



TECHNISCHE UNIVERSITÄT MÜNCHEN  
INSTITUT FÜR INFORMATIK

Diplomarbeit

# Dynamic Protocol Analysis for Network Intrusion Detection Systems

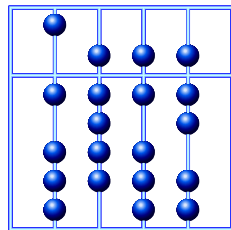
Michael Mai

Aufgabensteller: Prof. Anja Feldmann, Ph.D.

Betreuer: Dipl.-Inf. Holger Dreger

Dipl.-Inf. Robin Sommer

Abgabedatum: 15. September 2005





Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15. September 2005

(Michael Mai)



## **Abstract**

Many Network Intrusion Detection Systems (NIDSs) perform application layer protocol analysis. These systems typically infer the protocol from the ports in the TCP or UDP headers. This is not a reliable technique since many protocols do not use fixed ports. On the other hand there exist better methods to identify used application layer protocols e.g. signatures. In this thesis we present design and implementation of an architecture for NIDSs which supports the integration of these advanced methods for dynamic protocol analysis. The design is suitable for analyzing tunneled connections as well. Our implementation for the open source system Bro uses its existing signature matching engine as additional protocol detection method. On the basis of this prototype we show the results under the aspects of detection rate, need of performance and the interaction of both.

# Contents

<b>I</b>	<b>Introduction</b>	
1.1	Motivation . . . . .	1
1.2	Outline . . . . .	2
<b>II</b>	<b>Background</b>	
2.1	Network Intrusion Detection Systems . . . . .	3
2.1.1	Concepts . . . . .	3
2.1.2	Snort . . . . .	4
2.1.3	Bro . . . . .	5
2.2	Protocol Analysis . . . . .	7
2.2.1	TCP/IP Reference Model . . . . .	7
2.2.2	Internet Layer . . . . .	8
2.2.3	Transport Layer . . . . .	9
2.2.4	Application Layer . . . . .	11
2.2.5	Tunneled Connections . . . . .	12
2.3	Related Work . . . . .	13
<b>III</b>	<b>Design of an Architecture for Dynamic Protocol Analysis</b>	
3.1	Requirements . . . . .	15
3.2	Concepts . . . . .	16
3.3	Application Layer Switch . . . . .	20
3.4	Analyzer Structure . . . . .	21
3.5	Application Layer Switch Analyzer . . . . .	24
3.5.1	Responsibilities . . . . .	24
3.5.2	Detection Methods . . . . .	25
3.5.3	Decision Process . . . . .	26
3.6	Design Weaknesses . . . . .	28
<b>IV</b>	<b>Implementation</b>	
4.1	Protocol Analysis in Bro . . . . .	29
4.2	Application Layer Switch . . . . .	32
4.3	Analyzer Base Classes . . . . .	36
4.3.1	Protocol Analyzer . . . . .	36
4.3.2	Auxiliary Analyzer . . . . .	39

4.4	Application Layer Switch Analyzer . . . . .	40
4.4.1	Tasks . . . . .	40
4.4.2	State Machine . . . . .	41
4.4.3	Tunneled Connections . . . . .	46
4.4.4	Reassembler Analyzer . . . . .	47

## **V Adaption of the IRC Analyzer**

5.1	IRC Protocol . . . . .	49
5.2	IRC Analyzer . . . . .	52
5.2.1	IRC Analysis . . . . .	52
5.2.2	Configuration . . . . .	53
5.2.3	Signatures . . . . .	54
5.3	Auxiliary Analyzers . . . . .	57
5.3.1	ContentLine Analyzer . . . . .	57
5.3.2	ZIP Analyzer . . . . .	58
5.4	Evaluation . . . . .	59
5.4.1	Protocol Detection Tests . . . . .	59
5.4.2	Performance Tests . . . . .	61

## **VI Conclusion**

6.1	Summary . . . . .	67
6.2	Future Work . . . . .	68

## **Bibliography**

# Chapter I

## Introduction

We begin our thesis with the motivation of network security and techniques of Intrusion Detection. This puts across why the need for the proposed enhancements to Intrusion Detection is of great interest. After that, we explain the structure of this work.

### 1.1 Motivation

Network Intrusion Detection Systems (NIDSs) are the most used mechanism to detect attacks and analyze their proceeding and purpose. They are controlled and maintained at a central point and are not visible to the surrounding network. They monitor the whole data exchange between the internal and the external network and scan the data stream for significant attack patterns. Typical NIDSs use a signature matching unit to find previously defined attack patterns, usually given as regular expressions, in a network conversation.

More sophisticated systems search for malicious behaviour in network connections through the technique of protocol analyzing. This requires the system to decode protocols. This approach offers two different kinds of attack detection. On the one hand the NIDS may detect violations in a protocol session and on the other it is possible to perform signature matching leveraging the results of the protocol analysis.

But such a NIDS has to fulfill one condition to perform application layer analysis: it must classify data by their protocol. Therefore, a method to identify those protocols is required. Current systems make this decision by ports. But this is not as reliable as it should be. More and more protocols either do not use fixed ports, e.g. some peer-to-peer protocols, or use unprivileged ports, e.g. IRC, which might also utilized by other protocols like e.g. FTP-DATA connections. Thus, a connection might be analyzed by an inappropriate module or not at all.

In this work we propose a design of an architecture for NIDSs that may integrate far more powerful methods of protocol detection than relying on fixed ports and makes the analysis process dynamic especially for application protocols. This improves the rate of correctly analyzed connections and reduces the amount of inappropriate analyses concurrently.

Another purpose of dynamicalizing analysis is that it is a fundamental requirement to analyze tunneled connections. These connections have a deviating protocol stack and therefore are more complex to analyze. Some tunnel types even can not be classified by a static protocol detection method at all. The consequence is that encapsulated protocols might not be analyzed. This fact could lead to undetected attacks and thus needs a redesign of the analyzing subsystem.

We present a system independent architecture for dynamic protocol analysis and protocol detection first. For that, we take a closer look at requirements from which we derive the needed components. We point out the responsibilities of each module in a generic way. After that, we transfer the approach to the open source NIDS Bro and show the details of our prototypical implementation. To reach significant results we have adapted the existing analysis module for the Internet Relay Chat (IRC) protocol. Based on this prototype, we have examined both the costs and the use of the design.

## 1.2 Outline

This work is structured as follows. Chapter 2 explains the meaning of Intrusion Detection and different techniques in more detail. We focus on Snort and Bro, two widely used open source systems, and their kind of attack detection and protocol analysis. At the end of the chapter there is an overview about related work. Chapter 3 shows the design of the new developed architecture for NIDSs in its details. It contains the requirements to the design as well as their derivation into single components. In the following chapter the design is applied to Bro and gives more information about the implementation-specific demands. Chapter 5 covers the adjustment of the IRC analyzing module. We explain some needful information about the protocol. Through that knowledge it is possible to understand the used patterns for signature matching. At the end of this chapter we show the results for both the identification tests and performance tests. Finally we give a summary and an outlook to additional improvements and future work.

# Chapter II

## Background

First we explain the generic concepts of Intrusion Detection. Our focus are real-time Network Intrusion Detection Systems (NIDSs) that handle live traffic and report irregularities instantly. We take a closer look at two open source systems: Snort and Bro. Both of them perform a kind of protocol analysis. The background information about protocols and their hierarchies are illustrated in the TCP/IP Reference Model in the following paragraph. Afterwards, the way of layer-specific protocol analysis can be found. Finally we complete this chapter through making a digression to a related work - the l7-filter project - and its solution.

### 2.1 Network Intrusion Detection Systems

In this work we focus on connection-oriented NIDSs which raise to an important tool in network surveillance. Such a NIDS listens passively to the network traffic mirrored from a central network link. Receiving the raw packet stream, the NIDS reconstructs the byte stream that both endpoints exchange. Therefore, the system has to assign the data contained in a single packet to the related connection.

The responsibilities of the NIDS are laid down in the policy for network security. This policy must be transferred to a corresponding configuration for the NIDS. The configuration depends on the system's implementation and its approach to detect malicious activity. So we give a summary of different techniques of Intrusion Detection.

#### 2.1.1 Concepts

There are three ways of detecting malicious traffic. The first is an anomaly-based approach. In this case, the behaviour of message transfer is compared to a previously defined profile. The profile is created through a study about normal behaviour in a network. It may include e.g. time, user, file, host, network and

protocol specific statistics. If a deviation to the profile occurs, the NIDS interprets this as a policy violation. An example might be the unexpected increase of accesses to a special file.

The second approach tries to detect misuses through a set of attack patterns. Usual NIDSs own a powerful signature matching engine for this purpose. This allows to define attack patterns through expressions. During operation these systems search the content of each connection for all given expressions. This could be e.g. the string 'etc/passwd' to appear in network traffic.

The last method of Intrusion Detection represents protocol analysis. This means that protocol-specific parts of the conversation are analyzed by verifying syntax and semantics of the protocol. It is necessary e.g. to allow 'etc/passwd' to appear on websites but to detect its appearance within the Uniform Resource Locator (URL) of the Hypertext Transfer Protocol (HTTP).

The advantages of protocol analysis are to allow a more exact inspection and to have the potential of detecting unknown attacks. But on the other hand, this approach requires one fundamental element to become effective - a classification unit for application layer protocols. While actual systems use the connection's ports as criteria for protocol detection, we introduce a design that supports more sophisticated detection methods.

Our target is the enhancement of protocol analysis. Therefore, the next two paragraphs give an overview about two widely used systems with the ability of protocol analysis.

### **2.1.2 Snort**

A wide-spread Network Intrusion Detection System is Snort [4][5]. It is distributed under the GNU General Public License and available for most newer operation systems even Linux and Windows. Snort is maintained by a great community and is kept very up to date by the Sourcefire Vulnerability Research Team (VRT). Several new attack schemes are covered within a few hours.

Snort is a real-time NIDS that uses an advanced signature matching unit. It comes with an extensive set of attack patterns given as regular expressions. These are matched against the data stream. The configuration is done by specifying a list of rules. A rule contains a rule header and a rule option. The rule header defines conditions to the connections and the type of action. The rule option lays down a list of parameters to the rule. A sample Snort rule is:

```
alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|";  
                                     msg: "mountd access");
```

Figure 2.1: A sample Snort Rule (adopted from [6])

The given rule header limits the rule only to be applied to TCP connections with a destination host in the net 192.168.1.0 with the subnet mask 255.255.255.0 and destination port 111 (Remote Procedure Call). For all matching connections the signature matching unit searches the given byte code in the data part of all TCP segments belonging to this conversation. In the case of a match, the system fires an alarm. Additionally to the common informations (source, destination,...) a message is appended to the notification.

Since version 1.5 Snort provides a preprocessor technology to execute protocol specific analysis. A preprocessor mangles data before the signature matching unit performs the pattern search. The task of a preprocessor is to perform protocol analysis for certain protocols and to normalize the content so that Snort's signature engine interprets the data correctly. E.g. for HTTP there is a preprocessor called `http_decode`. It translates an obscure character set, like unicode or hex, into characters that are used in Snort's signatures. It is possible to encode the Uniform Resource Identifier (URI) with different character sets so that the signature matching engine does not react although there is a matching pattern. The method of hiding information to the NIDS is called evasion.

Preprocessors are statically configured to listen to certain destination ports. The `http_decode` e.g. may work on 80/TCP and 8080/TCP. For HTTP connections on other ports the preprocessor is not used. An attack is able to exploit this fact to evade a deeper analysis.

### 2.1.3 Bro

A far more complex and powerful real-time NIDS is Bro [7][8]. It was developed by Vern Paxson at the Lawrence Berkeley National Laboratory (LBNL) and International Computer Science Institute (ICSI), Berkeley, CA. It is written in C++ and is an open source project as well. Bro has already build up a community of researchers world wide including a small group at our research unit.

Bro provides an event-based approach. Figure 2.2 illustrates the structure of the Bro system. The security policies are transferred into policy scripts that are written in Bro's own powerful scripting language. These are user-definable and can easily be integrated into the system. There are also a lot of predefined scripts to support the user. A policy script controls the Event Engine and determines all necessary events that are implemented. Another configuration that is done by

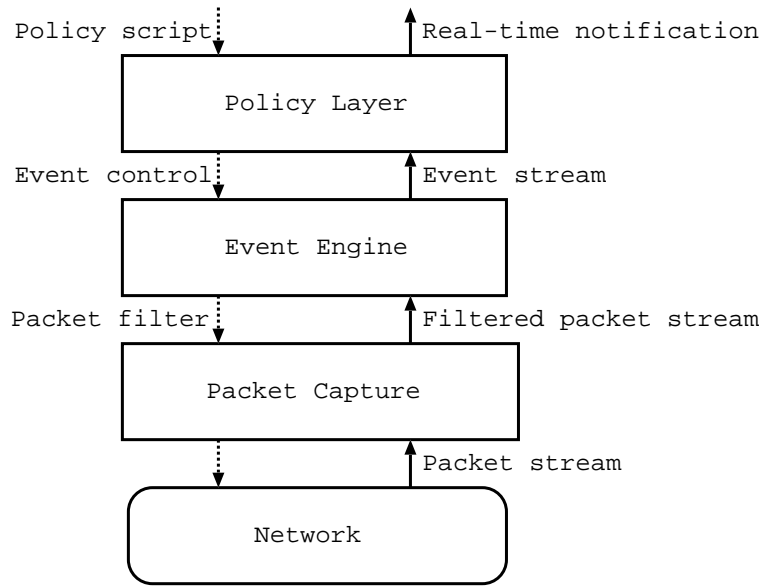


Figure 2.2: Design of Bro

policy scripts is the construction of a network filter to limit the traffic to those connections that should be analyzed. This filter is given to the Packet Capture subsystem. It receives the mirrored packets and filters them. Then the Event Engine abstracts the filtered packet stream to high-level events like e.g. a request message of a certain protocol. Processing an event and performing an action is done at the Policy Layer.

Bro provides two analyzing mechanisms in parallel. It comes with a high performance signature matching engine to scan traffic for malicious content equally to Snort. There exist some definitions for known attacks and worm detection. A match results in a fired event which performs the action defined in a policy script. The main feature making Bro so powerful is its sophisticated protocol analyzing technique. It offers a rich set of protocol analyzers which process the application data stream of a certain protocol and throw events on the basis of the content. An analyzer is responsible for detecting irregularities in a protocol session having the potential to detect even unknown attacks. Those two methods trigger the action when something unusual is found. Like in Snort the analysis for a protocol depends on the destination port of a connection. This is a security hole as the analysis can be evaded by using non-standard ports.

As protocol analysis is obviously an important aspect of Intrusion Detection we go into more detail of protocol decoding and analysis in the following paragraph.

## 2.2 Protocol Analysis

To understand the problem of what must be done within the analysis we need a definition of the term protocol and therefore the basics of communication. After that, the demands of an analysis are explained individually depending on the protocol layer.

### 2.2.1 TCP/IP Reference Model

The mostly known reference model for communication is called the OSI Model. According to the fact that we analyze traffic based on the Internet Protocol (IP) we focus the TCP/IP Reference Model. It defines only four layers instead of seven in the OSI Model. Figure 2.3 presents the structure of the communication architecture.

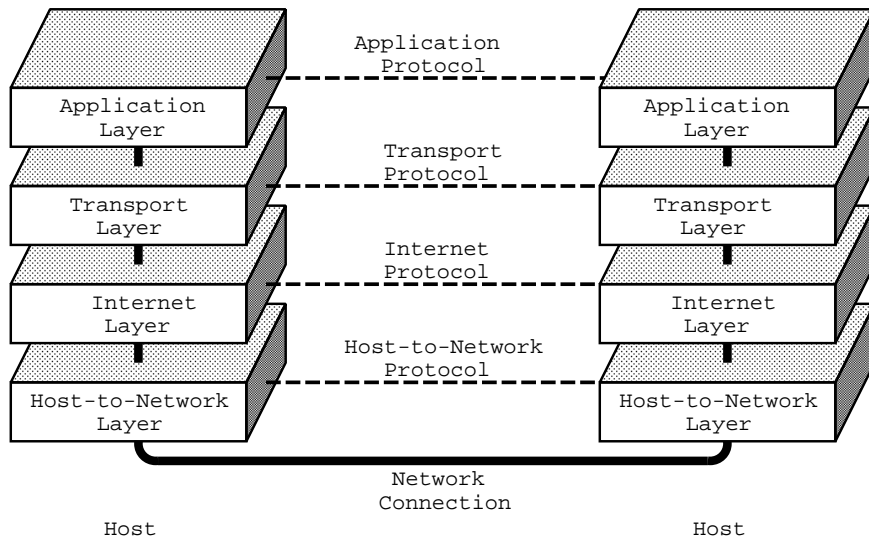


Figure 2.3: The TCP/IP Reference Model

This figure shows the communication path between two applications on different hosts connected through a network link. We differentiate between the vertical and the horizontal communication. For the vertical communication each layer defines an interface called the Service Access Point (SAP). This allows the interaction of neighbouring layers. On the other hand the TCP/IP Model requires a virtual communication between the peer entities on each layer. In this case the peers talk to the other one indirectly. The syntax and semantics of message exchange between those peers is defined by a protocol.

