

Jaypee University of Information Technology
Waknaghat, Distt. Solan (H.P.)

Learning Resource Center

CLASS NUM:

BOOK NUM.:

ACCESSION NO.: SP08021 / SP0812021

This book was issued is overdue due on the date stamped below. If the book is kept over due, a fine will be charged as per the library rules.

Due Date	Due Date	Due Date

TCP STREAM NORMALIZATION

Submitted by:

ISHA JAIN (081212)

NUPUR GOSWAMI (081409)

Guided by

(Brig.) SATYA PRAKASH GHRERA



Submitted in partial fulfillment of the Degree of
Bachelor of Technology
(MAY 2012)

DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING/INFORMATION TECHNOLOGY

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,
WAKNAGHAT

TABLE OF CONTENTS

Chapter No.	TOPIC	PAGE NUMBER
	Certificate From Supervisor	iii
	Acknowledgement	iv
	Summary	V
	List of figures	vi
1).	Objective	1
2).	Scope of the project	2
3).	Introduction	3
	3.1. What is a normalizer?	3
	3.2. Various types of normalizers implemented and related research done in this field	4
4).	Literature Review	6
	4.1. Evasion Attacks	6
	4.2. Network Intrusion Detection Systems	11
5).	Requirement Specification and Analysis	13
	5.1. Requirements	13
	5.2. Study	13
	• Network and Transport layer protocols	13
	• Detailed working of TCP/IP protocol	15

TCP STREAM NORMALIZATION

	• Linux	18
	• Socket programming in Linux	19
	• Packet Sniffer	24
	• SHA-1	27
6).	System Design and Implementation	28
	6.1. Implementation of packet sniffer	28
	6.2. Implementation of SHA-1	30
	6.3. Implementation of Client-server	35
	6.4. Implementation of Normalizer	36
7).	Testing	38
8).	Limitations	41
9).	Summary	42
10).	Conclusion	43
11).	References	44
	Annexure-A	45-65

CERTIFICATE

This is to certify that the project work done on **TCP STREAM NORMALIZATION** is a bonafide work carried out by **Isha Jain (081212)** and **Nupur Goswami (081409)**, in partial fulfillment for the award of degree of Bachelor of Technology of Jaypee University of Information Technology, Waknaghat, under my supervision and guidance. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor:



Name of Supervisor: (Brig.) Satya Prakash Ghrera

Designation: Head of Department (Computer Science & Engineering)


Date: 30/5/2012

Students' names and Signatures:

Isha Jain (081212)



Nupur Goswami (081409)



ACKNOWLEDGEMENT

We take immense pleasure in thanking **(Brig.) Satya Prakash Ghrera**, Associate Professor and HOD CSE/IT, for giving us the wonderful opportunity to work on this project under his comprehensive guidance. This project would not have been possible without his able guidance and useful suggestions. He has been a constant source of inspiration in the carrying out of all the project work in a timely manner.

We would also like to extend our heartfelt thanks to **Dr. Nitin**, Associate Professor and Project Coordinator, for his timely guidance in the conduct of our project work and for providing us with the much needed direction in this regard.

We wish to express our deep sense of gratitude to **Ms. Rajni Mohana**, Lecturer, for taking out time from her busy schedule and providing guidance in the implementation phase of the project.

Finally, yet importantly, we would like to thank our faculty and peers for some invaluable insight into certain key concepts that have led to the successful completion of our project.

SUMMARY

Even though a lot of effort is made to secure our networks by various mechanisms, they still remain vulnerable to a number of crafty attacks. There is a particular type of attack in which the attackers try to confuse the network security systems by introducing ambiguities in the network i.e. sending inconsistent TCP retransmissions. Our effort will be to design a normalizer that buffers only the hashes of incoming packets, to detect any inconsistent retransmissions in any TCP byte stream. In this way, the memory requirements are less and thus it can be implemented on high speed networks without affecting the performance. Our normalizer shall also be robust to attacks that attempt to prevent its correct operation or exhaust its resources.

LIST OF FIGURES

SL. NO.	FIGURE DETAILS	PG. NO.
1	Fig 1: Relationship between network, transport and application layers	14
2	Fig 2: TCP Header	16
3	Fig 3: IP (version4) Header	17
4	Fig 4:- Network Layers and socket Interface	20
5	Fig 5:- Socket API's used during server-client session	23
6	Fig 6: Packet Sniffer Flowchart	29
7	Fig 7: SHA-1 Flow diagram	30
8	Fig 8:- Flow diagram of the client side program to transfer file to a remote server	35
9	Fig 9:- Flow diagram of the server side program to receive file sent from a remote client	35
10	Fig 10:- Flowchart for design of normalizer	37
11	Fig 11:- Screenshot for the testing of SHA-1	38
12	Fig 12:- Screenshot for the starting of the normalizer program when the first file is sent	39
13	Fig 13:- Screenshot for the calculation of hash codes for different packets for file 1	39
14	Fig 14:- Screenshot of normalizer program when a second file is sent	40
15	Fig 15:- Screenshot for hash code calculation for packets of file 2 while being sent	40

1.) OBJECTIVE

The objective of this project is to implement a network element that detects and blocks inconsistent retransmissions in any TCP byte stream, in a manner that takes care of the memory requirements and is also resistant to attacks.

The programs will run on Linux operating system and the entire project will be developed and coded using C language and socket programming.

2.) SCOPE OF THE PROJECT

We have envisaged our normalizer to work on a Linux platform on a three node network with a simple client-server architecture. Our endeavor is to implement the functionality of this essential network element, the normalizer in software using the C programming language. The same can be extended to hardware implementation with the aid of assembly language. The scope of this normalizer can be extended to other dynamic network configurations and on other platforms (Windows, Mac etc.).

There are still possibilities of exploring a different approach to defending against evasion attacks in which the network monitor may proactively determine how specific end systems and network paths will resolve potential ambiguities. While this approach can be a valid point in the overall design space, eliminating ambiguities, rather than attempting to correctly guess their outcome, seems to provide a more robust foundation for security monitoring technology.

3.) INTRODUCTION

3.1. What is a Normalizer?

Network intrusion detection systems' fundamental property is the ability of a skilled attacker to evade detection by exploiting ambiguities in the traffic stream as seen by the monitor. Network intrusion detection and prevention systems are widely used to improve the security of networks used by providers, enterprises, and even home users. Such monitors usually operate on the path between the protected network and the rest of the Internet. They observe all traffic coming in and out of the network and flag or block activities that appear malicious.

A normalizer is a network element that prevents evasion attempts by removing ambiguities in network traffic. It sits directly in the path of traffic into a site and observes the packet stream to eliminate potential ambiguities before going into the network.

Normalizer differs from a firewall in several ways. It does not prevent access to services on internal hosts, but ensures that access to these hosts takes place in a secure manner that is unambiguous to the site's Network Intrusion Detection System. Normalizers can prevent known attacks, or shut down access to internal machines from an external host when it detects a probe or an attack. It can shut down and discard state for flows that do not appear to be making progress, while passing and normalizing those that do make progress.

In the next section we briefly discuss the possible ways in which a normalizer can be implemented, the various types of normalizers as well as the techniques which are used to develop them.

3.2. Various types of Normalizers implemented and related research done in this field

➤ Recent work done by G. Varghese, J. A. Fingerhut, and F. Bonomi, “Detecting Evasion Attacks at High Speeds without Reassembly,” addresses one type of evasions, namely an attacker attempting to prevent a specific signature match against text they transmit. The authors developed a scheme based on introducing a modest change in end-system TCP behavior in order to allow a monitor to detect attempts to ambiguously transmit byte sequences that match a given set of signatures. Their scheme is appealing in that by exploiting the introduced end-system change, they avoid needing to reassemble TCP byte streams. However, their scheme is also significantly limited in that it only applies to evasions that correspond to directly manipulating a known byte-sequence signature. As such, the scheme does not handle cases where the ambiguity does not constitute an actual attack in itself, but only confuses the monitor’s protocol parsing and obscures the occurrence of an attack later in the stream.

➤ Y. Sugawara, M. Inaba, and K. Hiraki in their paper “High-speed and Memory Efficient TCP Stream Scanning Using FPGA,” describe an FPGA-based solution to efficient TCP stream-level signature detection. Their system detects inconsistent retransmissions by storing hashes of transmitted packets. To handle retransmissions that do not overlap with original segment boundaries, the authors simply propose holding onto the partial overlaps till other packets that “fill the gap” arrive. However, our trace evaluation shows that such an approach will result in a significant number of connections stalling on pending consistency checks, RoboNorm addresses this problem with the ACK promotion mechanism.

➤ Normalization as a general feature has been incorporated into secure operating systems and commercial products. Some of these latter include explicit options to check for inconsistent retransmissions, but do not provide technical details as to how such detection works. From informal discussions with other vendors, it appears that a common approach is to use payload hashes, but without addressing the crucial problem of misaligned retransmissions for which the hashes cannot be matched.

➤ Shankar and Paxson explored a different approach to defending against evasion attacks which they term “Active Mapping”. Here, the idea is for the network monitor to proactively determine how specific end systems and network paths will resolve potential ambiguities. While this approach is a valid point in the overall design space, we argue that eliminating ambiguities rather than attempting to correctly guess their outcome, provides a more robust foundation for security monitoring technology.

➤ Work by Levchenko et al. demonstrates in formal terms that many security detection tasks (e.g., detecting SYN flooding, port scans, connection hijacking and evasion attacks) fundamentally require maintaining per-connection state. This finding highlights the importance of reducing the amount of per-connection state.

➤ In work done by S. Dharmapurikar and V. Paxson, “Robust TCP Stream Reassembly in the Presence of Adversaries,” explores how to robustly reassemble TCP byte streams when faced with adversaries who attempt to overwhelm the accompanying state management. Reassembly involves maintaining out-of-order data only until sequence “holes” are filled, while normalization requires maintaining data until it is acknowledged and hence requires a different solution.

4.) LITERATURE REVIEW

4.1. Evasion Attacks

The reviewed literature presents a keen insight into another important class of network security attacks i.e. the evasion attacks. Evasion is a term used to describe techniques of bypassing an information security device in order to deliver an exploit, attack or other malware to a target network or system, without detection. Evasions are typically used to counter network-based intrusion detection and prevention systems (IPS, IDS) but can also be used to by-pass firewalls. A further target of evasions can be to crash a network security device, rendering it in-effective to subsequent targeted attacks. Evasions can be particularly nasty because a well-planned and implemented evasion can enable full sessions to be carried forth in packets that evade IDS. Attacks carried in such sessions will happen right under the nose of the network and service administrators. The security systems are rendered ineffective against well-designed evasion techniques, in the same way a stealth fighter can attack without detection by radar and other defensive systems.

Network attackers often use network IPS evasion techniques to attempt to bypass the intrusion detection, prevention, and traffic filtering functions provided by network IPS sensors. Some commonly used network IPS evasion techniques are listed below:

- Encryption and Tunneling
- Timing Attacks
- Resource Exhaustion
- Traffic Fragmentation
- Protocol-level Misinterpretation
- Traffic Substitution and Insertion

4.1.1. Encryption and Tunneling

One common method of evasion used by attackers is to avoid detection simply by encrypting the packets or putting them in a secure tunnel. As discussed now several times, IPS sensors monitor the network and capture the packets as they traverse the network, but network based sensors rely on the data being transmitted in plaintext. When and if the packets are encrypted, the sensor captures the data but is unable to decrypt it and cannot perform meaningful analysis. This is assuming the attacker has already established a secure session with the target network or host. Some examples that can be used for this method of encryption and tunneling are:

- Secure Shell (SSH) connection to an SSH server
- Client-to-LAN IPsec (IP Security) VPN (virtual private network) tunnel
- Site-to-site IPsec VPN tunnel
- SSL (Secure Socket Layer) connection to a secure website

There are other types of encapsulation that the sensor cannot analyze and unpack that attackers often use in an evasion attack. For example, GRE (Generic Route Encapsulation) tunnels are often used with or without encryption.

4.1.2. Timing Attacks

Attackers can evade detection by performing their actions slower than normal, not exceeding the thresholds inside the time windows the signatures use to correlate different packets together. These evasion attacks can be mounted against any correlating engine that uses a fixed time window and a threshold to classify multiple packets into a composite event. An example of this type of attack would be a very slow reconnaissance attack sending packets at the interval of a couple per minute. In this scenario, the attacker would likely evade detection simply by making the scan possibly unacceptably long.

4.1.3. Resource Exhaustion

A common method of evasion used by attackers is extreme resource consumption, though this subtle method doesn't matter if such a denial is against the device or the personnel managing the device. Specialized tools can be used to create a large number of alarms that consume the resources of the IPS device and prevent attacks from being logged. These attacks can overwhelm what is known as the management systems or server, database server, or out-of-band (OOB) network. Attacks of this nature can also succeed if they only overwhelm the administrative staff, which does not have the time or skill necessary to investigate the numerous false alarms that have been triggered.

Intrusion detection and prevention systems rely on their ability to capture packets off the wire and analyze them quickly, but this requires the sensor has adequate memory capacity and processor speed. The attacker can cause an attack to go undetected through the process of flooding the network with noise traffic and causing the sensor to capture unnecessary packets. If the attack is detected, the sensor resources may be exhausted but unable to respond within a timely manner due to resources being exhausted.

