Protocol Compiler Design & Implementation in Network Security through Deep Packet Inspection

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Software Engineering

Thapar University, Patiala

By:
Anranya Yadav
(80631002)

Under the supervision of

Dr. Maninder Singh

Assistant Professor

MAY 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Protocol Compiler Design & Implementation in Network Security through Deep Packet Inspection"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted to Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Maninder Singh* and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(**Anranya Yadav*)***
Roll No. 80631002

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Maninder Singh)
Supervisor
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. (Mrs.) Seema Bawa)
Professor & Head
Computer Science & Engineering Department
Thapar University
Patiala

(Dr. R.K. Sharma)
Dean, Academic Affairs
Thapar University
Patiala

Acknowledgement

"Ideals are like stars: you will not succeed in touching them with your hands, but like the seafaring man on the ocean desert of waters, you choose them as your guides, and following them, you reach your destiny."

-Carl Schurz (1829 - 1906)

I am highly thankful to my teacher and guide Dr. Maninder Singh, Assistant Professor, Computer Science & Engineering Department, Thapar University, Patiala, for his advice, motivation, guidance, moral support, efforts and the attitude with which he solved all of my queries in making this thesis possible. It has been a great honor to work under him.

I am also thankful to Dr.(Mrs.) Seema Bawa, Professor and Head**,** Computer Science and Engineering Department, Thapar University, Patiala, for providing us with adequate infrastructure in carrying the research work.

I am also thankful to Mr. Gyan Shukla, Mr. Gobind Bansal (Amazon India Development Center, Bangalore) for providing me proper support and resources.

I am also thankful to Ms. Damandeep Kaur (PG Coordinator), Dr. Varinder Pal Singh and Mr. Balwinder Singh for their contribution in making this thesis possible. Besides the above dignitaries, I am also thankful to my friends Sanmeet Kaur, Maninder Singh, Navdeep Singh, Anju Sharma, Shashi Bhanwar, Shilpi Gupta, Gurpreet Kaur, Neha Sood, Anchal Jain, Tarun k Agrawal, Digumber Dutt Joshi, Chetan Jain, Ashish Jain, Neeraj, Nikhil, Maina for their kind and noble support.

Anranya Yadav

ABSTRACT

A key step in the semantic analysis of network traffic is to parse the traffic stream according to the high-level protocols it contains. This process transforms raw bytes into structured, typed, and semantically meaningful data fields that provide a high-level representation of the traffic. However, constructing protocol parsers by hand is a tedious and error-prone affair due to the complexity and sheer number of application protocols.

Because of the drastically increasing need of rapid protocol deployment, automatic protocol implementation has attracted a lot of attention. Although there are some available commercial tools for protocol compiling, usually generated code by these tools is used only for prototyping due to its poor quality, and optimized protocol compiling is the only way to bring out the maximal potential of generated code.

In this thesis work we will develop a protocol compiler, a declarative language and the compiler designed, will be used to parse the network traffic's payload or contents and to find the various anomalies hidden inside the network data packets. This work can work as a full fledged layer in Network Security.

TABLE OF CONTENTS

LIST OF FIGUERS

Chapter 1

Introduction

1.1 Security

1.1.1 Network Security

Computer Network is collection of two or more than two computers connected together to communicate via any wired or wireless media. *Network security* is the process of preventing and detecting unauthorized use of your computer. Prevention measures help you to stop unauthorized users (also known as "intruders") from accessing any part of your computer system. Detection helps you to determine whether or not someone attempted to break into your system, if they were successful, and what they may have done[1].

The computers and other internet devices are used for everything from banking and investing to online shopping and communicating with others through

email or using some chat programs. Network security consists of the provisions made in an underlying computer network infrastructure, policies adopted by the network administrator to protect the network and the network-accessible resources from unauthorized access and the effectiveness of these measures combined together.

1.11.1.2 Comparison with Computer Security

Securing network infrastructure is like securing possible entry points of attacks on a country by deploying appropriate defense. Computer security is more like providing means to protect a single PC against outside intrusion. The preventive measures attempt to secure the access to individual computers--the network itself--thereby protecting the computers and other shared resources such as printers, network-attached storage connected by the network. Attacks could be stopped at their entry points before they spread. As opposed to this, in

computer security the measures taken are focused on securing individual computer hosts. A host which is compromised is likely to infect other hosts connected to a potentially unsecured network. A computer host's security is vulnerable to users with higher access privileges to those hosts[2].

1.21.1.3 Attributes of a Secure Network

Network security begins from authentication of any user, most likely an username and a password are used for this purpose. Once authenticated, firewall enforces access policies such as what services are allowed to be accessed by the network users. Even being effective to prevent unauthorized access, this component fails to check potentially harmful contents such as computer worms, Trojan horses being transmitted over the network. An intrusion prevention system (IPS) helps detect and prevent such threats. Honeypots, essentially decoy network-accessible resources, could be deployed in

a network as surveillance and early-warning tools[2].

1.31.1.4 Security Management

Security Management for networks is different for all kinds of situations. A small home would only require basic security. While large businesses, if there are a no. of computers and high performance servers like mail servers, DNS are used then it will require high maintenance and advanced software and hardware to prevent malicious attacks from hacking and spamming[2].

- A basic firewall.
- A basic Antivirus software.
- When using a wireless connection, we should use a robust password.
- A strong firewall and proxy to keep unwanted people out.
- A strong Antivirus software and Internet Security Software.
- For authentication, use strong passwords and change it on a weekly/bi-weekly basis.
- Prepare a network analyzer or network monitor and use it when needed.
- Implement physical security management like closed circuit television for entry areas and restricted zones.
- Security fencing to mark the company's perimeter.
- When using a wireless connection, use a robust password.

1.2Network Security Tools and Techniques

The following tools and techniques can be used to for making a secure network.

1.2.1 Using Firewall for Network Security

A firewall is a system or group of systems that enforces an access control policy between two or more networks. The actual means by which this is accomplished

varies widely, but in principle, the firewall can be thought of as a pair of mechanisms: one which exists to block traffic, and the other which exists to permit traffic. Some firewalls place a greater emphasis on blocking traffic, while others emphasize permitting traffic. Probably the most important thing to recognize about a firewall is that it implements an access control policy. It's also important to recognize that the firewall's configuration, because it is a mechanism for enforcing policy, imposes its policy on everything behind it. Administrators for firewalls managing the connectivity for a large number of hosts therefore have a heavy responsibility[3]. .

1.2.1.1 Need of Firewall

The Internet, like any other society, is plagued with the kind of jerks who enjoy the electronic equivalent of writing on other people's walls with spray paint, tearing their mailboxes off, or just sitting in the street blowing their car horns. Some people try to get real work done over the Internet, and others have sensitive or proprietary data they must protect. Usually, a firewall's purpose is to keep the protect a private network from the security threats that may come form the other external public networks. Generally firewalls are combination of hardware and software.

Many traditional-style corporations and data centers have computing security policies and practices that must be followed. In a case where a company's policies dictate how data must be protected, a firewall is very important, since it is the embodiment of the corporate policy. Frequently, the hardest part of hooking to the Internet, if you're a large company, is not justifying the expense or effort, but convincing management that it's safe to do so. A firewall provides not only real security--it often plays an important role as a security blanket for management.

Firewall is used to protect our private internal network from offensive Web sites and potential hackers.

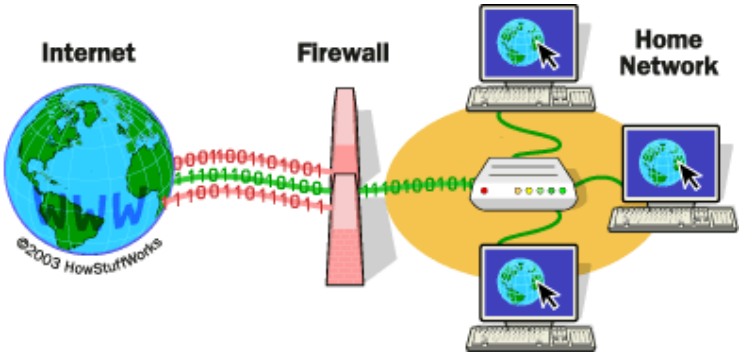

Figure 1.1. Firewall [3]

Basically, a firewall is a barrier to keep destructive forces away from our property. In fact, that's why its called a firewall. Its job is similar to a physical firewall that keeps a fire from spreading from one area to the next. Some firewalls permit only email traffic through them, thereby protecting the network against any attacks other than attacks against the email service. Other firewalls provide less strict protections, and block services that are known to be problems. Generally, firewalls are configured to protect against unauthenticated interactive logins from the external world. More elaborate firewalls block traffic from the outside to the inside, but permit users on the inside to communicate freely with the outside. The firewall can protect a private network against most of the network-borne attack if you unplug it[3].

1.2.1.2 Basic types of Firewall

Conceptually, there are three types of firewalls:

1. Network layer
2. Application layer
3. Hybrids

It’s depends on network administrators which type of firewall they want to use. Which is which depends on what mechanisms the firewall uses to pass traffic from one security zone to another. The International Standards Organization (ISO) Open Systems Interconnect (OSI) model for networking defines seven layers, where each layer provides services that higher-level layers depend on. In order from the bottom, these layers are physical, data link, network, transport, session, presentation, application. The important thing to recognize is that the lower-level the forwarding

mechanism, the less examination the firewall can perform. Generally speaking, lower-level firewalls are faster, but are easier to fool into doing the wrong thing. These days, most firewalls fall into the hybrid category, which do network filtering as well as some amount of application inspection. The amount changes depending on the vendor, product, protocol and version, so some level of digging and/or testing is often necessary[2].

- **Network Layer Firewall**

These generally make their decisions based on the source, destination addresses and ports in individual IP packets. A simple router is the traditional network layer firewall, since it is not able to make particularly sophisticated decisions about what a packet is actually talking to or where it actually came from. Modern network layer firewalls have become increasingly sophisticated, and now maintain internal information about the state of connections passing through them, the contents of some of the data streams, and so on. One thing that's an important distinction about many network layer firewalls is that they route traffic directly though them, so to use one you either need to have a validly assigned IP address block or to use a private internet address block. Network layer firewalls tend to be very fast and tend to be very transparent to users[3].

Screened Host Firewall:

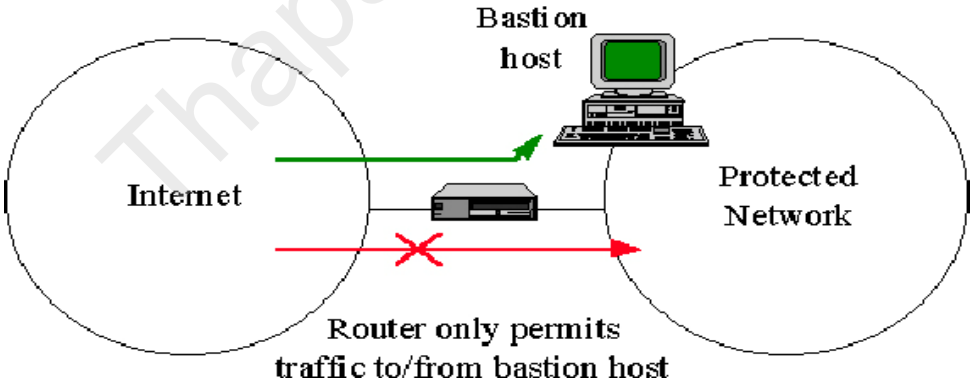

Figure 1.2 Screened Host Firewall[3]

In the above Figure 1.2 a network layer firewall called a Screened Host Firewall is represented. In a screened host firewall, access to and from a single host is controlled by means of a router operating at a network layer. The single host is a bastion host; a highly-defended and secured strong-point that can resist attack.

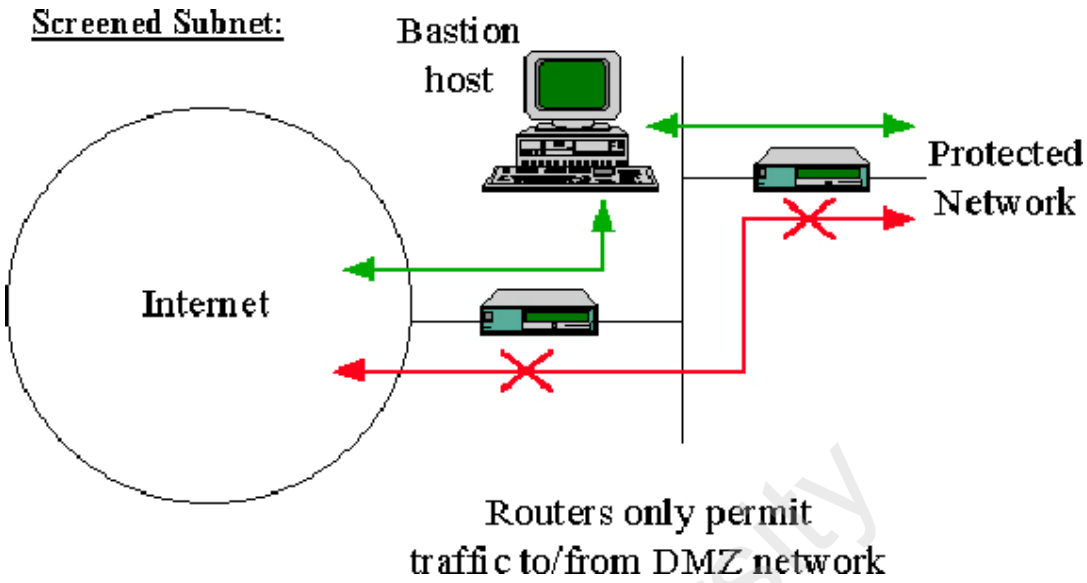

Figure 1.3 Screened Subnet Firewall [3]

In Figure 1.3, a network layer firewall called a screened subnet firewall is represented. In a screened subnet firewall, access to and from a whole network is controlled by means of a router operating at a network layer[3]. It is similar to a screened host, except that it is, effectively, a network of screened hosts.

- **Application Layer Firewalls**

These generally are hosts running proxy servers, which permit no traffic directly between networks, and which perform elaborate logging and auditing of traffic passing through them. Since the proxy applications are software components running on the firewall, it is a good place to do lots of logging and access control. Application layer firewalls can be used as network address translators, since traffic goes in one side and out the other, after having passed through an application that effectively masks the origin of the initiating connection. Having an application in the way in some cases may impact performance and may make the firewall less transparent.

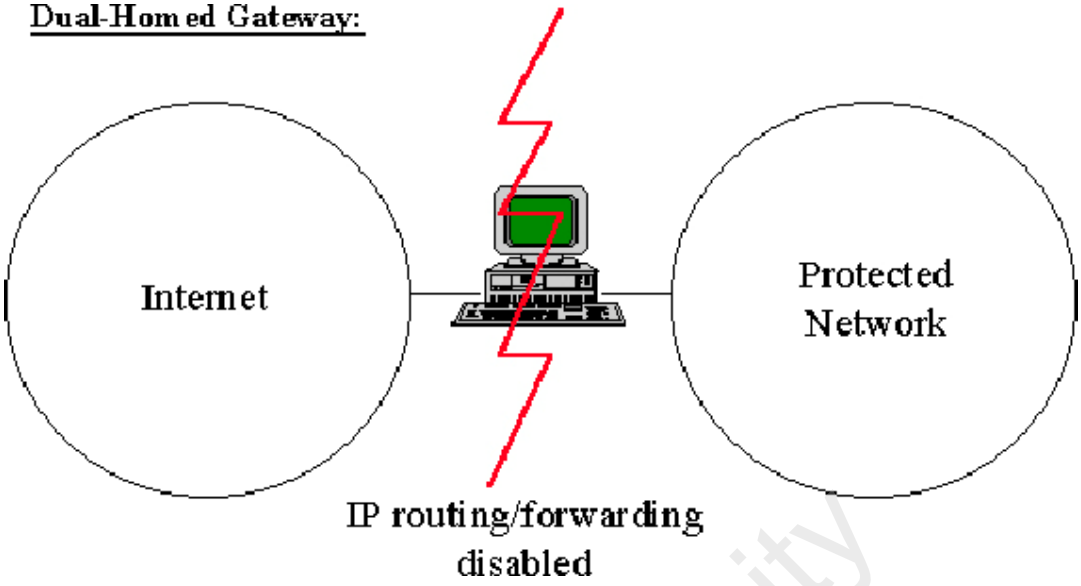

Figure 1.4 Dual Homed Gateway[3]

Early application layer firewalls are not particularly transparent to end users and may require some training. Modern application layer firewalls are often fully transparent. Application layer firewalls tend to provide more detailed audit reports and tend to enforce more conservative security models than network layer firewalls.