Each layer has its own tasks and thus its own requirements to the data. As every protocol needs additional protocol-specific data the raw message is encapsulated in a way the peer entity can understand. This information is packed into a header and connected with the message from the upper layer. On the opposite, side the message is unwrapped step by step. So at first our interest is directed towards the protocol headers and finally to the application-specific data.

Inspecting the TCP/IP Model we will see that the Host-to-Network Layer is not fixed to specific protocols or network architectures. The only condition is providing to send IP messages across the network. This becomes clear when taking a look at the heterogeneous environment in the Internet. For generality we start the analysis of Internet traffic on internet layer.

### 2.2.2 Internet Layer

The TCP/IP Model knows only one protocol on that layer - the Internet Protocol. IP builds the interface to send data in unreliable datagrams from one host to the other. For this reason IP has to meet the following demands:

- Unique addressing of every communicating instance:  
Every device on internet layer possesses a unique address in the entire network. It is called the IP address and is 32 bits long in version 4 and 128 bits in version 6 of the IP protocol.
- Routing mechanisms:  
As the Internet is constructed as a highly meshed net of transit systems and hosts, internet layer devices must support techniques to route datagrams through the network. The path selection is done at every system performing a lookup in the routing table.
- Fragmentation and Reassembly:  
The Maximum Transfer Unit (MTU) of a network architecture limits the size of a datagram. This means that data that is too long to send it in a single IP packet must be splitted into more packets. These are called fragments. On the other side the receiver has to reassemble all fragments before it delivers the whole message to the transport layer.

The analysis of IP datagrams confines itself to the inspection of the header fields shown in figure 2.4. The most important one is the Version field because the header is structured differently in version 6 of the IP protocol. During the IP analysis we have to determine the next layer protocol. This is laid down in the Protocol field.

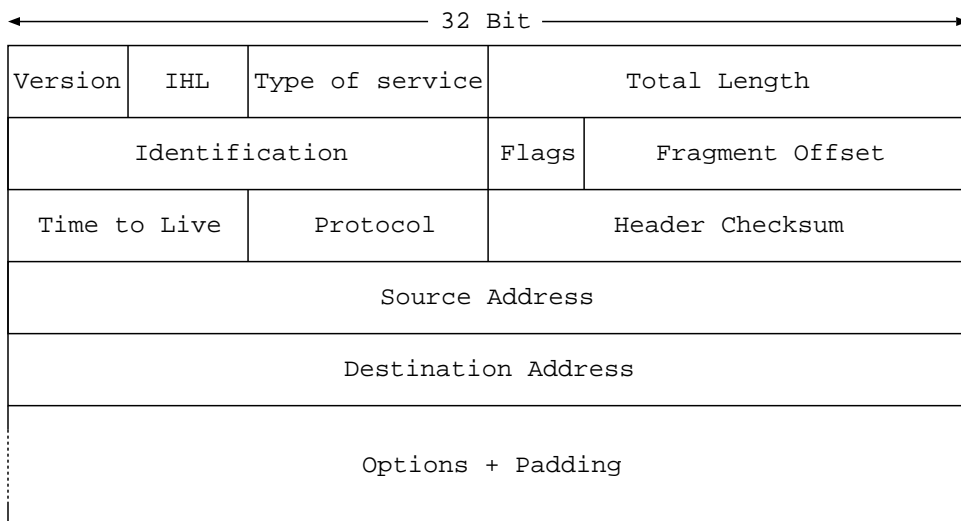


Figure 2.4: The IPv4 Header

### 2.2.3 Transport Layer

The transport layer establishes a virtual connection between the processes of the receiver and the sender. The TCP/IP Model defines two transport protocols named Transport Control Protocol (TCP) and User Datagram Protocol (UDP). Both target different types of communications. UDP offers a minimal and unreliable datagram-oriented transport service for applications. In opposition TCP provides a reliable and connection-oriented stream exchange. In this work we focus on TCP connections which are used when it is important that the application stream arrives completely and in-order at the receiver's side. A typical example for that is file transfer. If TCP messages, also known as segments, have been lost or corrupted during transfer, the file might be useless to the receiver. TCP uses following mechanisms to achieve this:

- Reassembly:  
TCP numbers each byte of data consecutively through a sequence number. The receiver acknowledges receipt of data by sending an acknowledgment number. As IP datagrams need not to arrive in the right order at the receiving side, TCP reorders contained segments due to its sequence numbers and compose data before delivery to application layer. The second purpose of reassembly is to discard duplicates.
- Retransmission:  
If packets get lost during transfer, the sender obtains no acknowledgment from the receiver causing it to retransmit the missing segments.
- Checksum:  
TCP uses a checksum to ensure that segments are transferred correctly.

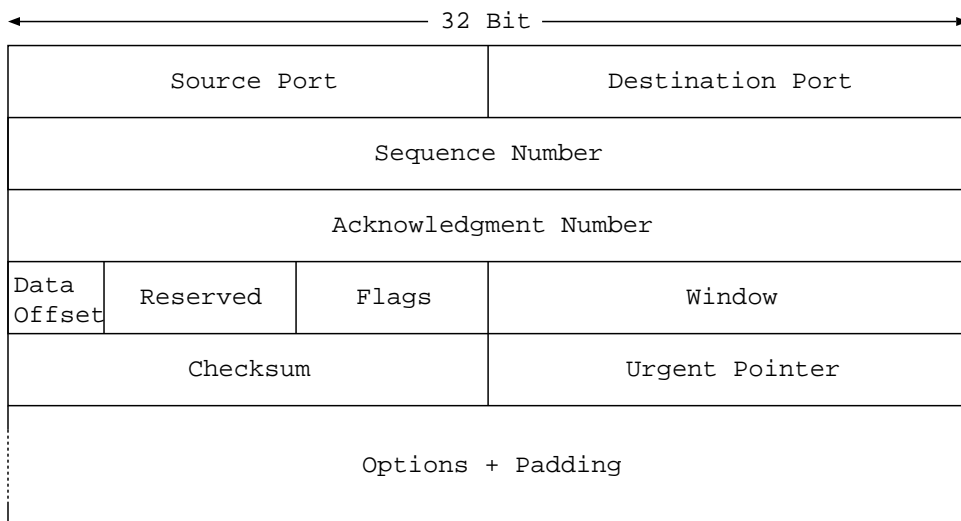


Figure 2.5: The TCP Header

Whereas analysis of UDP is restricted to verify the checksum, analysis of TCP is more complex. The first task is to reconstruct the application data stream. This means we have to discard duplicates and perform the reordering process. The needed informations like sequence numbers and acknowledgment numbers are stored in the TCP header like illustrated in figure 2.5. Another fact that complicates the analysis is the state management due to the connection-oriented approach. The consequence is that we have to inspect the flags in the TCP header, like a SYN - indicating a new connection. But there is a lot of misuse in this field. Therefore we must verify the state management through rebuilding the TCP state machine in the NIDS.

Together with the IP addresses, port numbers build the key to associate following segments to the existing connection. The problem is now to identify the correct application layer protocol. Port numbers specify the communicating processes on both hosts. Whereas the internet and transport layer are part of the operation system and thus are strictly defined, the processes are started by users. This fact allows a free choice of application layer protocols.

For the purpose of standardization and easier usage of unknown servers, there is a set of privileged or well known ports (0-1023) for which there exist an assignment of port to application protocols. This association is controlled by the Internet Assigned Numbers Authority (IANA) [15]. Processes running on these ports usually have to be executed by the system or privileged users. Whereas the application layer protocol detection by port might be successful for many protocols using well-known ports like HTTP, the detection may fail when protocols like IRC use unprivileged ports for which there are only recommendations. Then

we have to rely on widely used ports for specific protocols. The result is that connections are either analyzed by an inappropriate module or not analyzed at all. If the port identification succeeds we perform the analysis on the byte stream of the conversation.

## 2.2.4 Application Layer

Analyzing an application layer protocol differs enormously from one to the other. As there is no common header on this layer, the analysis must be individually adapted to the protocol. Application layer protocols are classified into two main groups - protocols using plain text to transfer messages and protocols that exchange binary encoded messages.

Most of the application layer protocols used in the Internet are plain text protocols. The advantage is the normed data representation and the easier readability to humans. The main part of them transfer their messages as a sequence of commands that causes the correct processing on the other side. Examples are HTTP, IRC or FTP. They have the following structure:

Command#1	EOL	Command#2	EOL	...
-----------	-----	-----------	-----	-----

The sequence of commands, eventually with a list of parameters, are separated by a line break (EOL). It is easy to divide the stream into single commands through searching for the next EOL. The representation of the data is defined e.g. to use the ASCII coding. Analyzing this protocol means now to verify the correct syntax of the data just like the correct semantics to the connection.

Binary protocols transfer a list of objects of a predefined data structure. Examples for that type are the Remote Procedure Call (RPC) or Napster protocol. The generic structure of binary messages looks as follows:

Object#1	Object#2	Object#3	...
----------	----------	----------	-----

These protocols have to handle the problem of different data representation on different architectures like e.g. a different word length or big endian versus little endian representation. To avoid this, a special network format must be defined to synchronize interpretation. The advantage is that binary protocols are usually more restrictive concerning the syntax than plain text ones.

The last point we have to look at is analyzing non-linear protocol stacks as used in tunneled connections.

## 2.2.5 Tunneled Connections

Tunneled connections build an exception to the reference model as they may use multiple instances of layers. Tunneling means the encapsulation of one protocol into an other. There are different purposes to use tunneling:

- Combination of networks through a different network architecture or constructing a virtual network over the existing network infrastructure like Virtual Private Networks (VPN) . Examples for that are L2TP [16] or IP-in-IP [17].
- Insertion of an additional encryption layer. Examples are the TLS [18] or the IPSec [19] Protocol.
- Evading the firewall filter through using a different application protocol to deliver data behind the firewall. Examples are filesharing protocols like Kazaa tunneled in HTTP.

The generic structure for a tunneled connection needs two edges that perform the encapsulation and decapsulation. These are called edge routers. The abstract communication path looks then as follows:

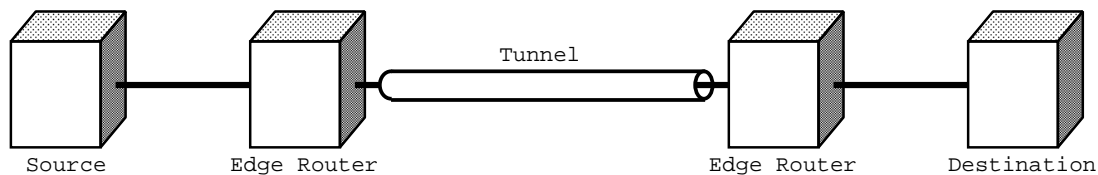


Figure 2.6: Communication Structure for Tunneled Connections

Tunneling results in a change in the protocol stack. Figure 2.7 shows three alternatives. The left protocol stack models IP-in-IP tunneling. It is used to connect two IP networks that are not able to communicate otherwise. An enhancement is made by Cisco Systems and is named Generic Routing Encapsulation (GRE) [20]. The second example shows a stack with an additional security layer. TLS is based on the development of Secure Socket Layer (SSL) from Netscape [21]. It adds encryption to the transfer and is used in connections containing sensible data. The last one is an example for tunneling of a filesharing protocol in a HTTP connection. This is a solution to evade a restrictive firewall as HTTP traffic is usually allowed.

What now should be clear is that performing protocol analysis is a dynamic process because at the beginning of a connection a decision of the right protocol stack is not always possible. A solution to this problem is presented in the following chapters through combining the problem of dynamicalizing the analysis and the problem of protocol identification.

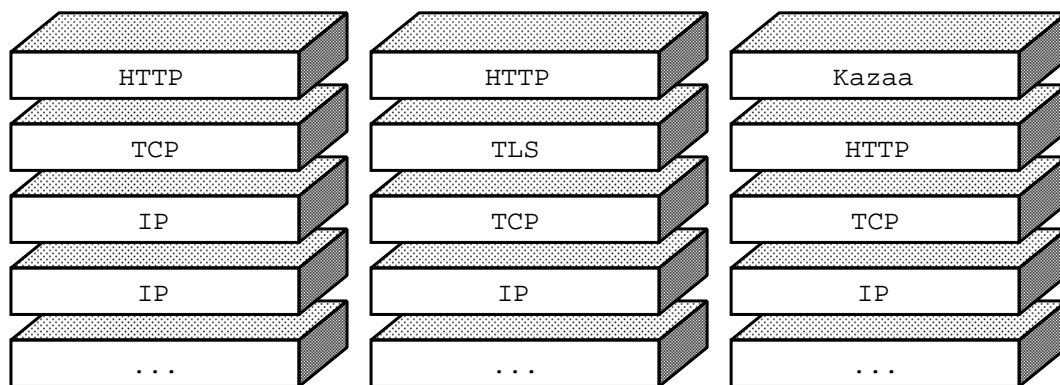


Figure 2.7: Sample Protocol Stacks in Tunneled Connections

## 2.3 Related Work

During the development of this work we have found no other NIDS that tries to do application protocol classification in a different way than port-driven. But there is a similar project for firewalls. The iptables firewall for the Netfilter framework under Linux deals with the same problem of fixed port to protocol matching. So a new project was brought to life which is called L7-filter [22]. It is an extension to iptables which tries to classify traffic on application layer.

The development for this project has already reached the status stable but note that the developers themselves highly recommend not to use it as filtering technique. The intention of this package is to control bandwidth for certain protocols. Bandwidth restriction is a wide-spread technique in ATM networks and is known as traffic shaping.

The underlying decision subsystem is based on a signature matching engine. L7-filter uses a single state in a Deterministic Finite Automaton (DFA) per connection. This means that both requests and replies are matched in the order as they come across. To reach a decision, the first eight packets of a connection or the first two kilobytes of content, whichever applies first, are matched against the protocol signatures.

The signatures consist of regular expressions. There are already a lot of protocols covered by patterns. A list of them can be found at [23]. For example the current pattern to detect the HTTP protocol is defined in figure 2.8. The given pattern tries to identify the HTTP server reply containing the HTTP version (0.9, 1.0 or 1.1), the status reply number (100-599) and informations about the following content. The second part searches for POST methods of the HTTP

client. If no pattern matches within the decision process the protocol 'unknown' is chosen. It can be used analogously to every other protocol like 'http' or 'ftp'.

```
http/(0\9|1\0|1\1) [1-5] [0-9] [0-9] [\x09-\x0d -~]*  
(connection:|content-type:|content-length:)|  
post [\x09-\x0d -~]* http/[01]\.[019]
```

Figure 2.8: Signature for HTTP Connections

It should be clear that this technique is too unreliable to use it as filter. One point is that signatures are not sure up to 100 percent or are so lazy that there are too many false positives. Another reason relies on the kind of implementation. Due to performance, the decision process is limited. Thus evading any signature is really simple through splitting the first command into more than eight packets. This has the consequence that a decision after eight packets is never possible.

We have completed the background and continue with the universal design of a dynamic analysis architecture for NIDSs.

# Chapter III

## Design of an Architecture for Dynamic Protocol Analysis

In this chapter we present an object-oriented design for a NIDS architecture that supports a dynamic construction of the analyzing process. Therefore we start to gather the requirements to this approach. After that, we illustrate the architecture in a short overview to impart the basic knowledge how the architecture works. Finally we go into details of the different aspects and look at the weaknesses of the design.

### 3.1 Requirements

First we look at the requirements. This helps us to find out what we need and what has to be done. We separate the process of analysis into distinctive steps. These correspond to individual protocols. For this, we introduce the term analyzer. This should be a module that performs the analysis for a single protocol. We come to the detailed view of an analyzer later in this chapter.

- Protocol detection unit:  
The architecture has to support different approaches of protocol detection. This helps us to classify traffic depending on more than port numbers like e.g. signatures or statistical protocol detection. Depending on the detection method it might be useful to buffer data until the decision is made. Then the system is able to replay the data for analysis.
- Flexible customization of application layer analyzers:  
An analyzer on that layer must be configured by the criteria that lead to its activation. These are e.g. the ports it should listen to, patterns for the application layer protocol detection or custom decision functions.
- Dynamic activation and shutdown of analyzers:  
If the analysis process should be dynamic, it requires that analyzers may

be activated at any time while the connection is active. On the other hand, if the analysis turns out to be uninteresting or wrong active analyzers are shut down.

