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. 1993. “Efficient Software-Based Fault Isolation.” 14th ACM Symposium on Operating Systems Principles.

[12] David Evans. May 1996. “Static Detection of Dynamic Memory Errors.” SIGPLAN Conference on Programming Language Design and Implementation.

[13] David Larochelle and David Evans. August 2001. “Statically Detecting Likely Buffer Overflow Vulnerabilities.” Proceedings of the 10th USENIX Security Symposium.

[14] Rita C. Summers. 1997. “Secure Computing Threats and Safeguards.” McGraw-Hill.

[15] Allen Householder, Kevin Houle, and Chad Dougherty. January 2002. Security & Privacy: “Computer Attack Trends Challenge Internet Security.” IEEE Computer Society.

[16] David Moore, Geoffrey Voelker, Stefan Savage. August 2001. “Inferring Internet Denial-of-Service Activity,” University of California, San Diego. Proceedings of 10th USENIX Security Symposium.

[17] Drew Dean, Adam Stubblefield. August 2001. “Using client puzzles to protect TLS.” Proceedings of the 10th USENIX Security Symposium.

[18] [Anonymous]. (n.d.). “Overview of Scans and DDoS Attacks.” <http://www.nipc.gov/ddos.pdf>. Accessed 2002.

[19] B. Clifford Neuman and Theodore Ts’o. September 1994. “Kerberos: An Authentication Service for Computer Networks.” Institute of Electrical and Electronics Engineers. ISI/RS-94-399.

[20] [Anonymous]. Open Socket Secure Layer. <http://www.openssl.org>. Accessed 2002.

[21] Richard A. Kemmerer and Giovani Vigna. January 2002. Security & Privacy: “Intrusion Detection: A Brief History and Overview.” IEEE Computer Society.

[22] Rusty Russell. February 2002. “Linux netfilter Hacking HOWTO.” <http://netfilter.samba.org/documentation/HOWTO//netfilter-hacking-HOWTO.html>.

[23] [Anonymous]. (n.d.) Encyclopedia dedicated to computer technology. <http://www.webopedia.com>

[24] Robert Zalenski. February/March 2002. "Firewall Technologies." IEEE Potentials p 24-27.

[25] [Anonymous]. (n.d.) http://www-900.ibm.com/developerWoryks/education/linux/l-fw/tutorial_eng/l-fw-4-1.shtml. Accessed February 2002.

[26] McAffee Corportatoin. McAffee website. <http://www.mcaffee.com>. Accessed 2002.

[27] Samuel Patton, David Doss, William Yurcik. 2000. "Open Source versus Commercial Firewalls: Functional Comparison." Proceedings of the 25th Annual IEEE Conference on Local Computer Networks (LCN '00).

[28] Christoph L. Schuba. 1997. "A Reference Model for Firewall Technology." Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC '97).

[29] James Fischer. "CiNIC – Calpoly Intelligent NIC," California Polytechnic State University, Master's Thesis, San Luis Obispo.

[30] D. Newman. August 1999. RFC 2647: "Benchmarking Terminology for Firewall Performance."

[31] James P. Anderson, Sheila Brand, Li Gong, Thomas Haiigh. September/October 1997. "Firewalls: An Expert Roundtable." September/October 1997. IEEE Software Magazine Vol 14, No. 5: 60-66.

[32] Giovanni Vigna. (n.d.). "A formal Model for Firewall Testing." Dipartamento di Elettronica Politecnico di Milano.

[33] Alian Mayer, Avishai Wool, and Elisha Ziskind. 2000. "Fang: A Firewall Analysis Engine." Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000).

[34] Scott Hazelhurst, Adi Attar, Raymond Sinnapan. June 25 - 28, 2000. "Algorithms for improving the Dependability of Firewall and Filter Rule Lists." International Conference on Dependable Systems and Networks (DSN 2000).

[35] Michael R. Lyu, Lorrien K.Y. Lau. 2000. "Firewall Security: Policies, Testing and Performance." Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC '00).

[36] Molitor, Andrew. (n.d.). "Measuring Firewall Performance." Network Systems Corporation. <http://web.ranum.com/pubs/fwperf/molitor.htm>.

[37] E Testing Labs. October 2001. "P-Cube SE1000: Layer 4 and Layer 7 Performance Tests." <http://www.etestinglabs.com/main/reports/pcube10_01.pdf>. Accessed May 2002.

[38] [Anonymous]. July 1999. Network World Fusion: "Performance tests turn up big differences." <http://www.nwfusion.com/reviews/0719fireperf.html>. Accessed April 2002.

[39] [Anonymous]. (n.d.). "Iptables Performance." <http://industrial-linux.org/mlug/2001-10-13/iptables_thruput.txt>. Accessed May 2002.

[40] Peter Xie. June 1999. "Network Protocol Performance Evaluation of IPv6 for Windows NT." California Polytechnic State University, San Luis Obispo. 5 p.