In Figure 1.4 an application layer firewall called a dual homed gateway is represented. A dual homed gateway is a highly secured host that runs proxy software. It has two network interfaces, one on each network, and blocks all traffic passing through it.

- **Hybrid Firewall**

In Hybrids combination of both network layer firewall and application layer firewall is used. These hybrid firewalls now lie some place between network layer firewalls and application layer firewalls. As expected, network layer firewalls have become increasingly aware of the information going through them, and application layer firewalls have become increasingly low level and transparent. The end result is that now there are fast packet-screening systems that log and audit data as they pass through the system. Increasingly, firewalls(network and application layer) incorporate encryption so that they may protect traffic passing between them over the Internet.

1.2.1.3 Sample filtering rule for a Router

Figure 1.5 shows one possible configuration for using the filtering router. It shows the implementation of as specific policy. These policies will undoubtedly vary from organization to organization..

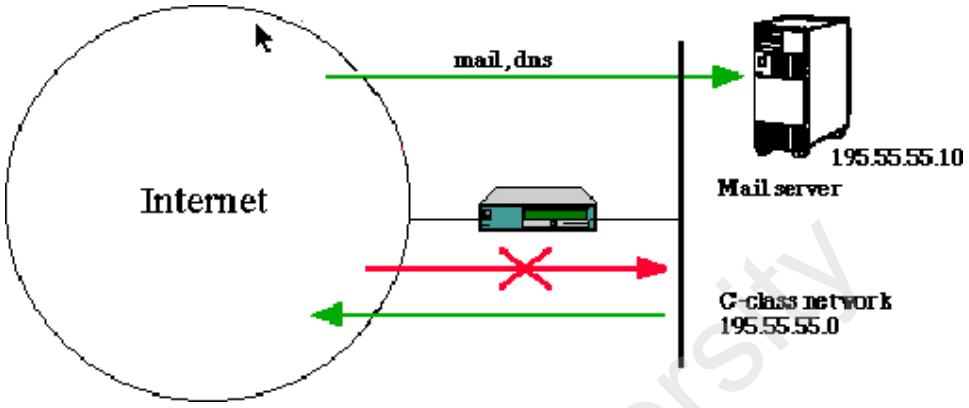

Figure1.5 Router Filtering[3]

In this Figure 1.5 an organization has Class C network address 190.50.50.0. Organization's network is connected to Internet via IP Service Provider. Organization's policy is to allow everybody access to Internet services, so all outgoing connections are accepted. All incoming connections go through mail host . Mail and DNS are only incoming services. Only incoming packets from Internet are checked in this configuration. Rules are tested in order and stop when the first match is found. There is an implicit deny rule at the end of an access list that denies everything.

1.2.1.4 Shortcomings of Firewalls

- Firewalls cannot enforce strong access policies with router access lists. Users can easily install backdoors to their systems to get over no incoming telnet or no X11 rules. Also crackers install telnet backdoors on systems where they break in.

- It is not sure that what services are listening for connections on high port numbers.

- Checking the source port on incoming FTP data connections is a weak security method. It also breaks access to some FTP sites. It makes use of the service more difficult for users without preventing bad guys from scanning target hosts.

1.2.2 Using Proxy Servers for Network Security

A proxy server (sometimes referred to as an application gateway or forwarder) is an application that mediates traffic between a protected network and the Internet. Proxies are often used instead of router-based traffic controls, to prevent traffic from passing directly between networks. Many proxies contain extra logging or support for user authentication. Since proxies must understand the application protocol being used, they can also implement protocol specific security (e.g., an FTP proxy might be configurable to permit incoming FTP and block outgoing FTP)[1].

Proxy servers are application specific. In order to support a new protocol via a proxy, a proxy must be developed for it. One popular set of proxy servers is the TIS Internet Firewall Toolkit ("FWTK) which includes proxies for Telnet, rlogin, FTP, the X Window System, HTTP/Web, and NNTP/Usenet news[4]. SOCKS is a generic proxy system that can be compiled into a client-side application to make it work through a firewall. Its advantage is that it's easy to use, but it doesn't support the addition of authentication hooks or protocol specific logging. Various proxies are available such as web proxies those could provide HTTP tunneling and ICMP tunneling. In HTTP tunneling traffic for banned site can be transferring by hiding in HTTP packets those are intended for some other site, in this case it will be difficult to judge just by looking on port no[2].

1.2.3 Reactive Network Security Tools & Techniques

Those tools and techniques which takes generates alerts only after something malicious activity has happened, comes under the category of reactive tool and techniques category. One of the common reactive approach for network security is called IDS.

Intrusion Detection System (IDS)

An intrusion detection system (IDS) generally detects unwanted manipulations of computer systems, mainly through the Internet. The manipulations may take the form

of attacks by crackers. An intrusion detection system is used to detect several types of malicious behaviors that can compromise the security and trust of a computer system. This includes network attacks against vulnerable services, data driven attacks on applications, host based attacks such as privilege escalation, unauthorized logins and access to sensitive files, and malware (viruses, trojan horses, and worms)[31].

An IDS is composed of several components: **Sensors** which generate security events, a **Console** to monitor events and alerts and control the sensors, and a central **Engine** that records events logged by the sensors in a database and uses a system of rules to generate alerts from security events received. There are several ways to categorize an IDS depending on the type and location of the sensors and the methodology used by the engine to generate alerts. In many simple IDS implementations all three components are combined in a single device or appliance. An intrusion detection system (IDS) inspects all inbound and outbound network activity and identifies suspicious patterns that may indicate a network or system attack from someone attempting to break into or compromise a system[33].

1.2.3.1 Intrusion Detection System Categories

There are several ways to categorize an IDS:

- **Misuse detection** vs. **Anomaly detection**: in misuse detection, the IDS analyzes the information it gathers and compares it to large databases of attack signatures. Essentially, the IDS looks for a specific attack that has already been documented. Like a virus detection system, misuse detection software is only as good as the database of attack signatures that it uses to compare packets against. In anomaly detection, the system administrator defines the baseline, or normal, state of the network's traffic load, breakdown, protocol, and typical packet size. The anomaly detector monitors network segments to compare their state to the normal baseline and look for anomalies[32].
- **Network-based** vs. **Host-based systems**: in a network-based system, or NIDS, the individual packets flowing through a network are analyzed. The NIDS can detect malicious packets that are designed to be overlooked by a firewall's simplistic filtering rules. In a host-based system, the IDS examines at the activity on each individual computer or host[32].

- **Passive system** vs. **Reactive system**: in a passive system, the IDS detects a potential security breach, logs the information and signals an alert. In a reactive system, the IDS responds to the suspicious activity by logging off a user or by reprogramming the firewall to block network traffic from the suspected malicious source[32].

1.2.3.2 Differences from Firewall

Though they both relate to network security, an IDS differs from a firewall in that a firewall looks out for intrusions in order to stop them from happening. The firewall limits the access between networks in order to prevent intrusion and does not signal an attack from inside the network. An IDS evaluates a suspected intrusion once it has taken place and signals an alarm. An IDS also watches for attacks that originate from within a system.

1.2.3.3 Types of Intrusion Detection System

Intrusion detection system in general are of these types

In a NIDS, the sensors are located at choke points in the network to be monitored, often in the demilitarized zone (DMZ) or at network borders. The sensor captures all network traffic and analyzes the content of individual packets for malicious traffic. In systems, PIDS and APIDS are used to monitor the transport and protocols illegal or inappropriate traffic or constructs of language (say SQL). In a host-based system, the sensor usually consists of a software agent, which monitors all activity of the host on which it is installed. Hybrids of these two systems also exist[32].

- **Network Intrusion Detection System**

A network intrusion detection system is an independent platform which identifies intrusions by examining network traffic and monitors multiple hosts. Network Intrusion Detection Systems gain access to network traffic by connecting to a hub,

network switch configured for port mirroring, or network tap. An example of a NIDS is Snort[30].

- **Protocol Based Intrusion Detection System**

A protocol-based intrusion detection system consists of a system or agent that would typically sit at the front end of a server, monitoring and analyzing the communication protocol between a connected device (a user/PC or system). For a web server this would typically monitor the HTTPS protocol stream and understand the HTTP protocol relative to the web server/system it is trying to protect. Where HTTPS is in use then this system would need to reside in the "shim" or interface between where HTTPS is un-encrypted and immediately prior to it entering the Web presentation layer[33].

- **Application Protocol Based Intrusion Detection System**

An application protocol-based intrusion detection system consists of a system or agent that would typically sit within a group of servers, monitoring and analyzing the communication on application specific protocols. For example; in a web server with database this would monitor the SQL protocol specific to the middleware/business-login as it transacts with the database[32].

- **Host Based Intrusion Detection System**

A host-based intrusion detection system consists of an agent on a host which identifies intrusions by analyzing system calls, application logs, file-system modifications (binaries, password files, capability/acl databases) and other host activities and state.

- **Hybrid Intrusion Detection System**

A hybrid intrusion detection system combines two or more approaches. Host agent data is combined with network information to form a comprehensive view of the network.

1.2.4 Proactive Tools & Techniques for Network Security

Those tools and techniques which can take actions before something malicious is going to happen are called proactive approaches for network security. One of the most common example for proactive approach for network security is IPS.

Intrusion Prevention System (IPS)

An intrusion prevention system is a computer security device that could be combination of hardware and software or purely software running on a hardware, that monitors network and/or system activities for malicious or unwanted behavior and can react, in real-time, to block or prevent those activities. Network-based IPS, for example, will operate in-line to monitor all network traffic for malicious code or attacks. When an attack is detected, it can drop the offending packets while still allowing all other traffic to pass. Intrusion prevention technology is considered by some to be an extension of intrusion detection (IDS) technology. Intrusion prevention system is a considerable improvement over firewall. IPS make access control decisions based on application content, rather than IP address or ports as traditional firewalls had done. As IPS systems were originally a literal extension of intrusion detection systems, they continue to be related[5].

An Intrusion Prevention system must also be a very good Intrusion Detection system to enable a low rate of false positives. Some IPS systems can also prevent yet to be discovered attacks, such as those caused by a Buffer overflow.

1.2.4.1 Types of Intrusion Prevention System

The Intrusion prevention systems could be of following types and their combinations could also be used differing from organization to organization. Some of those are as follows .

- **Host Based Intrusion Prevention System**

In this IPS is installed on a specific host or on a single computer. These are also known as HIPS. As with Host IDS systems, the Host IPS relies on agents installed directly on the system being protected. It binds closely with the operating system kernel and services, monitoring and intercepting system calls to the kernel or APIs in

order to prevent attacks as well as log them. It may also monitor data streams and the environment specific to a particular application (file locations and Registry settings for a Web server, for example) in order to protect that application from generic attacks for which no "signature" yet exists. One potential disadvantage with this approach is that, given the necessarily tight integration with the host operating system, future OPERATING SYSTEMupgrades could cause problems. Since a Host IPS agent intercepts all requests to the system it protects, it has certain prerequisites - it must be very reliable, must not negatively impact performance, and must not block legitimate traffic. Any HIPS that does not meet these minimum requirements should never be installed in a host, no matter how effectively it blocks attacks[5].

- **Network Based Intrusion Prevention System**

A network based IPS is one where the IPS application/hardware and any actions taken to prevent an intrusion on a specific network host(s)is done from a host with another IP address on the network (This could be on a front-end firewall appliance). Network intrusion prevention systems (NIPS) are purpose-built hardware/software or their combination that are designed to analyze, detect, and report on security related events and even to take proper actions. NIPS are designed to inspect traffic and based on their configuration or their security policy, they can drop malicious traffic. The Network IPS combines features of a standard IDS, an IPS and a firewall, and is sometimes known as an *In-line IDS* or *Gateway IDS (GIDS)*[6]. The next generation firewall - the *deep inspection firewall* - also exhibits a similar feature set, though researchers do not believe that the deep inspection firewall is ready for mainstream deployment yet. As with a typical firewall, the NIPS has at least two network interfaces, one designated as *internal* and one as *external*. As packets appear at the either interface they are passed to the detection engine, at which point the IPS device functions much as any IDS would in determining whether or not the packet being examined poses a threat. However, if it should detect a malicious packet, in addition to raising an alert, it will discard the packet and mark that flow as bad[1].

- **Content Based Intrusion Prevention System**

A content based IPS (CBIPS) inspects the content of network packets for unique sequences, called signatures, to detect and hopefully prevent known types of attack such as worm infections and hacks.

- **Protocol Based Intrusion Prevention System**

A key development in IDS/IPS technologies is the use of protocol analyzers. Protocol analyzers can natively decode application-layer network protocols, like HTTP or FTP. Once the protocols are fully decoded, the IPS analysis engine can evaluate different parts of the protocol for anomalous behavior or exploits. For example, the existence of a large binary file in the User-Agent field of an HTTP request would be very unusual and likely an intrusion. A protocol analyzer could detect this anomalous behavior and instruct IPS engine to drop the offending packets. Not all IPS/IDS engines are full protocol analyzers. Some products rely on simple pattern recognition techniques to look for known attack patterns. While this can be sufficient in many cases, it creates an overall weakness in the detection capabilities. Since many vulnerabilities have dozens or even hundreds of exploit variants, pattern recognition-based IPS/IDS engines can be evaded. For example, some pattern recognition engines require hundreds of different signatures to protect against a single vulnerability. This is because they must have a different pattern for each exploit variant. Protocol analysis-based products can often block exploits with a single signature that monitors for the specific vulnerability in the network communications[2,32].

- **Rate Based Intrusion Prevention System**

Rate based IPS (RBIPS) are primarily intended to prevent denial of service and Distributed Denial of Service attacks. They work by monitoring and learning normal network behaviors. Through real-time traffic monitoring and comparison with stored statistics, RBIPS can identify abnormal rates for certain types of traffic e.g. TCP, UDP or ARP packets, connections per second, packets per connection, packets to

specific ports etc. Attacks are detected when thresholds are exceeded. The thresholds are dynamically adjusted based on time of day, day of the week etc., drawing on stored traffic statistics. Unusual but legitimate network traffic patterns may create false alarms. The system's effectiveness is related to the granularity of the RBIPS rulebase and the quality of the stored statistics[2].

Once an attack is detected, various prevention techniques may be used such as rate-limiting specific attack-related traffic types, source or connection tracking, and source-address, port or protocol filtering (black-listing) or validation (white-listing).

- Host Based Vs Network Based Intrusion Prevention System

  - HIPS can handle encrypted and unencrypted traffic equally, because it can analyze the data after it has been decrypted on the host.
  - NIPS does not use processor and memory on computer hosts but uses its own CPU and memory.
  - NIPS is a single point of failure, which is considered a disadvantage; however, this property also makes it simpler to maintain. A Bypass Switch can be implemented to alleviate the single point of failure disadvantage though. This also allows the NIPS appliance to moved and taken off-line for maintenance when needed.

NIPS can detect events scattered over the network and can react, whereas with a HIPS, only the hosts data itself is available to take a decision, respectively it would take too much time to report it to a central decision making engine and report back to block[2].

1.3 Network Security Future

1.3.1 Deep Packet Inspection (DPI)

DPI is the foremost technology for identifying and authenticating protocols and applications conveyed by IP. The standard packet inspection process extracts basic protocol information such as IP addresses (source, destination) and other low-level connection states. This information typically resides in the packet header itself and

consequently reveals the principal communication intent. The inspection level in the shallow inspection process is insufficient to reach any application-related deductions. DPI, on the other hand, provides application awareness. This is achieved by analyzing the content in both the packet header and the payload over a series of packet transactions[34].