- **Performance:**  
The dynamic approach for NIDSs targets analyses in high-speed networks. Thus the performance of the system should be as efficient as possible. Of course we expect the cost of a dynamic approach to be higher than the static equivalent. Therefore we want to support a static protocol detection by port furthermore. In this case we demand that the performance of the system is equal to comparable systems.
- **Parallel analyses:**  
Due to the unreliability of the protocol detection unit it might be possible that multiple analyzers are forced to analyze the same data. For this reason our design should allow parallel analyses.
- **Analyzing tunneled connections:**  
As we demand a dynamic approach it should be designed to support analyses of tunnels as well.
- **Standardized interfaces for analyzers:**  
The construction of the complete analysis process should be dynamic. So we connect analyzers in series corresponding to the protocol stack. This requires an analyzer to process input data and to produce an output for the following analyzer. Common interfaces simplify this fact enormously.
- **Extendibility of analyzers:**  
We want analyzers to have a plug-in interface that allows the reuse of modules for common jobs. These modules mangle the data before the analysis of the analyzer.

Now, that we know what has to be realized, we are able to derive the architectural design from the requirements and introduce them in an example.

## 3.2 Concepts

First we have to understand the basic concept of the architecture. For this, we give an overview about the architecture based on a sample IRC connection.

One requirement is that the architecture should support parallel analyses. At first sight this does not make sense when we look at the structure of a protocol stack. But we integrate protocol detection methods that are not reliable up to 100 percent. This means we have to expect a number of undetected and false

detected connections. Let us consider the following example. We listen to an established connection and have an IRC analyzer as application layer analyzer. What should be done if the protocol detection unit tells us to instantiate a HTTP analyzer? This results in two assumptions. Either the IRC analyzer inspects a HTTP connection or the HTTP analyzer will process an IRC connection due to a false identification. Which way is correct we do not know. This argumentation leads to the structure of an analyzer tree to store the arrangement of analyzers for a single connection. If the same situation as in the mentioned example happens now we can duplicate the application layer stream and give it to both in the hope that one of them is correct. Another advantage of the analyzer tree is the extendibility to other analyzers than protocol analyzers. Now it is possible to add an analyzer which executes a statistical evaluation in parallel. This might not be so unusual for a NIDS.

Now we introduce the architecture and its components on the basis of an IRC connection. For this example we use a signature matching engine as additional detection method to identify application layer protocols. We will see other protocol detection mechanisms later in this chapter. We start our analysis on the internet layer again.

When the first IP packet of the IRC connection arrives at the NIDS we have to create a new object which represents this connection and stores the root of the analyzer tree - the IP analyzer. All following packets are delivered directly to this node. The IP analyzer performs the IP header analysis for each IP message. This results in the initial analyzer tree like illustrated in figure 3.1

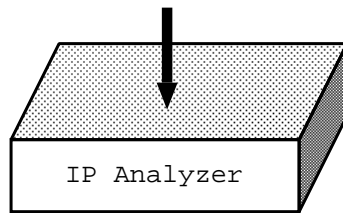


Figure 3.1: Initial Analyzer Tree

The IP analyzer gets the packets through the connection object after the NIDS has assigned a packet to this IRC connection. For each packet it extracts the header, verifies the fields and composes fragmented packets. Then it takes a look at the Protocol field in the header which specifies the transport layer protocol. As IRC is based on TCP, the IP analyzer creates a new TCP analyzer object and appends it at the analyzer tree. See figure 3.2. Then it delivers the data without the IP header to the TCP analyzer.

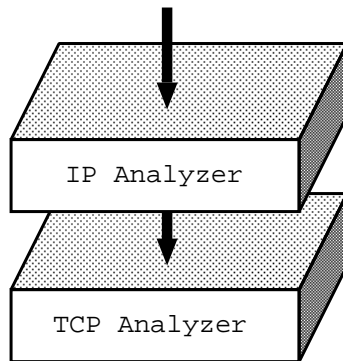


Figure 3.2: Appending the TCP Analyzer

The TCP analyzer inspects the header of all TCP segments. It reorders the segments due to their sequence number and discards duplicates. Now we face the problem of identifying the inner protocol. As decision by port is too unreliable we enhance detection with signatures. This job should be done by an analyzer called Application Layer Switch Analyzer (ALSA). It is adapted to the analyzer structure and is appended to the TCP analyzer. Its task is to create and stop application layer analyzers. We receive a configuration as shown in figure 3.3.

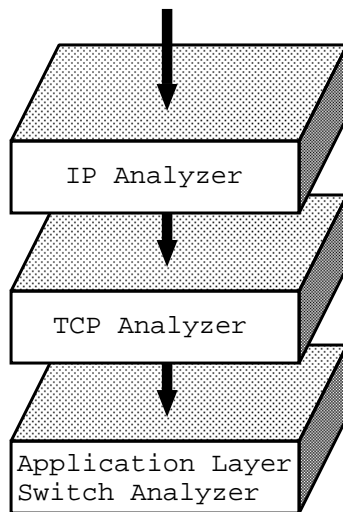


Figure 3.3: Appending the ALSA

This analyzer has to decide which application analyzer is to apply for this connection. For this, it needs its own configuration consisting of port configurations of application layer analyzers, patterns for the protocol detection and custom decision functions. As this is needed for all ALSAs in every connection, we set up a single container called Application Layer Switch (ALS) that holds the configurations of every analyzer. This is shown in figure 3.4.

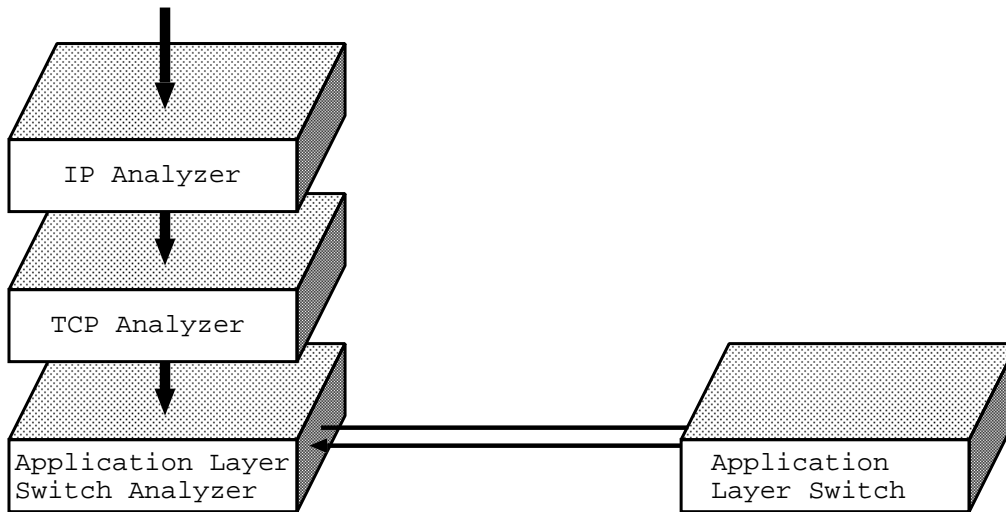


Figure 3.4: Retrieving the Criteria for the Decision Process

The ALSA is able to make a first decision now. Based on the received port configurations and custom decision functions, it might instantiate new application layer analyzers. But the result of protocol detection by pattern matching may lead to a classification in the middle of the connection. This is a problem as the analyzer gets only the part of conversation after the detection. This should be avoided through buffering data in the ALSA. This technique is needed whenever protocol detection is not able to make a decision based on the first packet. According to that fact the ALSA has to replay old data for every new analyzer. When at least one of our detection methods has succeeded, the ALSA instantiates an IRC analyzer. This is illustrated in figure 3.5. The ALSA remains in the analyzer tree until the connection is finished. It delivers incoming data after the signature matching to all its successors.

The IRC analyzer performs the IRC analysis on the byte stream of the connection. It has to verify the syntax as well as the semantics. All following packets have to get through the complete analyzer tree. As it is a dynamic analyzing structure it might be possible that analyzers are shut down or that the signature matching engine of the ALSA identifies another protocol. Then the ALSA has to instantiate a new analyzer object and append it as peer of the IRC analyzer. If the buffer still holds the complete elapsed conversation, then it replays it to the new analyzer as well.

We have traced the analysis process of a single connection and address the global ALS, the analyzer structure and the decentral ALSA now in more detail.

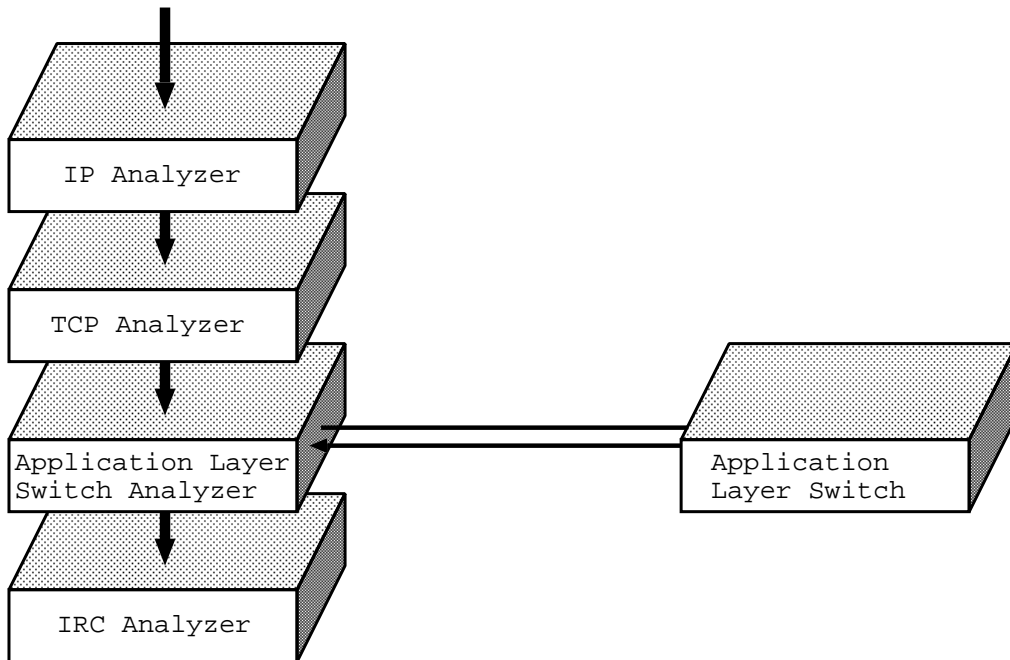


Figure 3.5: Appending the IRC analyzer

### 3.3 Application Layer Switch

What we need for the architecture to become dynamic is a global storage container which holds the criteria for the ALSAs to make their decision of the application layer protocol. So we collect all points that should influence the decision:

- Positive Port Set: Although the target is a high dynamical architecture we still recommend to use ports as a hint for the right protocol. When a protocol defines a positive port list like "HTTP" = [ 80/TCP, 8080/TCP ] every new connection with a destination port of those leads to the instantiation of a HTTP analyzer object.
- Negative Port Set: This set defines ports whose connections will not be analyzed by the given analyzer.
- Positive Detection Pattern: If a connection is found which matches a defined pattern (e.g. signature or statistical pattern) the corresponding analyzer has to be activated.
- Negative Detection Patterns: To avoid false positives it is possible to define negative patterns as well. These must not appear within a protocol conversation otherwise the corresponding analyzer is shut down.
- Positive Customized Function: This is a function of every analyzer which decides whether it should be started or not. A use case for this might

be either that we have a web server given by its IP address and we want this server to always be analyzed by the HTTP analyzer even when the detection methods fail or, on the other hand, that we want to overwrite the decision mechanism.

- **Negative Customized Function:** The same function with an opposed result. When this function returns true for a pair of connection and analyzer this connection will not be inspected by this analyzer anymore.
- **Accept Partial:** This is a field that separates application layer analyzers in two distinctive groups. All analyzers in this list accept partial connections too. These are connections whose beginning is not seen by the analyzer. This can happen e.g. at the NIDS's start time when there are already established connections or due to packet loss.
- **Prediction Table:** This table holds temporary informations about future connections. An use case is e.g. the announcement of a FTP-DATA connection while analyzing the control connection.

Note that the ALS itself is not an acting component in this architecture. It gets its configuration parameters from the analyzers at starting time. During the NIDS's operation only the prediction table may change its state.

We have completed the required criteria for the configuration of the ALSA. But before we describe the ALSA in detail we have to explain the structure of the analyzers.

### 3.4 Analyzer Structure

As mentioned the layers respectively their protocols should be analyzed one by one. For that purpose we need an analyzer that has an interface acting as input and an interface functioning as output channel. The analyzer processes the input stream and performs the layer-specific analysis. The analysis may result in an notification alerting any anomalous activity. The processed data is then delivered to the following analyzers in the tree.

The interfaces are common to all analyzers regardless to the layer where the analysis takes place. This might be evident for the IP and TCP analyzer but in the case of tunneled connections even an application layer analyzer may need an output interface. But reminding the message format between the layers we recognize that they have a different structure. The analysis on internet layer gets IP packets and produces defragmented TCP segments as output without the IP headers. So the TCP analyzer expects segments with sequence and acknowledgment numbers, otherwise it is not able to perform the stream reassembly. After

the TCP analysis all application layer analyzers expect to receive the reassembled byte stream. This fact requires the analyzer to have multiple input and output interfaces namely one for IP, one for TCP and one for the raw byte stream.

One of our requirements is that analyzers have to be extendible. Some analyzers may expect their data in a preformatted or normalized way. An example is a stream-to-line converter for line-based application layer analyzers. To reuse the splitting of byte stream into lines for protocols like Telnet, HTTP, FTP, IRC or SMTP, it is useful to write an external module that can be plugged in to the protocol analyzer. We call this an auxiliary analyzer because it needs the same interfaces, processes an input and produces an output analogously to an analyzer. As we will see later, there are even more analyzers that utilize this feature. Of course it is possible to plug in more than one auxiliary analyzer at the same time. Those analyzers are working in series. The first gets the input from the protocol analyzer and give its output to the next one. The last auxiliary analyzer hands the correctly formatted data back to the protocol analyzer which now is able to perform its analysis.

So taking a deeper look at the analyzer structure it looks like figure 3.6 illustrates. The analyzer has three input interfaces corresponding to the layers. Then the data is delivered either to existing auxiliary analyzers or to the analysis module immediately. If there are irregularities in the protocol conversation the analysis module may create a notification for the NIDS's alert subsystem. Depending on the severity of the incident the analysis is aborted at this point. If we continue the analysis or if there is no error at all, the analyzer produces the output for the following analyzers.

A last point which adds even more flexibility is giving an analyzer the potential to identify its own protocol. This means that an analyzer can recognize that the input data is not conform to the protocol that it understands. For this, the analyzer replies to each received piece of data with an ACK or NACK. When it assumes that it is not the correct protocol, it replies with a NACK causing the previous analyzer to shut it down. This feature reduces multiple, unnecessary analyses. Now we are able to explain the functionality of the ALSA in a broad view.

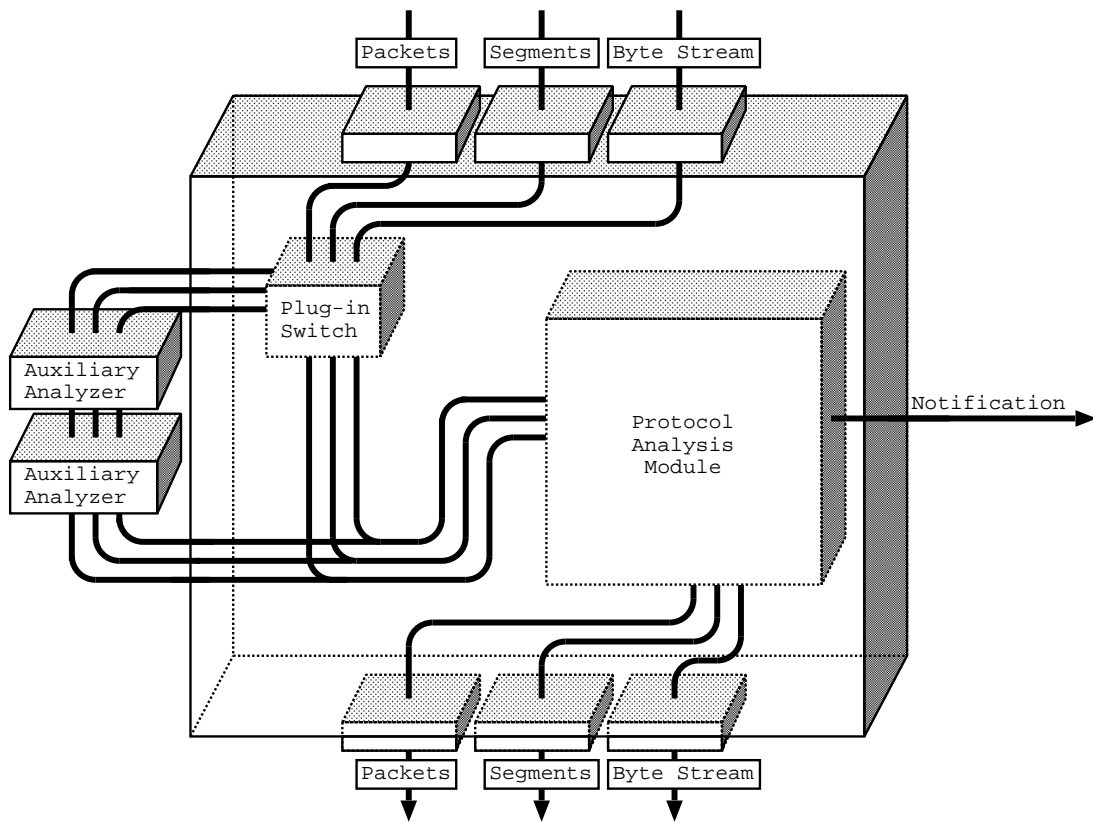


Figure 3.6: Structure of an Analyzer

## 3.5 Application Layer Switch Analyzer

For our design we have inserted an analyzer, called ALSA, into the analyzer tree after the TCP analysis. First we address the tasks and concepts of this analyzer in more detail. Then we give an overview about different protocol detection techniques that might be implemented in the ALSA. At the end we take a deeper look at the decision process and the precedences between the criteria.