[41] W. Richard Stevens. 1998. UNIX Network Programming. Network APIs: Sockets and XTI, Volume 1. Elementary TCP Sockets. Upper Saddle River, NJ. 86 p.

[42] Glen Herrin. May 2000. "Linux IP Networking. A guide to the Implementation and Modification of the Linux Protocol Stack." <http://www.cs.unh.edu/cnrg/gherrin/linux-net.html>. Accessed May 2002.

[43] D. P. Bovet and M. Cesati. 2001. "Understanding the Linux Kernel." O'Reilly. ISBN 0-596-00002-02.

Appendix A Performing tests in the co-host

Testing the firewall on the EBSA285 board will serve to find out how the performance in a co-host scales in comparison to the host. We were unable to benchmark the co-host, however, we build the software platform to perform the tests. Bellow we describe the hardware and software implementation on the host.

A.1 Hardware

Our DUT is Sextans, an EBSA285 board with a StrongArm (SA-110) Intel chip, see Figure A-2. The board is connected via a serial port to Hydra, the client. We connect Sextans to Hydra via the same 3Com switch used to make the tests for the host, see Figure A-1.

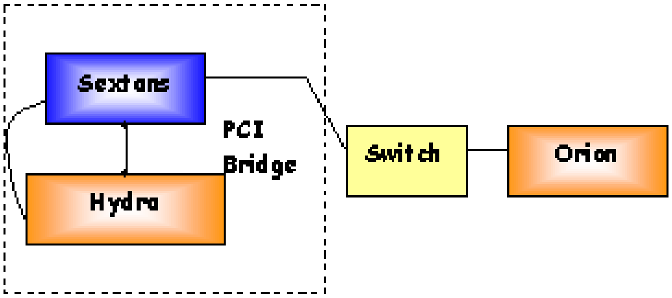


Figure A-1 Co-host test setup

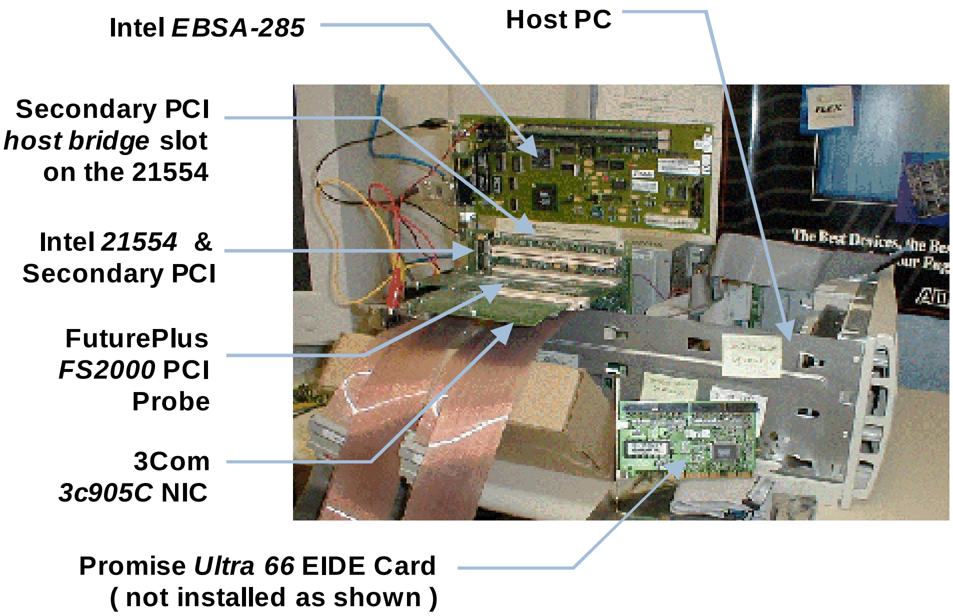

Intel *EBSA-285*
Host PC
Secondary PCI *host bridge* slot on the 21554
Intel *21554* & Secondary PCI
FuturePlus *FS2000* PCI Probe
3Com *3c905C* NIC
Promise *Ultra 66* EIDE Card (not installed as shown)

Figure A-2 CiNIC Architecture [22]

A.2 Software in the co-host

After having problems compiling a 2.4.7 version of the kernel for the co-host, we decided to use a 2.4.3-rmk2-bpa-jdf version of the kernel. To install iptables to the co-host's sources we first took the original sources for the 2.4.3 kernel and applied the iptables patch. Then, we applied Russell King's patch (i.e. rmk2), then the Big Physical Area (i.e. pgh) patch, and finally Jim Fisher's patch.