4.1.4. Traffic Fragmentation

Fragmentation of traffic was one of the early network IPS evasion techniques used to attempt to bypass the network IPS sensor. Any evasion attempt where the attacker splits malicious traffic to avoid detection or filtering is considered a fragmentation-based evasion by:

- Bypassing the network IPS sensor if it does not perform any reassembly at all.
- Reordering split data if the network IPS sensor does not correctly order it in the reassembly process.
- Confusing the network IPS sensor's reassembly methods which may not reassemble split data correctly and result in missing the malicious payload associated with it.
- A few classic examples of fragmentation-based evasion are below:

- TCP segmentation and reordering, where the sensor must correctly reassemble the entire TCP session, including possible corner cases, such as selective ACKs and selective retransmission.
- IP fragmentation, where the attacker fragments all traffic if the network IPS does not perform reassembly. Most sensors do perform reassembly, so the attacker fragments the IP traffic in a manner that it is not uniquely interpreted. This action causes the sensor to interpret it differently from the target, which leads to the target being compromised.

In the same class of fragmentation attacks, there is a class of attacks involving overlapping fragments. In overlapping fragments the offset values in the IP header don't match up as they should, thus one fragment overlaps another. The IPS sensor may not know how the target system will reassemble these packets, and typically different operating systems handle this situation differently.

4.1.5. Protocol-level Misinterpretation

Attackers also evade detection by causing the network IPS sensor to misinterpret the end-to-end meaning of network protocols. In this scenario the traffic is seen differently from the target by the attacker causing the sensor either to ignore traffic that should not be ignored or vice versa. Two common examples are packets with bad TCP checksum and IP TTL (Time-to-live) attacks.

A bad TCP checksum could occur in the following manner: An attack intentionally corrupts the TCP checksum of specific packets, thus confusing the state of the network IPS sensor that does not validate checksums. The attacker can also send a good payload with the bad checksum. The sensor can process it, but most hosts will not. The attacker follows with a bad payload with a good checksum. From the network IPS sensor this appears to be a duplicate and will ignore it, but the end host will now process the malicious payload.

The IP TTL field in packets presents a problem to network IPS sensor because there is no easy way to know the number of hops from the sensor to the end point of an IP session stream. Attackers can take advantage of this through a method of reconnaissance by sending a packet that has a very short TTL which will pass through the network IPS fine, but be dropped by a

router between the sensor and the target host due to a TTL equaling zero. The attacker may then follow by sending a malicious packet with a long TTL, which will make it to the end host or target. The packet looks like a retransmission or duplicate packet from the attacker, but to the host or target this is the first packet that actually reached it. The result is a compromised host and the network IPS sensor ignored or missed the attack.

4.1.6. Traffic Substitution and Insertion

Another class of evasion attacks includes traffic substitution and insertion. Traffic substitution is when that attacker attempts to substitute payload data with other data in a different format, but the same meaning. A network IPS sensor may miss such malicious payloads if it looks for data in a particular format and doesn't recognize the true meaning of the data. Some examples of substitution attacks are below:

- Substitution of spaces with tabs, and vice versa, for example inside HTTP requests.
- Using Unicode instead of ASCII strings and characters inside HTTP requests.
- Exploit mutation, where specific malicious shell code (executable exploit code that forces the target system to execute it) can be substituted by completely different shell code with the same meaning and thus consequences on the end host or target.
- Exploit case sensitivity and changing case of characters in a malicious payload, if the network IPS sensor is configured with case-sensitive signature.

Insertion attacks act in the same manner in that the attacker inserts additional information that does not change the payload meaning into the attack payload. An example would be the insertion of spaces or tabs into protocols that ignore such sequences.

4.2. Network Intrusion Detection Systems

Intrusion detection is a security technology that attempts to identify and isolate "intrusions" against computer systems. Different ID systems have differing classifications of "intrusion"; a system attempting to detect attacks against web servers might consider only malicious HTTP requests, while a system intended to monitor dynamic routing protocols might only consider RIP spoofing. Regardless, all ID systems share a general definition of "intrusion" as an unauthorized usage of or misuse of a computer system.

Intrusion detection is an important component of a security system, and it complements other security technologies. By providing information to site administration, ID allows not only for the detection of attacks explicitly addressed by other security components (such as firewalls and service wrappers), but also attempts to provide notification of new attacks unforeseen by other components. Intrusion detection systems also provide forensic information that potentially allow organizations to discover the origins of an attack. In this manner, ID systems attempt to make attackers more accountable for their actions, and, to some extent, act as a deterrent to future attacks.

4.2.1. The Need for Reliable Intrusion Detection

Because of its importance within a security system, it is critical that intrusion detection systems function as expected by the organizations deploying them. In order to be useful, site administration needs to be able to rely on the information provided by the system; flawed systems not only provide less information, but also a dangerously false sense of security. Moreover, the forensic value of information from faulty systems is not only negated, but potentially misleading.

Given the implications of the failure of an ID component, it is reasonable to assume that ID systems are themselves logical targets for attack. A smart intruder who realizes that an IDS has been deployed on a network she is attacking will likely attack the IDS first, disabling it or forcing it to provide false information (distracting security personnel from the actual attack in progress, or framing someone else for the attack).

In order for a software component to resist attack, it must be designed and implemented with an understanding of the specific means by which it can be attacked. Unfortunately, very little information is publicly available to IDS designers to document the traps and pitfalls of implementing such a system. Furthermore, the majority of commercially available ID systems have proprietary, secret designs, and are not available with source code. This makes independent third-party analysis of such software for security problems difficult.

The most obvious aspect of an IDS to attack is its "accuracy". The "accuracy" of an IDS is compromised when something occurs that causes the system to incorrectly identify an intrusion when none has occurred (a "false positive" output), or when something occurs that causes the IDS to incorrectly fail to identify an intrusion when one has in fact occurred (a "false negative"). Some researchers discuss IDS failures in terms of deficiencies in "accuracy" and "completeness", where "accuracy" reflects the number of false positives and "completeness" reflects the number of false negatives.

Other attacks might seek to disable the entire system, preventing it from functioning effectively at all. We say that these attacks attempt to compromise the "availability" of the system.

5.) REQUIREMENT SPECIFICATION AND ANALYSIS

5.1. Requirements

In order to develop and implement this project a study of following topics was done in detail of:

- Basic knowledge of network and transport layer protocols
- Detailed working of TCP/IP protocol
- Linux based C Programming
- Socket TCP/IP Programming in Linux
- Packet Sniffer
- SHA-1 (Secure Hash Standard)
- Implementation knowledge of a normalizer

A brief introduction to all the topics mentioned above is given in the next subsection.

5.2. Study

5.2.1. Basic Knowledge of network and transport layer protocols

The network layer offers its services to the transport layer which deals with the end-to-end communication in any network. The network layer deals with the transmission of packets from the source to the destination. More particularly, it is concerned with how to route the packets from one router to the other. The network layer may provide two types of services to the transport layer – connectionless and connection oriented. In the former, each packet travels on a different path based on the traffic conditions to maximize throughput. The packets may thus arrive out of order but are delivered to the higher layers in perfect order. In case of connection-oriented service, connection first needs to be established which sets up a virtual circuit i.e. a predetermined path which all packets must follow. It is thus a more reliable form of transmission

and the packets also arrive in order but it generally tends to consume more bandwidth and is prone to fatal errors. Their usage depends on the application and the higher level protocols.

An application of network and transport layer working is the Internet. Communication in the Internet works as follows. The transport layer takes data streams and breaks them up into datagrams. In theory, datagrams can be up to 64 Kbytes each, but in practice they are usually not more than 1500 bytes (so they fit in one Ethernet frame). Each datagram is transmitted through the Internet, possibly being fragmented into smaller units as it goes. When all the pieces finally get to the destination machine, they are reassembled by the network layer into the original datagram. This datagram is then handed to the transport layer, which inserts it into the receiving process' input stream.

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective service to its users, normally processes in the application layer. To achieve this goal, the transport layer makes use of the services provided by the network layer. The hardware and/or software within the transport layer that does the work is called the transport entity. The transport entity can be located in the operating system kernel, in a separate user process, in a library package bound into network applications, or conceivably on the network interface card. The (logical) relationship of the network, transport, and application layers is illustrated in the following figure.

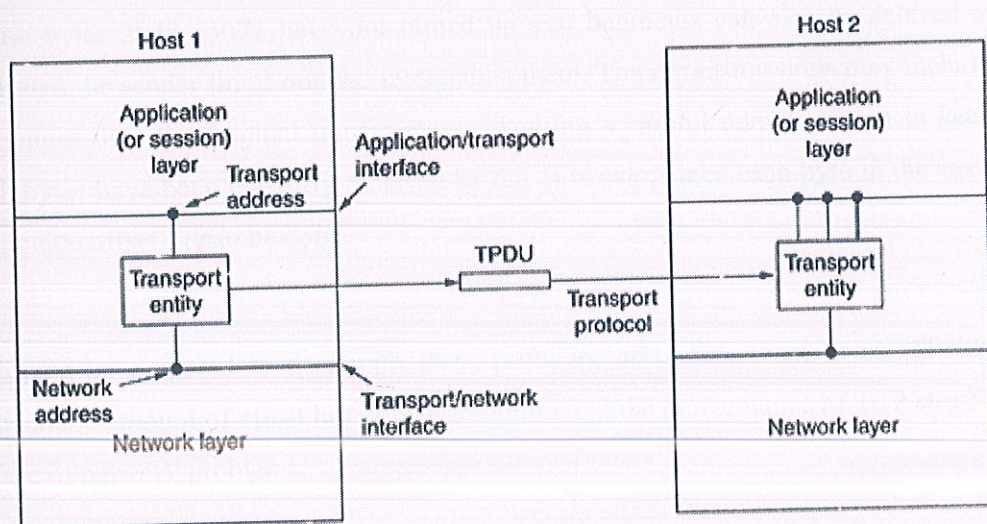


Figure 1: Relationship between network, transport and application layers

5.2.2. Detailed working of TCP/IP Protocol

A key feature of TCP, and one which dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. The sending and receiving TCP entities exchange data in the form of segments. A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515-byte IP payload. Second, each network has a maximum transfer unit, or MTU, and each segment must fit in the MTU. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.

The basic protocol used by TCP entities is the sliding window protocol. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again. Segments can arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte ranges than the original transmission, requiring a careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems.

Figure 2 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to

TCP STREAM NORMALIZATION

$65,535 - 20 - 20 = 65,495$ data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

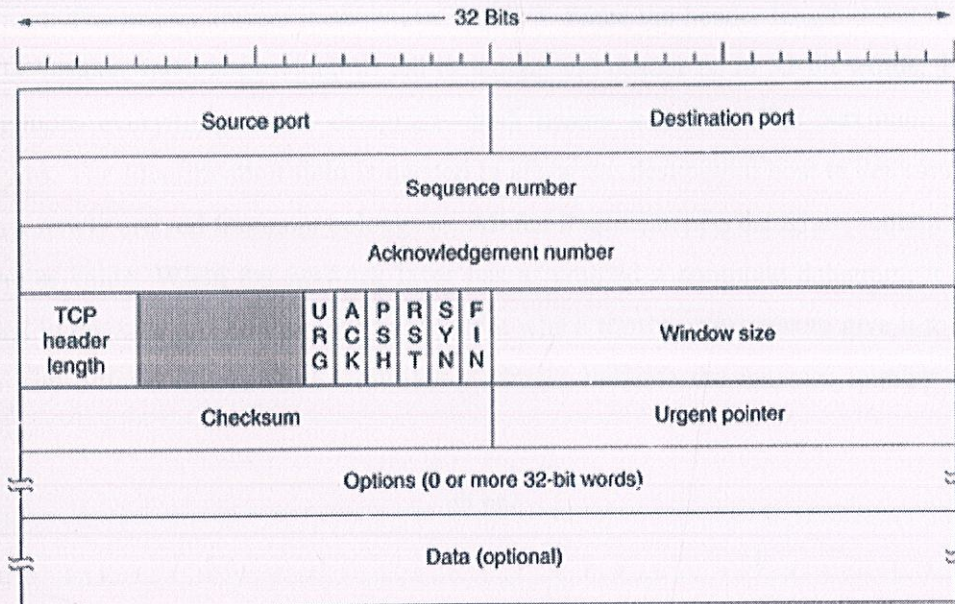


Figure 2: TCP Header

The Source port and Destination port fields identify the local end points of the connection. The TCP header length tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is, too. The ACK bit is set to 1 to indicate that the Acknowledgement number is valid. If ACK is 0, the segment does not contain an acknowledgement so the Acknowledgement number field is ignored. The SYN bit is used to establish connections. The connection request has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has SYN = 1 and ACK = 1. In essence the SYN bit is used to denote CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities. The other data fields are not of direct consequence to the scope of our project so we have primarily focused on these.

IPv4 Protocol:

The underlying protocol working on the network layer is the IPv4 protocol. An IP datagram consists of a header part and a text part. The header has a 20-byte fixed part and a variable length optional part. The header format is shown in Figure 3. Since the header length is not constant, a field in the header, IHL, is provided to tell how long the header is, in 32-bit words. The Total length includes everything in the datagram—both header and data. The maximum length is 65,535 bytes. The Identification field is needed to allow the destination host to determine which datagram a newly arrived fragment belongs to. All the fragments of a datagram contain the same Identification value. When the network layer has assembled a complete datagram, it needs to know what to do with it. The Protocol field tells it which transport process to give it to i.e. TCP or UDP. The Source address and Destination address indicate the network number and host number.