### 3.5.1 Responsibilities

The ALSA of a single connection has to fulfill a number of jobs.

- Retrieve configuration from ALS:  
The first is to retrieve all informations that are stored in the ALS and that may match its connections. This means the ALSA does not need to know which ports are accepted by an analyzer but which analyzer wants to analyze the actual connection because of the destination port. So it demands the port, the detection pattern and the customized function configuration.
- Send incoming data to protocol detection unit:  
The ALSA must send a copy of the data to the protocol detection unit. It compares the data with defined patterns. In the case of a match the detection unit alarms the ALSA.
- Buffering:  
If the protocol detection unit is not able to decide on the basis of the first packet, buffering any elapsed data is useful. In the case that the detection alarms in the middle of the connection, the ALSA is able to replay the complete connection for a new analyzer.
- Compute initial analyzer subtree:  
The configuration received from the ALS enables the ALSA to compute an initial set of analyzers that should be activated for the application layer analysis.
- Observe the protocol detection unit:  
The ALSA has to observe alarms of the protocol detection unit until the connection is finished. Whereas a positive pattern alarm results in the instantiation of an analyzer and maybe a buffer replay, a negative one leads to the shutdown of the corresponding analyzer.

We have seen the tasks and continue with the expected message format of an ALSA. As illustrated in figure 3.7 we want the ALSA to receive segments as well. This has two advantages. In our implementation we have shifted the TCP reassembly into the ALSA to save performance through omitting the expensive

reassembly for unclassified connections. Another advantage is the idea to reuse the ALSA to detect the inner protocol of application layer tunnels. In this case the encapsulating protocol has to recognize that it is used for tunneling. Then it appends a second ALSA to itself to classify the inner protocol. As the ALSA receives byte stream in this case we can distinct the location of the ALSA due to the input format. We address these measures in the next chapter in more detail.

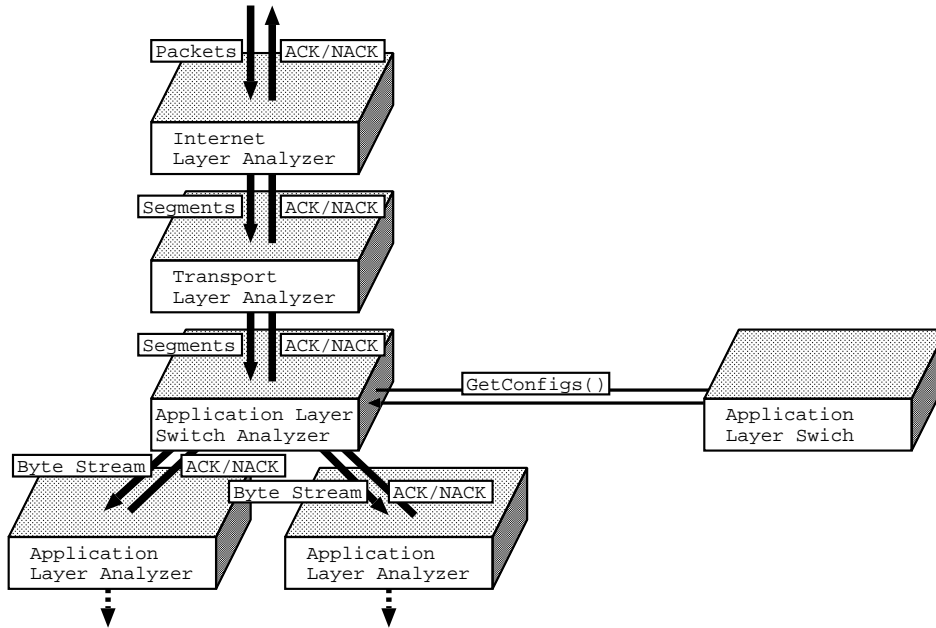


Figure 3.7: Application Layer Switch Analyzer

We have seen how the ALSA works and which criteria influence the protocol detection. In our example we have used a signature-based approach to identify a protocol. There exist other ones as well. These are discussed in the following paragraph.

### 3.5.2 Detection Methods

The detection mechanism is needed to identify the correct application layer protocol. Remind that the choice on the transport layer and beneath is unambiguous. As we assume that the application layer protocol detection is not reliable up to 100 percent, we have to expect a certain amount of false positives, meaning that an analyzer inspects another protocol, as false negatives, leading to omit an analysis although there is a corresponding analyzer. Nevertheless this approach will improve the rate of correctly analyzed connections.

There are three different techniques allowing us to identify protocols.

- Behavioural-based detection:  
Behavioural detection means a form of statistical identification. The foundation for this method is the creation of a reference profile. The probabilistic detection works on the probabilities of different protocols to specific users, hosts or servers. These values are laid down in the reference profile file. An enhancing approach is called the pure statistical detection which constructs its profile through Machine Learning [24] over a certain period of time. The newest variation is that the profile represents a neuronal network and tries to guess which protocol the next connection will use.
- Protocol-based detection:  
This approach bases on the idea of an intelligent analyzer that identifies the following protocol in the connection's protocol stack. This is done by verifying the regular structure of a protocol and comparing it to other protocol structures.
- Signature-based detection:  
This will be the most used detection mechanism as a NIDS often comes with a signature matching unit already. All protocols have to be defined through a set of patterns. These patterns are matched against the content of each connection.

An important fact is the difference in the point in time at which the detection method makes its decision. Whereas some behavioural-based approaches result in an immediate answer, all other solutions may alarm anytime within the connection. All those approaches should be used in combination with the mentioned concept of buffering.

What is missing yet is the deeper look at the decision process of the ALSA. We come to this in the following paragraph.

### **3.5.3 Decision Process**

We have seen a number of criteria that should lead to a decision. Now we have to declare a precedence between them.

We define the following sets:

$A$	:=	Set of all analyzers
$P^+$	:=	$\{x x \in A$ and the connection's destination port is in the positive port set of analyzer $x\}$
$P^-$	:=	$\{x x \in A$ and the connection's destination port is in the negative port set of analyzer $x\}$
$D^+$	:=	$\{x x \in A$ and the detection unit has identified the connection's protocol to be for analyzer $x\}$
$D^-$	:=	$\{x x \in A$ and the detection unit has identified the connection's protocol not to be for analyzer $x\}$
$C^+$	:=	$\{x x \in A$ and the positive customized function returned true for analyzer $x\}$
$C^-$	:=	$\{x x \in A$ and the negative customized function returned true for analyzer $x\}$
$AP$	:=	$\{x x \in A$ and the analyzer $x$ accepts partial connections }
$PT$	:=	$\{x x \in A$ and $x$ is in the the prediction table and should accept this connection

Then we compute the list of analyzers  $L_{complete}$  at the beginning of a complete connection as follows:

$$L_{complete} = (P^+ \cup D^+) - (P^- \cup D^-) \cup (C^+ \cup PT) - C^-$$

When applying to partial connections meaning that the beginning of the conversation has not been seen, we compute the initial  $L_{partial}$  in this way:

$$L_{partial} = L_{complete} - (A - AP)$$

$L_{complete}$  respectively  $L_{partial}$  build a list of those analyzers which should be activated when the connection is established.

Depending on the detection mechanism there are only two sets that may change during the process of analyzing. These are  $D^+$  and  $D^-$ . Whereas a new analyzer in  $D^-$  results in stopping this one, a change in  $D^+$  requires a new computation. If at least one piece of data is already lost it is a partial connection for all future analyzers. So we distinct three cases with  $x$  being the new analyzer:

Case #1: We are analyzing a complete connection and all elapsed data can be restored through buffering:

$$L'_{complete} = L_{complete} \cup (x - P^- - D^- - C^-)$$

Case #2: We are analyzing a complete connection and there is data lost yet. This happens when there is no buffer at all or the buffering is limited:

$$L_{partial} = L_{complete} \cup (x \cap AP - P^- - D^- - C^-)$$

Case #3: We are analyzing a partial connection:

$$L'_{partial} = L_{partial} \cup (x \cap AP - P^- - D^- - C^-)$$

So defining any negative characteristics should only be used to reduce false positives but note that this will cause not to load an analyzer for the rest of the connection.

## 3.6 Design Weaknesses

Although we have developed a flexible and powerful design of a NIDS architecture we have to face some exploitable weaknesses. For our point of view, this means that an attacker is able to prevent the detection of the used protocol. By doing this, hiding an attack may be possible.

Assuming that the analyzer is working correctly and therefore is able to detect this attack at all, preventing the protocol classification may occur due to two reasons. The first is exploiting holes in the detection pattern which is an implementational weakness. Although it is not easy to find high-quality patterns which identify as many connections as possible without having a high false negative rate, this represents the only solution. The other is a weakness of the design when integrating a detection method that alarms anytime within the connection. Then we have to buffer elapsed data to restore the complete connection for the new analyzer. But this represents a problem.

Unlimited buffering is not feasible due to need of performance and memory. So we have to limit the buffer size. Then we have to distinct two versions of protocols assuming that they are detectable by a certain characteristic.

If the protocol requires a conversation to start with this characteristic having a fixed length, we are able to identify this protocol by patterns as long as the buffer can hold the maximum size of the characteristic. These connections are no problem for the current design. Otherwise we have either to shorten the pattern or to face the second case.

The other case is that the pattern announces a matching anytime within the connection. As we have restricted the buffer size, this may lead to the fact that the beginning part of the connection can not be completely restored by the buffer. This missing part at the beginning of a connection may prevent an analyzer to perform its analysis. But this problem can not be solved with a limited buffer size.

The next chapter shows the details of our prototypical implementation for the NIDS Bro.

# Chapter IV

## Implementation

We have explained the design of an architecture for dynamic protocol analysis. In this chapter we transfer this approach to Bro. We begin this chapter with a summary of protocol analysis in Bro. This points up the functioning of the system and where we start the analysis according to the new approach. After this, we explain the implementation of the ALSA and the analyzers. Then we look at the realization of the ALSA and a way of using the ALSA to analyze tunneled connections. At the end of this chapter we present the reassembler analyzer which performs the stream reassembly for TCP connections.

### 4.1 Protocol Analysis in Bro

As it is a prototypical implementation we have not realized the complete design. We restricted the implementation to analyze complete TCP connections only and have omitted the extraction of IP and TCP analysis. This means we start the dynamic analysis with the ALSA as root node in the analyzer tree. We have tested the new approach with IRC connections. For all other protocols we have let the process of analysis untouched.

We have accomplished the overview of the basic architecture of Bro with separate layers for policies, events and capturing. Now we follow the way of a packet through the analysis process.

The main problem that real-time NIDSs have to face is lack of performance due to the expensive TCP reassembly and application layer protocol analysis. Therefore, all systems provide a network filter to limit the incoming traffic. Bro is reading the traffic by using the libpcap library [25]. Libpcap allows the extension with the BSD Packet Filter (BPF) [26] to limit traffic within the kernel. The filter is automatically constructed by the policy scripts or can be manually overridden. When a packet passes the filter Bro begins to analyze the IP header.

## IP Analysis

Bro starts its analysis on internet layer and begins with the conversion of raw bytes into IP datagrams. The first part of the IP analysis is to verify the content of the header fields through performing a checksum computation and comparing the result with the header's checksum. As seen in the background chapter IP packets may be fragmented caused by a small MTU. Thus the second job is to defragment a transport layer message that is splitted into several IP packets. Unless this message is not fully restored we abort the analysis at this point.

Beyond this step we determine the transport layer protocol. This is a clear decision as the values in the protocol field are laid down in [27]. When the value of this field equals `\x06` this represents a TCP connection.

From now on we distinguish different sessions of connections through the 4-tuple of IP addresses and ports. Bro defines a class hierarchy to represent different connections. This is illustrated in the following figure:

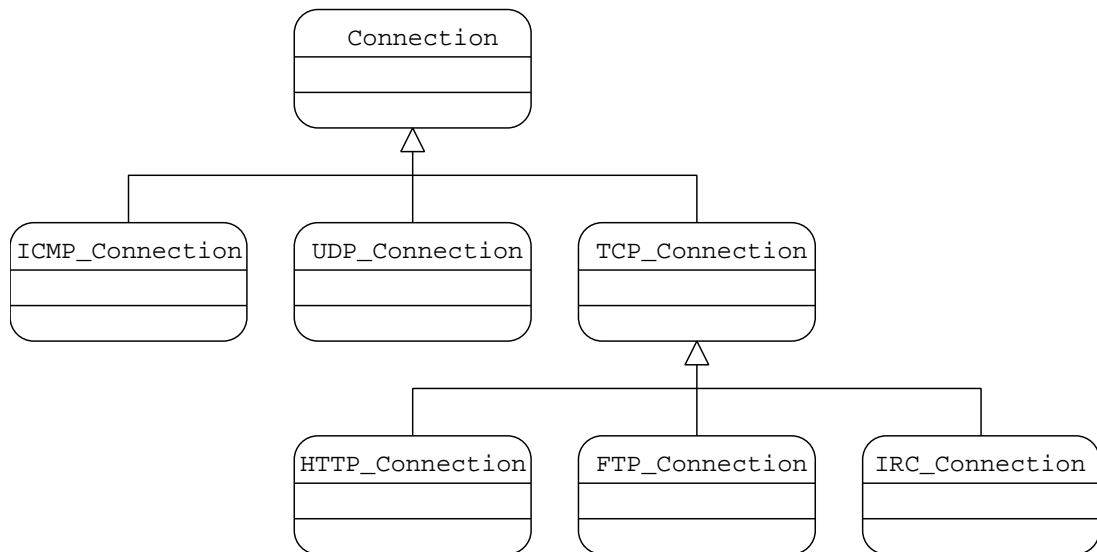


Figure 4.1: Connection's Class Hierarchy

Bro uses a static protocol detection method by port numbers. So it is able to instantiate the connection's object during the IP analysis because the system has already access to the ports. In the case that a packet does not belong to an existing connection, we create a connection corresponding to the destination port. E.g. we instantiate a `HTTP_Connection` if the destination port is 80/TCP. If there is no port entry for the used one, a simple `TCP_Connection` is set up. We omit the details of the connection classes and go further into the TCP analysis.

## TCP Analysis

The IP header is completely analyzed and the TCP segment is decapsulated. We continue the analysis of the segment now. Equally to IP even TCP uses a checksum to verify that the header is correct.

The main part of Bro's TCP analysis is the state management of TCP connections. This means we have to inspect the flags set in the header. For this, Bro uses two objects of the class `TCP_Endpoint` functioning as representatives for the sending and receiving TCP instance. Both hold their own TCP state like e.g. `TCP_SYN_SENT`, `TCP_ESTABLISHED` or `TCP_CLOSED`. Based on the previous state and the flags set in the current segment, Bro examines the acceptance and updates the state in the endpoints. If the connection's type is a `TCP_Connection` only we abort the analysis here, otherwise we shift the further processing of the segment into the endpoint that acts as sender of this segment. A part of TCP that is still missing is the stream reassembly. Bro tries to enhance performance through performing the reassembly only for classified connections. So we rank the reassembly of Bro among the application layer analysis.

## Application Layer Analysis

The endpoint delivers the TCP segment to a structure called `ContentProcessor`. This is an object that combines the work of reassembling and preformatting of the byte stream. Connections are reassembled either when it is a known protocol and is classified through the destination port or when the ports are configured to be reassembled on Policy Layer. Therefore, there are two port sets called `tcp_reassembler_ports_orig` and `tcp_reassembler_ports_resp` defined in the initial policy script `bro.init`. When the port matches to one of those in the sets, the connection is reassembled as well. The second job of the `ContentProcessor` is to preformat data. There is e.g. a class called `TCP_ContentLine` that allows the splitting of data into lines for line-based protocols. It is appended at the reassembler and manipulates the reassembled byte stream before it is delivered to the analyzer for the application stream.

We restrict the explanation for the analysis of the application data to the common aspects. The job of the analyzer is to abstract the byte stream to events. So the analyzer classifies the messages into different types. Each type corresponds to an event for which a handler might be implemented in a policy script. A resulting action due to a message is triggered by the execution of each implementation of this event in every policy script.

It is obvious that Bro follows a centralized concept of analyzing. After the creation of a connection object it controls and triggers all following work. This is a fast approach but has one important disadvantage. The architecture does not allow any dynamical changes in the analyzing process.