Usually we log-in to Sextans using a Telnet session, however, in order to avoid any traffic in the connection (just as we did for single-packet tests) we connect the serial port to a different PC and use a console to maintain communication with the CiNIC [29]. We tested the firewall only with the INPUT policy set to ACCEPT because otherwise, if the firewall's INPUT policy is DROP, the firewall will block all the ports (even the serial port) and we will not be able to log in. You may ask if the INPUT policy make a difference to the sensitivity of the firewall? As we saw earlier in the first section of Chapter 4, the INPUT policy does not make any difference. The test procedures are the same as the one explained in Appendix C.

A.3 Timestamp implementation on the EBSA21285

A.3.1 Clocks

The StrongArm SA-110 microprocessor does not have an internal clock fulfilling the same functionality that Time Stamp Counter of the x86 processor architecture. It, however, operates at any one of 16 core clock frequencies but its maximum frequency of operation is limited by the speed of the core clock of the EBSA-285 [33]. Thus, the maximum frequency of the EBSA-285 core clock is 233 MHz.

A.3.2 Timers

Since the SA-110 does not have an internal clock we use a timer Control register. The timer register should be able to provide us with accuracy in the microseconds. We found that the EBSA-285 has four 24-bit timers “that can be preloaded and either free-run, or decremented to zero and then reloaded [34].” In other words, we can use one of the EBSA’s timers to perform our measurements since we can control the start and stop times.

The timer block diagram for the EBSA-285 is shown in Figure A-3 [34]:

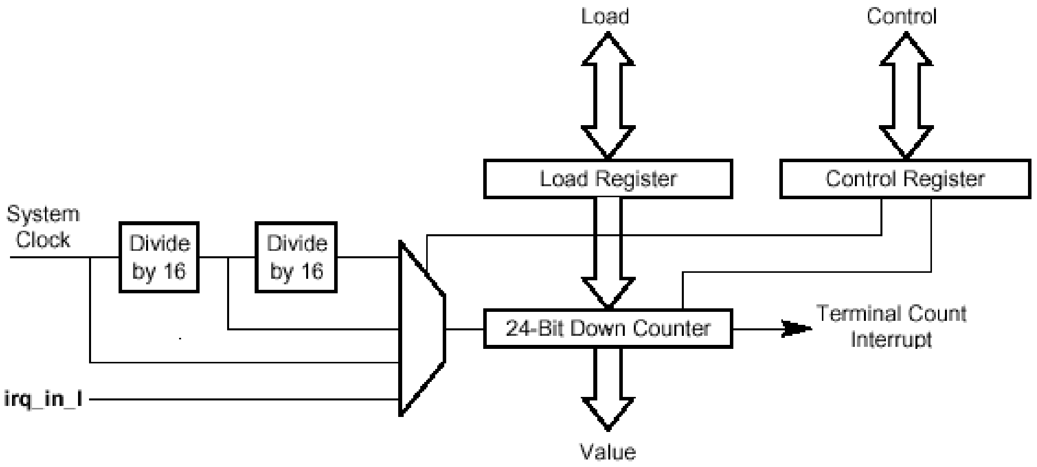


Figure A-3 EBSA-21285 Timer Block Diagram

The 21285 Core Logic Data Sheet [34] says that the four timers can be clocked in four different ways:

- fclk_in: 50 MHz
- fclk_in divided by 16
- fclk_in divided by 256
- External input: 3.6874 MHz

In order to obtain precise measurements we need at least microseconds resolution. Then, we must calculate the precision that a 24-bit register can provide as well as the roll-over time for we do not want the timer to roll over while the packet in passing through the stack.

The resolution can be calculated as follows:

CLOCKING 1: Inverse of (Input Frequency) = 1/(50 MHz) = 0.02 µs

CLOCKING 2: Inverse of (Input Frequency div 16) = 1/(50 MHz / 16) = 0.32 µs

CLOCKING 3: Inverse of (Input Frequency div 16) = 1/(50 MHz / 256) = 5.12 µs

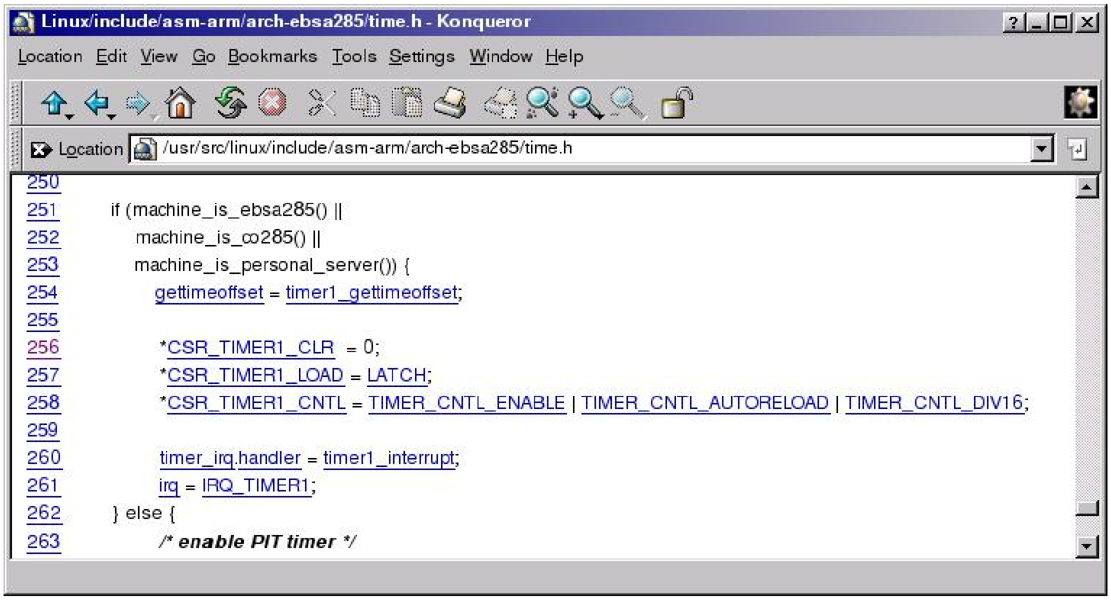
The rollover time is obtained by:

CLOCKING 1: Resolution * 2^{24} = 0.02 µs * 2^{24} = 0.34 seconds

CLOCKING 2: Resolution * 2^{24} = 0.32 µs * 2^{24} = 5.36 seconds

CLOCKING 3: Resolution * 2^{24} = 5.12 µs * 2^{24} = 85.9 seconds

We timed to a microsecond resolution by dividing the input frequency by 16, and as a matter of fact, that is what the kernel uses to control the number of jiffies for the EBSA. See Figure A-4.

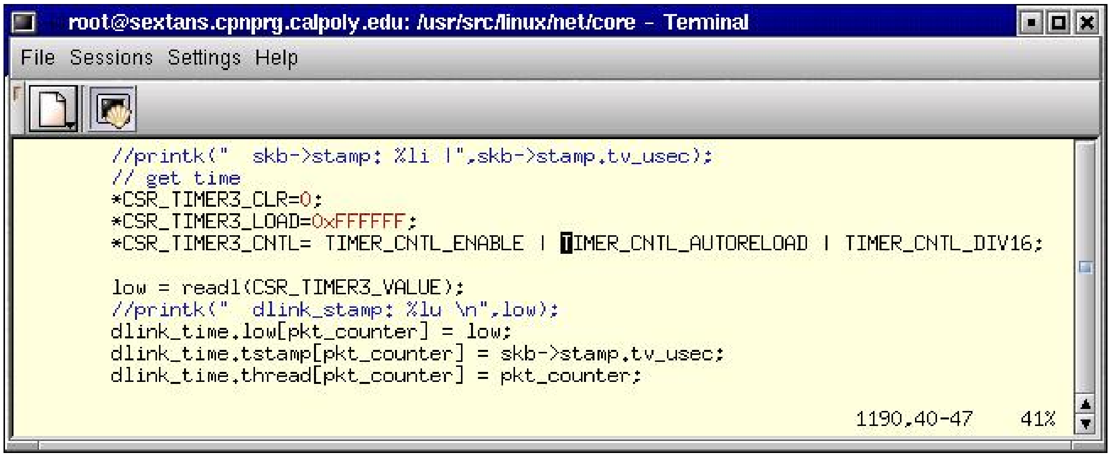

Linux/include/asm-arm/arch-ebsa285/time.h - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Location /usr/src/linux/include/asm-arm/arch-ebsa285/time.h

```
        if (machine_is_ebsa285() ||
            machine_is_co285() ||
            machine_is_personal_server()) {
                gettimeoffset = timer1_gettimeoffset;

                *CSR_TIMER1_CLR  = 0;
                *CSR_TIMER1_LOAD = LATCH;
                *CSR_TIMER1_CNTL = TIMER_CNTL_ENABLE | TIMER_CNTL_AUTORELOAD | TIMER_CNTL_DIV16;

                timer_irq.handler = timer1_interrupt;
                irq = IRQ_TIMER1;
        } else {
                /* enable PIT timer */
```

Figure A-4 ARM Linux use to control of timers

A.3.3 Controlling the time registers

The ARM Linux kernel provides a very simple way to access the Timer Control and Status registers. Figure A-5 shows how to control the third timer (i.e. TIMER3) to take timing measurements in the *dev.c* file.