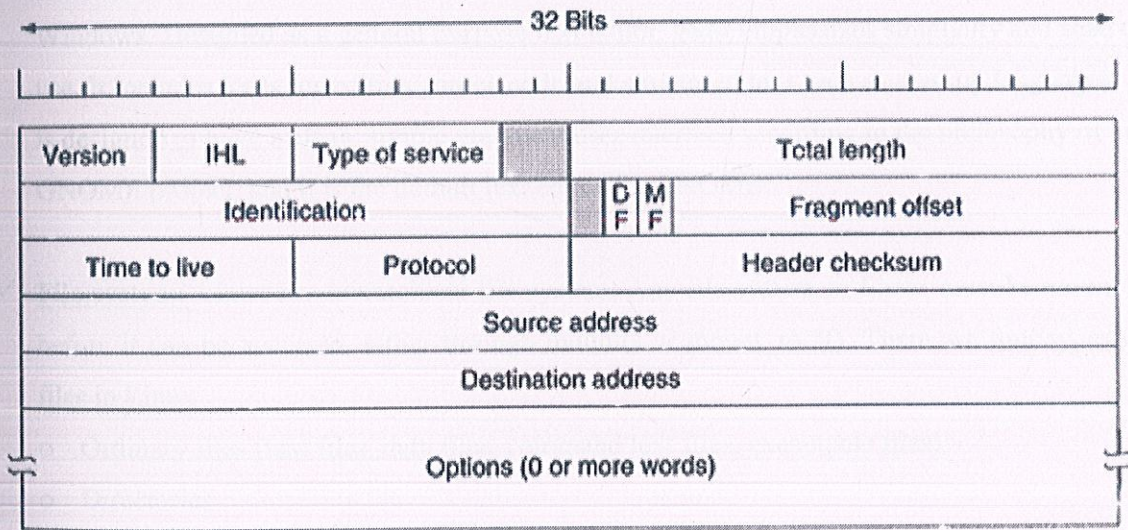


Figure 3: IP (version 4) Header

5.2.3. Linux

In order to gain expertise on Linux OS following topics were covered in detail:

- ✓ **Linux commands**
- ✓ **Text Editors: vi and gedit**
 - **Vi:** The VI editor is a screen-based editor used by many UNIX users. The VI editor has powerful features to aid programmers. It lets a user create new files or edit existing files. The command to start the VI editor is vi, followed by the filename. It operates in either insert *mode* or *command mode*. In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (*Escape*) key turns off the Insert mode.
 - **Gedit:** gedit is a text editor for the GNOME desktop environment, Mac OS X and Microsoft Windows. Designed as a general purpose text editor, gedit emphasizes simplicity and ease of use. It includes tools for editing source code and structured text such as markup languages. It is designed to have a clean, simple graphical user interface according to the philosophy of the GNOME project, and it is the default text editor for GNOME.
- ✓ **File system:** Linux has hierarchical file system. Any file system in Linux must be mounted before it can be accessed, either through mount() or mount_root(). There are four types of files in Linux
 - Ordinary files (text files, data files, command text files, executable files)
 - Directories
 - Links
 - Special device file (physical Hardware)
- ✓ **Directory structure:**

Directory is group of files. Directory is divided into two types:

Root directory - Strictly speaking, there is only one root directory in your system, which is denoted by / (forward slash). It is root of your entire file system and cannot be renamed or deleted.

Sub directory - Directory under root (/) directory is subdirectory which can be created, renamed by the user.

Directories are used to organize your data files, programs more efficiently.

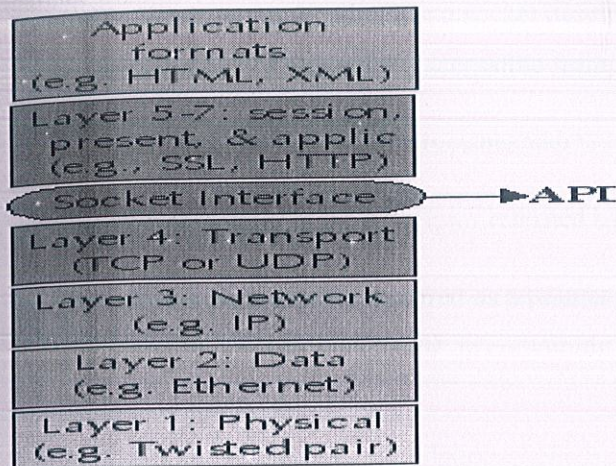
- ✓ **LAN and Internet Exploration tools:** telnet, ftp, netstat, who, ifconfig, whois, nslookup, dig, finger, ping, traceroute, ftp.
 - **Telnet:** Telnet allows you to login remotely from a remote computer to a host server running any unix or unix clone system.
 - **Ftp:** Ftp is the user interface to the **Internet** standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.
 - **Netstat:** This command shows all sorts of statistics for your LAN, including all internet connections.
 - **Ifconfig:** Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.
 - **Ping:** This command is used to find out if a distant computer is alive and runs diagnostic tests.
 - **Traceroute:** This command is similar to ping. It maps internet connections, reveals routers and boxes running firewalls.

- ✓ **Inter Process Communication Mechanisms:** Signals, Pipes, Message queues, Semaphores and shared memory (can be used for further enhanced performance in the next version of the project)

5.2.4. Socket Programming in Linux

Sockets are interfaces that can "plug into" each other over a network. Once so "plugged in", the programs so connected communicate.

A computer network is composed of a number of "network layers", each providing a different restriction and/or guarantee about the data at that layer. The protocols at each network layer generally have their own packet formats, headers, and layout.



The seven traditional layers of a network are divided into two groups: upper layers and lower layers. The sockets interface provides a uniform API to the lower layers of a network, and allows implementing upper layers within your sockets application.

Fig 4:- Network Layers and socket Interface

For doing Socket Programming in Linux study about the various libraries and functions that were required to create a server-client program to transfer file.

List of functions used for socket programming is given below:

- ✓ **socket():** The `socket()` function creates an endpoint for communications and returns a socket descriptor that represents the endpoint.

```
int socket(int protocolFamily, int type, int protocol)
```

- The first parameter `protocolFamily` determines the protocol family of the socket; we will always supply `PF_INET` for the protocol family specifying that the socket uses protocols from the internet family.
- The `type` determines the semantics of data transmission with the socket--for example, whether transmission is reliable, whether message boundaries are preserved, and so on. The constant `SOCK_STREAM` specifies a socket with reliable byte-stream semantics, whereas `SOCK_DGRAM` specifies a best-effort datagram socket.

c) The third parameter specifies the particular end-to-end protocol to be used. For the PF_INET protocol family, we want TCP (identified by the constant IPPROTO_TCP) for a stream socket and UDP (identified by IPPROTO_UDP) for a datagram socket.

✓ **bind():** When an application has a socket descriptor, it can bind a unique name to the socket. Servers must bind a name to be accessible from the network.

```
int bind (int socket, struct sockaddr* localAddress, unsigned int addressLength)
```

- a) The first parameter is the descriptor returned by an earlier call to socket().
- b) The address parameter is declared as a pointer to a sockaddr, but for TCP/IP applications, it will actually point to a sockaddr_in containing the Internet address of the local interface and the port to listen on.
- c) The third parameter addressLength is the length of the address structure, invariably passed as sizeof(struct sockaddr_in).

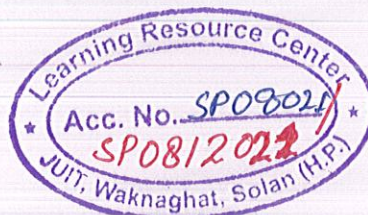
✓ **listen():** function indicates a willingness to accept client connection requests. When a listen() is issued for a socket, that socket cannot actively initiate connection requests. The listen() API is issued after a socket is allocated with a socket() function and the bind() function binds a name to the socket. A listen() function must be issued before an accept() function is issued.

```
int listen(int socket, int queueLimit)
```

- a) The first parameter is the descriptor returned by an earlier call to socket().
 - b) The queueLimit parameter specifies an upper bound on the number of incoming connections that can be waiting at any time.
- ✓ **Connect():** The client application uses a connect() function on a stream socket to establish a connection to the server.

```
int connect(int socket, struct sockaddr* foreignAddress, unsigned int addressLength)
```

- a) socket is the descriptor created by socket ().



- b) `foreignAddress` is declared to be a pointer to a `sockaddr` because the sockets API is generic; for our purposes, it will always be a pointer to a `sockaddr_in` containing the Internet address and port of the server.
 - c) `addressLength` specifies the length of the address structure and is invariably given as `sizeof(struct sockaddr_in)`.
- ✓ **Accept():** The server application uses the `accept()` function to accept a client connection request. The server must issue the `bind()` and `listen()` functions successfully before it can issue an `accept()`; `accept()` dequeues the next connection on the queue for socket. If the queue is empty, `accept()` blocks until a connection request arrives.

```
int accept(int socket, struct sockaddr* clientAddress, unsigned int* addressLength)
```

- a) The first parameter is the descriptor created by `socket()`.
- b) The second parameter is the address of the client at the other end of the connection.
- c) The third parameter `addressLength` specifies the maximum size of the `clientAddress` address structure and contains the number of bytes actually used for the address upon return.

When a connection is established between stream sockets clients and servers can transfer data using `send()` and `recv()`.

- ✓ **Send():** The function sends data to the connected machine on the other end.

Recv(): The function receives data sent from the machine connected on the other end.

```
int send(int socket, const void *msg, unsigned int msgLength, int flags)
```

```
int recv(int socket, void *rcvBuffer, unsigned int bufferLength, int flags)
```

- a) The first parameter of `send()` function `msg` points to the message to send.
- b) The second parameter of `send()` function `msgLength` is the length (bytes) of the message.
- c) The first parameter of the `recv()` function is the descriptor created by `socket()`.

- d) The second parameter of the `recv()` function `rcvBuffer` points to the buffer--that is, an area in memory such as a character array--where received data will be placed.
 - e) The third parameter `bufferLength` gives the length of the buffer, which is the maximum number of bytes that can be received at once.
 - f) The parameter `flags` parameter in both `send()` and `recv()` provides a way to change the default behavior of the socket call. Setting `flagsto 0` specifies the default behavior.
- ✓ **Close():** `close()` tells the underlying protocol stack to initiate any actions required to shut down communications and deallocate any resources associated with the socket,

`int close(int socket)`

- a) The parameter `socket` is the descriptor created by `socket()`.

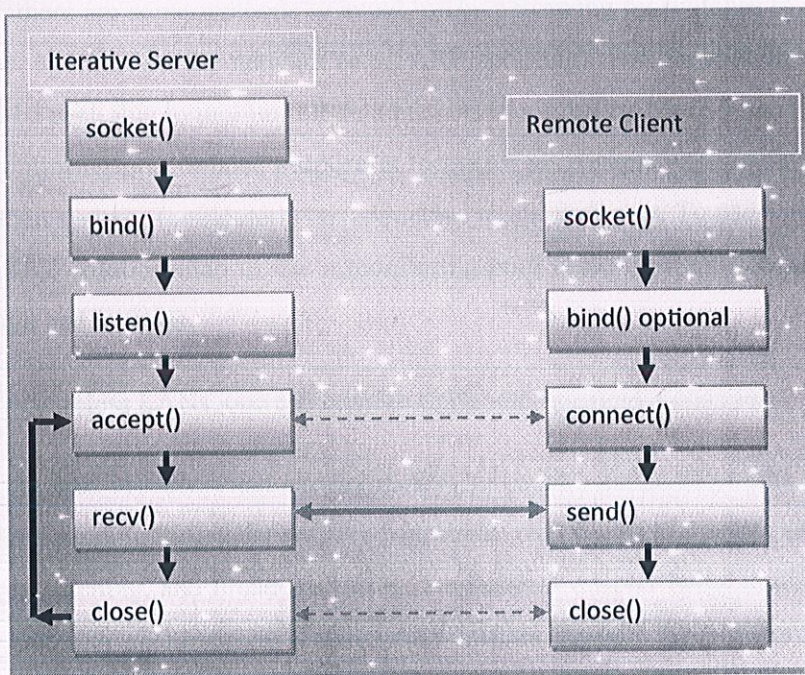


Fig 5:- Socket API's used during server-client session

5.2.5. Packet Sniffer

A **packet sniffer** is a computer program or a piece of computer hardware that can intercept and log traffic passing over a digital network or part of a network. As data streams flow across the network, the sniffer captures each packet and, if needed, decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content. It is effective on both switched and non-switched networks. In a non-switched network environment packet sniffing is an easy thing to do. This is because network traffic is sent to a hub which broadcasts it to everyone.

✓ Capabilities

On wired broadcast LANs, depending on the network structure (hub or switch), one can capture traffic on all or just parts of the network from a single machine within the network; however, there are some methods to avoid traffic narrowing by switches to gain access to traffic from other systems on the network (e.g., ARP spoofing). For network monitoring purposes, it may also be desirable to monitor all data packets in a LAN by using a network switch with a so-called monitoring port, whose purpose is to mirror all packets passing through all ports of the switch when systems (computers) are connected to a switch port. To use a network tap is an even more reliable solution than to use a monitoring port, since taps are less likely to drop packets during high traffic loads.