Chapter 2

Literature Survey

2.1 Protocol Compiler

2.1.1 Introduction to Protocol Compiler

A DPI based system would let an administrator write network policies based on not only the network and transport layer, but, also the content encompassed by application layer. Similarly functionality can be achieved using Firewall. A firewall policy is traditionally defined in terms of 5-tuples or parts thereof. For example: allow traffic from subnet 192.168.1.0/24 to any host with TCP destination port 80 and drop all the rest of traffic. Now imagine when a firewall needs to be extended to write policies with rules that cover application layers. The range of possibilities in application layers is virtually limitless. A firewall with such a support would either cover only a limited functionality or would never be in a finished state. DPI firewalls would need to be really flexible, in terms of letting users very finely specify what they are looking for. A DPI firewall which presents the user with the most flexible way of specifying “interesting” traffic will be the one that'll be most useful and powerful. DPI is an emerging field. There are products out there, both commercial as well as free ones, that do DPI. But, the technology isn't mature as yet. Most IDS/IPS products exhibit this limited form of DPI. They use pattern matching techniques to classify traffic into one or more of known classes or buckets. These patterns are also known as signatures, since they are unique to the traffic in question[35].

A key issue with doing DPI is that it's very computationally-intensive. This puts a high stake on the performance of the system to make it a viable solution. Protocol compiler, designed to capture, decode and monitor network traffic, are inconsistently implemented. These software tools are typically implemented as large, hand-crafted bodies of code, subject to individual programmer interpretation of the protocols. Security mechanisms in firewalls and intrusion detection systems suffer from the same problems, which also results in inconsistent implementations that are exploitable by knowledgeable attackers. The overall Result of this current state of affairs is to make large suites or families of protocol implementations highly duplicative,

expensive to maintain, and extremely expensive to secure [11]. In this overall process main challenge will be time. The concept of traffic parsing can be done in the following sequence.

- Packet Capturing : To save time packets must be captured at kernel level so that the context switching time could be saved . It will lead to save a lot of time too.
- Protocol Compiler : There should be some efficient protocol compiler that will be responsible for processing packets. When the packet is captured at kernel level, the packet's content should be parsed. For this some Context Free Grammar will be needed to defined on the basis of that packet's data will be parsed and categorized for further actions.
- IDS : Just after protocol compiler there will be some Intrusion detection system that will be able to generate alerts on the data categorized by our protocol compiler. The IDS will take decision of generating alerts on the basis of rules specified in regular expressions and signatures database of various threats.
- IPS : As soon as our Intrusion Detection System will identify security threat, our Intrusion Prevention System will take appropriate action on the basis of network security policy defined by Network Administrator. IPS comes under proactive kind of security approach.

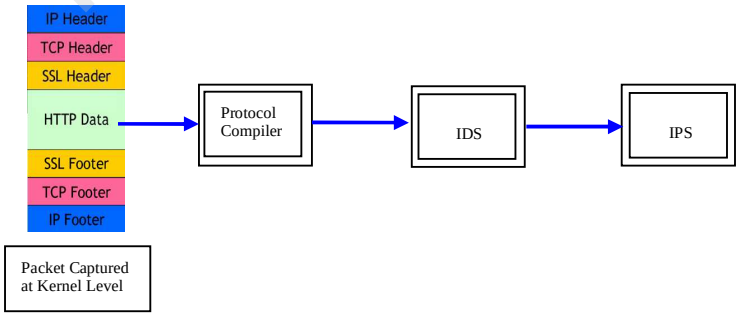

Figure 2.1 DPI Process on Network Traffic

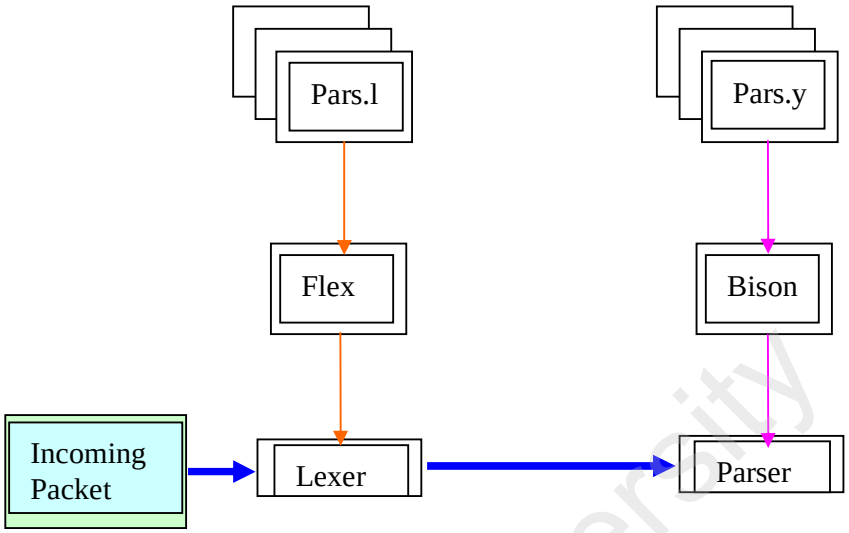

Figure 2.2 Protocol Compiler Internal Processing

Protocol Analyzers, designed to capture, decode and monitor network traffic, are inconsistently implemented. These software tools are typically implemented as large, hand-crafted bodies of code, subject to individual programmer interpretation of the protocols. Security mechanisms in firewalls and intrusion detection systems suffer from the same problems, which also results in inconsistent implementations that are exploitable by knowledgeable attackers. The overall result of this current state of affairs is to make large suites or families of protocol implementations highly duplicative, expensive to maintain, and extremely expensive to secure [11]. Writing a protocol compiler from scratch in itself is a very tedious task, some compiler generation tools like Flex for lexical analyzer generation and Bison for parser generation purpose are used for this purpose. These will generate most efficient compiler ever could be written by human programmer[12].

Many security protocols, using cryptographic techniques. They are supposed to succeed even in the presence of malicious Agents or some intruders or enemies, that can interfere with their correct execution. Unfortunately, many of them may fail when

an intruder intercepts some messages and exploits the information they contain in order to introduce new messages into the network [17].

Any specific property that the protocol must satisfy is not specified. Only it is observed, whether the intruder is able to modify (using some way) the flow of information among the honest participants. The task of understanding which is the set of interferences corresponding to attacks is not difficult, but the real problem is to specify the system.

In order to analyze the protocol, it is necessary to study its behavior in the presence of a malicious user, often known as the *enemy*, which can perform the following actions:

- intercept any sent message;
- store the information contained in intercepted messages for later uses;
- introduce into the system new messages fabricated using information he knows.

An enemy with the above features is able to use the information gained in the previously intercepted messages when he introduces new messages into the system [13].

2.1.2 Giving Protocol Specifications to the Protocol Compiler

The specification of a protocol is composed of four parts:

- **Declaration of types** (e.g., Agent, Nonce, Key) and their values, declaration of the structure of the protocol messages, declaration and definition of roles. In this part, we define an abstract description of the protocol, leaving the actual components of the session and the other parameters of the analysis unspecified [14].
- **Definition of the session.** In this part, we define the actual system by specifying the combination (sequential and/or parallel composition) of roles, where each one is instantiated with actual values, e.g., agent names and **Enemy setting.** In this part, we define the enemy component by specifying its initial knowledge and its memory capacity.

- **Definition of visible actions and secret values.** The actions specified in this section are those used to observe and detect enemy interferences[15].

2.2 Compiling Techniques

2.2.1 Lexical Analyzer

- **Lexical Analysis**

In computer science, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called lexical analyzers or lexers. A lexer is often organized as separate scanner and tokenizer functions, though the boundaries may not be clearly defined. Lexical analysis is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. *Tokens* are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words[16].

- **Lexical grammar**

The specification of a programming language will include a set of rules, often expressed syntactically, specifying the set of possible character sequences that can form a token or lexeme. The whitespace characters are often ignored during lexical analysis[17].

- **Token**

A **token** is a categorized block of text. The block of text corresponding to the token is known as a lexeme. A lexeme is the actual character sequence forming a token, the token is the general class that a lexeme belongs to. Some tokens have exactly one lexeme (e.g., the > character); for others, there are many lexemes (e.g.,integer constants). A lexical analyzer processes *lexemes* to categorize them according to

function, giving them **meaning**. This assignment of meaning is known as **tokenization**. A token can look like anything: English, gibberish symbols, anything; it just needs to be a useful part of the structured text[17].

A sample C program line *sum=3+2;* Tokenized in the following table:

Lexeme	**token type**
Sum	IDENT
=	ASSIGN_OP
3	NUMBER
+	ADD_OP
2	NUMBER
;	SEMICOLON

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer such as flex. The lexical analyzer reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

The first phase of the compiler is the lexical analyzer, also known as the scanner, which recognizes the basic language units, called tokens. The exact characters in a token is called its lexeme. Tokens are classified by token types, e.g. identifiers, constant literals, strings, operators, punctuation marks, and key words. Different types of tokens may have their own semantic attributes (or values) which must be extracted and stored in the symbol table. The lexical analyzer may perform semantic actions to extract such values and insert them in the symbol table. Classification of token types mainly depends on what form of input is needed by the next compiler phase, the parser. The scanner is tasked with determining that the input stream can be divided into valid symbols in the source language, but has no smarts about which token should come where. Few errors can be detected at the lexical level alone because the scanner has a very localized view of the source program without any context. The scanner can report about characters that are not valid tokens (e.g., an illegal or unrecognized symbol) and a few other malformed entities (illegal characters within a string constant, unterminated comments, etc.) It does not look for or detect garbled

sequences, tokens out of place, undeclared identifiers, misspelled keywords, mismatched types [17]. The mythical source language tokenized by the above scanner requires that reserved words be in all upper case and identifiers in all lower case. This convenient feature makes it easy for the scanner to choose which path to pursue after reading just one character.

We can use one of several ways to precisely express the classification. A common method is to use a finite automaton to define all character sequences (i.e. strings) which belong to a particular token type. The states, the starting state, the accepting states of a finite automaton. An accepting state is also called a final state. Given the definitions of different token types, it is possible for a string to belong to more than one type. Such ambiguity is resolved by assigning priorities to token types. Regular expressions are such a textual representation:

Regular expressions are equivalent to Deterministic Finite Automaton in that:
1) For any given regular expression, there exist a DFA which accepts the same set of strings represented by the regular expression.
2) For a given DFA, there exists a regular expression which represents the same set of strings accepted by the DFA[16].

A simplest lexer derives a stream of tokens by processing a stream of characters from left to light. Several tools that facilitate the construction of such lexers have been developed, including the well-known Unix tools flex it is also known as Flex. Flex tool supports an extension of regular expression notation as their pattern set. Each rule is associated with a pattern and used to generate a corresponding token from input, by looking at a particular pattern [18].

Since tokens generally have very limited contextual dependencies, the resulting differences in the token stream are typically limited to the area surrounding modification sites. In this setting it makes sense to retain the token stream as a persistent data structure and use it to decrease the time required for subsequent analyses. This is known as *incremental lexing* and the tool that performs it as an *incremental* lexer. This approach has several advantages. No changes to the generator's implementation are necessary. By capturing the state of the batch lexing

machine at the conclusion of each token's creation and saving it within the token, we are able to restart lexical analysis at point in the token stream. This provides extremely fine-grained incrementally. We can handle contextual dependencies by restricting the no of look ahead tokens. Incremental lexing algorithm operates in optimal time and is linear in the number of affected characters *(*the characters in modified tokens and in tokens whose look ahead reached modified tokens).

The overhead associated with locating modification sites depends on the representation in which incremental lexing is embedded. Incremental lexing is the incremental maintenance of the mapping between a textual stream and a token stream. Each character belongs to exactly one token. the lexer must partition the textual stream by locating the inter token boundaries and assigning a type to each resulting lexeme. Tokens persist until deleted explicitly through editing or implicitly when an invocation of incremental analysis fails to retain them in the resulting token stream. The beginning of the token stream is marked with a sentinel called bos, the end of the stream by eos, It is easy to see how a textual edit affects token dependencies: if a character is inserted to, deleted from, or overwritten within a token, then the previous look back tokens must be considered suspect[19].

2.2.2 Parser

- **Introduction to Parsing**

In computer science and linguistics, **parsing** (more formally: *syntactic analysis*) is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. A **parser** is the component of a compiler that carries out this task.

Parsing transforms input text into a data structure, usually a tree, which is suitable for later processing and which captures the implied hierarchy of the input. Lexical analysis creates tokens from a sequence of input characters and it is these tokens that are processed by a parser to build a data structure such as parse tree or abstract syntax trees.

➢ **Parser with Programming Languages**

The most common use of a parser is as a component of a compiler. This parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them. Parsers are usually not written by hand but are generated by parser generators.

1.1.1

1.1.2 Overview of process

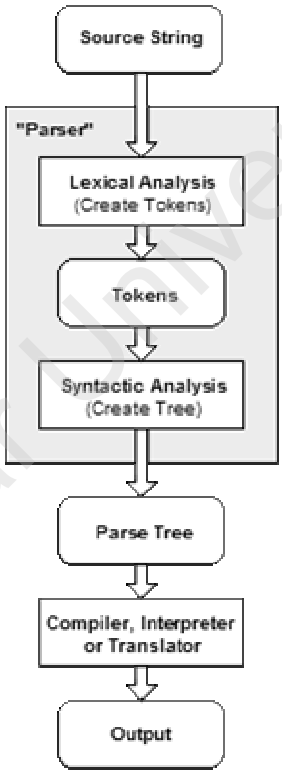

Figure 2.3 Parsing Process[20]

The following example in Figure 2.3 demonstrates the common case of parsing a computer language with two levels of grammar: lexical and syntactic. The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "12*(3+4)^2" and split it into the tokens 12, *, (, 3, +, 4,), ^ and 2, each of which is a meaningful symbol in the context

of an arithmetic expression. The parser would contain rules to tell it that the characters *, +, ^, (and) mark the start of a new token, so meaningless tokens like "12*" or "(3" will not be generated. The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars. The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator, the action is to evaluate the expression; a compiler, on the other hand, would generate code. Attribute grammars can also be used to define these actions.

- **Types of Parser**

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- **Top Down Parser**

A parser can start with the start symbol and try to transform it to the input. Intuitively, the parser starts from the largest elements and breaks them down into incrementally smaller parts. LL parsers are examples of top-down parsers. **Top-down parsing** is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages. Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Simple implementations of top-down parsing do not terminate for left-recursive grammars, and top-down parsing with backtracking may have exponential time complexity with respect to the length of the input for ambiguous

CFGs . However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially-exponential number of parse trees.

- **Programming language application**

A compiler parses input from a programming language to assembly language or an internal representation by matching the incoming symbols to Backus-Naur form production rules. An LL parser, also called a **top-bottom parser** or **top-down parser**, applies each production rule to the incoming symbols by working from the left-most symbol yielded on a production rule and then proceeding to the next production rule for each non-terminal symbol encountered. In this way the parsing starts on the Left of the result side (right side) of the production rule and evaluates non-terminals from the Left first and, thus, proceeds down the parse tree for each new non-terminal before continuing to the next symbol for a production rule.

For example:

$$A \rightarrow aBC$$

$$B \rightarrow c|cd$$

$$C \rightarrow df|eg$$

would match $A \rightarrow aBC$ and attempt to match $B \rightarrow c|cd$ next. Then $C \rightarrow df|eg$ would be tried. As one may expect, some languages are more ambiguous than others. For a non-ambiguous language in which all productions for a non-terminal produce distinct strings: the string produced by one production will not start with the same symbol as the string produced by another production. A non-ambiguous language may be parsed by an LL(1) grammar where the (1) signifies the parser reads ahead one token at a time. For an ambiguous language to be parsed by an LL parser, the parser must look ahead more than 1 symbol, e.g. LL(3)[20].

- **Accommodating Left Recursion in Top-down Parsing**

A formal grammar that contains left recursion cannot be parsed by a naive recursive descent parser unless they are converted to a weakly equivalent right-recursive form. However, recent research demonstrates that it is possible to accommodate left-recursive grammars (along with all other forms of general CFGs) in a more sophisticated top-down parser by use of curtailment. A recognition algorithm which accommodates ambiguous grammars and curtails an ever-growing direct left-recursive parse by imposing depth restrictions with respect to input length and current input position. That algorithm was extended to a complete parsing algorithm to accommodate indirect (by comparing previously-computed context with current context) as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars[17].