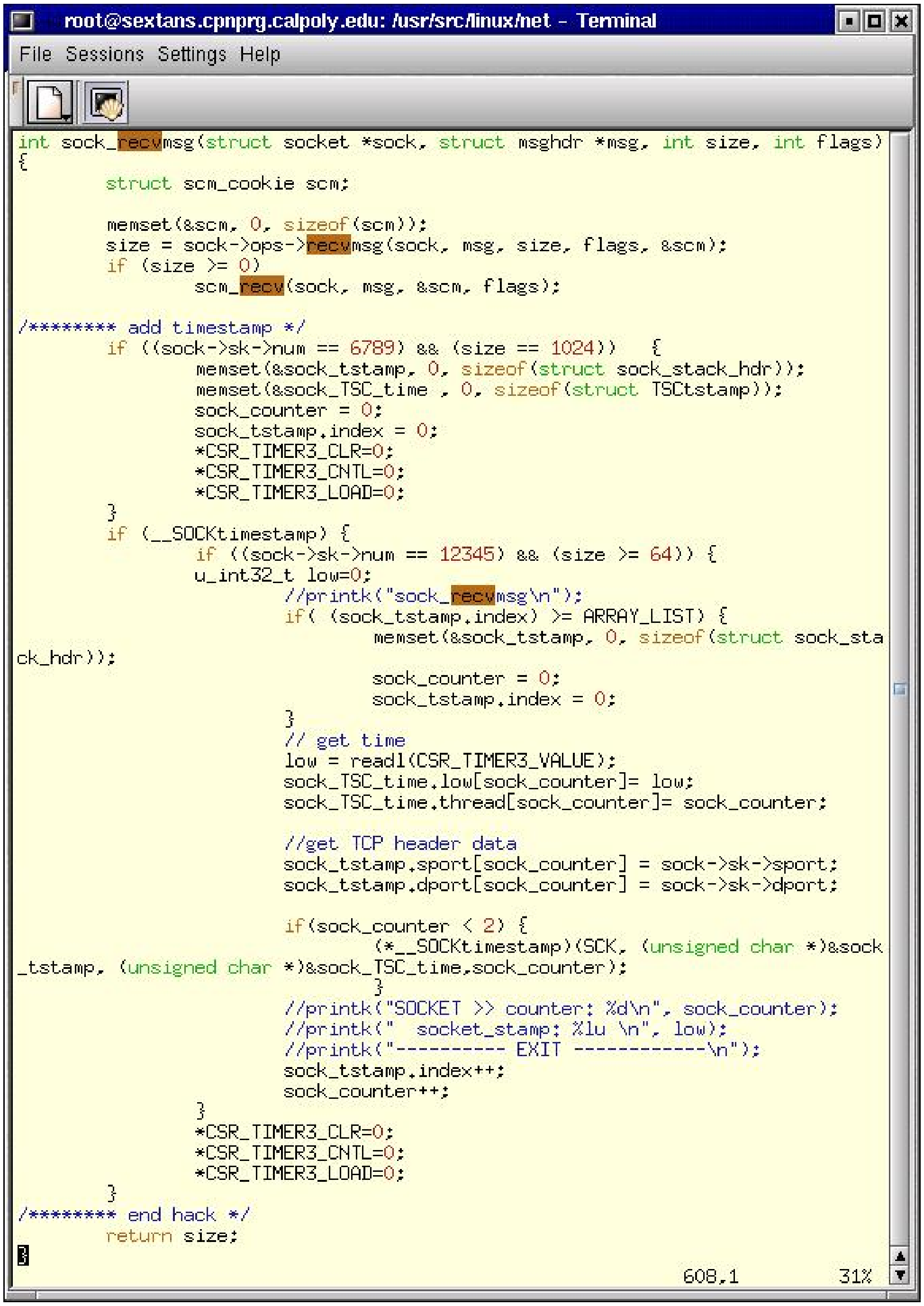

root@sextans.cpnprg.calpoly.edu: /usr/src/linux/net/core - Terminal
File Sessions Settings Help

```
        //printk("  skb->stamp: %li |",skb->stamp.tv_usec);
        // get time
        *CSR_TIMER3_CLR=0;
        *CSR_TIMER3_LOAD=0xFFFFFF;
        *CSR_TIMER3_CNTL= TIMER_CNTL_ENABLE | TIMER_CNTL_AUTORELOAD | TIMER_CNTL_DIV16;

        low = readl(CSR_TIMER3_VALUE);
        //printk("  dlink_stamp: %lu \n",low);
        dlink_time.low[pkt_counter] = low;
        dlink_time.tstamp[pkt_counter] = skb->stamp.tv_usec;
        dlink_time.thread[pkt_counter] = pkt_counter;
                                                            1190,40-47     41%
```

Figure A-5 Using Timer Control Registers in the EBSA285

First of all we clear/reset the register using the `*CSR_TIMER3_CLR=0`. Timers usually decrement, therefore, we have to load the 24 bit timer `*CSR_TIMER3_LOAD=0xFFFFFF`. The `*CSR_TIMER3_CNTL` controls the timer, sets the bits to autoreload, and set the bits that divide the clock by 16 in order to give us a microsecond resolution. The timers are read by using the `readl()` macro.