On wireless LANs, one can capture traffic on a particular channel.

On wired broadcast and wireless LANs, to capture traffic other than unicast traffic sent to the machine running the sniffer software, multicast traffic sent to a multicast group to which that machine is listening, and broadcast traffic, the network adapter being used to capture the traffic must be put into promiscuous mode; some sniffers support this, others do not. On wireless LANs, even if the adapter is in promiscuous mode, packets not for the service set for which the adapter is configured will usually be ignored. To see those packets, the adapter must be in monitor mode.

The captured information is decoded from raw digital form into a human-readable format that permits users of the protocol analyzer to easily review the exchanged information. Protocol

analyzers vary in their abilities to display data in multiple views, automatically detect errors, determine the root causes of errors, generate timing diagrams, etc.

Some protocol analyzers can also generate traffic and thus act as the reference device; these can act as protocol testers. Such testers generate protocol-correct traffic for functional testing, and may also have the ability to deliberately introduce errors to test for the DUT's ability to deal with error conditions.

Protocol Analyzers can also be hardware-based, either in probe format or, as is increasingly more common, combined with a disk array. These devices record packets (or a slice of the packet) to a disk array. This allows historical forensic analysis of packets without the users having to recreate any fault.

✓ Types of Packet Sniffing

There are basically three types of packet sniffing:

- **ARP Sniffing:** ARP sniffing involves information packets that are sent to the administrator through the ARP cache of both network hosts. Instead of sending the network traffic to both hosts, it forwards the traffic directly to the administrator.
- **IP Sniffing:** IP sniffing works through the network card by sniffing all of the information packets that correspond with the IP address filter. This allows the sniffer to capture all of the information packets for analysis and examination.
- **MAC Sniffing:** MAC sniffing also works through a network card which allows the device to sniff all of the information packets that correspond with the MAC address filter.

✓ Uses

The versatility of packet sniffers means they can be used to:

- Analyze network problems
- Detect network intrusion attempts
- Detect network misuse by internal and external users
- Documenting regulatory compliance through logging all perimeter and endpoint traffic
- Gain information for effecting a network intrusion
- Isolate exploited systems
- Monitor WAN bandwidth utilization
- Monitor network usage (including internal and external users and systems)
- Monitor data-in-motion
- Monitor WAN and endpoint security status
- Gather and report network statistics
- Filter suspect content from network traffic
- Serve as primary data source for day-to-day network monitoring and management
- Spy on other network users and collect sensitive information such as passwords (depending on any content encryption methods that may be in use)
- Reverse engineer proprietary protocols used over the network
- Debug client/server communications
- Debug network protocol implementations
- Verify adds, moves and changes
- Verify internal control system effectiveness (firewalls, access control, Web filter, Spam filter, proxy)

5.2.6. SHA-1

✓ Introduction

This Standard specifies a Secure Hash Algorithm, SHA-1, for computing a condensed representation of a message or a data file. When a message of any length $< 2^{64}$ bits is input, the SHA-1 produces a 160-bit output called a message digest. The message digest can then be input to the Digital Signature Algorithm (DSA) which generates or verifies the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process because the message digest is usually much smaller in size than the message. The same hash algorithm must be used by the verifier of a digital signature as was used by the creator of the digital signature.

The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify. SHA-1 is a technical revision of SHA (FIPS 180).

The Secure Hash Algorithm (SHA-1) is required for use with the Digital Signature Algorithm (DSA) as specified in the Digital Signature Standard (DSS) and whenever a secure hash algorithm is required for federal applications. For a message of length $< 2^{64}$ bits, the SHA-1 produces a 160-bit condensed representation of the message called a message digest. The message digest is used during generation of a signature for the message. The SHA-1 is also used to compute a message digest for the received version of the message during the process of verifying the signature. Any change to the message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.

The SHA-1 is designed to have the following properties: it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest.

6.) SYSTEM DESIGN AND IMPLEMENTATION

- 1) Design and implementation of Packet Sniffer
- 2) Implementation of SHA-1
- 3) Implementation of Socket Programs of server and client
- 4) Design and implementation of normalizer

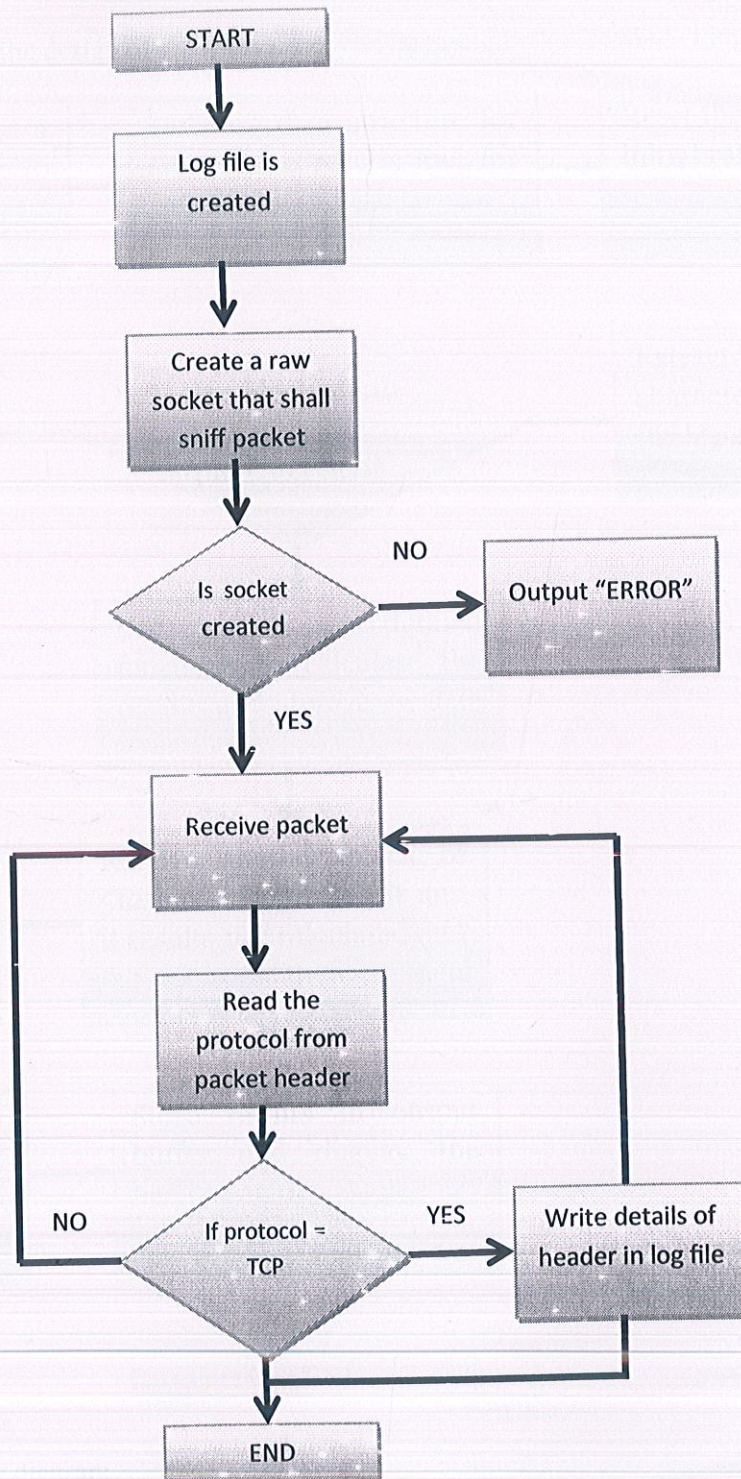
6.1. Implementation of Packet Sniffer

The first phase in the design and development of our normalizer was to implement a packet sniffer module. The first step is to set a TCP connection between the server and a client. The second step is to capture the data packet sent by the TCP client to TCP server. The packet is captured by the normalizer lying between client and server. Packet sniffing module can read different types of protocol such as TCP, IP, ICMP, UDP, however, it will extract header details of TCP packets only. The values of TCP protocol header are saved in the log file.

For the packet sniffer to work, it needs to listen constantly on the RAW socket. During execution, administrative rights need to be given via command 'sudo'. The packet sniffer can catch the details of all the packets that are transmitted. We are primarily interested in the details of the TCP and the IP header to know details like source port, destination port, packet size, sequence number, identification number, data payload etc.

The program makes use of inbuilt library functions that capture the details of TCP header and IP header in data structures. The program runs till the time there are packets coming through the network after which the raw socket is closed.

Figure 6: Packet Sniffer Flowchart



Description of Algorithm:**Introduction**

The SHA-1 is designed to have the following properties: it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest.

Bit Strings and Integers

The following terminology related to bit strings and integers will be used:

- A hex digit is an element of the set $\{0, 1, \dots, 9, A, \dots, F\}$. A hex digit is the representation of a 4-bit string. **Examples:** $7 = 0111$, $A = 1010$.
- A word equals a 32-bit string which may be represented as a sequence of 8 hex digits.

Example

1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

- An integer between 0 and $2^{32} - 1$ inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. **Example:** the integer $291 = 2^8 + 2^5 + 2^1 + 2^0 = 256 + 32 + 2 + 1$ is represented by the hex word, 00000123.
- If z is an integer, $0 \leq z < 2^{64}$, then $z = 2^{32}x + y$ where $0 \leq x < 2^{32}$ and $0 \leq y < 2^{32}$. Since x and y can be represented as words X and Y , respectively, z can be represented as the pair of words (X, Y) .
- Block = 512-bit string. A block (e.g., B) may be represented as a sequence of 16 words.

Operations on Word

The following logical operators will be applied to words:

- Bitwise logical word operations

$X \wedge Y$ = bitwise logical "and" of X and Y .

$X \vee Y$ = bitwise logical "inclusive-or" of X and Y .

$X \text{ XOR } Y$ = bitwise logical "exclusive-or" of X and Y .

$\sim X$ = bitwise logical "complement" of X .

- The operation $X + Y$ is defined as follows: words X and Y represent integers x and y , where $0 \leq x < 2^{32}$ and $0 \leq y < 2^{32}$. For positive integers n and m , let $n \bmod m$ be the remainder upon dividing n by m . Compute $z = (x + y) \bmod 2^{32}$. Then $0 \leq z < 2^{32}$. Convert z to a word, Z , and define $Z = X + Y$.
- The circular left shift operation $S^n(X)$, where X is a word and n is an integer with $0 \leq n < 32$, is defined by

$$S^n(X) = (X \ll n) \text{ OR } (X \gg 32-n).$$
- In the above, $X \ll n$ is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). $X \gg n$ is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus $S^n(X)$ is equivalent to a circular shift of X by n positions to the left.

Message Padding

Suppose a message has length $l < 2^{64}$. Before it is input to the SHA-1, the message is padded on the right as follows:

- "1" is appended. **Example:** if the original message is "01010000", this is padded to "010100001".
- "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Functions used

A sequence of logical functions f_0, f_1, \dots, f_{79} is used in the SHA-1. Each f_t , $0 \leq t \leq 79$, operates on three 32-bit words B, C, D and produces a 32-bit word as output. $f_t(B, C, D)$ is defined as follows: for words B, C, D ,

$$f_t(B, C, D) = (B \text{ AND } C) \text{ OR } ((\text{NOT } B) \text{ AND } D) \quad (0 \leq t \leq 19)$$

$$f_t(B, C, D) = B \text{ XOR } C \text{ XOR } D \quad (20 \leq t \leq 39)$$

$$f_t(B, C, D) = (B \text{ AND } C) \text{ OR } (B \text{ AND } D) \text{ OR } (C \text{ AND } D) \quad (40 \leq t \leq 59)$$

$$f_t(B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (60 \leq t \leq 79).$$

Constants Used

A sequence of constant words $K(0), K(1), \dots, K(79)$ is used in the SHA-1. In hex these are given by

$$K_t = 5A827999 \quad (0 \leq t \leq 19)$$

$$K_t = 6ED9EBA1 \quad (20 \leq t \leq 39)$$

$$K_t = 8F1BBCDC \quad (40 \leq t \leq 59)$$

$$K_t = CA62C1D6 \quad (60 \leq t \leq 79).$$

Computing the Message Digest

The message digest is computed using the final padded message. The computation uses two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. The words of the first 5-word buffer are labeled A,B,C,D,E. The words of the second 5-word buffer are labeled H_0, H_1, H_2, H_3, H_4 . The words of the 80-word sequence are labeled W_0, W_1, \dots, W_{79} . A single word buffer TEMP is also employed.

To generate the message digest, the 16-word blocks M_1, M_2, \dots, M_n defined in Section 4 are processed in order. The processing of each M_i involves 80 steps.

Before processing any blocks, the $\{H_i\}$ are initialized as follows: in hex,

$$H_0 = 67452301$$

$$H_1 = EFCDAB89$$

$$H_2 = 98BADCFE$$

$$H_3 = 10325476$$

$$H_4 = C3D2E1F0.$$

Now M_1, M_2, \dots, M_n are processed. To process M_i , we proceed as follows:

- a. Divide M_i into 16 words W_0, W_1, \dots, W_{15} , where W_0 is the left-most word.
- b. For $t = 16$ to 79 let $W_t = S^1(W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16})$.