Our goal is to abolish this deficiency. We have to change the connection concept of Bro to a more general form, to be able to change protocol analyzers within the connection. Thus it is necessary to extract all protocol-specific information out of the connection. After we have accomplished that, we can implement the new analyzer structure in Bro. Such an analyzer has to meet the demands mentioned in the previous chapter, must possess all necessary information of its protocol and has to work as autonomously as possible.

As our implementation in Bro is a prototype, only the IRC analyzer is adapted to the new analyzer design. In order to leave Bro as powerful as before, we do not touch the protocol analysis of other protocols. This means the prototype uses different types of analysis in parallel. The decision which method is applied depends on the port declarations. When a connection can be classified by a port number during the IP analysis the old technique is used. All unclassified connections benefit from the new architecture.

But first we have to consider where we start the dynamic analysis. We have decided to deliver the TCP segments to the root node and the TCP endpoints in parallel. Although we deliver the segments not via the endpoints to the analyzer, the endpoints are needed further on. They contain the information about sequence and acknowledgment numbers. These values are referenced by the re-assembling module. The functioning of this module is summarized in an auxiliary analyzer now as well as the TCP\_ContentLine module. As these become independently, the TCP\_ContentLine class is unnecessary for the new approach. In addition, the class concept for connections becomes needless too. We want to classify the traffic dynamically meaning that there are only objects of TCP\_Connection anymore. But as long as not all analyzer are adjusted to the architecture the hybrid implementation has to persist. The next paragraph explains the implementation of the ALS.

## 4.2 Application Layer Switch

Reminding the concepts of the ALS there are two important jobs that have to be realized. The first is that the ALS has to import the configuration of the analyzers and to convert this into an internal structure afterwards. This is done once at Bro's start time. The second task is to build up the functionality so that the connection's ALSA can easily inquire the necessary information.

Bro activates analyzers only when there are events defined for that analyzer. Taking IRC as example again this means if there is no `irc_request` or `irc_response` event on Policy Layer, we do not perform an IRC analysis. This is considered in the process of collecting analyzer configurations.

As the analyzer configuration should be user-definable on script layer we have implemented a data structure holding each configuration in the initializing script of Bro called `bro.init`. We have defined a table where every analyzer stores its configuration. This table is defined as follows:

```
const tcp_als_cfg: table[string] of prot_cfg = {} &redef;
```

This table is indexed by the name of the analyzer, e.g. “irc” or “http”. The corresponding value is of a new type called `prot_cfg` which is the data structure to save the configuration of a single analyzer. The attribute `redef` sets the table to be redefinable by the analyzer’s scripts. This means if Bro should load e.g. the IRC analyzer, the IRC script `irc.bro` registers its configuration in this table with index “irc”. The instance of `prot_cfg` holds the definition for the configuration variables. This type is defined as follows:

```
type prot_cfg: record {  
    pos_ports: set[port] &optional;  
    neg_ports: set[port] &optional;  
    partial: bool;  
};
```

The port sets are not necessary, so they are defined as optional. All ports in `pos_ports` define these destination ports that lead to the instantiation of an IRC analyzer while the negative ones prevent this. The `partial` variable must be declared by the analyzer. If set to `True`, this analyzer is able to analyze partial connections meaning it may start its analysis in the middle of a connection too.

Note that the signature patterns are not part of the ALS’s configuration. In Bro these are imported to the signature engine directly. So the analyzer adds its signature file in its script like shown in the following example:

```
redef signature_files += "patterns.sig";
```

Another feature is still missing namely the customized functions. These are defined in the C++ core of Bro only and not on script layer. Therefore we inspect these in the process of collecting all information then.

Now the ALS has to get the information from script layer and builds up the internal structure. First we address the port configurations again. If an ALSA wants to know if there is an analyzer that accepts a certain destination port, it

is useful to index the port configurations by the port number. The consequence is that the ALS needs two data structures - one for the positive port set and one for the negative one - which are indexed by port and result in a list of analyzers. A very good and fast data structure for that purpose are hash maps. For the value field we use a 32 bit unsigned integer. As there are only two states possible for an analyzer, we use a binary representation of analyzer IDs. For this, we associate every protocol analyzer with one bit in the integer value representing its ID (1, 2, 4, 8,...). A sample construction of the hash map for the positive port configuration can be seen in figure 4.2.

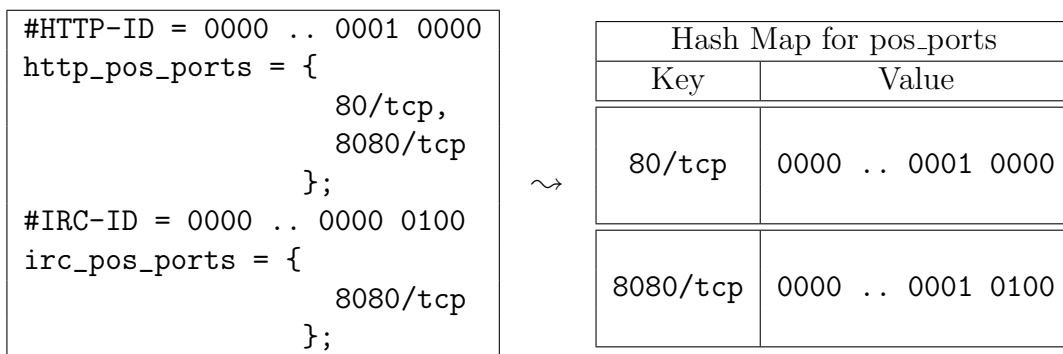


Figure 4.2: Construction of the Positive Port Hash Map

The next thing is to get a list of analyzers that accept partial connections too. This is stored in another 32 bit value in the binary representation. If one bit is set to true the corresponding analyzer is able to analyze partial connections. As the `accept_partial` value is initialized with 0 only those are interesting where the partial field on script layer is True. For them, their ID is added to the `accept_partial` value.

The last requirement to the ALS are the customized functions for every analyzer. We have decided to use the feature of C++ to define function pointers. We have arranged them in a list of pointers. If such a pointer is not 0, respectively there is a customized function for that analyzer, it is appended at the list of function pointers. So we can summarize the process of importing the analyzer configurations and storing them in the internal structure of the ALS like illustrated in figure 4.3.

As mentioned the second task of the ALS is to provide interfaces for the ALSA to retrieve its configuration. This is explained in the context of the ALSA later in this chapter. Previously, we have to look at the class concept for the analyzers.

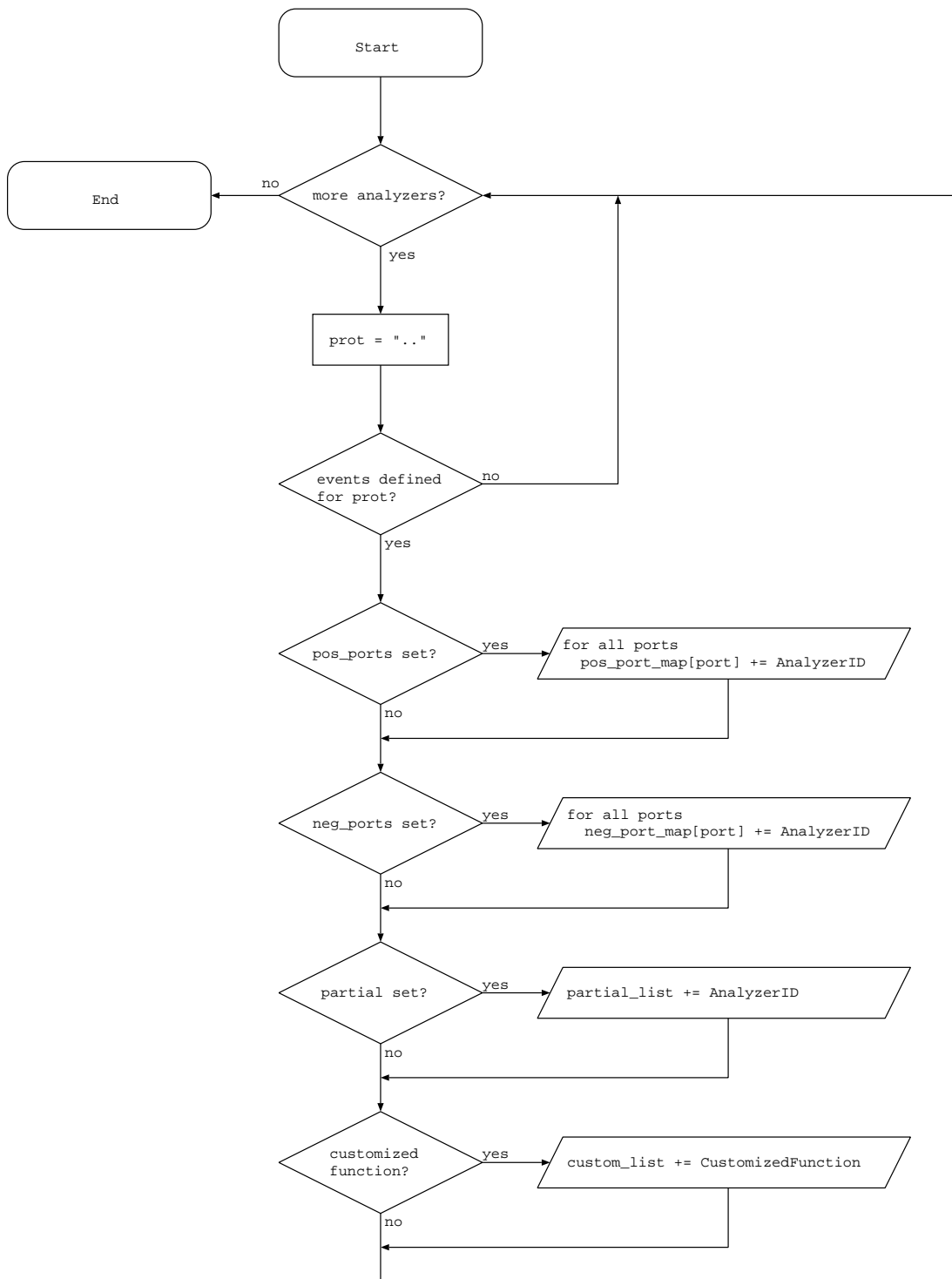


Figure 4.3: Building the ALS's Internal Structure

## 4.3 Analyzer Base Classes

We continue with the implementation of the analyzer structure. We adapted the structure to Bro-specific needs and present the workflow in the aimed analyzer tree. The second part shows the class definition of auxiliary analyzers.

### 4.3.1 Protocol Analyzer

We use the object-oriented feature of inheritance and therefore have defined a base class for all protocol analyzers which is called `TCP_ALS_Analyzer`. This class provides the basic interfaces that analyzers may utilize. Figure 4.4 shows the structure illustrated in the Unified Modeling Language (UML).

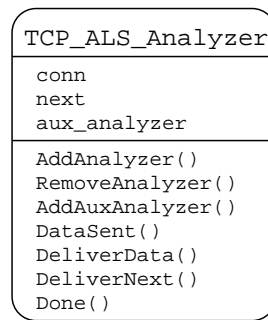


Figure 4.4: UML Class Declaration for `TCP_ALS_Analyzer`

Opposed to the original version of Bro, connections are no more distinguished by their application layer protocol. The class `Connection` inherits its attributes to three subclasses called `TCP_Connection`, `UDP_Connection` and `ICMP_Connection`. A further partitioning makes no sense for a dynamical approach. The needed attributes of application-specific connections must now be transferred into the analyzer. But of course there are common informations e.g. the source address that are stored in the `Connection`. So every analyzer possesses a pointer to the `TCP_Connection` object.

The attribute named `next` is typed as a map and stores pointers to all analyzers building the child level in the tree. We use the 32 bit binary representation again. A sample snapshot of the map is presented in the following figure:

Analyzer ID	Pointer to Analyzer Object
0000 0000 ... 0000 0001	0x12345678
0000 0000 ... 0000 0100	0x12345690

Figure 4.5: Snapshot of `next`

Assuming that the IRC analyzer has the ID 1 and the HTTP analyzer the ID 4, this map contains the information that there are two succeeding analyzers in the analyzer tree namely an IRC and a HTTP analyzer. For delivery of the data we have access to both analyzers by the stored pointers.

The attribute `aux_analyzer` is a pointer to a `TCP_AUX_Analyzer` object. This class is the base class of all auxiliary analyzers. We make up the explanation of this class at the end of this paragraph. If there is no auxiliary analyzer in use, this pointer is 0, otherwise the pointer targets the first auxiliary analyzer through which the incoming data should be piped.

Now we come to the methods of an analyzer. The first one is called `AddAnalyzer()` and is already implemented in the base class. It adds one or more analyzers to the map of successors. The necessary argument is another unsigned integer. For every bit set in this integer, the corresponding analyzer will be initialized. This means, to start several analyzers at the same time, execute this function with the sum of their IDs as argument. Additionally, this function verifies that applied to partial connections, only analyzers are added that support this feature. The opposed function is `RemoveAnalyzer()` and removes every analyzer whose bit is set in the parameter. To add an auxiliary analyzer to the protocol analyzer, there is an `AddAuxAnalyzer()` function. The function plugs the already instantiated auxiliary analyzer onto the analyzer through adapting the pointer `aux_analyzer`. If there is another auxiliary analyzer yet, it is appended at the end of the list.

The most important function is `DataSent()`. It acts as interface where the analyzer receives its input from and performs its analysis. We have demanded multiple interfaces for input and output depending to the layer in the protocol stack. As we start our dynamical analysis on transport layer, we need two kinds of the `DataSent()` method, one for segments and one for the byte stream. Another enhancement for analyzers is that it verifies that it is the correct protocol it analyzes. So we have required that an analyzer replies to the incoming data with an ACK or NACK. For this, the `DataSent()` method returns a boolean value. In the case that it returns `True`, it agrees to receive more traffic. If it returns `False`, the analyzer will be shut off by the predecessor.

The `DataSent()` function is not thought to be for direct usage. There is a wrapper method named `DeliverData()` which tries to deliver the data to an existing auxiliary analyzer at first. After the data has been passed to the last auxiliary analyzer, this one hands the data over to the `DataSent()` interface. If there is no auxiliary analyzer at all, `DeliverData()` passes the data directly to `DataSent()`. Of course the return value of `DataSent()` must be handed back to the wrapper function building the return value for the predecessor.

The method to deliver the data to all successors and to shut off an analyzer returning a NACK is called `DeliverNext()`. It is already implemented in the base class too and should not be overridden in derived classes. This function distributes the output data to all following analyzers contained in next and if one of them returns `False`, it shuts down this analyzer.

`Done()` is a Bro-specific function. It is executed for the Connection before it is destroyed. It could be used for clearance. The connection calls this function for the root analyzer in the analyzer tree. The base class implementation of `Done()` causes the execution of all successor's `Done()` functions. The workflow for delivering the data from one analyzer to all successors looks then as follows:

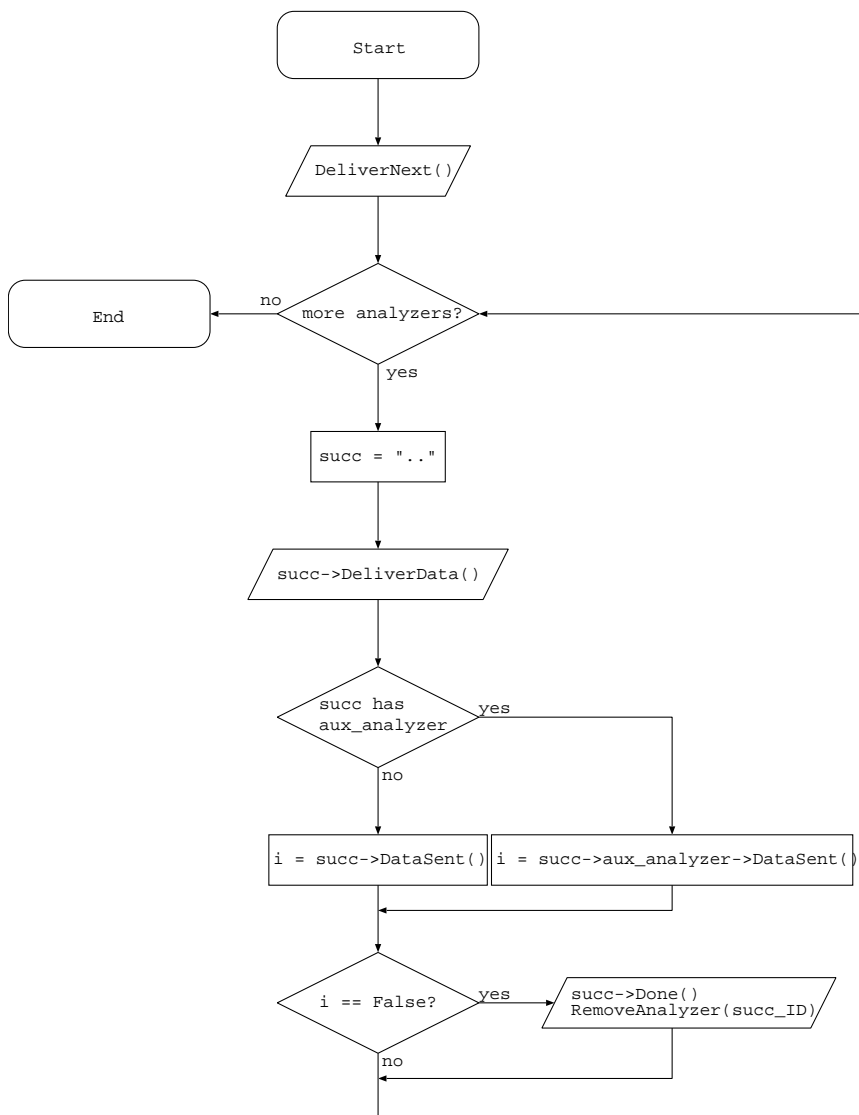


Figure 4.6: Process of Data Delivery through the Analyzer Tree

### 4.3.2 Auxiliary Analyzer

The second kind of analyzers are the auxiliary analyzers. At first we give the illustration as an UML diagram of the class called TCP\_AUX\_Analyzer.