Given that the timer will reset after 5.32 seconds, it is best to reset the timer at the socket layer. Figure A-6 shows how at the socket layer we read the timer and after all the data has been stored we reset and stop it.

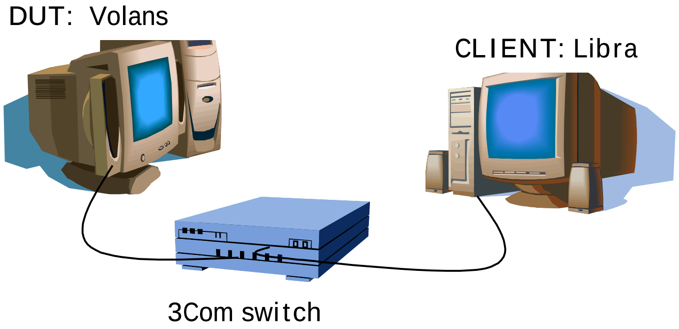

root@sextans.cpnprg.calpoly.edu: /usr/src/linux/net - Terminal
File Sessions Settings Help

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
{
        struct scm_cookie scm;

        memset(&scm, 0, sizeof(scm));
        size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
        if (size >= 0)
                scm_recv(sock, msg, &scm, flags);

/******** add timestamp */
        if ((sock->sk->num == 6789) && (size == 1024))   {
                memset(&sock_tstamp, 0, sizeof(struct sock_stack_hdr));
                memset(&sock_TSC_time , 0, sizeof(struct TSCtstamp));
                sock_counter = 0;
                sock_tstamp.index = 0;
                *CSR_TIMER3_CLR=0;
                *CSR_TIMER3_CNTL=0;
                *CSR_TIMER3_LOAD=0;
        }
        if (__SOCKtimestamp) {
                if ((sock->sk->num == 12345) && (size >= 64)) {
                u_int32_t low=0;
                        //printk("sock_recvmsg\n");
                        if( (sock_tstamp.index) >= ARRAY_LIST) {
                                memset(&sock_tstamp, 0, sizeof(struct sock_sta
ck_hdr));

                                sock_counter = 0;
                                sock_tstamp.index = 0;
                        }
                        // get time
                        low = readl(CSR_TIMER3_VALUE);
                        sock_TSC_time.low[sock_counter]= low;
                        sock_TSC_time.thread[sock_counter]= sock_counter;

                        //get TCP header data
                        sock_tstamp.sport[sock_counter] = sock->sk->sport;
                        sock_tstamp.dport[sock_counter] = sock->sk->dport;

                        if(sock_counter < 2) {
                                (*__SOCKtimestamp)(SCK, (unsigned char *)&sock
_tstamp, (unsigned char *)&sock_TSC_time,sock_counter);
                                }
                        //printk("SOCKET >> counter: %d\n", sock_counter);
                        //printk("  socket_stamp: %lu \n", low);
                        //printk("---------- EXIT ------------\n");
                        sock_tstamp.index++;
                        sock_counter++;
                }
                *CSR_TIMER3_CLR=0;
                *CSR_TIMER3_CNTL=0;
                *CSR_TIMER3_LOAD=0;
        }
/******** end hack */
        return size;
}
                                                               608,1         31%
```

Figure A-6 Reseting TIMER3 at the socket layer

Appendix B Setup for testing a firewall for a single host

B.1 Single packet tests



Switch:

3Com Office Connect
10/100 Dual Speed Switch 8
Serial: 7L5V016E08

DUT characteristics:
Name: Volans
Dell PowerEdge 2300 – Dual Pentium II

Operating System: Windows 2000
RedHat 7.1
Kernel 2.4.7

Lilo:

To perform TCP tests choose:
Lilo: tcptest
Image: /boot/ame/2002/02/06/2.4.7-tcp/1/vmlinyz-2.4.7-tcp

Lilo: printks > to watch the printk statements choose
Image: /boot/printks/2001/12/13/2.4.7-printks/2/vmlinuz-2.4.7-printks

To perform UDP tests choose:
Lilo: udperf > to perform the tests
Image: /boot/ame/udp/2002/03/10/2.4.7-udperf/1/vmlinuz-2.4.7-udperf

Lilo: udprint> to watch the printk statements choose
Image: /boot/ame/2002/

TCP server application:

/root/ametest/performance-PC/server
from C file: server.cc

UDP server application:

Directory:	/root/ametest/performance/udp_client_server
Execulable file:	server_udp
C file:	udp_server.cc

Loadable module – critical load to perform tests:

Binary file:	asm32_sys.o
C file:	asm32_sys.c

The reason behind naming this module “asm32_sys” is the following. The “sys” is because the module is “triggered” via a system call. You cannot load the functions in the kernel unless you trigger the module via the driver (see Driver below). The “asm32” is because at first I was going to use an assembly macro that read the entire 64 bits of the RTSC – I named that file asm64_sys.c. Later I decided that it would be better to read the lower 32 bits of the RTSC. So, I made a new file asm32_sys.c and never changed the name after that.

Driver:

Executable:	a32 [START | STOP]
C file:	driver.c

The asm32_sys loadable module is “triggered” via a system call.

Random generators: /root/ametest/iptables.tests/generators/

IP:	ip-random-generator
MAC:	mac-random-generator
TCP:	tcp-random-generator

List of scripts:

To run before tests are performed for both TCP and UDP:
/root/ametest/performance-PC/install.sh

Firewall rules:

```
/root/ametest/iptables.tests/
          MAC:  iptables.mac.accept_but_drop
                iptables.mac.drop_but_accept
          TCP:  iptables.tcp.accept_but_drop
                iptables.tcp drop_but_accept
          IP:   iptables.ip.accept_but_drop
                iptables. ip. drop_but_accept
          UDP:  iptables.udp.accept_but_drop
                iptables.udp drop_but_accept
```

Run to perform TCP tests:

```
/root/ametest/performance-PC/./autotest.sh
```

Run to perform UDP tests:

```
/root/ametest/performance-PC/udp_client_server/./autotest1.sh
/root/ametest/performance-PC/udp_client_server/./autotest2.sh
/root/ametest/performance-PC/udp_client_server/./autotest3.sh
```

How to run TCP tests

1. At BOOT time: choose the label 'tcptest'

The kernel must be set to boot to run level 3, that means that no X window should run. You can do the above either by typing 'tcptest 3' when you get to 'boot:' option as the kernel starts of if you want to set it up automatically change the /etc/inittab file to the following:

Find the line: id:5:initdefault and
Change it to: id:3:initdefault.
Save and exit.

2. Login as root and go to the /root/ametest/performance-PC directory. Run the "install.sh" script. This script STOPS a list of processes and also loads the asm32_sys.o module and triggers it to START. This means that the test is ready to run.

3. Check the running processes. In the prompt shell call the 'uptime' command. DO NOT perform any tests until the *load average* is 0.00 0.00 0.00.

4. Make sure no other processes are running. Use the command 'ps ax' to check that the 'install.sh' stopped all the processes. Also make sure that the PCs are not transferring any data. You can run a network sniffer (e.g. tcpdump) to test it.

5. Once uptime shows 0.00 0.00 0.00 load average, go to the /root/ametest/iptables_tests/ directory and choose the rules that you want to add. Inside every script (e.g. iptables.[FILTERTYPE].accept_but_drop or iptables.[FILTERTYPE].drop_but_accept) you may change the number of rules that you want to have.

   Go to 'increment_rule={xxxx}' variable and make the change. For example, in the iptables.ip.accept_but_drop script the INPUT policy is ACCEPT which means to ACCEPT everything but drop the following rules or matches to the rule. The 'increment_rule={ip_10_addr}' variable means to add only 10 rules to the table. When the variable 'increment_rule={ip_40_addr}' means to filter 40 rules. If 'increment_rule={ip_10_addr ip_40_addr}' the ip_10_addr rules will be called twice because ip_40_addr already contain the ip_10_addr rules. Take a close look to the script and you will understand what I mean!

   To add the rules to the firewall run the script, for example:
   ./iptables.ip.accept_but_drop

6. Check the 'uptime' to be 0.00 0.00 0.00 – yes, again!

7. To run the test run the script:
   ./autotest.tcp.sh

   The script creates a path to store the results in the /proc/TCPresults file

Client side
At boot time, Libra is also run in level 3.

8. Run /ametest/pktgen/./install script to shut down all other services

9. Run ./auto_pktgen on the client

10. Wait until everything is done and change the rule-set, change the variables in autotest.tcp.sh and perform the tests again.

How to run UDP tests

Running UDP tests is not much different than TCP

1. BOOT: udptest

2. Login as root
   cd /root/ametest/performance-PC/./install

3. Check 'uptime' to be 0.00 0.00 0.00

4. No processes should be running. Use the ‘ps ax’ command

5, 6, 7 are the same as TCP

8. cd /udp_client_n_server

9. ./autotest1.udp.sh **à** for test 1
   ./autotest2.udp.sh **à** for test 2
   ./autotest3.udp.sh **à** for test 3

These are the same as autotest.tcp.sh, the only difference is the variable ‘TEST’ I made a copy of each one to save time.

Client
In Libra:
10. Do the ‘install.sh’ script to shutdown all other services

11. /root/ametest/udp_client_n_server/./auto_pktgen

The script will automatically save the results in a directory specified in the variables in the script. You <u>have to change</u> the variables in the ‘autotest.tcp/udp.sh scripts to match the type of test that you’re going to do. Take a look a the scripts and will become clear.

For example, if you want to perform a test with the following parameters:
UDP PACKETS
FILTER 10 IP addresses
INPUT ACCEPT

Server

1. cd /root/ametest/performance-PC/./install
2. uptime – wait until it is 0.00 0.00 0.00
3. in the mean time run
4. cd ../iptables.tests/./iptables.ip.accept_but_drop
5. iptables –L **à** to see the list of rules
6. cd ../performance-PC/udp_client_n_server/
7. vi autotests1.udp.sh
   a. PROTO= “UDP”
   b. INPUT_POLICY = “ACCEPT”
   c. SPEED = “100Mbps”
   d. FILTER_TYPE = “ip”
   e. TEST_NO = “test1”
   f. RULES = “10”
8. ./autotest1.udp.sh

Client

9. cd /udp_client_n_server/./install
10. uptime
11. ./auto_pktgen
12. repeat for autotest2.udp.sh in server…and so on!

B.2 Multiple packets

Client

1. Load the module: `insmod –f asm32_sys.o`
2. Change to the UDP directory: `cd udp_client_n_server`
3. run the `./init.sh` script
4. Set up the packet information in the Smartbits
    a. MAC DST
    b. SRC IP
    c. PORT NUMBER
    d. PAYLOAD LENGTH (for a 64 bytes payload you must add 42 bits for the CRC)
    e. Set the RATE per packet
5. Run the `./flushser` script to clear all the counters
6. Run the `./server_udp`
7. Run Smartbits
8. After all the packets have been sent run ./readproc to read the /proc file system to read all the 4000 timestamps
9. Check if the file “filename” had data in it
10. Backup “filename”
11. Redo from step 3

Smartbits

1. Connect the Windows 95 PC to port 12 in the patch panel
2. Connect 2 SmartBits cards Model ML-7710 to the patch panel and from the patch panel to the Cisco 2900 series XL switch
3. Connect “Volans” to the Cisco 2900 series XL switch
4. Set the Smartbit cards to the same subnet as the client
5.
6. Set the Smartbits cards to “Smart Metrics mode” and ping the cards from Volans
7. Turn off the “Smart Metrics Mode” to perform the tests

8. Transmit Setup Window for Card 1:
   i. Mode: Timed
   ii. Time: 60
   iii. Length: 106
   iv. Background: UDP
   v. Rate: 5% - Units: % utilization

   b. Frame Editor: UDP EDIT
   i. MAC DEST: 00 50 da 26 b0 55
   ii. MAC SRC: 00 00 00 00 00 0a
   iii. SRC IP: 192.168.50.20
   iv. DST IP: 192.168.50.10
   v. dst: 12345

9. Transmit Setup Window for Card 2:
   i. Mode: single burst
   ii. Count: 4
   iii. Length – Fixed 106
   iv. Background: UDP
   v. Rate: 0.96% - util

   b. Frame Editor: UDP edit
   i. MAC DEST: 00 50 da 26 b0 55
   ii. MAC SRC: 00 00 00 00 00 02
   iii. SRC IP: 192.168.50.30
   iv. DST IP: 192.168.50.10
   v. dst: 6789

Appendix C IPTABLES rules random generators and scripts

File name: random-ip-generator.c