TCP STREAM NORMALIZATION

c. Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.

d. For $t = 0$ to 79 do

$TEMP = S^5(A) + f_t(B, C, D) + E + W_t + K_t$;

$E = D$; $D = C$; $C = S^{30}(B)$; $B = A$; $A = TEMP$;

e. Let $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$.

After processing M , the message digest is the 160-bit string represented by the 5 words H_0 H_1 H_2 H_3 H_4 .

6.3. Implementation of Client-Server

Client

The working of the client is shown in the following module:

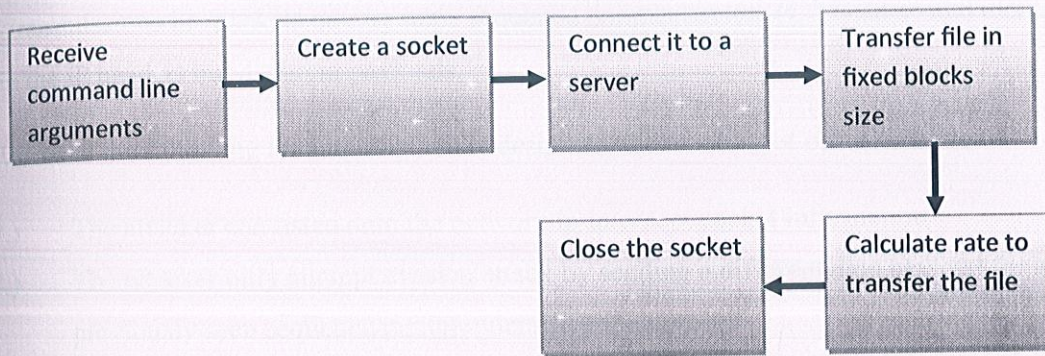


Figure 8:- Flow diagram of the client side program to transfer file to a remote server

Server

The working of the server is explained by the following.

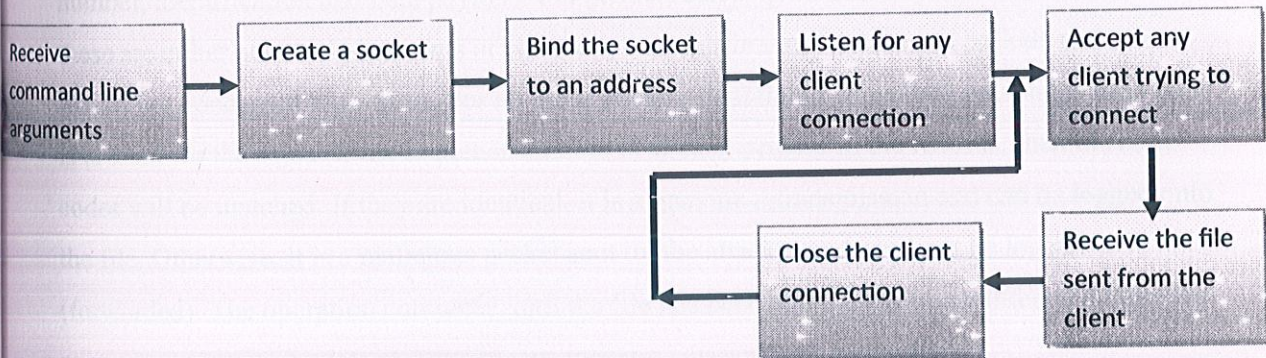


Figure 9:- Flow diagram of the server side program to receive file sent from a remote client

6.4. Implementation of Normalizer

The normalizer works on client/server end where the server/client tries to send a file through TCP connection. TCP protocol ensures that no retransmissions are taken into account while delivering the file to the application. So our concern is mainly to check for inconsistent retransmissions i.e. packets having same sequence number (in the TCP header) and identification (in the IP header) but different payloads.

We make the following assumptions while designing the working of our normalizer:-

- 7.) The attacker can listen onto the network to intercept packet information
- 8.) The attacker only attempt evasion attack by sending a different data payload for a previously seen sequence number/identification number
- 9.) On a 3 node implementation, the channel between normalizer and client is safe (if the server is sending a file)

The normalizer has a packet sniffing module integrated into it so that it can process the information that the packets are carrying.

First of all, every packet is processed to find out three key pieces of information – sequence number, identification and data payload. For packets carrying no data payload it is assumed that these are either acknowledgements or connection establishment/termination packets and hence we are not concerned with them. For all those packets containing some data payload, a hash code is created for the same. If the sequence number is already present in the records, then the hash codes will be matched. If they are identical, it is a genuine retransmission and can be logged onto the file. Otherwise, it is a malicious packet sent by the attacker and need not be logged (forwarded). The operation continues until the file has been completely transmitted and there are no more packets to be detected on the raw socket.

The entire design has been explained with the help of a flowchart in Figure 8.

TCP STREAM NORMALIZATION

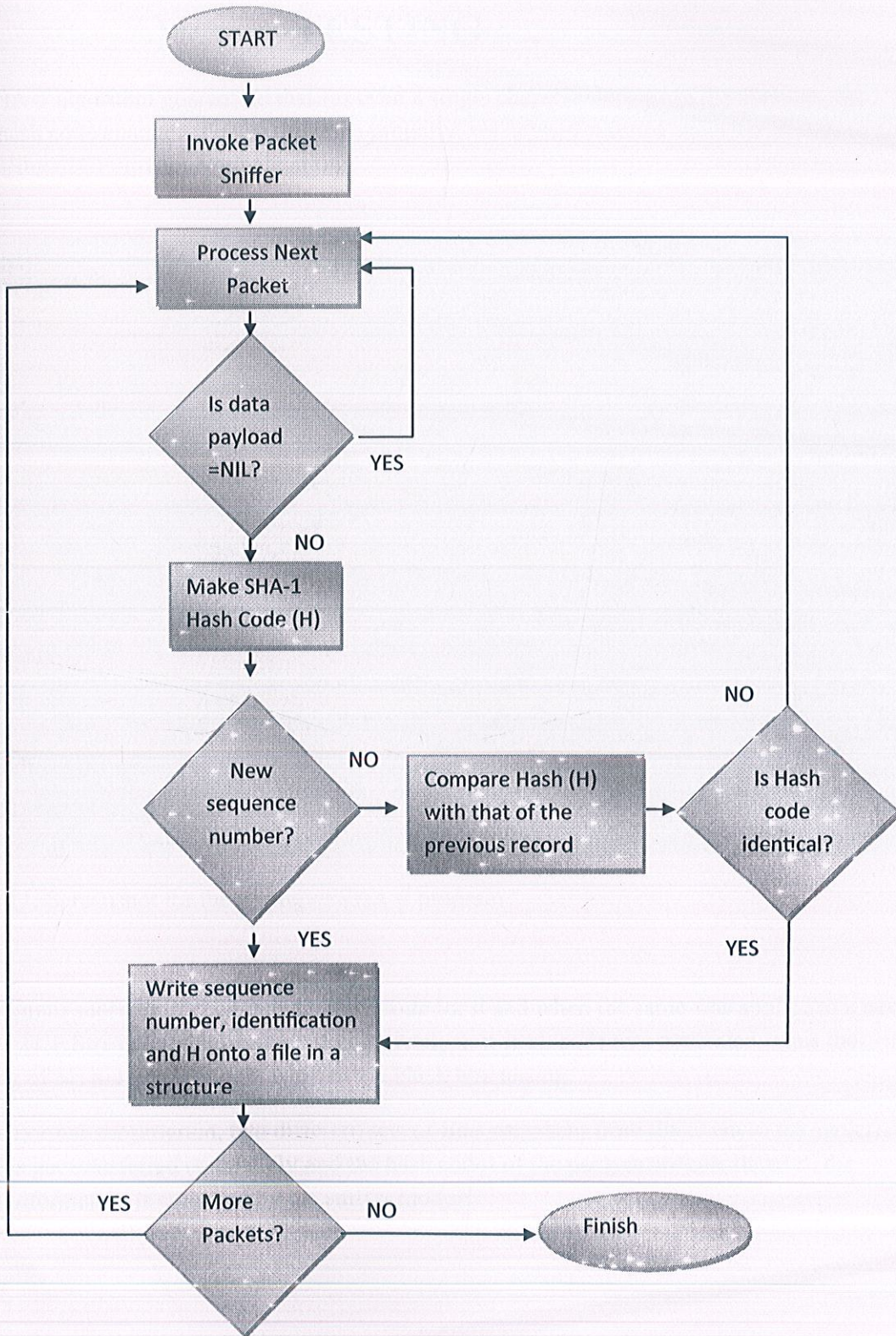
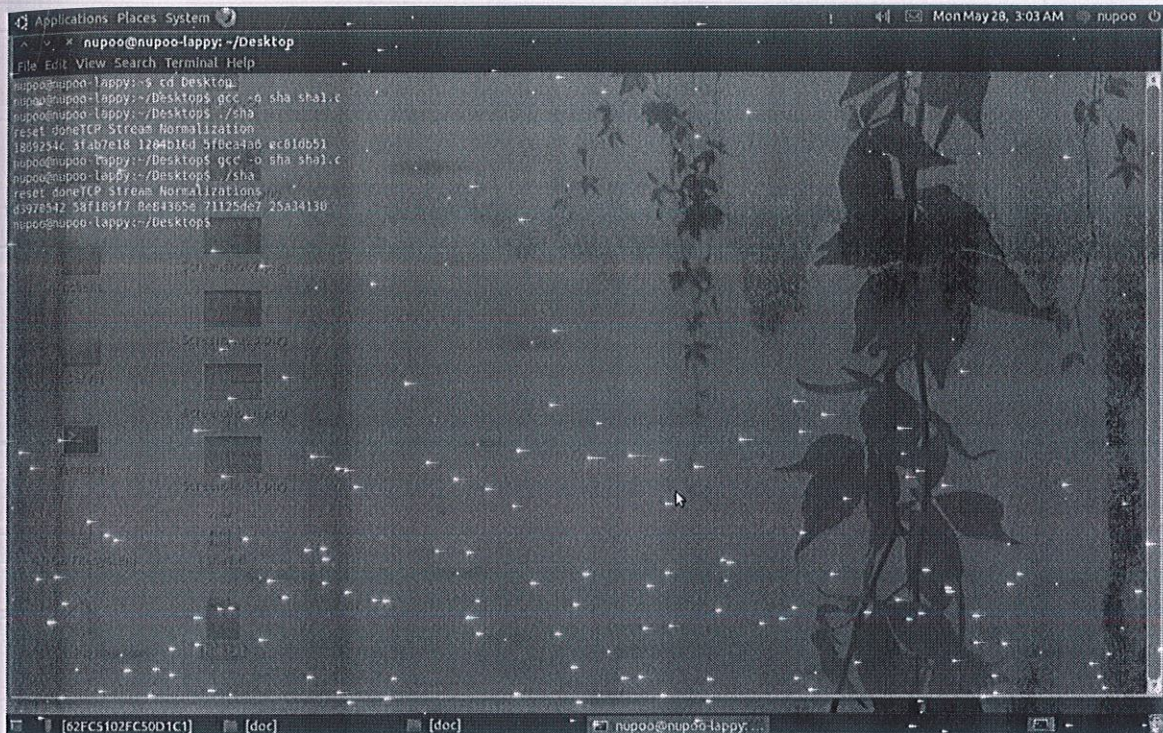


Figure 10: Flowchart for design of normalizer

7.) TESTING

The SHA-1 algorithm guarantees that for even a single character change in the message, the entire hash code changes. To test the same property, we applied a single character change to an input string- TCP Stream Normalization:



```

Applications Places System | Mon May 28, 3:03 AM | nupoo
nupoo@nupoo-lappy: ~/Desktop
File Edit View Search Terminal Help
nupoo@nupoo-lappy:~$ cd desktop
nupoo@nupoo-lappy:~/Desktop$ gcc -o sha sha1.c
nupoo@nupoo-lappy:~/Desktop$ ./sha
reset doneTCP Stream Normalization
1809254c 3fab7e18 1294b1e1 5f8ea4a0 ec010b51
nupoo@nupoo-lappy:~/Desktop$ gcc -o sha sha1.c
nupoo@nupoo-lappy:~/Desktop$ ./sha
reset doneTCP Stream Normalizations
d3978942 58f189f7 a684305e 71125de7 25a34130
nupoo@nupoo-lappy:~/Desktop$
  
```

Figure 11: Screenshot for the testing of SHA-1 program

The program successfully computed a hash code for it and when the same was applied to a new string – TCP Stream Normalizations, a completely new hash code was generated. Thus the working of SHA-1 program was verified via black box testing.

For the normalizer program, two different sets of files were sent from the client to the server. The files get transferred completely and the hash codes of the packets are calculated as the packet information is collected by the sniffer module.