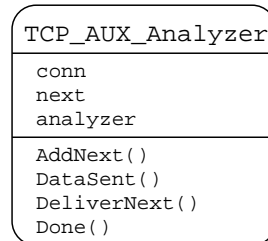


Figure 4.7: UML Class Declaration for TCP\_AUX\_Analyzer

An instance of this class has similar attributes as a protocol analyzers. We store pointers to the connection object, to the following auxiliary analyzer and to the protocol analyzer. The attribute next implements the linked list of auxiliary analyzers. If next equals 0, there is no successor.

To add an successor, there is a function called AddNext(). If no following auxiliary analyzer exists, it updates the aux\_analyzer attribute to the new object. If there has been an existing pointer before, this function call is handed over to the following object. Thus, the new object is appended at the end of the list.

An object of this class has to implement the DataSent() function as well. This is the interface where the input data arrives at the object. After processing the incoming data, the auxiliary analyzer sends the changed data via the DeliverNext() function to the next auxiliary analyzer or back to the protocol analyzer. Equivalently to the protocol analyzers, an auxiliary analyzer replies each piece of data with an ACK or a NACK. The reply is usually computed as follows. If there is an error while processing it returns a False. The result is passed backwards to the previous protocol analyzer which shuts down the analyzer, our auxiliary analyzer is plugged in. Otherwise it should return the value it receives from the successor.

The Done() function is called before destruction. It is initiated by the protocol analyzer and must be called for the next auxiliary analyzer. Of course an auxiliary analyzer is destructed when the corresponding protocol analyzer is destroyed.

This realization of the analyzer classes meets all demands we mentioned in the beginning of the last chapter. Now, we address the implementation of the ALSA.

## 4.4 Application Layer Switch Analyzer

We have completed the analyzer structure which is used for the ALSA as well. Now we continue with the implementational details of the ALSA. First this requires a transfer from the theoretical design to the implementational needs. Then we summarize the workflow in a state machine. After that we follow the idea to reuse the ALSA for application layer tunnels. At the end we make up a short explanation of the reassembling for TCP connections using the feature of auxiliary analyzers.

### 4.4.1 Tasks

As we start the dynamical approach after the TCP analysis, the ALSA builds the root node of our analyzer tree and expects segments at its input interface. Now we gather all jobs of the implemented ALSA:

- Retrieving Configuration of the ALS:  
During instantiation of the ALSA it has to retrieve its configuration from the ALS. For our implementation this concerns the port and customized function configurations.
- Protocol detection:  
We have gone to know three different types of protocol detection. For our implementation we have utilized the ability of Bro to perform signature matching. Therefore, we combine protocol detection by port, by signatures and by custom decision functions of the analyzers. So the ALSA has to send each incoming segment to the signature matching engine of Bro. The disadvantage of signatures is that connections are usually not identified instantly but during the connection. The consequence is that this requires either that analyzers handle partial connections or that we have to create a buffer which restores the connections for a new analyzer.
- Buffering:  
The motivation to use buffering should be clear. But buffering all connections requires a lot of memory and performance. So we decided to limit the buffer size to 4 KB. If the buffer size exceeds the 4 KB limit it is completely removed and no more data is saved.
- Dynamic creation and stopping of analyzer objects:  
The ALSA is able to compute the initial set of application layer analyzers through the static configuration received from the ALS. Based on the results of signature matching it has to instantiate new analyzers or shut down analyzers if a corresponding negative signature has been found.

- Reassembling:
 

The reassembling on transport layer is missing yet. We have already mentioned that the ALSA should expect segments that we can shift the reassembly process into the ALSA. This has the advantage that only classified connections must be reassembled. For this, we have adapted the existing reassembling module to the structure of an auxiliary analyzer. This is explained at the end of this chapter.

With these information we are able to set up a DFA that shows the different states that an ALSA needs.

### 4.4.2 State Machine

We have implemented five states to represent the ALSA. These can be differenced by the input format and by the state of the buffer. The following scheme shows the differences of each state:

		Buffering	No Buffering
<b>Input: Segments</b>	INIT	SIGMODE	ONLYMATCHMODE
<b>Input: Reassembled Byte Stream</b>		ANALYZERMODE	DELIVERMODE

Now we can build up the corresponding state machine of the ALSA for analyzing its connection:

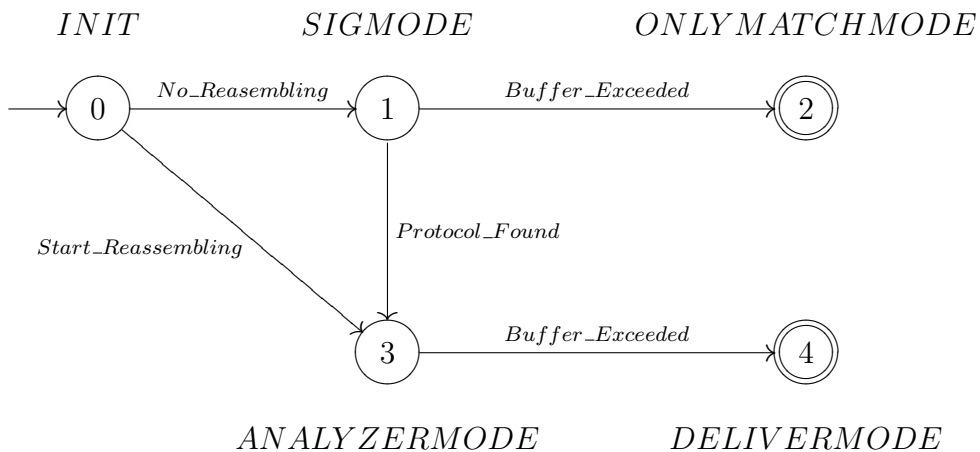


Figure 4.8: The ALSA's State Machine

At this point we take a deeper look at the single states.

## INIT

When the ALSA is instantiated its state is set to INIT and it retrieves the connection-specific analyzer configuration from the ALS. This is done in several steps.

First we retrieve the information concerning the connection's destination port. For this, the ALSA has two 32 bit values for the positive port configuration as well as for the negative one. The ALS offers a method to update both of them at the same time. It is called `GetPortConfig()` and needs as parameter the destination port of the actual connection and references to both port configuration values. Both values are updated then through a lookup of the ALS in the `pos_ports` and `neg_ports` hash maps for the port configurations with the current port as key.

The next is the assignment of the positive and negative results of the customized functions to two other variables. It is called `GetCustomConfig()` and also requires two references to these values. Then the ALS executes every implemented function. This updates both values with analyzer IDs announcing which analyzers want the connection and which not. At this point we have the static configuration.

As we work on TCP connection only, we have to remind that TCP performs a three way handshake before exchanging data. Although the ALSA never sees packets without content the SYN and the SYN/ACK messages are used to initialize the states of the signature matching automaton for both directions. This may already cause a signature to match before the first byte of content arrives. These patterns are independent to the content and match because of a combination of lower protocol (IP, TCP), port or header conditions. Keeping this in mind we have to evaluate the two values for positive and negative matches as well.

The computation of an initial set of analyzers is done when the first segment of data arrives at the ALSA. We store this in a variable called `initial_cfg` in binary representation. Like illustrated in Chapter 3 we compute:

```
initial_cfg = pos_port | pos_sig;  
initial_cfg = initial_cfg & ((neg_port | neg_sig) ^ 0xffffffff);  
initial_cfg = initial_cfg | pos_custom;  
initial_cfg = initial_cfg & (neg_custom ^ 0xffffffff);
```

Afterwards we have to decide whether to reassemble the connection or not. This fact is influenced by three aspects. As soon as there is an analyzer waiting for

this connection it must be reassembled. If `initial_cfg` is 0 meaning that there is no analyzer that wants that connection at this time, we test whether we should reassemble the connection anyway. This feature was taken over from the original version of Bro. The policy variables `tcp_reassembler_ports_orig` and `tcp_reassembler_ports_resp` consist of a defined set of ports whose connections should always be reassembled. Additionally to that we look at another configuration value for the ALSA that is named `reassemble_first_packets` which is also defined on Policy Layer. It is reserved for future use of a fast reassembly on the first packets until the buffer exceeds. If set at the moment, it causes to reassemble all connections. Based on these three conditions we either update the ALSA's state to SIGMODE or we instantiate the reassembler and maybe analyzers and change to ANALYZERMODE. The reassembler is implemented as an auxiliary analyzer. We explain its functionality at the end of this chapter. Now we send the first piece of data to the `DataSent()` interface again.

## **ANALYZERMODE**

In the ANALYZERMODE the ALSA expects application data. All data is reassembled before meaning we have a complete and correct application stream.

The work of this state contains matching, buffering, delivering and watching the buffer size. The first task is to match the current piece of stream. There exists a function called `MatchData()` for that purpose. This function hands the data over to Bro's signature matching engine. In the case of a match with a positive pattern we initialize the appropriate analyzer and replay elapsed data from the the buffer to it. When a negative pattern is found we shut off the corresponding analyzer.

To buffer data we have implemented a data structure called `TCP_Stream` containing the following fields:

- Size
- Data
- Direction

As long as the buffer does not exceed our limit of 4 KB, each incoming data is encapsulated in an object of `TCP_Stream` and appended to a linked list holding the elapsed part of the conversation.

Now we deliver the current data to all existing successors. This is done via the `DeliverNext()` function of analyzers.

The last work is watching the buffer size. For this we add the Sizes of all pieces of data. If this is lower than the 4 KB limit we abort here, otherwise we delete the whole list of TCP\_Stream objects, set the connection to partial and change to DELIVERMODE.

## **DELIVERMODE**

In this state all incoming data is only matched and delivered to succeeding analyzers. As our implementation is still a prototype partial connections are not supported yet. Thus, positive protocol patterns do not lead to the creation of new analyzer objects anymore. In opposition, matching negative patterns cause the analyzers to stop. This state is final and the ALSA remains in this state until the connection is finished.

## **SIGMODE**

The SIGMODE state is reached when the connection is not reassembled and thus not analyzed. In this state the ALSA expects unreassembled segments. This improves performance as we omit the expensive reassembling for all unclassified connections.

But buffering segments requires an extended data structure which saves following values:

- Sequence Number
- Size
- Data
- Direction

This structure is called TCP\_Packet and is derived from TCP\_Stream. The buffer for segments is equally structured to the buffer for application stream.

In this state the ALSA matches data and buffers it as long as the buffer is smaller than 4 KB. But the matching in this state is far more complex. Whereas negative pattern matchings are ignored due to the fact that there is no analyzer at all, a positive matching requires the instantiation of the reassembler and the analyzer first. Then we update the ALSA's state to ANALYZERMODE. But now we face the problem of buffering. As we have only segments in our buffer we have to send each TCP\_Packet object in the list back to the reassembler. The reassembler hands the data back to the ALSA in ANALYZERMODE. This means that the buffer is converted to a buffer of TCP\_Stream objects step by step. In

ANALYZERMODE the stream objects are then delivered to the new analyzer. Finally the current packet must be sent to the reassembler as well.

If no signatures have matched in SIGMODE, we buffer the current segment too. If the buffer is filled up we delete it and change to state ONLYMATCHMODE.

### ONLYMATCHMODE

In this final state the ALSA gives up to find a suitable analyzer. It only matches data immediately. We ignore positive matchings again but stop analyzers when we found its corresponding negative pattern in the content of the connection.

A requirement for our design was to be as fast as comparable systems when we use a static approach only. This means we have to keep Bro almost as fast as the original version when abstaining from the signatures-based protocol detection. So we have implemented a shortcut in the DFA of the ALSA.

Reminding the ALSA's configuration, we said that only the protocol detection unit may lead to a change in the analyzer tree in the middle of a connection. Then the buffering does not make sense anymore. So the ALSA is implemented in this way that if no protocol detection patterns are defined, we skip the states ANALYZERMODE and SIGMODE. The result is the smaller DFA as presented in the following figure:

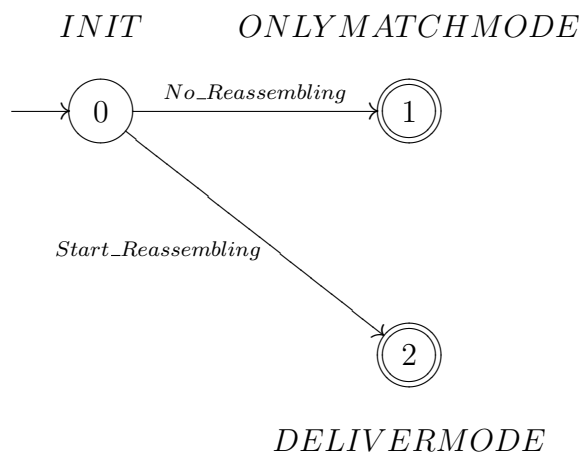


Figure 4.9: Static Detection of the ALSA

As we will see in the results we almost reached the performance of original version of Bro.

Of course we have to face the weakness of the design that limited buffering is exploitable. Although the technique could be enhanced using a sliding window technique, this is not implemented yet as it is still a prototype.

Another requirement for the design is to support the analysis of tunneled connections. This is discussed in the next paragraph.

### 4.4.3 Tunneled Connections

In the beginning we have learned that tunneled connections have abnormal protocol stacks. Thus, the analysis requires more intelligence in the analyzers. As our prototype does not have an autonomous IP or TCP analyzer, we focus on application layer protocols tunneled in another only.

It is obvious that the tunnel detection is the job of the encapsulating protocol analyzer. But this demand will be accomplishable. If an analyzer detects that its protocol is used to ship another protocol, there are two solutions possible. On the one hand, the analyzer may use an implementation of the protocol-based detection mechanism. This has the consequence that the outer analyzer knows the correct inner protocol and thus can initialize the following analyzer in the tree. The other case is when the analyzer detects that there this is a tunneled connection but does not know the inner protocol. Then the idea comes up to reuse the ALSA to detect the inner protocol.

Reminding the layer architecture we said the ALSA expects segments. But when the ALSA is appended to an application layer protocol analyzer it receives a byte stream. This aspect helps us to distinguish the location and purpose of the ALSA.

So we have added a new state called INIT2 that represents the initial state for detecting the encapsulated protocol of a tunnel. This results in the extended state machine shown in figure 4.10.

The state INIT2 has the job to initialize the ALSA to detect the inner protocol. So it has to retrieve the analyzer configuration from the ALS first. But a detection by port does not make sense anymore. Therefore we retrieve the customized function configuration only. A second necessary task is to initialize the state machine for signature matching. These two criteria may result in a new analyzer for the encapsulated protocol. Then we change to the ANALYZERMODE and resend the first piece of data to the DataSent() interface. Finally this and all following data is handled like before.

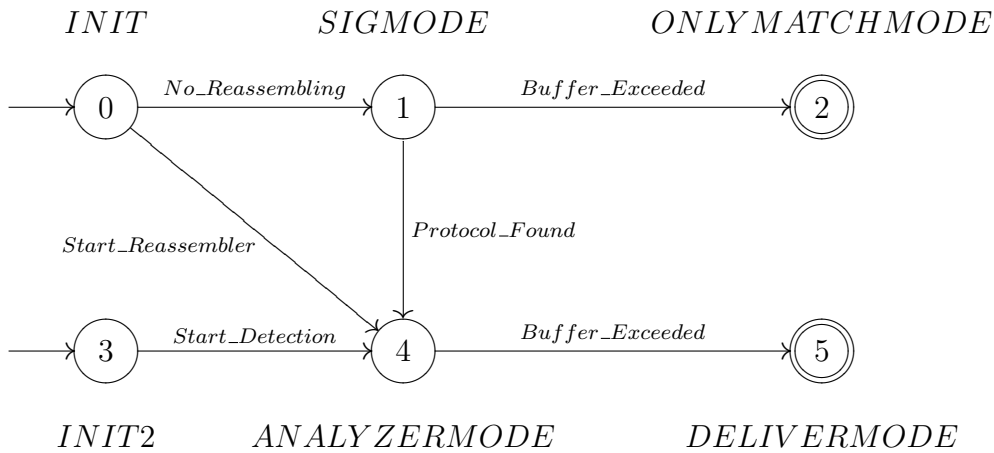


Figure 4.10: The complete ALSA's State Machine

This approach has already been implemented but is not tested yet due to the lack of suitable analyzers.