- **Time and Space Complexity of Top-down Parsing**

When top-down parser tries to parse an ambiguous input with respect to an ambiguous CFG, it may need exponential number of steps (with respect to the length of the input) to try all alternatives of the CFG in order to produce all possible parse trees, which eventually would require exponential memory space. The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions.

- **Bottom Up Parser**

A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing [21].

- **Bottom-up parsing**

As the name suggests, bottom-up parsing works in the opposite direction from top down. A top-down parser begins with the start symbol at the top of the parse tree and

works downward, driving productions in forward order until it gets to the terminal leaves. A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it *reduces* it, i.e., substitutes the left side non terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse. In general, bottom-up parsing algorithms are more powerful than top-down methods, but not surprisingly, the constructions required are also more complex[20]. It is difficult to write a bottom-up parser by hand for anything but trivial grammars, but fortunately, there are excellent parser generator tools like bison that build a parser from an input specification, not unlike the way lex builds a scanner to our spec. *Shift-reduce* parsing is the most commonly used and the most powerful of the bottom-up techniques. It takes as input a stream of tokens and develops the list of productions used to build the parse tree, but the productions are discovered in reverse order of a topdown parser. To illustrate stack-based shift-reduce parsing, consider this simplified expression grammar:

S –> E

E –> T | E + T

T –> id | (E)

The shift-reduce strategy divides the string that we are trying parse into two parts: an undigested part and a semi-digested part. The undigested part contains the tokens that are still to come in the input, and the semi-digested part is put on a stack[16].

Reduce: If we can find a rule A –> w , and if the contents of the stack are q w for some q (q may be empty), then we can reduce the stack to qA. We are applying the production for the non terminal A backwards.

Shift: If it is impossible to perform a reduction and there are tokens remaining in the undigested input, then we transfer a token from the input onto the stack. This is called a shift.

Error: If neither of the two above cases apply, error is generated. If the sequence on the stack does not match the right-hand side of any production, we cannot reduce.

And if shifting the next input token would create a sequence on the stack that cannot eventually be reduced to the start symbol, a shift action would be futile. A dead end is hit where the next token conclusively determines the input cannot form a valid sentence[20].

- **LR Parsers**

LR parsers ("L" for left to right scan of input, "R" for rightmost derivation) are efficient, table-driven shift-reduce parsers. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive LL parsers. In fact, virtually all programming language constructs for which CFGs can be written can be parsed with LR techniques. As an added advantage, there is no need for lots of grammar rearrangement to make it acceptable for LR parsing the way that LL parsing requires. The primary disadvantage with LR parser is the amount of work it takes to build the tables by hand, which makes it infeasible to hand-code an LR parser for most grammars. Fortunately, there are LR parser generators that create the parser from an unambiguous CFG specification. The parser tool does all the tedious and complex work to build the necessary tables and can report any ambiguities or language constructs that interfere with the ability to parse it using LR techniques[21].

An LR parser uses two tables:

1. The *action table* Action[s,a] tells the parser what to do when the state on top of the stack is s and terminal a is the next input token. The possible actions are to shift a state onto the stack, to reduce the handle on top of the stack, to accept the input, or to report an error.

2. The *goto table* Goto [s,X] indicates the new state to place on top of the stack after a reduction of the non terminal X while state s is on top of the stack. The two tables are usually combined, with the action table specifying entries for terminals, and the goto table specifying entries for non terminals[20].

- **LR Parser Tracing**

Suppose we start with the initial state s0 on the stack. The next input token is the terminal a and the current state is st. The action of the parser is as follows:

• If Action[st,a] is shift, we push the specified state onto the stack. We then call yylex() to get the next token a from the input.

• If Action[st,a] is reduce Y –> X1...Xk then we pop k states off the stack (one for each symbol in the right side of the production) leaving state su on top. Goto [su,Y] gives a new state sV to push on the stack. The input token is still a (i.e., the input remains unchanged)[20].

• If Action[st,a] is accept then the parse is successful and we are done.

• If Action[st,a] is error (the table location is blank) then we have a syntax error. With the current top of stack and next input we can never arrive at a sentential form with a handle to reduce.

➢ **LR Parser Types**

There are three types of LR parsers: *LR(k), simple LR(k),* and *look ahead LR(k)* (abbreviated to LR(k), SLR(k), LALR(k))). The k identifies the number of tokens of look ahead.

2.3 Tools for breaking the Input Packet's bit Stream into tokens

During the process of analyzing the contents of the data packets, it becomes very necessary to break the input bit's stream into tokens by looking at fixed pattern across various packets. For this purpose lexical analysis tool Flex could be used.

2.3.1 FLEX A Lexical Analyzer

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input. Each pattern in flex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser. To begin with, however, we will simply print the matched string rather than return a token value. We may scan for identifiers using the regular expression[21].

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter, and is followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*****" operator
- alternation, expressed by the "|" operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.

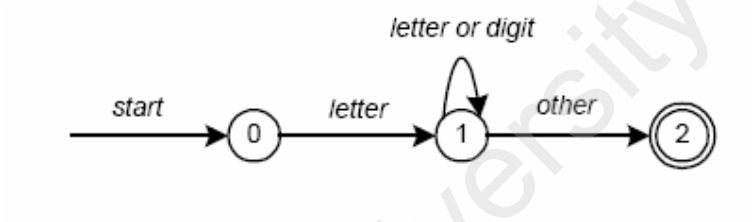

Figure 2.4 Finite State Automaton[21]

In above figure state 0 is the start state, and state 2 is the accepting state. As characters are read, we make a transition from one state to another. When the first letter is read, we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit, we transition to state 2, the accepting state. *Any* FSA may be expressed as a computer program.

start: goto state0
state0: read c
if c = letter goto state1
goto state0
state1: read c
if c = letter goto state1
if c = digit goto state1
goto state2
state2: accept string

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character, and *current state*, the next state is easily determined by indexing into a computer-generated state table.

2.3.2 Limitations of Flex

Flex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "**(**", we push it on the stack. When a "**)**" is encountered, we match it with the top of the stack, and pop the stack. Flex, however, only has states and transitions between states Since it has no stack, it is not well suited for parsing nested structures. Yacc augments an FSA with a stack, and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Bison is appropriate for more challenging tasks. The program Lex generates a so called `Lexer'. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action. A very simple example:

```
%{
#include <stdio.h>
%}
%%
stop printf("Stop command received\n");
start printf("Start command received\n");
%%
```

The first section, in between the %{ and %} pair is included directly in the output program. We need this, because we use printf later on, which is defined in stdio.h. Sections are separated using '%%', so the first line of the second section starts with the 'stop' key. Whenever the 'stop' key is encountered in the input, the rest of the line (a printf() call) is executed. Besides 'stop', we've also defined 'start', which otherwise does mostly the same. We terminate the code section with '%%' again[23].

2.3.3 Debugging with Flex

Flex has facilities that enable debugging. This feature may vary with different versions of flex, so you should consult documentation for details. The code generated by flex in file **lex.yy.c** includes debugging statements that are enabled by specifying command-line option "**-d**". Debug output may be toggled on and off by setting **yy_flex_debug**. Output includes the rule applied and corresponding matched text. If you're running lex and yacc together, specify the following in your yacc input file:

```
extern int yy_flex_debug;
int main(void) {
yy_flex_debug = 1;
yyparse();
}
```

2.4 Parse Tree Generation for Network Traffic

BISON – A Parser Generator

From the stream of input packets, it is required to generate the parse tree. To generate the parse tree, some kind of parser is required. For this purpose we can use pasrer generator Bison. Bison is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1). We can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change [24].

2.4.1 Languages and Context-Free Grammars

In order for Bison to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. The most common formal system for presenting such rules for humans to read is Backus- Naur Form or “BNF”, which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF. Bison is optimized for what are called LALR(1) grammars[24].

2.4.2 Bison Parser Algorithm

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting. For example, suppose the infix calculator has read `1 + 5 *', with a `3' to come. The stack will have four elements, one for each token that was shifted. But the stack does not always have an element for each token read. When the last *n* tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called **reduction**. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping[24].

For example, if the infix calculator's parser stack contains this:

1 + 5 * 3

and the next input token is a new line character, then the last three elements can be reduced to 15 via the rule:

expr: expr '*' expr;

Then the stack contains just these three elements:

1 + 15

At this point, another reduction can be made, resulting in the single value 16. Then the new line token can be shifted. The parser tries, by shifts and reductions, to reduce the

entire input down to a single grouping whose symbol is the grammar's start-symbol (see section Languages and Context-Free Grammars). This kind of parser is known in the literature as a bottom-up parser[24].

2.4.3 Bison Look Ahead Tokens

The Bison parser does *not* always reduce immediately as soon as the last *n* tokens and groupings match a rule. This is because such a simple strategy is inadequate to handle most languages. Instead, when a reduction is possible, the parser sometimes "looks ahead" at the next token in order to decide what to do[24].

When a token is read, it is not immediately shifted; first it becomes the **look-ahead token**, which is not on the stack. Now the parser can perform one or more reductions of tokens and groupings on the stack, while the look-ahead token remains off to the side. When no more reductions should take place, the look-ahead token is shifted onto the stack. This does not mean that all possible reductions have been done; depending on the token type of the look-ahead token, some rules may choose to delay their application. Here is a simple case where look-ahead is needed. These three rules define expressions which contain binary addition operators and postfix unary factorial operators (`!'), and allow parentheses for grouping.

```
expr:     term '+' expr
        | term
        ;
term:     '(' expr ')'
        | term '!'
        | NUMBER
        ;
```

Suppose that the tokens `1 + 2' have been read and shifted; what should be done? If the following token is `)', then the first three tokens must be reduced to form an expr. This is the only valid course, because shifting the `)' would produce a sequence of symbols term ')', and no rule allows this.

If the following token is `!', then it must be shifted immediately so that `2 !' can be reduced to make a term. If instead the parser were to reduce before shifting, `1 + 2' would become an expr. It would then be impossible to shift the `!' because doing so would produce on the stack the sequence of symbols expr '!'. No rule allows that sequence. The current look-ahead token is stored in the variable `yychar[24]`.

2.4.4 Reduce/Reduce Conflict

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar[24].

For example, here is an erroneous attempt to define a sequence of zero or more word groupings[24].

```
sequence: /* empty */
        { printf ("empty sequence\n"); }
    | maybeword
    | sequence word
        { printf ("added word %s\n", $2); }
    ;
maybeword: /* empty */
      { printf ("empty maybeword\n"); }
    | word
        { printf ("single word %s\n", $1); }
    ;
```

The error is an ambiguity: there is more than one way to parse a single word into a sequence. It could be reduced to a maybeword and then into a sequence via the second rule[24]. Alternatively, nothing-at-all could be reduced into a sequence via the first rule, and this could be combined with the word using the third rule for sequence. There is also more than one way to reduce nothing-at-all into a sequence. This can be done directly via the first rule, or indirectly via maybeword and then the second rule. You might think that this is a distinction without a difference, because it does not change whether any particular input is valid or not. But it does affect which actions are run. One parsing order runs the second rule's action; the other runs the first rule's

action and the third rule's action. In this example, the output of the program changes. Bison resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this[17,24]. Every reduce/reduce conflict must be studied and usually eliminated. Here is the proper way to define sequence:

```
sequence: /* empty */
            { printf ("empty sequence\n"); }
        | sequence word
            { printf ("added word %s\n", $2); }
        ;
```

Here is another common error that yields a reduce/reduce conflict:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    /* empty */
        | words word
        ;

redirects:/* empty */
        | redirects redirect
        ;
```

The intention here is to define a sequence which can contain either word or redirect groupings. The individual definitions of sequence, words and redirects are error-free, but the three together make a subtle ambiguity: even an empty input can be parsed in infinitely many ways. Consider: nothing-at-all could be a words. Or it could be two words in a row, or three, or any number. It could equally well be a redirects, or two, or any number. Or it could be a words followed by three redirects and another words[24]. And so on. Here are two ways to correct these rules. First, to make it a single level of sequence:

```
sequence: /* empty */
        | sequence word
        | sequence redirect
        ;
```

Second, to prevent either a words or a redirects from being empty[24]:

```
sequence: /* empty */

        | sequence words
        | sequence redirects
        ;
words:    word
        | words word
        ;
Redirects :redirect
        | redirects redirect
        ;
```

2.4.5 Stack Overflow Problem with Bison

The Bison parser stack can overflow if too many tokens are shifted and not reduced. When this happens, the parser function yyparse returns a nonzero value, pausing only to call yyerror to report the overflow. By defining the macro YYMAXDEPTH, you can control how deep the parser stack can become before a stack overflow occurs. Define the macro with a value that is an integer. This value is the maximum number of tokens that can be shifted (and not reduced) before overflow. It must be a constant expression whose value is known at compile time[24].

The stack space allowed is not necessarily allocated. If we specify a large value for YYMAXDEPTH, the parser actually allocates a small stack at first, and then makes it bigger by stages as needed. This increasing allocation happens automatically and silently. Therefore, we do not need to make YYMAXDEPTH painfully small merely to save space for ordinary inputs that do not need much stack. The default value of YYMAXDEPTH, if we do not define it, is 10000. We can control how much stack is

allocated initially by defining the macro YYINITDEPTH. This value too must be a compile-time constant integer. The default is 200[24].

2.4.6 Writing Specification for Bison

In order for Bison to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an expression. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion[24].

The most common formal system for presenting such rules for humans to read is **Backus-Naur Form** or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF[24].

Not all context-free languages can be handled by Bison, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1).

2.4.7 Bison Output : The Parser File

When we run Bison, you give it a Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a **Bison parser**. Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser that becomes part of our program. The job of the Bison parser is to group tokens into

groupings according to the grammar rules--for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses. The Bison parser file is C code which defines a function named `yyparse` which implements that grammar[24]. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called `main`; you have to provide this, and arrange for it to call `yyparse` or the parser will never run[17].

2.4.8 Stages in Using Bison for Protocol Compiler Writing

The actual language-design process using Bison, from grammar specification to a working compiler or interpreter has these parts:

1. Formally specify the grammar in a form recognized by Bison (see section Bison Grammar Files). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.
2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C (see section The Lexical Analyzer Function yylex). It could also be produced using Lex, but the use of Lex is not discussed in this manual.
3. Write a controlling function that calls the Bison-produced parser.
4. Write error-reporting routines.

To turn this source code as written into a runnable program, we must follow these steps:

1. Run Bison on the grammar to produce the parser.
2. Compile the code output by Bison, as well as any other source files.
3. Link the object files to produce the finished product[24].

2.4.9 The Over all Layout of a Bison Grammar

The input file for the Bison utility is a Bison grammar file[24]. The general form of a Bison grammar file is as follows:

%{
Prologue
%}
Bison declarations
%%
Grammar rules
%%
Epilogue

The '%%', '%{' and '%}' are punctuation that appears in every Bison grammar file to separate
the sections.

The prologue may define types and variables used in the actions. The preprocessor commands to define macros can be used there, and use #include to include header files that do any of these things. To declare the lexical analyzer yylex and the error printer yyerror here, along with any other global identifiers used by the actions in the grammar rules.

2.5 Related Work

BinPAC is an implementation very similar to the work we have done in our thesis project. It is closely resembled with bro. BinPAC is designed to be specific with bro (Intrusion Detection System) only. But our work is very generic and it can work well with existing network security systems.

2.5.1 BinPAC

A key step in the semantic analysis of network traffic is to parse the traffic stream according to the high-level protocols it contains. This process transforms raw bytes into structured, typed, and semantically meaningful data fields that provide a high-level representation of the traffic. However, constructing protocol parsers by hand is a tedious and error-prone affair due to the complexity and sheer number of application protocols. For this purpose BinPAC is used[14].

BinPAC a declarative language and compiler designed to simplify the task of constructing robust and efficient semantic analyzers for complex network protocols. We discuss the design of the binpac language and a range of issues in generating efficient parsers from high-level specifications. BinPAC is a high level language for describing protocol parsers. The BinPAC compiler generates C++ code. There are already written a few Bro protocol analyzers using BinPAC, including CIFS/SMB, DCE/RPC, HTTP, DNS, NCP, and Sun/RPC. The BinPAC -based parsers have almost same performance as the hand-written ones. On the other hand, writing parsers in BinPAC is easier and the code is more maintainable. BinPAC is open-source with the same license as Bro has[14].

Finally, while the BinPAC compiler is released with Bro, it is designed to be general-purpose and so can be used to generate parsers for other programs as well.

Figure 2.5 BinPAC [14]

- PAC grammar - .pac file written by user.
- PAC source - _pac.cc file generated by binpac
- PAC header - _pac.h file generated by binpac[14]

➢ **BinPAC Language Reference**

BinPAC language consists of:

- analyzer
- type - data structure like definition describing parsing unit. Types can built on each other to form more complex type similar to yacc productions.
- flow - *flow* defines how data will be fed into the analyzer and the top level parsing unit.

- Keywords
- Built-in macros[15]

2.5.2 Characteristics of Application Protocol Compilers

Characteristics of network protocols differ significantly from the sorts of languages targeted by traditional parser-generators. We discuss them in terms of syntax and grammar, input model, and robustness.[15]

➢ **Syntax and Grammar Issues**

In terms of syntax and grammar, application-layer protocols can be broadly categorized into two classes: *binary* protocols and human - readable *ASCII* protocols. The messages of binary protocols, like DNS and CIFS, consist of a (not necessarily fixed) number of data *fields*. These fields directly map to a set of basic data *types* such as integers and strings. Clear-text ASCII protocols, on the other hand, typically restrict their payload to a human-readable request/reply structure, using only printable ASCII-characters. Many of these protocols, such as HTTP and SMTP, are primarily line-based, i.e., requests/replies are separated by carriage-return/line-feed (CR/LF) tuples, and their syntax is usually specified with grammar production rules in protocol standards. While these two types of protocols appear to exhibit quite distinct language characteristics, we in fact find enough underlying commonality between binary and ASCII protocols that we can treat both styles in a uniform fashion within declarative binpac specifications, as we will develop below. On the other hand, there are some critical differences between the grammars of network protocols (binary as well as ASCII) and those of programming languages[14].

➢ **Concurrent Input**

A fundamental difference between a protocol parser and a Yacc style parser is their input model. A protocol-parser has to parse many connections simultaneously and, within each connection, the two flows on opposite directions, *in parallel*. For example, in persistent HTTP connections each request needs to be associated with the

correct reply. Similarly, the syntax of a SUN/RPC reply depends on program and procedure fields in the corresponding RPC Call[14].

Parsers generated by Yacc/Flex process input in a .pull. fashion. That is, when input is incomplete, the parser blocks, waiting for further input. Thus, a thread can handle only one input stream at a time. To handle flows simultaneously without spawning a thread for each one, the parsers must instead process input *incrementally* as it comes in, partially scanning and parsing incomplete input and resuming where the analysis left off when next invoked[14].

- **Robustness**

Parsing errors are inevitable when processing network traffic. Error scan be caused by irregularity in real-world traffic data (protocol deviations, corrupted contents) as well as by incomplete input due to packet drops when capturing network traffic, asymmetric routing (so that only one direction of a connection is captured), routing changes, or .cold start. (a connection was already underway when the monitor begins operation). Unlike compilers, protocol parsers cannot simply complain and stop processing, but must robustly detect and recover from errors. This is particularly important if we consider the presence of adversaries: an attacker might specially craft traffic to lead a protocol parser into an error condition[14].

- **Binpac Language**

This section describes the design of the binpac language and its compiler, which are specifically tailored to address these properties. Here we assume that a parser generated from the binpac language receives data from a lower-level protocol (such as TCP) analyzer. While one can also use the binpac language to build parsers for TCP/IP packets. How to manage states of TCP Connections and invoke corresponding application analyzers[14]

- **Data Model**

BinPAC's data model provides integral and composite types which allow us to describe basic patterns in protocol data layout, parameterized types to pass

information between grammar elements, and derivative data fields to store intermediate computation results. We discuss these in turn[14].

- **Integral Type and Composite Type**

A binpac *type* describes both the data layout of a consecutive segment of bytes and the resulting data structure after parsing. Type empty represents zero-length input. Elementary types int8, int16, int32 represent 8-, 16-, and 32-bit integers, respectively.

- **Byte Orders**

For use with binary protocols, binpac allows the user to specify the byte order using a & byteorder attribute. In most cases we also want to propagate the byte-order specification along the type hierarchy to the other types. Conceptually we can pass byte order between types as a parameter but in practice the byte order parameter is required universally for binary protocols[14].

- **State Management**

To model the state of a continuous communication, binpac introduces notions of *flow* and connection. A *flow* represents a sequence of messages and state to maintain between messages. A connection represents a pair of *flows* and state between flows.
For example, a DCE/RPC connection may correspond to a TCP connection on port 135, a UDP session to the Windows messenger port, or a CIFS .named pipe. (a DCE/RPC tunnel through the CIFS protocol). As shown in the HTTP example (line 38), the declaration of a connection consists of definitions of flow types for each flow. The up flow. refers to the flow from the connection originator to the responder, and the .down flow. refers to the flow in the opposite direction. Like types, connections and flows can be parameterized too[14].

2.5.3 Protocol Parser Generation

Two main considerations in parser generation are (1) handling incremental input on many flows at the same and (2) detecting and recovering from errors[24]

- **Incremental Input in Protocol Parser**

One approach to handle incremental input is to make the parsing process itself fully incremental, i.e., to make the parse function ready to stop anywhere, buffer unprocessed bytes at elementary type level, return, and resume on next invocation. The parsing state of a composite type, such as a record, can be kept by an indexing variable pointing to the member to be parsed next and a buffer storing unprocessed raw data.

However, incremental parsing at elementary type granularity is expensive, because boundary checks of adjacent fields can no longer be combined. It is also unnecessary for all the protocols we have encountered. As protocols are designed for easy processing, they often have a natural unit for buffering. Binary protocols (such as DCE/RPC) often have a .length. header field that denotes the total message length[14,24].

- **Error Detection and Recovery in Protocol Parser**

Protocol parsers have to robustly detect and recover from various kinds of errors. Errors can be caused by irregularity in real-world traffic data, including small syntax deviations from the standard, incorrect length fields, corrupted contents, and even payloads of a completely different protocol running on the standard port of the parsed protocol. Errors can also result from incomplete input, such as due to packet drops when capturing network traffic. In these cases, the parser might not know in the specific state of the dialog, e.g., whether what it now sees on HTTP flow is inside a data transferor not. Errors may also arise through incorrect binpac specifications[24].

- **Error Detection**

The following techniques are used for error checking

- **Efficient Boundary Checking.** Conceptually, boundary checking (whether scanning stays within the input buffer) only need take place before evaluating every elementary integer or character type field, because all other types are composed of elementary types. While it would be easy to generate the

boundary checking code this way, the generated code would be quite inefficient, too. Instead, the binpac compiler tries to minimize the number of boundary checks. The basic idea is: before generating boundary checking code for a record field, check recursively whether we can generate the checking for the next field. If so, we can combine them into one check. In this way, the compiler can determine the furthest field for which the boundary checking can be performed at a given point of parsing[14].

- **Handling dropped packets.** When capturing network traffic, packet drops cannot always be avoided. They can be caused by a high traffic volume, kernel scheduling issues, or artifacts of the monitoring environment. Such drops lead to *content gaps* in application-level data processed by protocol parsers. Facing content gaps, parsers not only are unable to extract data for the current message, but also may not even know where the next message starts[14].
- **Run-time type safety.** The only access to parsing results provided to binpac parsers is via typed interfaces. These leaves two aspects of type safety to enforce at run-time: (1) among multiple case fields in a case type, the generated code ensures that only the case that is selected during parsing can be accessed, otherwise it throws a run-time exception. (2) access to array elements is always boundary-checked[14].
- **User-defined error detection** A user may also define protocol specification error checking, using the &check attribute. For example, one may check the data against some protocol signature[14].

- **Error Recovery**

Currently errors are handled in a simple model: when the flow processing function catches an exception, it logs the error, discards the unfinished message as well as the unprocessed data, and initializes to resume on the next chunk of data. One potential problem with this approach is that, for stream based protocols, the next message might not be aligned with the next payload chunk. In the future we plan to add support for re-discovering message boundaries in such cases. Having such a mechanism will also help to further improve parsing performance, as we can then skip

large, semantically uninteresting messages, and re-align with the input stream afterwards[14,24].

2.6 A Common Integrated Solution to Network Security

Instead of using various tools and techniques from different vendors, it is better to use a common integrated solution. This integrated set of network security tools and techniques is called Unified Threat Management.

2.6.1 Unified Threat Management (UTM)

Threat and security co-exist one would not survive without the other. But as threats continue to become more sophisticated, corporate security strategies begin to take precedence. To take a historical look at the security solution market, firewalls were introduced with the onset of large computer networks, which eventually led to desktop Antivirus solutions and gateway antivirus, and most recently the advent of intrusion detection systems and intrusion prevention systems. Early solutions were mainly software specific, but dedicated hardware solutions coupled with software solutions and an underlying operating system have also surfaced[10].

Unified Threat Management (UTM) appliances are all-in-one security appliances for the small to medium business and branch office user market segments. They are fast replacing firewalls to offer comprehensive security to enterprises. They carry firewall, VPN, gateway anti-virus, gateway anti-spam, intrusion detection and prevention, content filtering as basic features. The complete solutions offer bandwidth management and multiple link load balancing and gateway fail over too[7]. A single UTM appliance makes it very easy to manage an organization's security strategy as it is one device to manage, providing one source of support that maintains the complete set of security features. UTM solutions are also a cost-effective investment, lowering the tax on resources and day-to-day costs to boost the bottom line. UTM leverages a host of tightly integrated security solutions that work in tandem systematically to provide comprehensive network security. As there is a customized operating system supporting the technology, the solutions work in unison and provide very high throughput. What makes UTM unique is its ability to bundle separate solutions that are designed to work together without competition. The solution's most important

feature is its single, centralized platform that allows administrators to monitor and configure each of the solutions to reduce resource-draining redundancies. UTM is used to describe network firewalls that have many features in one box, including e-mail spam filtering, anti-virus capability, an intrusion detection (or prevention) system (IDS or IPS), and World Wide Web content filtering, along with the traditional activities of a firewall. These are application-layer firewalls that use proxies to process and forward all incoming traffic, though they can work in a transparent mode that disguises this fact. However, if this uses too much processor time, the higher-level inspection can be disabled so that the firewall functions like a much simpler network address translation (NAT) gateway. Unified threat management (UTM) refers to a comprehensive security product that includes protection against multiple threats. A UTM product typically includes a firewall, antivirus software, content filtering and a spam filter in a single package[9].

Unified Threat Management is an emerging trend in the firewall appliance security market. It is the evolution of the traditional firewall into a product that not only guards against intrusion but performs content filtering, spam filtering, intrusion detection and anti-virus duties traditionally handled by multiple systems. When hackers were the primary focus of an IT enterprise, a firewall was sufficient to protect most networks. Then as viruses became more prevalent, corporate took to anti-virus gateways that scanned for viruses followed by Web content filtering, and later, spam filtering. This resulted in a mess of systems that were costly to administer and took up valuable rack space. As the hardware that powered today's enterprise firewalls became more robust it became viable to add functions that were traditionally off the box right into the firewall. Firewalls became 'firewall appliances'. This is where Unified Threat Management comes in. Rather than administer multiple systems that handle anti virus, content filtering, intrusion detection and Spam filtering, companies can purchase a Unified Threat Management firewall appliance that integrates all of the above into a single rack mountable network appliance. The multiple functionality of the Unified Threat Management appliance can be the justification for replacing older more basic firewalls. UTM appliance must consist of to be regarded as such. First, it must have a operating system and an installation process that requires a minimum of human intervention. The appliance must have the ability to perform network firewalling, intrusion detection and prevention (IDS/IPS) and gateway anti-virus (AV). All

capabilities need not be utilized, but the functions must exist in the appliance. A UTM appliance may also include other features such as security management and policy management by group or user. Unified threat management (UTM) is an emerging trend in the network security market[9]. UTM appliances have evolved from traditional firewall/VPN products into a solution with many additional capabilities, including:

- Spam blocking
- Gateway antivirus
- Spyware prevention
- Intrusion prevention
- URL filtering

These are functions that previously had been handled by multiple systems. UTM solutions also provide integrated management, monitoring, and logging capabilities to streamline deployment and maintenance[7].

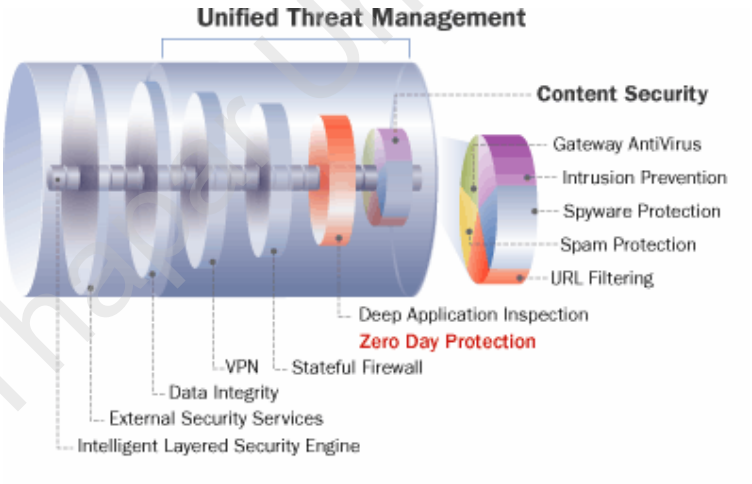

Figure 2.6 A general UTM architecture[7]

In UTM we can also include powerful security right out of the box. Some architectures of the UTM may have built-in, proactive zero day protection to defend against new and emerging threats. Many vendors only provide signature-based protection. These reactive solutions actually leave their customers exposed to new types of threats during the "zero day" timeframe - between an exploit's discovery and a signature update's release [7]. The Unified Threat Management market has emerged

to address complex security challenges**.** In order to solve the security problems for businesses and service providers, the Unified Threat Management (UTM) market has emerged. UTM devices incorporate firewall, intrusion prevention and gateway antivirus, at a minimum. Many vendors have attempted to provide UTM capabilities through various implementations on the theme of UTM. Some vendors have re-packaged existing firewall and VPN offerings with antivirus and intrusion detection and/or prevention technologies from other vendors. Others have simply relabeled their existing network security products, which offer limited threat management capabilities across all three technology areas. In order to address these challenges, an effective UTM solution must deliver a network security platform comprised of robust and fully integrated security and networking functions. Protection must be provided against next generation application layer threats and offer centralized management from a single console, all without impairing the performance of the network.

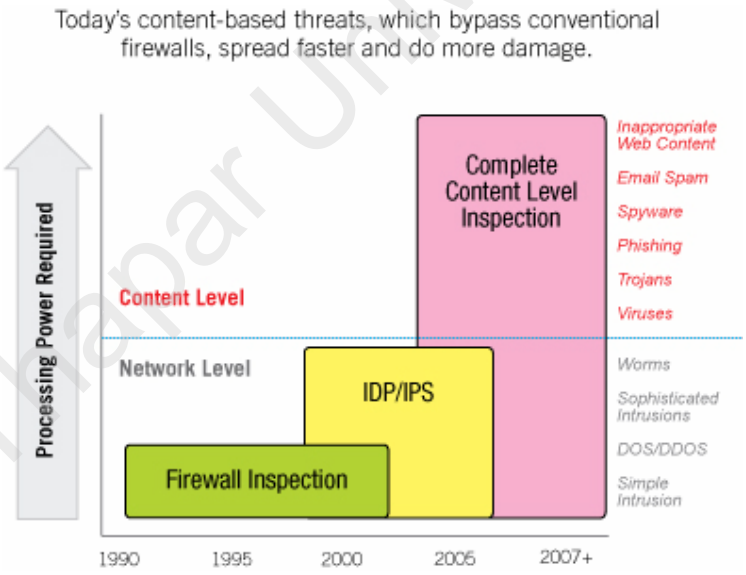

Figure 2.7 Change in Technology requirements with time [8]

Unified Threat Management (UTM), also known as integrated security appliance, refers to a gateway security appliance that embeds multiple security functionality in order to stop multilevel threats that would otherwise require the use of dedicated solutions. A UTM device needs to offer built-in multi-security functionality even though the UTM solution may have been implemented for a specific purpose. These

technologies include firewall and some form of content filtering (e.g. gateway anti-virus, anti-spam, anti- spyware, web filtering, etc.) as basic functions. Furthermore, there are additional vendor specific functionalities such as: SSL and/or IPSec virtual private networks (VPN); intrusion detection and/or prevention systems (IDS/IPS); and user authentication [8]. The most common function that a UTM system performs is traffic inspection and control. Therefore in many cases, UTM systems are replacing dedicated firewall appliances as an integrated and more cost effective solution. UTM systems can either inherently include all of these different technologies and then allow the user to decide which of these functions to turn on or simply offer a platform that can be expanded with the addition of dedicated security software from a range of specialist vendors[9].

2.6.2 Key Benefits of UTM

- **Integration of multiple security technologies into a single device**

This approach provides a seamless integration of different security technologies into a single device. This is especially true in the case of vendors that develop all the hardware and software components themselves. Therefore users avoid the risk of having conflicting security products that cancel out each other's benefits[8].

- **Simplified management**

Vendors normally ship their security products together with a dedicated management platform to control and monitor that specific security product. Therefore, companies often find themselves unable to manage all of these different platforms and devices. In addition, they are unable to store and correlate the different logs adequately. UTM vendors offer a central management console for controlling all the different technologies included in their UTM devices, regardless of the choice of vendors. This reduces the number of management platforms needed to monitor and manage the security infrastructure[9].

- **Lower cost of ownership**

The UTM approach is very appealing from the cost of ownership perspective. Consolidating several gateway security products into one system is cheaper than purchasing, monitoring, updating and managing multiple devices[7].

2.6.3 Constraints of UTM

- **Not suitable for best of breed approach**

Those companies that are looking for the best possible solution in every product category for securing their gateways may find that UTM is not an interesting option for them. UTM systems have been designed to perform multiple functionalities but do not normally offer much flexibility with regards to the choice of technology[10].

- **Challenges of integrating different vendor technologies**

UTM vendors face the challenge of integrating third party software into their appliances. This means that integration work can go in different directions, since the participants in the UTM vendor's eco-system are unlikely to evolve their products in the same way. This can make product development challenging and adversely impact the overall quality of the UTM solution[9]

- **Represents a single point of failure**

The fact that the UTM device can be compromised means that in the event of failure, user left vulnerable to all the threats that the UTM system is supposed to protect against. Alternatively, implementing dedicated security appliances to handle each component separately (e.g. firewall, content security, and IPS amongst others) reduces the overall security risk. If one device becomes affected, the remaining devices may continue to function normally and thus the overall impact is limited. A potential workaround involves the use of load balancing with a back-up UTM appliance, but this would have a significant impact on costs and defy its purpose[10].

- **Possible Effects on Network Performance**

Network performance can be affected as traffic needs to go through several security filters before it reaches the client network. Despite the fact that the different components within the appliance are meant to work seamlessly together, it is also true

that unless the appliance can handle data traffic comfortably (i.e. having sufficient throughput to handle traffic inspection without impacting on overall network performance), the UTM system may represent a potential bottleneck in the network [10].

Chapter 3
Design & Implementation of Protocol Compiler

3.1 Thesis Objective

Thesis objective is to investigate and implement a representative suite of protocols among layers two through seven of the OSI protocol stack, employing formal specifications coupled with automatic code generation techniques. Under this thesis, we will design and implement and a protocol compiler as well as a small, but representative suite of formal protocol specifications that, when input to the automatic protocol generator, are compiled into efficient table-driven implementations. This implementations will be complete, in the sense that they address lexical analysis, grammar-based parsing, and finite state engine transitions in a consistent, systematic fashion. The over all goal of this thesis is to develop an approach for efficiently generating efficient network protocol compilers those will be able to detect various anomalies in network system, for use in network monitoring, network intrusion detection, prevention and response, and other network centric applications. In my thesis work we have three objectives to meet. These objectives are as follows :-

- Create a Protocol Description Language.
- Designing Compiler for the same.
- Use these to create a versatile Protocol Parser.

3.1.1 Creating a Protocol Description Language

Many previous protocol description languages have been designed for Verification. Protocol language description in fact are formal description techniques. Protocol description language is used to describe various protocol that could be understood by any protocol compiler. Finite state machines make specifications complicated and difficult to change, even for carefully layered protocols. In protocol description language Context free grammar are used to specify a protocol specification.

The protocol description language might be incomplete and it will be difficult to use for some protocols. The most important issue will be of safety as this protocol

compiler and language intended to ensure the reliable operation of applications in a potentially adversarial environment, it is imperative that protocol analyzers do not introduce new sources of failure or vulnerabilities. We therefore wanted to avoid memory corruption errors and provide resistance to denial-of-service attacks. All of the analysis will be performed within a memory safe language, and we introduced further restrictions into the language to limit the amount of memory and CPU used by protocol analysis. The second goal should be of providing a real time analysis and response.

The language and runtime must be designed based on a streaming data model, parsing, analyzing, and enacting decisions on potentially incomplete application messages. In terms of performance, our prototype implementation can process data at speeds matching the traffic demands of busy web servers, allowing for online operation. Our goal will also be on rapid development or generation of protocol analyzers. We will try to structure the domain-specific language to be similar to the BNF specifications found in many RFCs that describe protocols.

A protocol compiler written in this language will take care of these three things First, it specifies how to parse the message format used by a protocol. Second, it needs to correctly track session and application context. Finally, it needs to perform analysis based on the message content and the application context, and potentially carry out decisions, such as terminating a connection. The security will be the main concerned that is why proper bound checking must be done. The language must be complete enough to express many important protocols and vulnerabilities in these protocols, that the amount of effort required for the specification should be reasonable, and that the language features are helpful in this task.

- **Protocol Description Language (PDL)**

Some examples for protocol description language are as follows

- **PDL for Internet Protocol (IP)**

Here is an example of the PDL for the IP protocol:

```
ipAddress     FIELD
              SYNTAX            BYTESTRING(4)
              DISPLAY-HINT      "1d."
              LOOKUP            HOSTNAME
              DESCRIPTION
```

```
 "IP address"
ipversion     FIELD
              SYNTAX            INT(4)
              DEFAULT           "4"
ipHeaderLength    FIELD
                  SYNTAX INT(4)
ipTypeOfService   FIELD
                  SYNTAX            BITSTRING(8) { minCost(1),
                                    maxReliability(2),
                                    maxThruput(3),
                                    minDelay(4) }
ipLength      FIELD
              SYNTAX UNSIGNED INT(16)
ipFlags       FIELD
              SYNTAX            BITSTRING(3) { moreFrags(0),
                                dontFrag(1) }
IpFragmentOffset  FIELD
                  SYNTAX            INT(13)
ipProtocol    FIELD
              SYNTAX INT(8)
              LOOKUP FILE "IpProtocol.cf"
ipData FIELD
              SYNTAX            BYTESTRING(0..1500)
              ENCAP             ipProtocol
              DISPLAY-HINT      "HexDump"
ip        PROTOCOL
          SUMMARIZE
          "$FragmentOffset != 0"
           "IpFragment ID=$Identification Offset=$Fragmentoffset"
          *"Default" :
           "IP Protocol=$Protocol"
          DESCRIPTION
           "Protocol format for the Internet Protocol"
          REFERENCE "RFC 791"
::= { Version=ipVersion, HeaderLength=ipHeaderLength,
          TypeOfService=ipTypeOfService, Length=ipLength,
          Identification=UInt16, IpFlags=ipFlags,
          FragmentOffset=ipFragmentOffset, TimeToLive=Int8,
          Protocol=ipProtocol, Checksum=ByteStr2,
          IpSrc=ipAddress, IpDest=ipAddress, Options=ipOptions,
          Fragment=ipFragment, Data=ipData }
ip        FLOW
          HEADER { LENGTH=HeaderLength, IN-WORDS }
          NET-LAYER {
              SOURCE=IpSrc,
              DESTINATION=IpDest,
              FRAGMENTATION=IPV4,
              TUNNELING
          }
          CHILDREN { DESTINATION=Protocol }
ipFragData    FIELD
              SYNTAX            BYTESTRING(1..1500)
              LENGTH            "ipLength - ipHeaderLength * 4"
              DISPLAY-HINT      "HexDump"
ipFragment    GROUP
              OPTIONAL          "$FragmentOffset != 0"
::= { Data=ipFragData }
ipOptionCode  FIELD
              SYNTAX INT(8) { ipRR(0x07), ipTimestamp(0x44),
                                    ipLSRR(0x83),
                                    ipSSRR(0x89) }
```

```
                DESCRIPTION
                 "IP option code"
    ipOptionLength    FIELD
                      SYNTAX UNSIGNED INT(8)
                      DESCRIPTION
                       "Length of IP option"
    ipOptionData  FIELD
                      SYNTAX            BYTESTRING(0..1500)
                      ENCAP              ipOptionCode
                      DISPLAY-HINT      "HexDump"
    ipOptions     GROUP
                  LENGTH           "(ipHeaderLength * 4) - 20"
    ::= { Code=ipOptionCode, Length=ipOptionLength, Pointer=UInt8,
              Data=ipOptionData }[37]
```

- **PDL for HTTP**

While designing PDL for HTTP this thesis work referred the work of [37] as per PDL for IP as described in [37].

```
    httpData FIELD
        SYNTAX    BYTESTRING(1..1500)
        LENGTH     "($ipLength - ($ipHeaderLength * 4)) -
                   ($tcpHeaderLen * 4) "
    DISPLAY-HINT      "Text"
    FLAGS             NOLABEL
    http        PROTOCOL
                SUMMARIZE
                    "$httpData m/ GET| HTTP| HEAD|
     POST/" :
                       "HTTP $httpData"
                    "$httpData m/ [Dd]ate| [Ss]erver|
     [Ll]ast-
                    [Mm]odified/" :
                       "HTTP $httpData"
                    "$httpData m/ [Cc]ontent-/" :
                       "HTTP $httpData"
                    $httpData m/ <HTML>/" :
                       "HTTP [HTML document]"
                    $httpData m/ GIF/" :
                       "HTTP [GIF image]"
                    "Default" :
                       "HTTP [Data]"
                DESCRIPTION
                    "Protocol format for HTTP."
    ::= { Data=httpData }
    http FLOW
    HEADER { LENGTH=0 }
    CONNECTION { INHERITED }
    PAYLOAD { INCLUDE-HEADER, DATA=Data, LENGTH=256 }
    STATES
                    "S0: CHECKCONNECT, GOTO S1
                       DEFAULT NEXT S0
                    S1: WAIT 2, GOTO S2, NEXT S1
                       DEFAULT NEXT S0
                    S2: MATCH
.backslash.n.backslash.r.backslash.n` 900 0 0 255 0, NEXT S3
`.backslash.n.backslash.n` 900 0 0 255 0, NEXT S3
                          `POST /tds?`       50 0 0 127 1,
                                            CHILD sybaseWebsql
```

```
                    `.hts HTTP/1.0`    50 4 0 127 1,
                                       CHILD sybaseJdbc
                    `jdbc:sybase:Tds`  50 4 0 127 1,
                                       CHILD sybaseTds
                    `PCN-The Poin`     500 4 1 255 0,
                                       CHILD pointcast
                    `t: BW-C-`         100 4 1 255 0,
                                       CHILD backweb
                    DEFAULT NEXT S3
                s3: MATCH
`.backslash.n.backslash.r.backslash.n`  50 0 0  0 0,  NEXT S3

`.backslash.n.backslash.n`  50 0 0  0 0, NEXT S3
                    `Content-Type:`    800 0 0 255 0,
                                       CHILD mime
                    `PCN-The Poin`     500 4 1 255 0,
                                       CHILD pointcast
                    `t: BW-C-`         100 4 1 255 0,
                                       CHILD backweb
                    DEFAULT NEXT S0"
sybaseWebsql        FLOW
                    STATE-BASED
sybaseJdbc          FLOW
                    STATE-BASED
sybaseTds           FLOW
                    STATE-BASED
pointcast           FLOW
                    STATE-BASED
backweb             FLOW
                    STATE-BASED
mime                FLOW
                    STATE-BASED
                    STATES
" S0: MATCH
`application`       900 0 0  1 0,
                    CHILD mimeApplication
`audio`             900 0 0  1 0,
                    CHILD mimeAudio
`image`              50 0 0  1 0,
                    CHILD mimeImage
`text`               50 0 0  1 0,
                    CHILD mimeText
`video`              50 0 0  1 0,
                    CHILD mimeVideo
`x-world`           500 4 1 255 0,
                    CHILD mimeXworld
DEFAULT GOTO S0"
mimApplication      FLOW
                    STATE-BASED
mimeAudio         FLOW
                  STATE-BASED
                  STATES
                   "S0: MATCH
                       `basic`           100 0 0 1 0,
                                         CHILD pdBasicAudio
                       `midi`            100 0 0 1 0,
                                         CHILD pdMidi
                       `mpeg`            100 0 0 1 0,
                                         CHILD pdMpeg2Audio
                       `vnd.rn-realaudio` 100 0 0 1 0,
                                         CHILD pdRealAudio
```