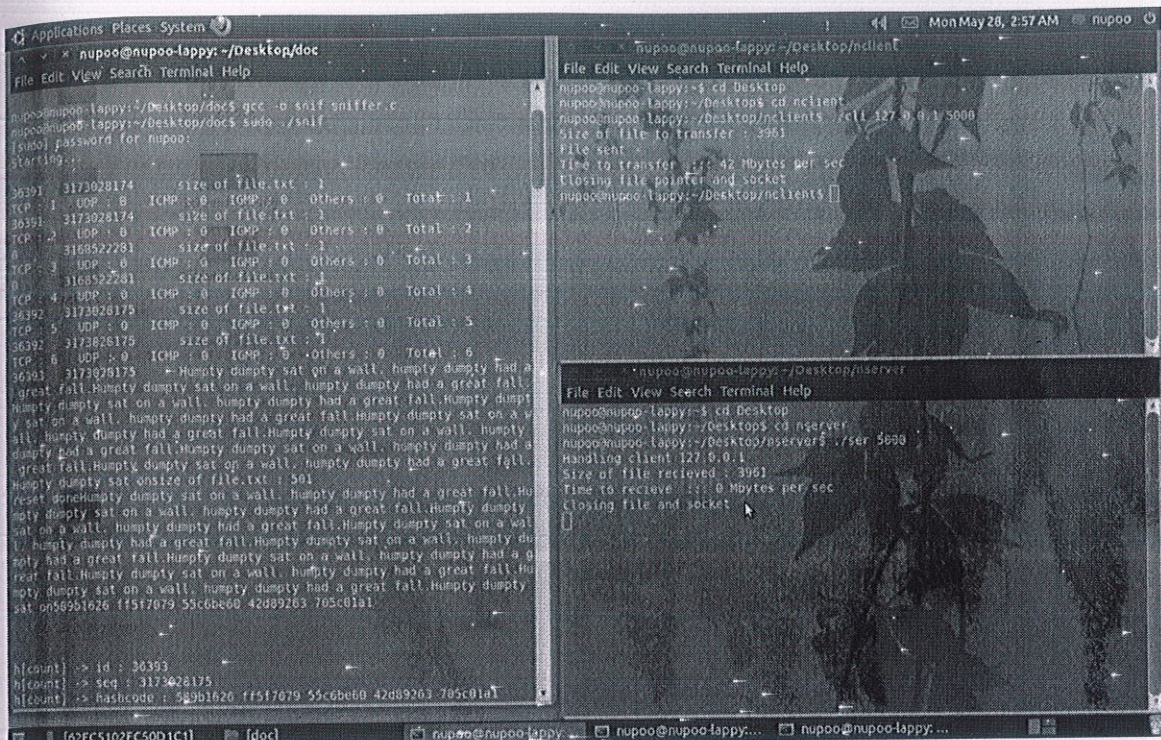


Figure 12: Screenshot for the starting of the normalizer program when the first file is sent

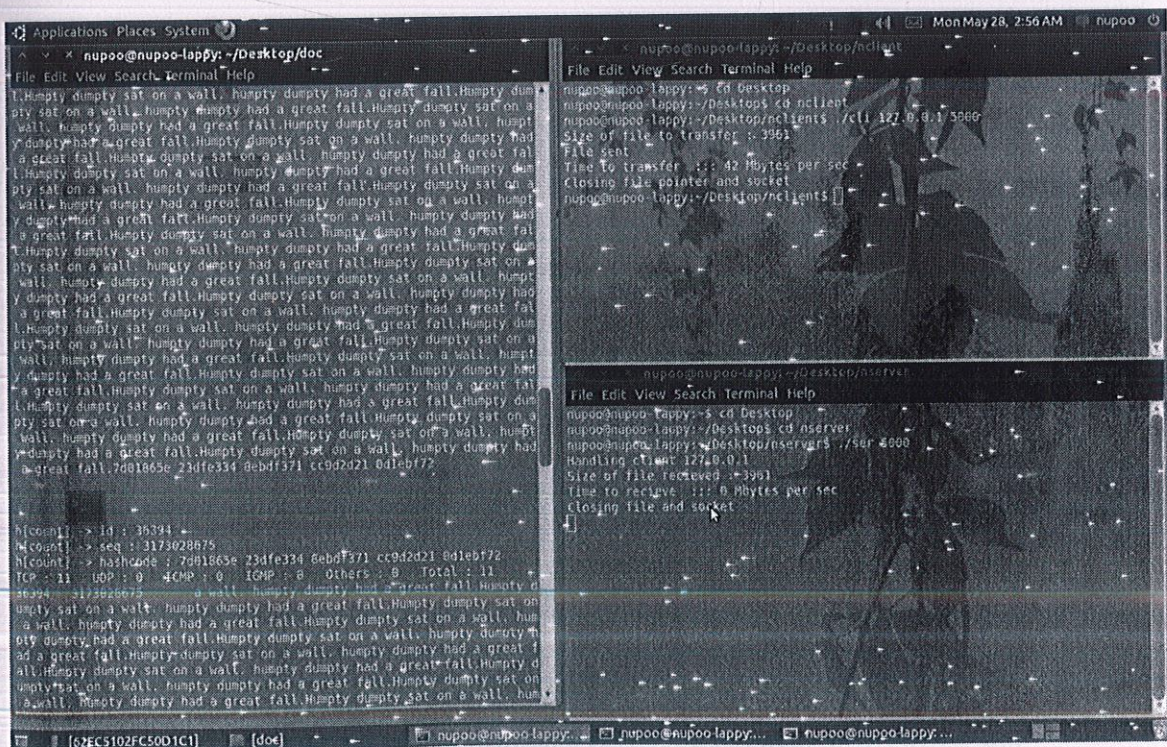


Figure 13: Screenshot for the calculation of hash codes for different packets for file 1

8.) LIMITATIONS

We have implemented the normalizer for TCP at user-level. For high performance a production normalizer would need to run in the kernel rather than at user level, but our current implementation makes testing, debugging and evaluation much simpler.

The application of this project seems more probable on a high speed network where the internal network of a home user/ organization needs to be protected from any malicious activity by an attacker. That would require full control on the transmission of packets so that the inconsistent retransmissions can be blocked and not allowed to pass by the normalizer.

9.) SUMMARY

Defending networks against today's attackers is especially challenging for modern intrusion detection/prevention systems for two reasons: the sheer amount of state they must maintain, and the possibility of resource exhaustion attacks on the defense system itself. Our work shows how to cope with these challenges in the context of a TCP stream normalizer whose job is to detect all instances of inconsistent TCP retransmissions. The two currently used methods to detect inconsistent retransmissions, maintaining complete contents of unacknowledged data, or maintaining only the corresponding hashes suffer from a set of flaws each. Systems that maintain complete contents consume an amount of memory problematic for high-speed operation. Systems that maintain hashes cannot verify the consistency of the 20-30% of retransmissions that fail to preserve original segment boundaries; as a result attackers can easily encode their evasions in these unverified segments. Our normalizer stores hashes of data and verifies the consistency of all retransmissions. The resulting design is necessarily somewhat complex. In considering resource exhaustion attacks, the observation that provisioning for a worst-case traffic pattern is simply impractical led us to develop a simple framework to evict connections when space is at a premium. Thus, our most important conclusion is that TCP stream normalization does not have to choose between correctness and implement ability; it can achieve both goals, while resisting a range of resource exhaustion attacks.

10.) CONCLUSION

The work done in this semester brings us to the end of our Project. Having prepared the three key components namely, the packet sniffer program, the SHA -1 implementation and the setting up of a client server network through socket programming, we were able to combine them to prepare our TCP stream normalizer. These three components essentially form the major part of the requirements for implementing a robust and efficient network normalizer. Next came the logical implementation of how the normalizer works to prevent the typical kind of attack that we discussed i.e. the evasion attack. The client server configuration served the purpose of simulating the actual working of a hardware implementation of the normalizer where the normalizer helps prevent our internal network from any inconsistent retransmissions. The packet sniffer allowed the capturing of essential details required to track and distinguish genuine packets from the others. With the help of the SHA-1 hashing algorithm, the memory requirements of our normalizer were reduced great deal. Comparing the incoming packets with the hash codes of previously monitored packets enabled the working of the normalizer that prepares a table for each incoming packet and holds three key entities – the sequence number, identification field and the hash of data payload. The normalizer was thus successful in demarcating genuine packets forwarded by the TCP protocol from any malicious packet that an attacker may have introduced in the TCP stream.

11.) REFERENCES

- 1) M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in *Proc. USENIX Security Symposium, Aug. 2001*.
- 2) G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting Evasion Attacks at High Speeds without Reassembly," in *Proc. ACM SIGCOMM, Sept. 2006*.
- 3) "Configuring TCP Normalization," 2006,
http://www.cisco.com/en/US/products/ps6120/products_configuration_guide_chapter09186a008054ecb8.html#wp1051891.
- 4) Mythili Vutukuru, H. Balakrishnan, "Efficient and Robust TCP Stream Normalization", IEEE Symposium on Security and Privacy, May 2008.
- 5) Anderson, Ross (2008). "Security Engineering – A Guide to Building Dependable Distributed Systems – 2nd edition. John Wiley & Sons.
- 6) Burns, David (2011). "CCNP Security IPS 642-627 Official Cert Guide". Cisco Press.
- 7) Thomas H. Ptacek, Timothy N. Newsham, "Insertion, Evasion and denial of service: Eluding Network Intrusion Detection". Jan 1998.
- 8) Tanenbaum, Andrew S., "Computer Networks" 4th edition. Pearson Education.
- 9) "Secure Hash Standard", FIPS – 180, National Institute of Standards and Technology.

ANNEXURE- A

This appendix contains implementation of SHA-1 algorithm, the client server programs and network normalizer.

Program 1: Implementation of SHA-1

```
//SHA-1 is designed to work with messages less than 2^64 bits long.

#include <stdio.h>
#include <string.h>
#include <fcntl.h>

//circular shift definition word size is always 32 bits i.e. represented by an unsigned int
#define CircularShift(bits,word) (((word) << (bits)) | (((word) >> (32-(bits))))

typedef struct sha1result
{
    unsigned int output[5];           /* Output (Message digest) */
    unsigned int len_Low;             /* Message length in bits */
    unsigned int len_High;           /* Message length in bit */
    /*length low is for least significant 32 of length and high is for most significant 32 bits of
length*/
    unsigned char message[64];       /* 512-bit message blocks */
    int index;                       /* index into message block array */
    int computed;                    /* Is the digest computed? */
    int flag;                         /* if size of msg is more than 2^64 then flag is
set 1 and output is not computed*/
}DS;

//Function Prototypes
void Reset(DS *);
int Result(DS *);
void Input( DS *, unsigned char *, unsigned);
void ProcessMB(DS *);
void PadMessage(DS *);

int main()
{
    DS sha;                          /* Data structure in which result is stored */
    FILE *fp;                        /* File pointer for reading files*/
    unsigned char c;                 /* Character read from file */
    int r;

    fp=fopen("file.txt","r");
    if(fp==NULL)
    {
```



```

        printf("cannot open file : error");
        exit(0);
    }

    Reset(&sha);          /*Resetting the datastructure and process input*/

    c=fgetc(fp);
    while(!feof(fp))
    {
        Input(&sha, &c, 1);
        c=fgetc(fp);
    }
    fclose(fp);

    r=Result(&sha)
    if(r==0)
    {
        printf("size of file greater than 2^64 : output not calculated");
    }
    else
    {
        printf("%08x %08x %08x %08x %08x \n", sha.output[0], sha.output[1],
            sha.output[2], sha.output[3], sha.output[4]);
    }
    return 0;
}
//definition of Reset function
void Reset(DS *sha)
{
    sha->len_low    = 0;
    sha->len_high   = 0;
    sha->message     = 0;
    sha->output[0]  = 0x67452301;
    sha->output[1]  = 0xEFCDAB89;
    sha->output[2]  = 0x98BADCFE;
    sha->output[3]  = 0x10325476;
    sha->output[4]  = 0xC3D2E1F0;
    sha->computed   = 0;
    sha->flag       = 0;
}

//definition of SHA1Result (it returns 1 if result is computed else returns 0)
int Result(DS *sha)
{
    if (sha->flag)
    {
        return 0;
    }
    if (!sha->computed)

```



```

    {
        PadMessage(sha);
        sha->computed = 1;
    }
    return 1;
}

//Definition of Input
//This function accepts the input in form of octets (one unsigned char)

void Input(DS *sha, unsigned char *c, unsigned length)
{
    if (!length)
    {
        return;
    }
    if (sha->computed || sha->flag)
    {
        sha->flag = 1;
        return;
    }
    while(length-- && !sha->flag)
    {
        sha->message[sha->index++] = (*c & 0xFF);
        sha->len_Low += 8;
        /* Force it to 32 bits */
        sha->len_Low &= 0xFFFFFFFF;
        if(sha->len_Low == 0)
        {
            sha->len_High++;
            /* Force it to 32 bits */
            sha->len_High &= 0xFFFFFFFF;
            if (sha->len_High == 0)
            {
                /* Message is too long */
                sha->flag = 1;
            }
        }
        if(sha->index == 64)
        {
            ProcessMB(sha);
        }
    }
}

//Definition of PadMessage()
//According to the standard, the message must be padded to an even 512 bits. The first padding bit
//must be a '1'.
//The last 64 bits represent the length of the original message. All bits in between should be 0.