What is still missing is the overview about the already mentioned reassembler. Now we catch up on this aspect in the next section.

#### 4.4.4 Reassembler Analyzer

The reassembler is needed for TCP connections to reconstruct the complete and correct application stream. It is a part of Bro's ContentProcessor which has been duplicated and adapted to the structure of an auxiliary analyzer. The reassembler needs two implemented interfaces. The input interface receives segments and the output interface produces the reassembled byte stream.

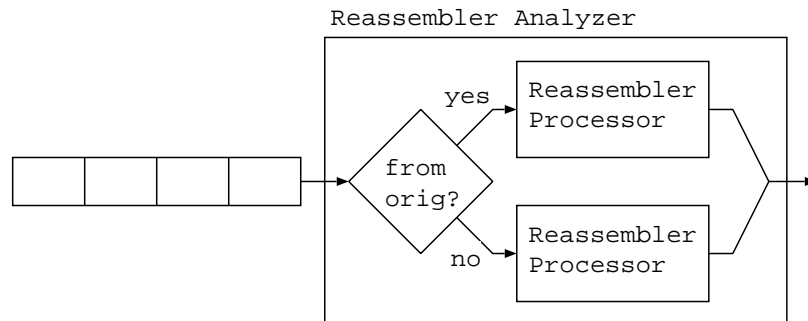


Figure 4.11: The Reassembler Analyzer

The reassembler analyzer consists of two reassembler processors corresponding to both directions. This is shown in figure 4.11. The analyzer distributes each incoming segment to the adequate processor. The output of the processors is pipelined through the analyzer and returned to the ALSA.

The reassembly is done by the processors with their separate receiving buffers. The buffer looks as follows:

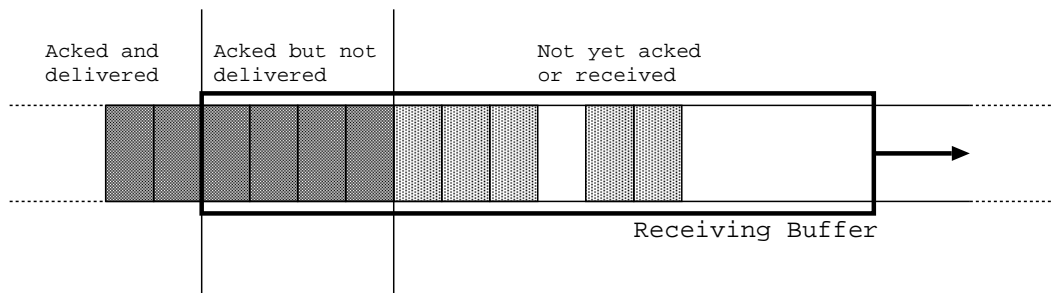


Figure 4.12: The Reassembler Buffer

The tasks of a reassembler processor is obvious now. An incoming segment must be inserted in the receiving buffer at the right place. This is done until the opposed endpoint acknowledges data up to a certain sequence number. At this time the processor can deliver all data up to this number to the reassembler analyzer which delivers data to the ALSA. As packets with an acknowledgment but without content never arrives at the ALSA, the acknowledgment is announced to the reassembler via the TCP\_Endpoints. All delivered data is finally removed within the reassembler buffer. There is a configuration variable of the reassembler processors called `stop_on_gap`. This handles the case of proceeding after a packet loss. If set, we abort the whole analysis for this connection at this point. It is now set to stop reassembling in this case. For our evaluation we unset this flag. Otherwise this had influenced the results enormously because of too many abortions.

We have completed the implementation of the architecture and address the adaption of the IRC analyzer now.

# Chapter V

## Adaption of the IRC Analyzer

To use the new features we have modified the existing IRC analyzer to be consistent with the new analyzer structure. We have chosen IRC because it is a protocol which does not use strict ports. Therefore IRC is a good example for the need of the new architecture.

First we take look at the protocol itself before we present analyzer. After that we show two auxiliary analyzers that support the IRC analyzer. At the end of this chapter we come to the results on real traffic and their evaluation.

### 5.1 IRC Protocol

The standard for the Internet Relay Chat Protocol was first published in 1993 in [28]. It defines a simple but powerful network protocol to provide near real-time conferencing and data exchange between clients. There exist lots of client applications to simplify usage to users. This has lead to a daily growing utilization of message exchange via IRC.

The protocol is based on the client-server model that makes it highly scalable and flexible. The IRC network is built as a spanning tree over the underlying network architecture. Clients connect to servers which are responsible that all messages arrive at its destination. For that IRC uses its own simple form of routing mechanism. As messages are transported over servers which lay on the direct path from one client to the other only, traffic is reduced enormously. Of course some messages must be broadcasted to all servers. This is e.g. the registration of a new user.

Another feature of the IRC protocol is to group members who talk about the same topic. These groupings are called channels. A channel is created when the first user joins it and deleted when the last one leaves. The first one who opens

a new channel gets the channel operator status. He has more rights than other users and is allowed e.g. to kick members out of the channel or to give others operator status too. The second form of control are irc-ops. These are administrators for the IRC network itself.

The most famous and public IRC networks are IRCnet [33] and QuakeNet [34]. IRCnet has about 100 000 registered users with over 60 000 channels. The infrastructure consists of 50 servers - one of them inside the TU München - and is maintained by 200 operators. In contrast, QuakeNet handles with about 150 000 users offering 180 000 channels. The network is distributed on 40 servers and 90 operators guarantee a smooth procession. For more facts about IRC networks, see [35].

IRC commands are transferred in plain text and follow an easy to understand structure. As IRC is a line-based protocol every command is written in a separate line. The commands can mainly be subdivided into three different types of messages:

- Client messages
- Server replies
- Server messages

Client messages are all messages sent from clients to their servers. The syntax for this type is as follows:

```
[:<prefix>] <command> [<parameters>]
```

That what is called prefix is the personal information about the user and its host. It is standardized to the following scheme:

```
<nick name>!<real name>@<hostname>
```

Most client messages result in a reply of the server with a certain reply code. This tells the client whether the command was successful or errors have occurred. These form the second type of messages - the server replies. They are declared similarly to client messages but instead of a command name it contains a reply code consisting of three digits. So the scheme for that kind of messages is:

```
:<prefix> <reply_code> [<parameters>]
```

The third part of messages are those from one server to another. If chat messages of a client appear at the server it must be delivered to the server the target is

logged in. These messages always contain a prefix because elsewhere the receiver would not know who sent this message. For these the command structure looks like this:

```
:<prefix> <command> [<parameters>]
```

Further informations about the IRC protocol and its commands can be found in [31] and [32]. We want to impart a basic understanding of the protocol only. So we continue with a sample client-to-server session which illustrates the procedure in IRC. It shows all messages from the client and its server. This example shows how to log in, join a channel, send a message and terminate the session to the IRC server. We have inserted several commentaries for explanation.

```
# client1 connects to server irc.server
host:~ # telnet irc.server 6667
Trying 192.168.0.99...
Connected to 192.168.0.99.
Escape character is '^]'.

# client registers on the server
USER client1 0 * :User1 Name
NICK client1

# server replies with welcome message
:irc.server 001 client1 :Welcome to the Internet Relay Network
                        client1!~client1@192.168.0.1
:irc.server 002 client1 :Your host is irc.server,
                        running version 2.10.3p3
:irc.server 003 client1 :This server was created Tue Sep 23 2003
                        at 17:17:58 UTC
:irc.server 004 client1 irc.server 2.10.3 ao0irw abeiIklmno0pqrstv
:irc.server 251 client1 :There are 2 users, 0 services on 1 server
:irc.server 254 client1 12 :channels formed
:irc.server 255 client1 :I have 2 users, 0 services and 0 servers
:irc.server 375 client1 :- irc.server Message of the Day -
:irc.server 372 client1 :- 23/9/2003 19:18
:irc.server 372 client1 :-
:irc.server 376 client1 :End of MOTD command.

# client wants to join #channel1
JOIN #channel1

# server message broadcasting that client1 has joined #channel1
:client1!~client1@192.168.0.1 JOIN :#channel1
```

```

# server replies with list of members in channel
:irc.server 353 client1 = #channel1 :client1 @client2
:irc.server 366 client1 #channel1 :End of NAMES list.

# client1 orders more information about client2
WHOIS client2

# server replies with information
:irc.server 311 client1 client2 ~client2 192.168.0.2 * :User2 Name
:irc.server 319 client1 client2 :@#channel1
:irc.server 317 client1 client2 126 :seconds idle
:irc.server 318 client1 client2 :End of WHOIS list.

# client1 sends a message to client2
PRIVMSG client2 :Hello client2!

# client2 answers
:client2!~client2@192.168.0.2 PRIVMSG client1 :Hello client1!

# client1 leaves channel
PART #channel1
:client1!~client1@192.168.0.1 PART #channel1 :client1

# client terminates connection to server
QUIT
ERROR :Closing Link: client1[~client1@192.168.0.1] (I Quit)
Connection closed by foreign host.
host:~ #

```

## 5.2 IRC Analyzer

A first version of the IRC analyzer was developed in an earlier project at our research unit [36]. The IRC analyzer was now adapted to the new analyzer design. We give a short overview of the analysis without going into detail. Then we take a deeper look at the configuration and signatures of the IRC analyzer. Finally, we inspect whether it is possible to evade the signatures.

### 5.2.1 IRC Analysis

For our analyzer module we assume that it receives a single command at the `DataSent()` interface. How this could be reached is explained when we discuss the `ContentLine` analyzer later in this chapter.

We have to differentiate the analysis in the core of Bro and the event-based analysis on Policy Layer. In the core we have to execute common parts of analysis and the division into different message types. This means we have to verify the

acceptance of a message. This is e.g. the test that the size of a command must not exceed the mandatory maximum length of 512 bytes. Violations that occur in the analysis of common characteristics are a hint that an analyzer receives a false protocol. Although it is not suggestive to abort the analysis instantly, we count the number of incidents. When reaching a predefined threshold (20 in IRC) we abort the further analysis here. This means the `DataSent()` function returns `False` causing the ALSA to stop the IRC analyzer. Of course this results in a notification in the corresponding log file named `irc.log`.

Beyond the common analysis we have to classify messages into the three mentioned types. As they are nearly equivalent in structure we try to extract an existing prefix first. The standard requires a prefix to begin with a `'`. If a command line has a leading `'`, we extract the prefix ending with the first space character. Now we have to extract either a command name or a reply number. So we test whether the next word in line consists of three digits only. Then we have found the reply code of a server reply. Otherwise this word must be a command name.

At this point we are able to distinct single messages through the command name or the reply number. So we abstract the current message to an event. Examples for realized events are `irc_privmsg_message` or `irc_join_message`. The further analysis is executed on Policy Layer. Therefore, the analyzer fires an event. The implemented handler in `irc.bro` performs the deeper inspection and triggers the notification of abnormal activity.

Now we continue with the configuration parameters for the IRC analyzer on Policy Layer.

### 5.2.2 Configuration

For IRC the TCP ports from 6665 to 6669 are registered by IANA [15]. But this is not consistent with the Internet. Most IRC servers in the Internet use ports from 6660/TCP to 6669/TCP but there are several others using any unprivileged port. So we use the set of widely used ones and hope that the remaining connections are identified by signatures. What may happen due to the port configuration is that the analyzer receives another protocol using one of these ports at random. But in this case the ALSA will shut down the analyzer when the IRC analyzer recognizes this fact.

We have to set up a configuration for the IRC analyzer that the ALS imports at Bro's start. The current configuration in the policy script `irc.bro` looks as follows:

```

# Loading IRC signature-file
redef signature_files += "sigs/irc-signatures.sig";

#ALS-Configuration:
# Defining the port set
global set_pos_ports = {
    6660/tcp,
    6661/tcp,
    6662/tcp,
    6663/tcp,
    6664/tcp,
    6665/tcp,
    6666/tcp,
    6667/tcp,
    6668/tcp,
    6669/tcp
};

# Defining the prot_cfg-record for IRC
const irc_cfg: prot_cfg = [
    $pos_ports = set_pos_ports,
    $partial = F
];

# Add IRC config to the ALS's config
redef tcp_als_cfg += [{"irc"} = irc_cfg];

```

We have defined only a positive port set that causes the ALSA to load this analyzer. Negative ports have not been used and are omitted as the record field is optional. As the architecture is in a prototypical state partial connections are not supported yet. Therefore we configured the IRC analyzer only to accept complete connections. Finally, we insert the created configuration into the configuration table of the ALS.

### 5.2.3 Signatures

To have the advantage of dynamic protocol detection, signatures have to be defined. The architecture provides positive and negative signatures. The positive ones should be as generic as possible that no regular IRC connections evade the signature matching unit. But on the other hand the pattern must be as specific as possible to reduce false detections. If this is not compatible additional negative patterns may help. These have to be applied when it is not possible to differentiate between a positive signature for one protocol and a special data stream in

another one. Then the negative protocol pattern causes the ALSA not to load this analyzer for the rest of the connection

There are three different patterns that will result in a positive IRC matching. The first two of them are to identify client-to-server connections. The IRC RFC recommends that a client sends the following messages in the given order to register at the server:

- PASS <parameter> (optional)
- NICK <parameter>
- USER <parameters>

The problem is that a PASS command is optional and that the order of the NICK/USER combination is only recommended but not mandatory. Thus we have to allow any NICK/USER combination. A second problem that must be considered are space (\x20) and tab (\t) characters. These can appear at the beginning of a line and between words many times. This results in the following patterns:

```
signature irc_sig1 {
  ip-proto == tcp
  payload /(.*\x0a)*
    (\x20|\t)*[Uu] [Ss] [Ee] [Rr] .+(\x0a)+
    (\x20|\t)*[Nn] [Ii] [Cc] [Kk] (\x20|\t)+.*\x0a/
positive-irc
}

signature irc_sig2 {
  ip-proto == tcp
  payload /(.*\x0a)*
    (\x20|\t)*[Nn] [Ii] [Cc] [Kk] (\x20|\t)+.+(\x0a)+
    (\x20|\t)*[Uu] [Ss] [Ee] [Rr] (\x20|\t)+.*\x0a/
positive-irc
}
```

The additional restriction to TCP connections prevents the signature matching engine to match these patterns in UDP or ICMP connection as well. The first pattern expects that after an undefined number of line breaks (\x0a) and spaces (\x20) or tabs (\t) the word USER appears followed by at least one line break. After more spaces or tabs the word NICK must be found. This command must also be closed by a line break. The second pattern is responsible for the detection of the registration in the opposite order. The word 'positive-irc' defines the kind of action. For us, this means announcing the result to the corresponding ALSA.

Then the ALSA loads the IRC analyzer for this connection.

The last pattern covers server-to-server connections. The sequence of commands to register at another server is:

- PASS <parameters> (optional)
- SERVER <parameters>

In this case both servers registers at the other server using the SERVER command. The consequence is that this keyword must be seen in both directions. With that knowledge we have to construct the pattern in two rules:

```
signature irc_sig3 {
  ip-proto == tcp
  payload /(.*\x0a)*(\x20\t)*
           [Ss] [Ee] [Rr] [Vv] [Ee] [Rr] (\x20\t)+.\x0a/
}

signature irc_sig4 {
  ip-proto == tcp
  payload /(.*\x0a)*(\x20\t)*
           [Ss] [Ee] [Rr] [Vv] [Ee] [Rr] (\x20\t)+.\x0a/
requires-reverse-signature irc_sig3
positive-irc
}
```

So we defined two rules with the same regular expression for each direction. But the second which causes the notification to the ALSA requires the first pattern in the opposite direction.

Now we want to discuss how these patterns may be evaded. We have differentiated two cases - evasion due to the architecture and due to the signatures. For this it is necessary to look at the details of the IRC protocol. IRC requires the registration of both clients as servers at the target server. Although it is possible to send as many commands as wanted to the server before registration the server neither accepts nor forwards any other commands. This fact makes clear why signatures covering the registration are unevadable for doing malicious work via IRC. On the other hand our current signatures can easily be evaded because of the prototypical implementation. After transferring at least 4 KB to fill the ALSA's buffer the detection of the registration does not lead to the loading of the IRC analyzer due to the fact that the connection is partial then. This deficiency is abolished as soon as the architecture and the IRC analyzer support the analysis of partial connections. The next paragraph handles the implementation of auxiliary analyzers for the IRC analyzer.

## 5.3 Auxiliary Analyzers

We have learned the advantages of auxiliary analyzers. Now we present the implementation of two auxiliary analyzers - the ContentLine Analyzer and the ZIP analyzer.