```
/*
* Random IP address generator
* Max Roth <modified by Americo Melara>
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){

     int ret;
     int c;

     if(argv[1] == NULL) {
          printf("usage: ./ip-random-generator [missing
number]\n");
          return 0;
     }

     srand(time(0));

     for(c = 0; c < atoi(argv[1]); c++){

          printf("%d.%d.%d.%d\n", (rand() % 255),(rand() %
255),(rand() % 255),(rand() % 255));
     }

     return 0;
}
```

File name: random-mac-generator.c

```
/*
* Random MAC address generator
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){

    int ret;
    int c;
    char hex[]="0123456789ABCDEF";

    if(argv[1] == NULL) {
        printf("usage: ./mac-random-generator [missing
number]\n");
        return 0;
    }

    srand(time(0));

    for(c = 0; c < atoi(argv[1]); c++){

        printf("00:%C%C:%C%C:%C%C:%C%C:%C%C\n",
                (hex[rand() % 16]),(hex[rand() % 16]),
                (hex[rand() % 16]),(hex[rand() % 16]),
                (hex[rand() % 16]),(hex[rand() % 16]),
                (hex[rand() % 16]),(hex[rand() % 16]),
                (hex[rand() % 16]),(hex[rand() % 16]));
    }

    return 0;
}
```

File name: random-tcp-generator.c <also used for udp>

```
/*
* Random TCP address generator
* Americo Melara
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


int main(int argc, char *argv[]){

     int ret;
     int c;


     if(argv[1] == NULL) {
          printf("usage: ./tcp-random-generator [missing
number]\n");
          return 0;
     }

     srand(time(0));

     for(c = 0; c < atoi(argv[1]); c++){

          printf("%d\n", (rand() % 6555));
     }

     return 0;
}
```