```



```

void PadMessage(DS *sha)
{
    /* If current message block is too small to hold the initial padding bits and length then we
    will pad the block, process it, and then continue padding into a second block.*/
    if (sha->index > 55)
    {
        sha->message[sha->index++] = 0x80;
        while(sha->index < 64)
        {
            sha->message[sha->index++] = 0;
        }
        ProcessMB(sha);
        while(sha->index < 56)
        {
            sha->message[sha->index++] = 0;
        }
    }
    else
    {
        sha->message[sha->index++] = 0x80;
        while(sha->message_index < 56)
        {
            sha->message[sha->index++] = 0;
        }
    }

    /* Store the message length as the last 8 octets */

    sha->message[56] = (sha->len_High >> 24) & 0xFF;
    sha->message[57] = (sha->len_High >> 16) & 0xFF;
    sha->message[58] = (sha->len_High >> 8) & 0xFF;
    sha->message[59] = (sha->len_High) & 0xFF;
    sha->message[60] = (sha->len_Low >> 24) & 0xFF;
    sha->message[61] = (sha->len_Low >> 16) & 0xFF;
    sha->message[62] = (sha->len_Low >> 8) & 0xFF;
    sha->message[63] = (sha->len_Low) & 0xFF;
    ProcessMB(sha);
}

//ProcessMB() definition. This function will process the next 512 bits of the message
//stored in the message array.

void ProcessMB(DS *sha)
{
    const unsigned K[] = {0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC,
                          0xCA62C1D6};
    /* Constants defined in SHA-1 */
    int t; /* Loop counter */
    unsigned temp; /* Temporary word value */
    unsigned W[80]; /* Word sequence */
}

```



```

unsigned A, B, C, D, E; /* Word buffers */

/*Initialize the first 16 words in the array W*/
for(t = 0; t < 16; t++)
{
    W[t] = ((unsigned) sha->message[t * 4]) << 24;
    W[t] |= ((unsigned) sha->message[t * 4 + 1]) << 16;
    W[t] |= ((unsigned) sha->message[t * 4 + 2]) << 8;
    W[t] |= ((unsigned) sha->message[t * 4 + 3]);
}

for(t = 16; t < 80; t++)
{
    W[t] = CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
}

A = sha->output[0];
B = sha->output[1];
C = sha->output[2];
D = sha->output[3];
E = sha->output[4];

for(t = 0; t < 20; t++)
{
    temp = CircularShift(5,A) + ((B & C) | ((~B) & D)) + E + W[t] + K[0];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 20; t < 40; t++)
{
    temp = CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 40; t < 60; t++)

```



```

    {
        temp = CircularShift(5,A) + ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 60; t < 80; t++)
    {
        temp = CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = CircularShift(30,B);
        B = A;
        A = temp;
    }

    sha->output[0] = (sha->output[0] + A) & 0xFFFFFFFF;
    sha->output[1] = (sha->output[1] + B) & 0xFFFFFFFF;
    sha->output[2] = (sha->output[2] + C) & 0xFFFFFFFF;
    sha->output[3] = (sha->output[3] + D) & 0xFFFFFFFF;
    sha->output[4] = (sha->output[4] + E) & 0xFFFFFFFF;
    sha->index = 0;
}

```

Program 2: To create a client program using socket APIs to transfer file to a remote server and calculate time taken to send a file

```

/*----- CLIENT SIDE PROGRAM TO SEND FILE-----*/
/*including the various libraries*/

#include <stdio.h> /* for printf() and fprintf() */
#include <string.h> /*for memset()*/
#include <unistd.h> /* for close() */
#include <stdlib.h> /* for atoi() */
#include <errno.h> /*for perror()*/
#include <netinet/in.h>
#include <arpa/inet.h> /* for sockaddr_in and inet_addr() */
#include <sys/time.h> /*for gettimeofday()*/
#include <sys/types.h>
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */

/*-----*/
#define bufsize 50 /*send buffer size*/

```



```

/*-----*/
void DieWithError(char *msg)          /*function to print error and terminate the program
as soon as error is found*/
{
    perror(msg);
    exit(1);
}
/*-----*/
int main(int argc, char *argv[])
{
    int s;                          /*socket descriptor*/
    struct sockaddr_in serverAddr;    /*Server Address*/
    unsigned short serverPort;       /*Server Port*/
    char *serverIP;                  /*dotted quad*/
    char sendstr[bufsize];           /*strings from file to send*/
    char ch;                          /*variable for extracting characters from file*/
    long size=0.0,count=0.0;         /*contains size of file*/
    unsigned short int q;
    FILE *fp;                         /*file pointer to open the file to be sent*/
    struct timeval tv;                /*Timeval structure as an input to gettimeofday()*/
    long start=0,end=0;              /*variable for storing the microseconds before and
after the sending of file*/
    long rate=0;                      /*variable for calculating transfer rate*/

    if(argc < 2 || argc >3) /*if incorrect no of arguments then exit after printing appropriate
message*/
    {
        printf("Incorrect arguments");
        exit(1);
    }
    serverIP=argv[1]; /*storing server IP address received as command line argument*/
    if(argc == 3) /*if port no is also entered then save it else initialize a default value*/
        serverPort=atoi(argv[2]);
    else
        serverPort=4000; /*default port number*/

    /*creates reliable stream socket using TCP*/
    if((s=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
        DieWithError("socket() failed");

    /*construct the server address*/
    memset(&serverAddr,0,sizeof(serverAddr)); /*zero out structure*/
    serverAddr.sin_family=AF_INET; /*Internet address family*/
    serverAddr.sin_addr.s_addr=inet_addr(serverIP); /*server IP address*/
    serverAddr.sin_port=htons(serverPort); /*server Port*/

    /*establish the connection to the echo server*/
    if(connect(s,(struct sockaddr*)&serverAddr, sizeof(serverAddr))<0)
        DieWithError("connect() failed");
    /*find the size of file*/

```



```

fp=fopen("hello.txt", "r");          /*opening file to calculate size*/
if(fp==NULL)
{
    DieWithError("fopen() failed");
}
ch=fgetc(fp);
while(!feof(fp))                    /*this loop finds the size of the input file*/
{
    size++;
    ch=fgetc(fp);
}
printf("Size of file to transfer : %ld\n", size);
fclose(fp);                          /*closing the file pointer*/
count=size; /*initializing count equal to size. This variable keeps track of how many
bytes are sent*/
fp=fopen("hello.txt", "r");          /*opening file to send*/
gettimeofday(&tv, NULL);             /*record the time before sending the file*/
start=tv.tv_usec+tv.tv_sec*1000000; /*storing begin time in microseconds*/
while(1)
{
    if(count>=bufsize) /*if count is greater than or equal to the size of buffer*/
    {
        q=fread(&sendstr,bufsize,1,fp);/*read data from file into the buffer*/
        if(send(s,sendstr,bufsize,0)!=bufsize)/*send the data read*/
            DieWithError("send() sent a different no of bytes than expected");
        count-=bufsize; /*decrement the count by size of buffer(this gives the
remaining bytes to send)*/
    }
    else /*if data left to send is less than the buffer size*/
    {
        q=fread(&sendstr,count,1,fp);/*read the amount of data that is left to send*/
        if(send(s,sendstr,count,0)!=count)/*send the data read*/
            DieWithError("send() sent a different no of bytes than expected");
        break; /*when no data is left to send then break from the while loop*/
    }
}
gettimeofday(&tv, NULL);             /*record the time after sending the file*/
end=tv.tv_usec + tv.tv_sec*1000000; /*storing end time in microseconds*/
printf("File sent \n");
rate=(size/(end-start));
printf("Time to transfer ::: %ld Mbytes per sec\n", rate);/*print rate of transfer*/
printf("Closing file pointer and socket\n");
fclose(fp);                          /*close file pointer*/
close(s);                             /*close socket*/
return 0;
}

```

Program 3: To create a server program using socket APIs to receive file sent from the client and

calculate time taken to receive the file

```

/*-----SERVER SIDE PROGRAM TO RECIEVE FILE-----*/
/*including the various libraries*/

#include <stdio.h>                /*for printf() and fprintf() */
#include <string.h>                /*for memset()*/
#include <unistd.h>                /*for close() */
#include <stdlib.h>                /*for atoi() */
#include <errno.h>                 /*for perror()*/
#include <netinet/in.h>
#include <arpa/inet.h>            /*for sockaddr_in and inet_addr() */
#include <sys/time.h>             /*for gettimeofday()*/
#include <sys/types.h>
#include <sys/socket.h>           /*for socket(), connect(), send(), and recv() */
/*-----*/
#define RCVBUFSIZE 500           //receive buffer size
/*-----*/
void HandleTCPClient(int clntSocket); //function declaration
void DieWithError(char *msg)       //function to print error and terminate the program
as soon as error is found
{
    perror(msg);
    exit(1);
}
/*-----*/
int main(int argc, char *argv[])
{
    int servSock;                /* Socket descriptor for server */
    int clnSock;                 /* Socket descriptor for client */
    struct sockaddr_in serverAddr; /*Local address*/
    struct sockaddr_in clientAddr; /*Client address*/
    unsigned short serverPort;    /* Server port */
    unsigned int clntLen;         /* Length of client address data structure */

    if(argc!=2)                  /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage : %s <server Port>\n", argv[0]);
        exit(1);
    }
    serverPort=atoi(argv[1]);   /* First argument: local port */

    /*create socket for incoming connections*/
    if((servSock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
        DieWithError("socket() failed");
    /*construct local address structure*/
    memset(&serverAddr,0,sizeof(serverAddr)); /* Zero out structure */
    serverAddr.sin_family=AF_INET;           /* Internet address family */
    serverAddr.sin_addr.s_addr=htonl(INADDR_ANY); /* Any incoming interface */
    serverAddr.sin_port=htons(serverPort);   /* Local port */

```



```

/*Bind to the local address*/
if(bind(servSock,(struct sockaddr*)&serverAddr, sizeof(serverAddr))<0)
    DieWithError("bind() failed");

/*Mark the socket so it will listen for incoming connections*/
if(listen(servSock,5)<0)
    DieWithError("listen() failed");

while(1)          /*Run Forever*/
{
    /*set the size of the in-out parameter*/
    clntLen=sizeof(clientAddr);
    if((clnSock=accept(servSock,(struct sockaddr*)&clientAddr,&clntLen))<0)
        DieWithError("Accept() failed");

    /*Client Sock is connected to a client!*/
    printf("Handling client %s \n",inet_ntoa(clientAddr.sin_addr));
    HandleTCPClient(clnSock);

}
close(servSock);
}

/*-----*/
/*function for handling clients*/
void HandleTCPClient(int clntSocket)
{
    FILE *fp;                /*File Pointer*/
    long sz=0;               /*for storing how many bytes received*/
    int recvMsgSize=0,q;
    char Buffer[RCVBUFSIZE]; /* Buffer for receiving file sent from client */
    struct timeval tv;
    long start=0,end=0,rate=0; /*variable for storing time at the beginning and at the
end of receive()*/

    fp=fopen("hello.txt", "w");

    /*receive file from client*/
    gettimeofday(&tv, NULL); /*record time at the start of receive file*/
    start=tv.tv_usec+tv.tv_sec*1000000; /*storing begin time in microseconds*/
    if((recvMsgSize = recv(clntSocket,Buffer,RCVBUFSIZE,0))<0)/* start receiving file*/
        DieWithError("recv() Failed");
    fwrite(&Buffer,recvMsgSize,1,fp); /*writing the data received to a file*/
    sz+=recvMsgSize; /*increment the size of file received by the no of
bytes received*/
    while(recvMsgSize>0) /*till data left to be received*/
    {

        if((recvMsgSize = recv(clntSocket,Buffer,RCVBUFSIZE,0))<0)/* receive data */
            DieWithError("recv() Failed\n");
    }
}

```



```

        fwrite(&Buffer,rcvMsgSize,1,fp);/*write received data to file*/
        sz+=rcvMsgSize; /*increment the size of file received by the no of bytes
received*/
    }

    gettimeofday(&tv, NULL); /*record time at the end of receive file*/
    end=tv.tv_usec+tv.tv_sec*1000000; /*storing end time in microseconds*/
    rate=(sz/(end-start)); /*finding rate of transfer in MBps*/
    printf("Size of file received : %ld\n", sz);
    printf("Time to receive ::: %ld Mbytes per sec\n", rate);/*print rate of transfer*/

    printf("Closing file and socket\n");

    fclose(fp); /*close file pointer*/
    close(clntSocket); /*close client socket*/
}
/*-----*/

```

Program 4: To create the normalizer that prevents inconsistent TCP retransmissions

```

/*-----NORMALIZER PROGRAM-----*/

#include<netinet/in.h>
#include<errno.h>
#include<netdb.h>
#include<stdio.h> //For standard things
#include<stdlib.h> //malloc
#include<string.h> //strlen
#include<netinet/ip_icmp.h> //Provides declarations for icmp header
#include<netinet/udp.h> //Provides declarations for udp header
#include<netinet/tcp.h> //Provides declarations for tcp header
#include<netinet/ip.h> //Provides declarations for ip header
#include<netinet/if_ether.h> //For ETH_P_ALL
#include<net/ethernet.h> //For ether_header
#include<sys/socket.h>
#include<arpa/inet.h>
#include<sys/ioctl.h>
#include<sys/time.h>
#include<sys/types.h>
#include<unistd.h>
#include <fcntl.h>

//circular shift definition word size is always 32 bits ie represented by an unsigned int
#define CircularShift(bits,word) (((word) << (bits)) & 0xFFFFFFFF) | ((word) >> (32-(bits)))

typedef struct sha1result
{