```
                        `wav`              100 0 0 1 0,
                                           CHILD pdWav
                        `x-aiff`           100 0 0 1 0,
                                           CHILD pdAiff
                        `x-midi`           100 0 0 1 0,
                                           CHILD pdMidi
                        `x-mpeg`           100 0 0 1 0,
                                           CHILD pdMpeg2Audio
                        `x-mpgurl`         100 0 0 1 0,
                                           CHILD pdMpeg3Audio
                        `x-pn-realaudio`   100 0 0 1 0,
                                           CHILD pdRealAudio
                        `x-wav`            100 0 0 1 0,
                                           CHILD pdWav
                     DEFAULT GOTO S0"
mimeImage         FLOW
                  STATE-BASED
mimeText          FLOW
                  STATE-BASED
mimeVideo         FLOW
                  STATE-BASED
mimeXworld        FLOW
                  STATE-BASED
pdBasicAudio         FLOW
                  STATE-BASED
pdMidi               FLOW
                  STATE-BASED
pdMpeg2Audio         FLOW
                  STATE-BASED
pdMpeg3Audio         FLOW
                  STATE-BASED
pdRealAudio          FLOW
                  STATE-BASED
pdWav                FLOW
                  STATE-BASED
pdAiff               FLOW
                  STATE-BASED
```

3.1.2 Designing Compiler for generated PDL

Each protocol defines a particular message format for exchanging data. These formats can roughly be classified into text and binary. Text formats use ASCII text to encode both the structure and the content of messages, following some sort of grammar that is often formalized in Backus-Naur form (BNF). Binary formats use machine structures to encode data, overlaying C-like constructed types onto the data stream. A generic protocol analyzer must, of course, be able to parse both types of protocols. In particular, network monitoring tools such as Ethereal [29] and tcpdump [28], intrusion detection systems such as Snort [30] and Bro [36], and application-level firewalls all perform some form of protocol analysis, but each tool involves hand-coding the analyzers in a general-purpose, low-level language such as C.

Protocols can be layered on top of other protocols, with a specification at each layer performing protocol analysis and sending data up to the next layer.

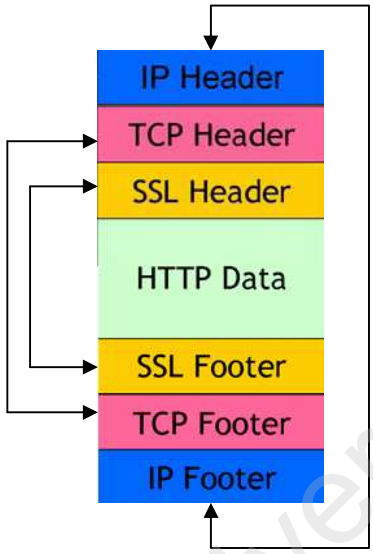

Figure 3.1 Protocol Layered Architecture

A specification can indicate a lower layer specification with a uses statement, or directly bind to the transport layer with a transport statement.

3.1.3 Versatile Protocol Parser

The analysis engine will be protocol compiler or protocol parser generated on the basis of protocol description language. It parses messages using a recursive descent parser. The protocol compiler first finds the appropriate description language specification for the current packet to be analyzed. The compiler then follows the grammar that specifies the message format to generate a parse tree. Since the engine may receive packets containing incomplete messages, it performs parsing incrementally, saving parsing state between packets.

During parsing, the protocol parser executes code fragments. The handlers use both the parse tree and any other session state updated by the code fragments to carry out task-specific logic and to update the current protocol state. Whenever the analysis engine fails to parse a message, it alerts the application containing protocol parser,

which can then react appropriately. For example, a firewall would likely drop the message, while a network monitor such as Ethereal could display an uninterpreted byte stream. In the remainder of the section, we focus on our techniques to limit the state in order to resist state-holding attacks and to achieve fidelity in the engine's interpretation of the current communication state of an application.

To prevent state-holding, we will structure our protocol analyzer to perform filtering decisions as quickly as possible with the use of incremental execution. After receiving each packet, the appropriate handler will be executed, even if the application-level message is incomplete. The handler will run until it references a message field that is not yet filled with a value. At that point, its execution will be suspended and a continuation will be saved, to be resumed when the next packet arrives. If the next packet contains the referenced field, the handler execution continues, otherwise, it will suspended once again. If the handler completes, the rest of the message is parsed without saving any state.

A protocol description for a particular protocol at a particular layer level includes any child protocols of the particular protocol, where in the packet, information related to the particular child protocol may be found. A protocol description also includes any protocol specific operations to be performed on the packet for the particular protocol at the particular layer level. The method includes performing the protocol specific operations on the packet specified by the set of protocol descriptions based on the base protocol of the packet and the children of the protocols used in the packet. A particular embodiment includes providing the protocol descriptions in a high-level protocol description language, and compiling to the descriptions into a data structure. The protocol specific operations may include parsing and extraction operations to extract identifying information.

In order for a network monitor to be able to analyze different packet or frame formats, the monitor needs to be able to perform protocol specific operations on each packet with each packet carrying information conforming to different protocols and related to different applications. For example, the monitor needs to be able to parse packets of different formats into fields to understand the data encapsulated in the different fields.

As the number of possible packet formats or types increases, the amount of logic required to parse these different packet formats also increases.

3.2 HTTP Traffic

The nature of network traffic is critical in order to properly design and implement computer networks and network services like the World Wide Web. Recent examinations of LAN traffic and wide-area network traffic have challenged the commonly assumed models for network traffic. Traffic that is bursty on many or all time scales can be described statistically using the notion of *self-similarity*. Self similarity is the property we associate with one type of fractal—an object whose appearance is unchanged regardless of the scale at which it is viewed. In the case of stochastic objects like time series, self-similarity is used in the distributional sense.

Now the issue arises when we are applying the firewall policies or our network policies just by checking the port numbers. But it might possible that the payload in itself may contain some another url to some banned site or the packets to/from banned destination or source.

3.3 Capturing Network Traffic

In order to detect network security threats, the first step is to capture the network traffic and then do it's analysis. For capturing the packets from the network traffic flowing in the traffic various tools can be used like tcpdump, ethereal, wireshark. These tools can capture the data packets and these captured packets can be used for active analysis or passive analysis. In active analysis the data packets at the same time when they are captured, on the other hand in passive analysis packets once captured, they are stored and analyzed later in future. So both active as well as passive network traffic analysis is important [25].

Some of the tools which can capture the network traffic or packets flowing in the network are as tcpdump, ethereal, snort. All of these tools are developed by using popular programming library called *libpcap* that provides a high level interface to packet capture [25,26].

3.3.1 Using tcpdump for Network Traffic Capturing

Tcpdump prints out a description of the contents of packets on a network interface that match the Boolean *expression*. It can also be run with the **-w** flag, which causes it to save the packet data to a file for later analysis, and/or with the **-r** flag, which causes it to read from a saved packet file rather than to read packets from a network interface. In all cases, only packets that match *expression* will be processed by *tcpdump*[28].

When *tcpdump* finishes capturing packets, it will report counts of: packets captured. Packets received by filter (the meaning of this depends on the operating system on which *tcpdump* is running, and possibly on the way the Operating system was configured - if a filter was specified on the command line, on some Operating system it counts packets regardless of whether they were matched by the filter expression and, even if they were matched by the filter expression, regardless of whether *tcpdump* has read and processed them yet, on other Operating systems it counts only packets that were matched by the filter expression regardless of whether *tcpdump* has read and processed them yet, and on other Operating systems it counts only packets that were matched by the filter expression and were processed by *tcpdump*), packets dropped by kernel[25].

3.3.2 Using Ethereal for Network Traffic Capturing

Ethereal is a network packet analyzer. A network packet analyzer will try to capture network packets and tries to display that packet data as detailed as possible. *Ethereal* is a very powerful well featured packet analyzer. It captures packets and decodes them into their component parts for analysis. The range of decodes is very large, if we've heard of a protocol there's a good chance that *Ethereal* has a decoder for it, and if you haven't there's still likely to be a decoder for it. *Ethereal* is available for use on UNIX systems and for Microsoft Windows also. We can think of a network packet analyzer as a measuring device used to examine what's going on inside a network cable[27].

➢ **Some Intended purposes to use Ethereal**

Here are some examples people use Ethereal for:

- network administrators use it to troubleshoot network problems
- network security engineers use it to examine security problems
- developers use it to debug protocol implementations
- people use it to learn network protocol internals

Beside these examples, Ethereal can be helpful in many other situations too.

➢ **Packet Capturing using Ethereal from many different media**

Despite its name, Ethereal can capture traffic from network media other than Ethernet. Ethereal can save packets captured in a large number of formats of other capture programs. Ethereal is an open source software project, and is released under the GNU General Public License.

➢ **Start Packet Capturing**

Ethereal can be used in two ways to start capturing packets with Ethereal:

1. From the command line using the following:

ethereal -i eth0 -k

This will start Ethereal capturing on interface eth0.

2. By starting Ethereal and then selecting Start... from the Capture menu, this brings up the Capture Options dialog box.

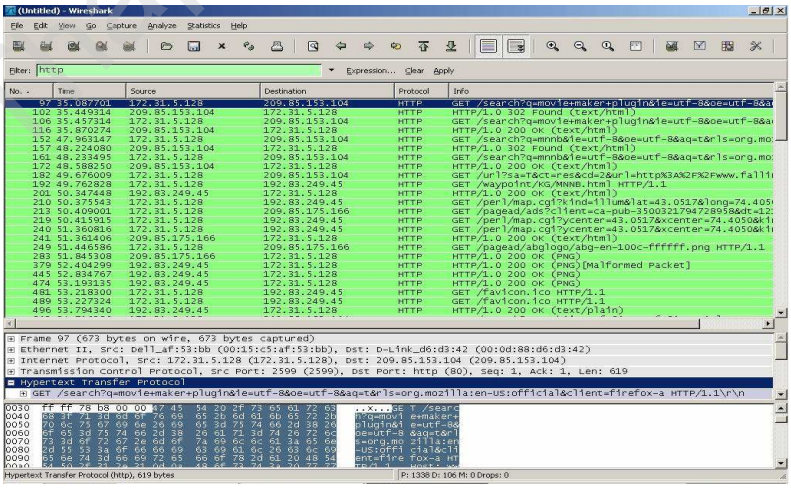

Figure 3.2 Packet capturing using Ethereal

3.4 Milestones

The following milestones were set for this thesis.

3.4.1 Developing a Compiler for /etc/network/interface file

This compiler will parse the /etc/network/interface file and will present the information in human readable format. The /etc/network/interface file contains information regarding ip address. This compiler will be able to parse this file and will be able to produce the following information

1. Ip Address :- IP address of the machine
2. Subnet Mask :- Subnet mask information
3. Interface Name :- Name of the interface on which particular ip is assigned.
4. Ip Type : Like Static or Dynamic

- **Lexical Analyzer Generation**

  We have written a flex program that will generate a lexical analyzer for out target /etc/network/interface file. This is saved as Network_Interface.l. This file will work as input to lexical analyzer generator. The output of Flex will be a file named as lex.yy.c.

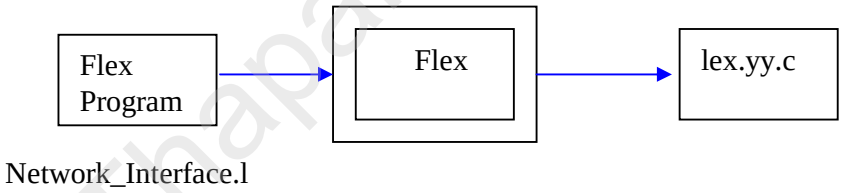

Figure 3.3 Lexical Analyzer Generation Process.

```
/******* LEXICAL ANALYZER GENERATION PROGRAM ********/
%{

/* flex programs that returns tokens  */

#include"Network_Interface.tab.h"
#include<string.h>
%}
colon [ ":" ]+
dot[“.”]+
```

```
%%

[A-Za-z0-9]+ {
                strcpy(otherText,yytext);
        return WORD;
     }
{colon} {return COLON; }
{dot } { return DOT;}
[\n] {

   return NEWLINE;
}
%%
#ifndef YYERRCODE
#define YYERRCODE 256
#endif
```

- **Parser Generator**

We have written a bison program that will generate a parser for out target /etc/network/interface file. This is saved as Network_Interface.y. This file will work as input to parser generator(Bison).

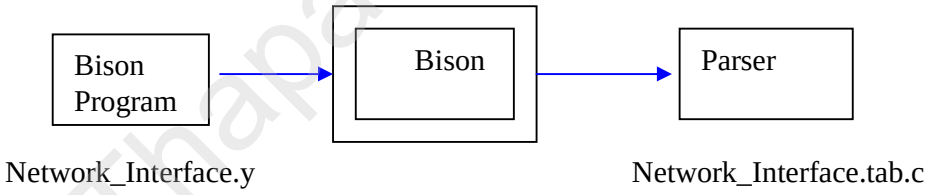

Figure 3.4 Parser Generation Process

```
/************ PARSER GENERATION PROGRAM *************/

%{
#include <stdio.h>
extern FILE* yyin;
%}
%union {
          char text[1024];
}
%token <text> WORD
```

```
%token SEP
%type <text> word line
%%

file: line
        | file'\n'line
        | file'\n'
        ;
line: word SEP word SEP word SEP word SEP word SEP word {
        printf("\n Host Name :- %s ",$1);
        printf("IP Address :- %s ",$3);
        printf("Subnet Mask  :- %s ",$5);
        printf("Interface Name :- %s ",$7);
        printf("IP Type :- %s ",$9);
        }
        ;
word: WORD
        | { $$[0] = '\0'; }
        ;
%%

main()
{

 char fileName[32];
 FILE *fpt;
printf("Enter the file name to be parsed  ");
scanf("%s",fileName);

 fpt = fopen(fileName,"r");
 yyin =fpt;
 if(!fpt){
     fprintf(stderr, "\n could not open the file interface");
 }
 else
 {
  printf("\n file is ready to open\n");
  }
        do {
                yyparse();
```

```
        } while (!feof(yyin));
        fclose(fpt);
}

yyerror(char* s)
{
  printf("\n Error occured  %s \n",s);

}
```

- **Compilation and Execution**

The following steps are needed to be followed to compile and run a compiler . To generate a lexical analyzer the following steps will be followed

Step 1:- flex Network_Interface.l
It's output will be a lex.yy.c file.

Step 2:- bison -d Network_Interface.y
It's output will be a Network_Interface.tab.c

Step 3:- cc lex.yy.c Network_Interface.tab.c -o Network_Interface.out -lfl
It's output will be a file Network_Interface.out

Step 4: ./ Network_Interface.out

3.4.2 Developing a Compiler to Validate Numerical Expressions

Proposed compiler will parse the input file in which some numerical expression will be written and will be processed to check that is/are these valid numerical expressions. This will act as a major milestone in writing compiler for HTTP in which we will require to find various tokens(like ?, \\ etc.) and to find a common signature across various packets of input stream.

➢ **Lexical analyzer generator**

We have written a flex program that will generate a lexical analyzer validation of numeric expressions. This is saved as Numeric.l. This file will work as input to lexical analyzer generator. In this file our rules are written to generate required tokens.

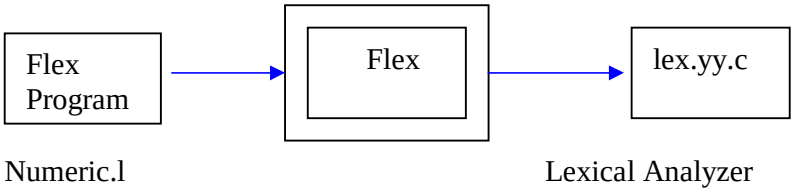

Figure 3.5 Lexical Analyzer Generation Process

```
/******* LEXICAL ANALYZER GENERATION PROGRAM ********/

%{
/* flex programs that returns tokens  */
#include"Numeric.tab.h"
#include<string.h>
char otherText[100];
%}

operator [+*-\/]

%%
[0-9]+ { strcpy(yylval.text,yytext);
                return OPERAND;
               }

[ \t]+ {  return SPACE;
        }
{operator} { return OPERATOR;
           }
. { };
\n { return '\n'; }

%%
```

```
#ifndef YYERRCODE
#define YYERRCODE 256
#endif
```

- **Parser Generator**

  We have written a bison program that will generate a parser for out target file to evaluate numerical expression written in it. This is saved as Numeric.y. This file will work as input to parser generator(Bison).

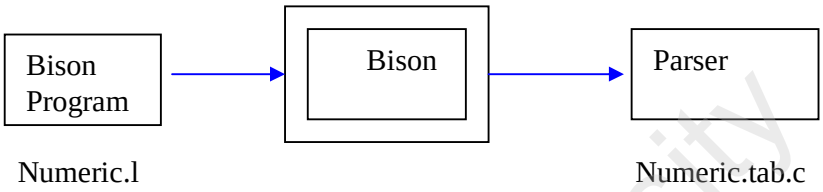

Figure 3.6 Parser Generation Process

```
/************ PARSER GENERATION PROGRAM *************/

%{
#include <stdio.h>

extern FILE* yyin;
extern char otherText[100];

%}

%union {
        char text[1024];
}

%token <text> OPERAND
%token OPERATOR SPACE
%type <text> word line

%%

file: line
        | file'\n'line
```

```
            | file'\n'
            ;

line: word OPERATOR word {
                printf("\n Valid Expression of type 1");

            }
      |
    word SPACE OPERATOR word {
            printf("\n Valid Expression of type 2 ");
      }
      |
    word OPERATOR SPACE word {
            printf("\n Valid Expression of type 3 ");
      }
      |
    word SPACE OPERATOR SPACE word {
            printf("\n Valid Expression of type 4 ");
      }
      |
    word SPACE word {
            printf("\n Inalid Expression ");
      }
    ;
 word: OPERAND
            | { $$[0] = '\0'; }
            ;
%%

main()
{
 char fileName[32];
 FILE *fpt;

 printf("Enter the file name to be parsed  ");
 scanf("%s",fileName);

 fpt = fopen(fileName,"r");
 yyin =fpt;
 if(!fpt)
```

```
 {
 fprintf(stderr, "\ncould not open the file numeric pass with numeric expressions entries");
   }
 else
 {
  printf("\nfile is ready to open\n");
  }
          do {
                  yyparse();
          } while (!feof(yyin));
          fclose(fpt);
}

yyerror(char* s)
{
 printf("\n ** Unexpected Expression Error occured  %s \n",s);
}
```

- **Compilation and Execution**

The following steps are needed to be followed to compile and run a compiler.

Step 1:- flex Numeric.l

It's output will be a lex.yy.c file.

Step 2:- bison -d Numeric.y

It's output will be a Numeric.tab.c

Step 3:- cc lex.yy.c Numeric.tab.c -o Numeric.out -lfl

It's output will be a file Numeric.out

Step 4: ./Numeric.out

3.5 Integrating Protocol Compiler with the Existing Network Systems

This protocol compiler can be integrated with existing network security tools and techniques like IDS, IPS, Firewall, UTM. It's output will be given as input to the some Intrusion Detection System. The IDS can generate alerts on the basis of security policies specified by network administrator.

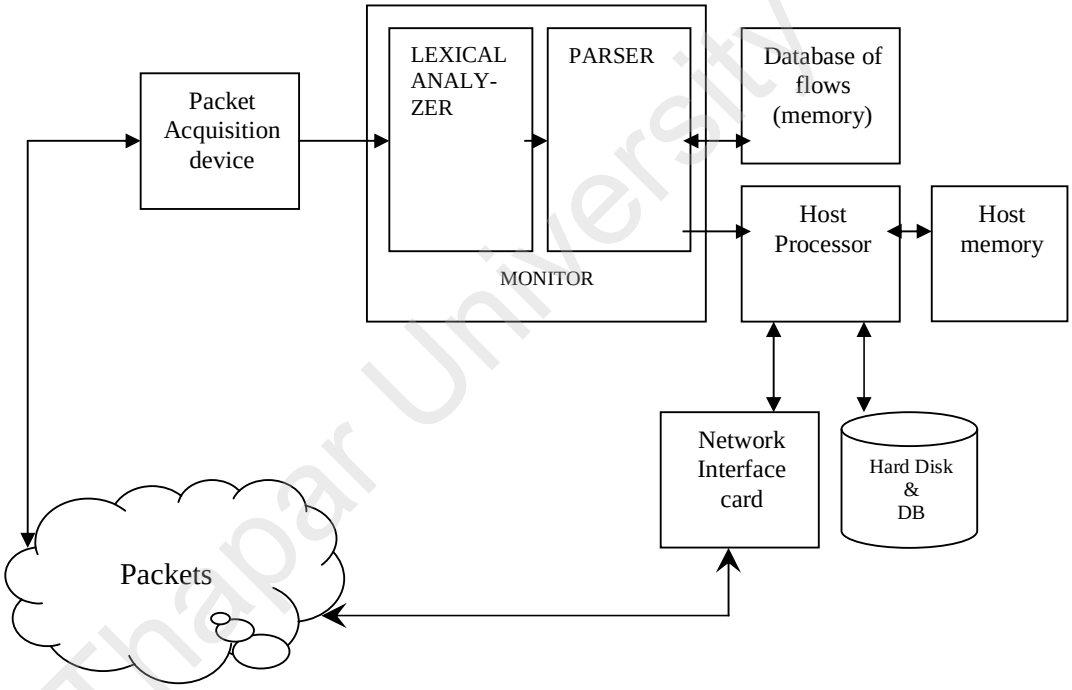

Figure 3.7 Protocol Compiler Integrated with Network System

The protocol parser performs well and can be integrated with existing network systems. It has following features

- **Packet Capturing**

To save time packets must be captured at kernel level so that the context switching time could be saved . It will lead to save a lot of time too. Packets are captured from

network using some packet acquisition device or say some packet capturing tool. These captured packets are parsed with the help of protocol parser and protocol description language.

- **Packet Parsing**

When the packet is captured at kernel level, the packet's content is parsed. For this each protocol a protocol description language is needed to defined on the basis of that packet's data will be parsed and categorized for further actions. Protocol packets are parsed with the help of protocol parser and using protocol description language to find some pattern across multiple packets.

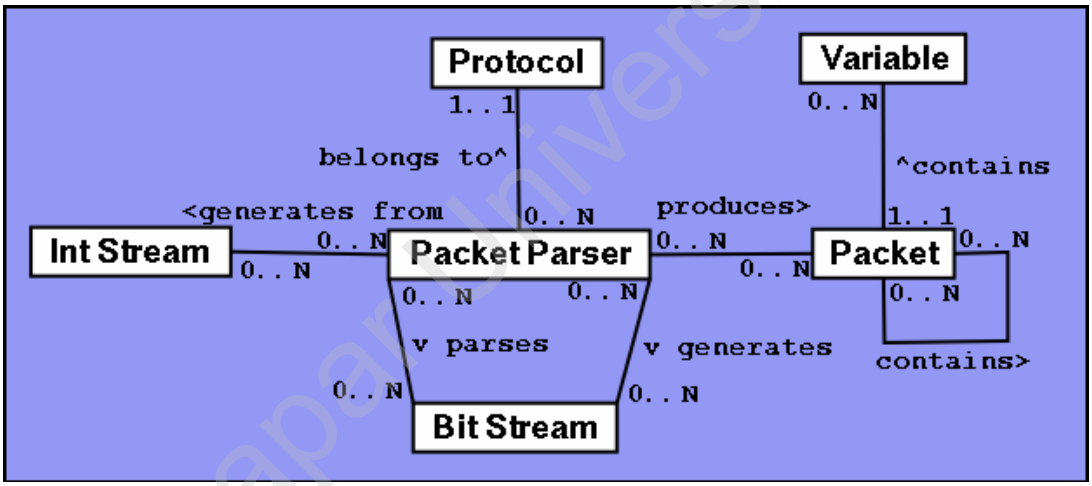

Figure 3.8 Packet Compiling Process

Packets are processed on the basis of protocol description. The input packet stream is given as input to parser. The parser will analyze both packet header and the content of the packet.

DPI is very computationally-intensive. This puts a high stake on the performance of the system to make it a viable solution. Protocol compiler, designed to capture, decode and monitor network traffic, are inconsistently implemented. These software tools are typically implemented as large, hand-crafted bodies of code, subject to individual programmer interpretation of the protocols.

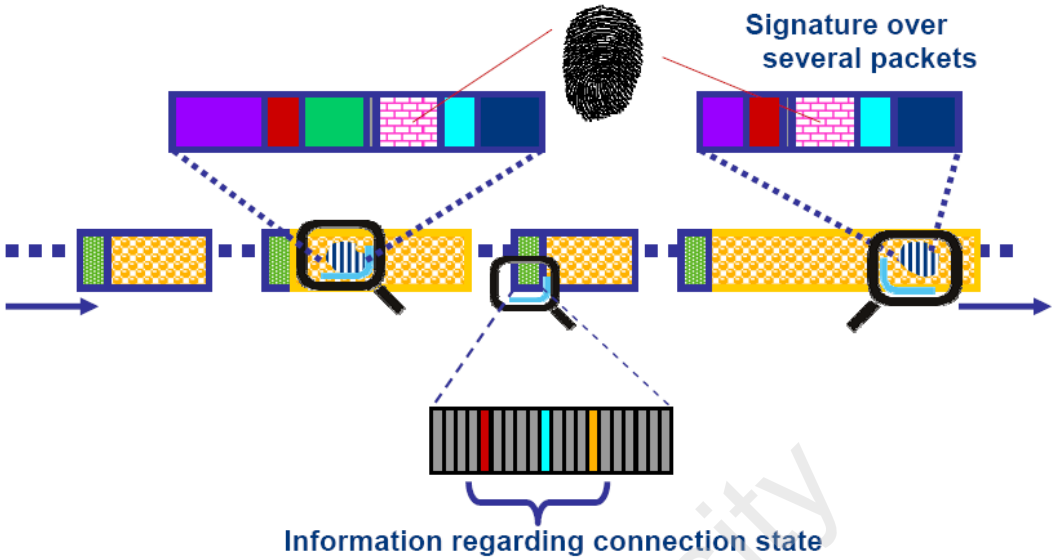

Figure 3.9 Deep Packet Inspection Using Protocol Compiler

3.5.1 Advantages of Protocol Compiler Integration with Hardened Operating System

- **Analysis by Port Number**

  In this packets are classified on the basis of port no attached. The reasoning is the simple fact that many applications use either default ports or some chosen ports in a specific manner.

- **Analysis by String Match**

  In this search for a sequence of textual characters or numeric values within the contents of the packet. Furthermore, string matches may consist of several strings distributed within a packet or several packets.

- **Analysis by Numerical Properties**

  Analysis by numerical properties involves the investigation of arithmetic and numerical characteristics within a packet, and of a packet or several packets. Some examples of properties analyzed include payload length, the number of packets sent in response to a specific transaction, and the numerical offset of some fixed string value within a packet.

- **Intrusion Detection System**

Just after protocol compiler the Intrusion detection system that will generate alerts on the data categorized by our protocol compiler. The IDS will take decision of generating alerts on the basis of rules specified in regular expressions and signatures database of various threats.

- **Intrusion Prevention System**

In proactive security approach the Intrusion Detection System will identify security threat on the basis of alerts generated by IDS. Intrusion Prevention System will take appropriate action on the basis of network security policy defined by Network Administrator.

3.5.2 Anomalies Detected in HTTP traffic

Anomalies on the network traffic are previously unseen traffic behaviors. Identifying the anomalies in a timely fashion is a fundamental part of this thesis. The length of a request can be a good feature for evaluating the correctness of a payload.The following anomaly was detected

A Denial-of-Service event, usually flood based. In that instance: an outbound flood of 40-byte TCP packets was found from a campus host with ip address 172.31.5.64 that had its security compromised was found.

3.5.3 Collective Efforts towards Real Product

The collective work done with my colleague researchers can be well described with the help of following diagram.

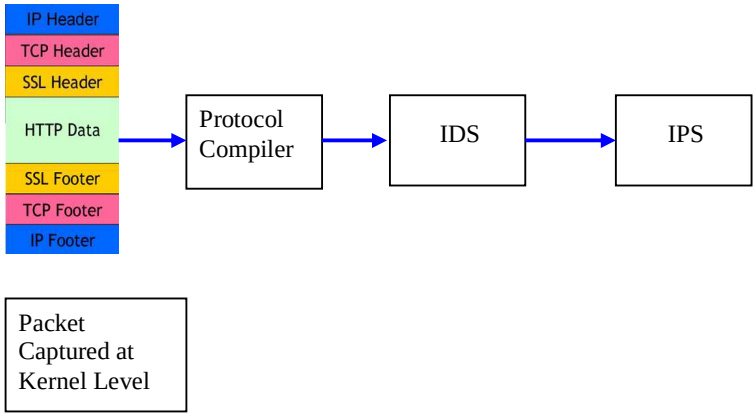

Figure 3.10 Collective Efforts

- **Packet Control**

This module will perform the task of packet routing, bridging, packet capturing, packet modification, firewalling. It will be the first module to handle packets.

- **Packet Parsing**

This module is Protocol Compiler module and it performs the task of performing the packet parsing. It parses the contents of the HTTP packet and looks for patterns of the protocol with the help of protocol description language. The output result is given as input to IDS.

- **Intrusion Detection System**

On the basis of results from the protocol compiler , IDS will generate corresponding alerts. The output of the IDS works as input to IPS.

- **Intrusion Prevention System**

The Intrusion Prevention System works as a proactive approach for Network Security. The IPS will take the appropriate actions on the basis of alerts generated by Intrusion Detection System.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

Network security is a vast field. A lot of work has been done by a no of people. A lot of tools and techniques are existing in the market. We have firewalls, IDS, IPS and some UTM also available. But the only these things are not sufficient to make our network secure. Most of these techniques work by the looking at the packet headers. So to make a secure network system we need to look inside the packets also to check that what actually is flowing inside the packets. In this work we are using protocol compilers and parser to parse these packets, the result indicates that these kind of parser can be integrated with other kernel modules to help in achieving better security.

By using these kind of protocol compiler we can detect various kind of anomalies threats hidden inside the packet payload and a lot of malicious activities can be stopped from happening. This kind of protocol compiler can parse the data and can give input to some IDS so that alerts can be generated for it and on the basis of alert generated by IDS our IPS can take appropriate action and this will help us a lot in making our network secure.

4.2 Future Work

The proposed solution is for HTTP protocols only. This work can be extended for more protocols like SMTP, SNMP, TFTP,IMAP(Internet Message Access Protocol), Bit Torrent(A peer to peer file transfer protocol) and many more. This work can work as a full fledged layer in Network Security. This solution can work well with other existing N/W security tools and techniques. By the implementation of this work a lot of security threats can be avoided like ICMP tunneling which even could lead to DOS (Denial of Service) attack

References

[1] Bruce Schneier, “Secrets and Lies”, John Wiley& Sons, 2000.

[2] My Audit PC
URL:http://www.auditmypc.com/freescan/readingroom/Network_Security_overview.asp

[3] Paul D. Robertson, Matt Curtin, Marcus J. Ranum
URL:**http://www.interhack.net/pubs/fwfaq/firewalls-faq.html**

[4] TIS FWTK
URL**:** http://www.fwtk.org

[5] IBM Internet Security Systems
URL: http://www.iss.net/blackice/

[6] NSS Group
URL: http://www.nss.co.uk

[7] WatchGuard Products
URL: www.watchguard.com/appliances

[8] FORTIGATE Unified Threat Management
URL: http://www.fortinet.com/products/

[9] Mick Johnson “Building a High performance Unified Threat Management appliance” analog ZONE

[10] Frost & Sullivan “Unified Threat Management”

[11] Information System Security Operation Automatic Generation of Protocol Analyzers. URL: http://www.isso.sparta.com/research

[12] Eddie Kohlar “Prolac: A language for protocol compilation” In department of Electrical Engineering and Computer Science, MIT, August 29, 1997

[13] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4.19, 1993.

[14] Ruoming Pang, Vern Paxon, Robin Sommer, Larry Peterson “binpac: A Yacc for writing Application Protocol Compiler”.

[15] Tommy M MacGuire, M.G. Gouda “The Austin Protocol Compiler” In Kluwer Academic Publishers, 2005.
URL: http://www.crsr.net

[16] Mart´ın Abadi, Bruno Blanchet "Analyzing Security Protocols with Secrecy Types and Logic Programs".

[17] Antonio Durante, Ricardo Focardi, Roberto Gorrieri "A Compiler for Analyzing Cryptographic Protocols Using Noninterference".

[18] TIM A. WAGNER and SUSAN L. GRAHAM University of California, Berkeley "General Incremental Lexical Analysis".

[19] Maggie Johnson and Julie Zelenski "Lexical Analysis" Autumn 2007, September 26, 2007.

[20] Maggie Johnson and revised by Julie Zelenski "Bottom-Up Parsing" Autumn 2007, October 8, 2007.

[21] Lesk, M. E. and E. Schmidt [1975]. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey.

[22] Johnson, Stephen C. [1975]. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.

[23] THOMAS NIEMANN "A Guide to Lex & Yacc".

[24] Bison : The Yacc Compatible Parser Generator, 30 May 2006, Bison Version 2.3. URL: http://www.gnu.org/software/bison/manual/

[25] Mark E. Crovella, *Member, IEEE,* and Azer Bestavros, *Member, IEEE* "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes" IEEE/ACM Transactions On Networking, VOL. 5, NO. 6, December 1999.

[26] Luca Deri "Improving Passive Packet Capture: Beyond Device Polling" URL: http://luca.ntop.org/

[27] Richard Sharpe "Ethereal User's Guide V2.00 for Ethereal 0.10.5"

[28] *Van Jacobson, Craig Leres and Steven McCanne* "tcpdump Manual" URL: *http://www.tcpdump.org/*

[29] Ethereal Home, last accessed April 2008 URL: http://ethereal.com.

[30] How to use Snort for packet capturing, last accessed March 2007 URL http://www.snort.org/

[31] Przemyslaw Kazienko, Piotr Dorosz "Intrusion Detection Systems (IDS) Classification; methods; techniques",2004.

[32] Rafeeq Ur Rehman, "Intrusion Detection Systems with Snort", New Jersey

[33] Theuns Verwoerd , "Active Network Security", Honours Report 5 November, 1999.

[34] Allot Communications, "Digging Deeper Into Deep Packet Inspection (DPI)", White Paper.

[35] eSoft, "Modern Network Security: The Migration to Deep Packet Inspection", White Paper.

[36] V. Paxson. Bro: A system for detecting network intruders in real-time. Computer Networks, Dec 1999.

1.1.3 [37] "Processing protocol specific information in packets specified by a protocol description language"

URL: http://www.patentstorm.us/

[38] libPDL - Protocol Definition Language
URL: http://nmedit.sourceforge.net/

List of Papers Published/Communicated

[1] Anranya Yadav, Maninder Singh, "Using Protocol Compiler for Network Security Through DPI", in TENCON 2008 (International Conference on Innovative Technologies on Social Transform), IEEE Hyderabad Section (18-21 November, 2008). (Status: Submitted).