```



```

    unsigned int output[5];      /* Output (Message digest) */
    unsigned int len_Low;       /* Message length in bits */
    unsigned int len_High;      /* Message length in bit */
    /*length low is for least significant 32 of length and high is for most significant 32 bits of
length*/
    unsigned char message[64]; /* 512-bit message blocks */
    int index;                  /* index into message block array*/
    int computed;               /* Is the digest computed? */
    int flag;                   /* if size of msg is more than 2^64 then flag is set 1 and output is not
computed */
}DS;
//Fuction Prototypes
void Reset(DS *);
int Result(DS *);
void Input( DS *, unsigned char *,unsigned);
void ProcessMB(DS *);
void PadMessage(DS *);

void ProcessPacket(unsigned char*, int);
void print_ip_header(unsigned char*, int);
void print_tcp_packet(unsigned char *, int );
void PrintData (unsigned char*, int);

typedef struct hash
{
    unsigned short id;
    unsigned int seq;
    unsigned int hashcode[5];
}hash;

    unsigned short id1;
    unsigned int seq1;
    struct hash h[10000];

FILE *logfile,*b;
int count=0;
struct sockaddr_in source,dest;
int tcp=0,udp=0,icmp=0,others=0,igmp=0,total=0,i,j;

int main()
{
    int saddr_size , data_size;
    struct sockaddr saddr;
    unsigned char *buffer = (unsigned char *) malloc(65536);
    logfile=fopen("logfile.txt","w");

```



```

b=fopen("result.txt","w");
if(logfile==NULL)
{
    printf("Unable to create log.txt file.");
}
if(b==NULL)
{
    printf("unable to create result file");
}
printf("Starting...\n");
int sock_raw = socket( AF_PACKET , SOCK_RAW , htons(ETH_P_ALL));

if(sock_raw < 0)
{
    //Print the error with proper message
    perror("Socket Error");
    return 1;
}
while(1)
{
    saddr_size = sizeof saddr;
    //Receive a packet
    data_size = recvfrom(sock_raw , buffer , 65536 , 0 , &saddr , (socklen_t*)&saddr_size);
    if(data_size < 0 )
    {
        printf("Recvfrom error , failed to get packets\n");
        return 1;
    }
    //Now process the packet
    ProcessPacket(buffer , data_size);
}
close(sock_raw);
printf("Finished");
return 0;
}

void ProcessPacket(unsigned char* buffer, int size)
{
    //Get the IP Header part of this packet , excluding the ethernet header
    struct iphdr *iph = (struct iphdr*)(buffer + sizeof(struct ethhdr));
    ++total;
    switch (iph->protocol) //Check the Protocol and do accordingly..
    {
        case 1: //ICMP Protocol
            ++icmp;
            break;

        case 2: //IGMP Protocol
            ++igmp;
            break;
    }
}

```



```

case 6: //TCP Protocol

    ++tcp;
    print_tcp_packet(buffer , size);
    break;

case 17: //UDP Protocol
    ++udp;
    break;

default: //Some Other Protocol like ARP etc.
    ++others;
    break;
}
printf("TCP : %d UDP : %d ICMP : %d IGMP : %d Others : %d Total : %d\r", tcp , udp , icmp ,
igmp , others , total);
}

void print_ip_header(unsigned char* Buffer, int Size)
{
    unsigned short iphdrlen;
    struct iphdr *iph = (struct iphdr*)(Buffer + sizeof(struct ethhdr) );
    iphdrlen = iph->ihl*4;
    memset(&source, 0, sizeof(source));
    source.sin_addr.s_addr = iph->saddr;
    memset(&dest, 0, sizeof(dest));
    dest.sin_addr.s_addr = iph->daddr;
    fprintf(logfile, "\n");

    fprintf(logfile, "\n |-Identification : %d\t", ntohs(iph->id));
    id1=ntohs(iph->id);
    h[count].id=id1;
    printf("\n%d\t", id1);
}

void print_tcp_packet(unsigned char* Buffer, int Size)
{
    unsigned short iphdrlen;
    struct iphdr *iph = (struct iphdr*)( Buffer + sizeof(struct ethhdr) );
    iphdrlen = iph->ihl*4;
    struct tcphdr *tcph=(struct tcphdr*)(Buffer + iphdrlen + sizeof(struct ethhdr));
    int header_size = sizeof(struct ethhdr) + iphdrlen + tcph->doff*4;

    fprintf(logfile, "\n\n*****TCP Packet*****\n");
    print_ip_header(Buffer,Size);

    fprintf(logfile, " |-Sequence Number : %u\t", ntohl(tcph->seq));
    seq1=ntohl(tcph->seq);
    h[count].seq=seq1;
}

```



```

printf("%u\t", seq1);

//fprintf(logfile, "Data Payload\t");
PrintData(Buffer + header_size, Size - header_size);
fprintf(logfile,
"\n#####");
}

void PrintData(unsigned char* data, int Size)
{
    int i, j;
    FILE *hh,*fp,*pc;
    DS sha; /* Data structure in which result is stored */
    unsigned char c; /* Character read from file */
    int r,a;
    hh=fopen("file.txt","w");
    if(hh==NULL)
    {
        printf("Unable to create file.txt");
    }

    for(i=0 ; i < Size ; i++)
    {
        if( i%16 ==0) //if one line of hex printing is complete...
        {
            //fprintf(logfile, " ");
            for(j=i-16 ; j<i ; j++)
            {
                if(data[j]>=32 && data[j]<=128)
                {
                    fprintf(logfile, "%c", (unsigned char)data[j]); //if its a
number or alphabet
                    fprintf(hh, "%c", (unsigned char)data[j]);
                    printf("%c", (unsigned char)data[j]);
                }
                else
                { //fprintf(logfile, "."); //otherwise print a dot
                }
            }
            //fprintf(logfile, "\n");
        }
        if(i%16!=0) //fprintf(logfile, " ");
        {
            //fprintf(logfile, " %02X", (unsigned int)data[i]);
        }

        if( i==Size-1) //print the last spaces
        {
            for(j=0;j<15-i%16;j++)
            {

```



```

        //fprintf(logfile, " "); //extra spaces
    }
    //fprintf(logfile, " ");
    for(j=i-i%16; j<=i; j++)
    {
        if(data[j]>=32 && data[j]<=128)
        {
            fprintf(logfile, "%c", (unsigned char) data[j]);
            fprintf(hh, "%c", (unsigned char) data[j]);
            printf("%c", (unsigned char) data[j]);
        }
        else
        {
            //fprintf(logfile, ".");
        }
    }
    //fprintf(logfile, "\n");
}
}
fclose(hh);
int size=0;
fp=fopen("file.txt", "r");
if(fp==NULL)
{
    printf("cannot open file file.txt in read mode 1: error");
    exit(0);
}
while(!feof(fp))
{
    fgetc(fp);
    size++;
}
printf("size of file.txt : %d\n", size);
fclose(fp);
if(size!=1)
{
    pc=fopen("file.txt", "r");
    if(pc==NULL)
    {
        printf("Cannot open file.txt in read mode 2: error");
        exit(0);
    }
    Reset(&sha); //Resetting the datastructure and process input
    printf("reset done");
    c=fgetc(pc);
    while(!feof(pc))
    {
        printf("%c", c);
        Input(&sha, &c, 1);
    }
}

```



```

        c=fgetc(pc);
    }
    fclose(pc);
    r=Result(&sha);
    if(r==0)
    {
        printf("size of file greater than 2^64 : output not calculated");
    }
    else
    {
        printf("%08x %08x %08x %08x %08x\n",sha.output[0],sha.output[1],sha.output[2],sha.output[3],sha.output[4]);
        h[count].hashcode[0]=sha.output[0];
        h[count].hashcode[1]=sha.output[1];
        h[count].hashcode[2]=sha.output[2];
        h[count].hashcode[3]=sha.output[3];
        h[count].hashcode[4]=sha.output[4];
        printf("\n\nh[count] -> id : %d\n",h[count].id);
        printf("h[count] -> seq : %u\n",h[count].seq);
        printf("h[count] -> hashcode : %08x %08x %08x %08x %08x\n",h[count].hashcode[0],h[count].hashcode[1],h[count].hashcode[2],h[count].hashcode[3],h[count].hashcode[4]);
        fwrite(&h[count],sizeof h,1, b);
        count++;
    }
}
}
//definition of Reset function
void Reset(DS *sha)
{
    sha->len_Low = 0x00000000;
    sha->len_High = 0x00000000;
    strcpy(sha->message,"");
    sha->output[0] = 0x67452301;
    sha->output[1] = 0xEFCDAB89;
    sha->output[2] = 0x98BADCFE;
    sha->output[3] = 0x10325476;
    sha->output[4] = 0xC3D2E1F0;
    sha->computed = 0;
    sha->flag = 0;
}

//definition of SHA1Result (it returns 1 if result is computed else returns 0)
int Result(DS *sha)
{
    if (sha->flag)
    {
        return 0;
    }
}

```



```

    if (sha->computed == 0)
    {
        PadMessage(sha);
        sha->computed = 1;
    }
    return 1;
}
//definition of Input
//This function accepts the input in form of octets (one unsigned char)

void Input(DS *sha, unsigned char *c, unsigned length)
{
    if (length==0)
    {
        return;
    }
    if (sha->computed || sha->flag)
    {
        sha->flag = 1;
        return;
    }
    while(length-- && !sha->flag)
    {
        sha->message[sha->index++] =(*c & 0xFF);
        sha->len_Low += 8;
        // Force it to 32 bits
        sha->len_Low &= 0xFFFFFFFF;
        if(sha->len_Low == 0)
        {
            sha->len_High++;
        }
        // Force it to 32 bits
        sha->len_High &= 0xFFFFFFFF;
        if (sha->len_High == 0)
        {
            //Message is too long
            sha->flag = 1;
        }
    }
    if(sha->index == 64)
    {
        ProcessMB(sha);
    }
}

//definition of PadMessage()
//According to the standard, the message must be padded to an even 512 bits.The first padding bit
must be a '1'.
//The last 64 bits represent the length of the original message. All bits in between should be 0.

```



```

void PadMessage(DS *sha)
{
//If current message block is too small to hold the initial padding bits and length then we will pad
the block, process it, and then continue padding into a second block.
    if (sha->index > 55)
    {
        sha->message[sha->index++] = 0x80;
        while(sha->index < 64)
        {
            sha->message[sha->index++] = 0;
        }
        ProcessMB(sha);
        while(sha->index < 56)
        {
            sha->message[sha->index++] = 0;
        }
    }
    else
    {
        sha->message[sha->index++] = 0x80;
        while(sha->index < 56)
        {
            sha->message[sha->index++] = 0;
        }
    }
    // Store the message length as the last 8 octets
    sha->message[56] = (sha->len_High >> 24) & 0xFF;
    sha->message[57] = (sha->len_High >> 16) & 0xFF;
    sha->message[58] = (sha->len_High >> 8) & 0xFF;
    sha->message[59] = (sha->len_High) & 0xFF;
    sha->message[60] = (sha->len_Low >> 24) & 0xFF;
    sha->message[61] = (sha->len_Low >> 16) & 0xFF;
    sha->message[62] = (sha->len_Low >> 8) & 0xFF;
    sha->message[63] = (sha->len_Low) & 0xFF;
    ProcessMB(sha);
}
//ProcessMB definition. This function will process the next 512 bits of the message stored in the
message array.
void ProcessMB(DS *sha)
{
    const unsigned int K[4] = {0x5A827999,0x6ED9EBA1,0x8F1BBCDC,0xCA62C1D6};
    //Constants defined in SHA-1
    int t; // Loop counter
    unsigned int temp; // Temporary word value
    unsigned int W[80]; // Word sequence
    unsigned int A, B, C, D, E; //Word buffers

    //Initialize the first 16 words in the array
    for(t = 0; t < 16; t++)
    {

```



```

    W[t] = ((unsigned) sha->message[t * 4]) << 24;
    W[t] |= ((unsigned) sha->message[t * 4 + 1]) << 16;
    W[t] |= ((unsigned) sha->message[t * 4 + 2]) << 8;
    W[t] |= ((unsigned) sha->message[t * 4 + 3]);
}
for(t = 16; t < 80; t++)
{
    W[t] = CircularShift(1,(W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]));
}
A = sha->output[0];
B = sha->output[1];
C = sha->output[2];
D = sha->output[3];
E = sha->output[4];

for(t = 0; t < 20; t++)
{
    temp = CircularShift(5,A) + ((B & C) | ((~B) & D)) + E + W[t] + K[0];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 20; t < 40; t++)
{
    temp = CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 40; t < 60; t++)
{
    temp = CircularShift(5,A) + ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 60; t < 80; t++)
{

```


TCP STREAM NORMALIZATION

```
temp = CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
temp &= 0xFFFFFFFF;
E = D;
D = C;
C = CircularShift(30,B);
B = A;
A = temp;
}
sha->output[0] =(sha->output[0] + A) & 0xFFFFFFFF;
sha->output[1] =(sha->output[1] + B) & 0xFFFFFFFF;
sha->output[2] =(sha->output[2] + C) & 0xFFFFFFFF;
sha->output[3] =(sha->output[3] + D) & 0xFFFFFFFF;
sha->output[4] =(sha->output[4] + E) & 0xFFFFFFFF;
sha->index = 0;
}
/*-----*/
```