### 5.3.1 ContentLine Analyzer

We have already seen how to analyze IRC traffic and how to customize an analyzer. But for the IRC analysis we have made the assumption that the analyzer receives the stream line by line. Therefore we duplicated the existing ContentLine class and adapted it to the structure of an auxiliary analyzer. Its job is to split the input stream into separated lines.

We have built up a structure similar to the reassembler analyzer. We have the ContentLine analyzer consisting of two components called ContentLine processors. Each processor handles the line-splitting job for one direction. So the analyzer has to distribute the traffic to the corresponding ContentLine processor according to their source. This is illustrated in figure 5.1.

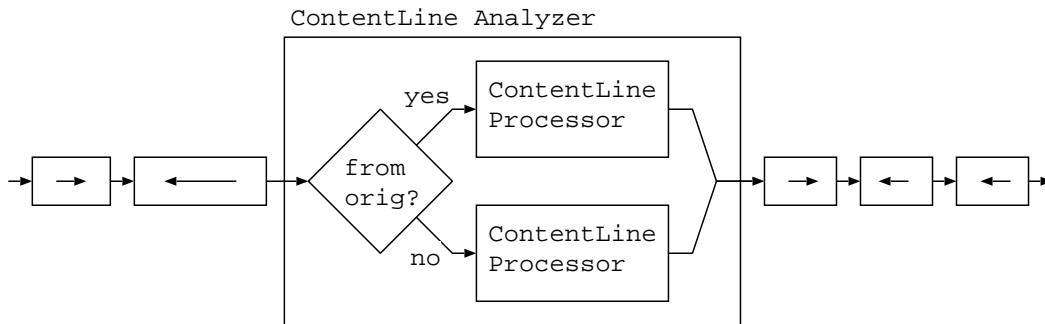


Figure 5.1: Model of the ContentLine Analyzer

The ContentLine processor possesses a buffer to realize this. Note that the stream is already reassembled. So we store the new data in a buffer and deliver line by line through searching the next EOL. It is possible that a line stretches across multiple packets. This requires to keep parts of lines in the buffer until the next part with an EOL arrives.

Now we can use this analyzer to split the IRC stream into single lines. Of course other protocol analyzers will use this analyzer as well. These are analyzers for plain text, line-based protocols like HTTP, FTP, Telnet or SMTP. In the following paragraph we inspect another auxiliary analyzer - the ZIP analyzer.

### 5.3.2 ZIP Analyzer

While we adapt the IRC analyzer we added another auxiliary analyzer for the reason of an IRC protocol feature. The IRC protocol allows to compress server-to-server connections. For that we have to insert an analyzer which inflates the stream before data is delivered to the ContentLine Analyzer.

To compress a conversation, both servers must agree with that. This is done via the PASS command of the IRC protocol as laid down in [32]. When establishing a server-to-server connection PASS holds more parameter fields than only the connection's password. The syntax is as follows.

PASS <password> <version> <flags> [<options>]

The interesting part is hidden in the optional field "options". This entry consists of at most two characters at this time. If this field contains a 'Z' the authenticating server offers compression. Note that more than one PASS command can be sent. Then all PASSES but the last are ignored. The authentication phase ends when the SERVER command is accepted. From this point on all following data is compressed. Figure 5.2 shows the automaton to decide whether a server provides compression or not.

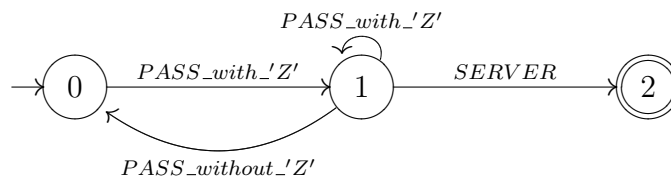


Figure 5.2: Automaton to detect IRC Servers providing Compressed Transfer

A server offers compression if the last PASS command before the SERVER command contains the option 'Z'. This means if both servers has reached state 2 the inflating analyzer called ZIP analyzer must be initialized immediately and put in the chain of auxiliary analyzers before the ContentLine analyzer. Note that the ZIP Analyzer can not function as a protocol analyzer since not the whole connection is compressed.

The structure of the ZIP Analyzer is very simple. The analyzer is again splitted into two ZIP processors which inflate the data stream for a single transfer direction. This is illustrated in figure 5.3.

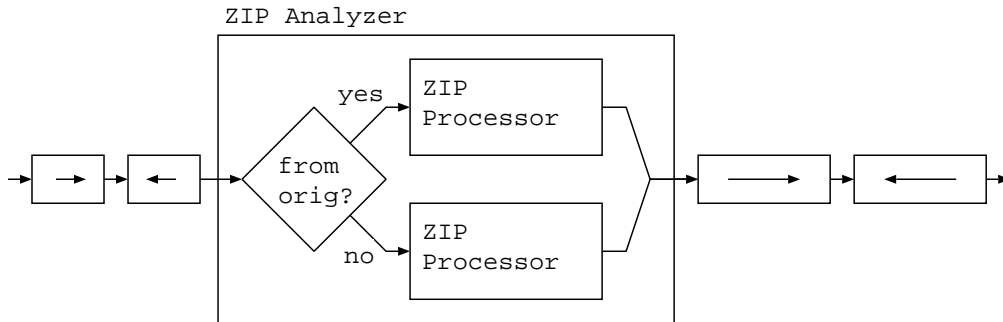


Figure 5.3: Structure of the ZIP Analyzer

The process of inflating data is implemented by using the C implementation of the zlib library [38]. This library simplifies the stream inflation in the implementation enormously. Pieces of stream are stored in a buffer. After the execution of the inflate function provided by the zlib library, we receive the unzipped data in another buffer which is handed over to the ZIP analyzer and then to the ContentLine analyzer. If an error occurs while unzipping like e.g. an error in the zipped stream, the ZIP analyzer is shut down resulting that the IRC analyzer is shut down as well.

We have completed the adaption of the IRC analyzer and continue with the evaluation of our architecture based on the results of the IRC analyzer.

## 5.4 Evaluation

We have performed a number of tests with the developed architecture in combination with the IRC analyzer. On the one hand we want to show the rate of correct as well as incorrect protocol detections of the signature-based approach. On the other hand we took measurements how much performance penalty the new architecture introduces.

### 5.4.1 Protocol Detection Tests

The idea to evaluate the quality of the IRC signatures is to capture two live traffic traces. The first one is filtered so that the trace contains only IRC connections. In opposition, the second trace consists of all but IRC traffic. Based on these traces we can see the correct as the false detection rate.

So we have captured two traces on the uplink of the Münchner Wissenschaftsnetz (MWN) [39] into the Deutsches Forschungsnetz (DFN) [40]. For the IRC-only trace we captured 5 GB of traffic from one of the IRCnet servers called irc.leo.org

in the MWN. The filter requires to have irc.leo.org as source or destination and allows only ports from 6660/TCP to 6669/TCP. These server ports listen for incoming IRC connections. The second trace consists of 10 GB data captured using a filter that allows only TCP connections where neither the source nor the destination is a known IRC server. The list of known IRC servers was obtained from the current server map of the IRC server of the Technische Universität Ilmenau [41]. Additionally, we filtered out all connections using ports from 6660/TCP to 6670/TCP as they mostly transport IRC traffic.

### **Protocol Detection Test #1:**

For the detection tests we have configured the IRC analyzer in this way that it should only be activated through its signatures. The pos\_port set has been omitted for these tests. The ALSA was configured to reassemble all connections from the beginning to exclude that a connection is not identified due to no reassembly. As partial connections are not supported yet, these cannot be detected and thus are filtered out too. The summary of the results can be seen in figure 5.4.

We have counted the number of connections that could be detected at all. These amount to 24566 connections. Then we have tested the IRC signatures. This results in 24538 identified IRC connections. This corresponds approximately 99,9%. Inspecting the remaining connections we have found out, that these are all IRC connections but without correct authentication. These connections are all shut down finally, either by the client through a QUIT message or by the server through a Ping Timeout. This means that all correct IRC connections have been identified by the signature engine.

Without reassembly we received the same result. Although the same amount of connections was identified, omitting the reassembly can be used to evade the signatures.

### **Protocol Detection Test #2:**

The second test should show how often our IRC signatures match on other protocols. The configuration of the analyzer and the ALSA is the same as in the first test. Our result is that in the other trace with 782898 TCP connections the signature engine has not found a single connection matching our IRC signatures. This equals a false positive rate of 0,0%. The results of both tests are summarized in the following figure:

	Detectable Connections	With Reassembly	Without Reassembly
Test #1	24566 (100,0%)	24538 (99,9%)	24538 (99,9%)
Test #2	782898 (100,0%)	0 (0,0%)	0 (0,0%)

Figure 5.4: Results of the Protocol Detection Tests

The second aspect that is of our interest are the performance costs.

## 5.4.2 Performance Tests

Now we are interested in the costs of performance and memory that the dynamic architecture demands. For these tests, we captured another 10 GB trace in the MWN environment without any filters. Based on this, we want to get an overview on how our architecture has influenced the performance of Bro depending on different configurations.

To get significant results we have used a policy script called `pkt-profile` that yields time slices needed to analyze one second of the network trace on the one hand and the amount of used memory on the other. Based on these values, we draw our conclusion<sup>1</sup>.

### Performance Test 1:

The first test shows the comparison between the original version of Bro and the new architecture using a port-based protocol detection. A requirement of the design was to slow down Bro only marginally in this case. So we disabled the detection of protocols by signatures. This leads to the smaller DFA of the ALSA presented in 4.4.2 that prevents buffering. Additionally, the `reassemble_first_packets` flag was set to `False`, having the consequence that only connections that are analyzed on application layer, are reassembled. For this test, Bro should only inspect IRC connections.

We have summarized the results of this test in figure 5.5. We computed the average over the time slices yielding to an average performance value. We have compared this to the original version of Bro. So our new concept requires additional costs of 3,6% relatively to the original. We put this fact down to the instantiation of the ALSAs and the interaction with the ALS as this is the only overhead costing performance. Comparing the need of memory, we reduced the demand a little as figure 5.5 shows.

---

<sup>1</sup>As mentioned we set the variable `stop_on_gap` of the reassembler analyzer to `false`. Elsewhere all connection with a packet loss are skipped. That would influence the results and making them incomparable.

	Original	New
Average of Time Slices	0,251 (100%)	0,260 (103,6%)
Used Memory	45,4 MB	44,6 MB

Figure 5.5: Results of Performance Test #1

But note that this is the basic overhead the new approach. The ALSA is instantiated for every TCP connection, whether analyzed or not. The graphical evaluation of this test is shown in figure 5.8.

### **Performance Test 2:**

In the second test we run Bro equally configured like in the test before, but now with the additional IRC signatures. We have compared this to the original Bro two times. Once in the standard configuration like in test #1 and on the other hand to a Bro with similar signatures to get a significant result of memory usage. The results can be seen in figure 5.6.

	Original	New
Average of Time Slices	0,251 (100%)	0,532 (212,1%)
Used Memory	45,4 MB	122,4 MB

	Original with Signatures	New
Average of Time Slices	0,498 (100%)	0,532 (106,8%)
Used Memory	92,5 MB	122,4 MB

Figure 5.6: Results of Performance Test #2

The resulting overhead in computation is now 112,1% relatively to the original version. Although this sounds very much comparing this to the original Bro with a similar signature we get only 6,8%. This means that the high costs of performance are caused by the signature matching engine. The buffering of connections causes only a small part of this additional costs. The same result can be seen looking at the used memory. As the most part of the memory is needed by Bro and the signature matching engine, the extra costs due to buffering are comparatively low (approx. 30 MB).

The problem that signature matching is so expensive is that protocol signatures are not limited to a few connections but to all TCP connections. Therefore, the computation of pattern matching must be done to nearly all connections. This requires a lot of performance and memory. The graphical results of the original version of Bro with enabled signatures compared to our new approach are shown in figure 5.9.

### **Performance Test 3:**

For the last test we additionally enabled `reassemble_first_packets`. This means that all TCP connections are reassembled completely. The results are summarized in figure 5.7.

	Original	New with Reassembly
Average of Time Slices	0,251 (100%)	0,588 (234,4%)
Used Memory	45,4 MB	152,3 MB

	Original with Signatures	New with Reassembly
Average of Time Slices	0,498 (100%)	0,588 (118,0%)
Used Memory	92,5 MB	152,3 MB

Figure 5.7: Results of Performance Test #3

We now get a relative increase of 18,0% to the original version with signatures. This is not surprising as reassembling all TCP connections is expensive. Figure 5.10 illustrates the result. Compared to the original without any signatures it amounts to 134,4% extra costs.

Setting this flag to true is only recommended when the load of the network link is low. For high-speed analysis this must be switched off.

What these tests have shown is that a dynamic approach needs more resources than a static approach. But on the other hand, the usage of the new architecture without signatures for protocol detection slows down the system only marginally. The consequence is that a further development in this direction is recommended as the system is as powerful as before, almost as fast as before and has the ability of performing additional protocol detection by signatures. How to configure the system depends on the load of the network link as well as the power of the NIDS's host.

Our results also show that the performance penalty is acceptable if other signatures are already used. Now we come to the last chapter summarizing this work and giving an overview about future works.

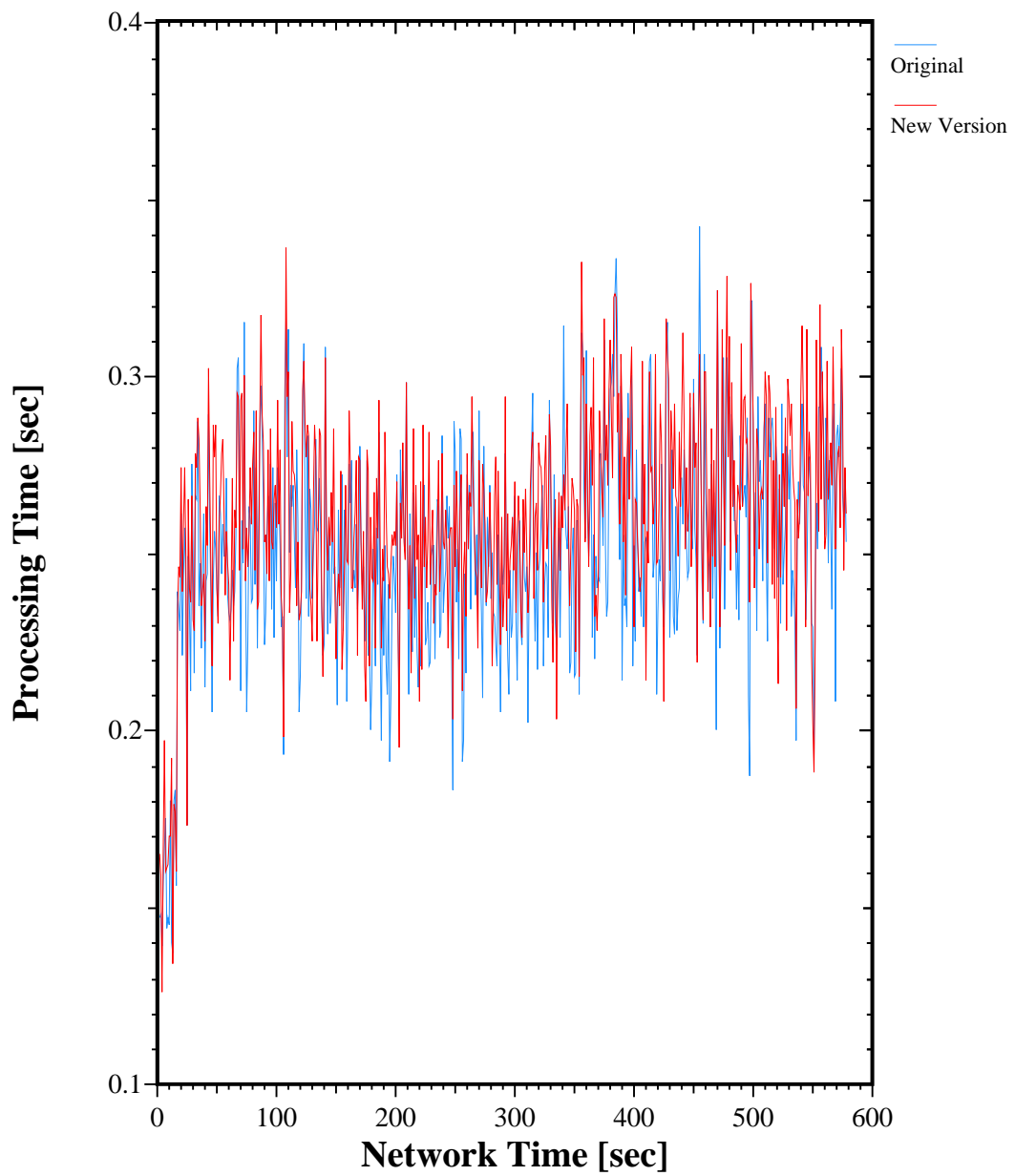


Figure 5.8: Results of Performance Test 1

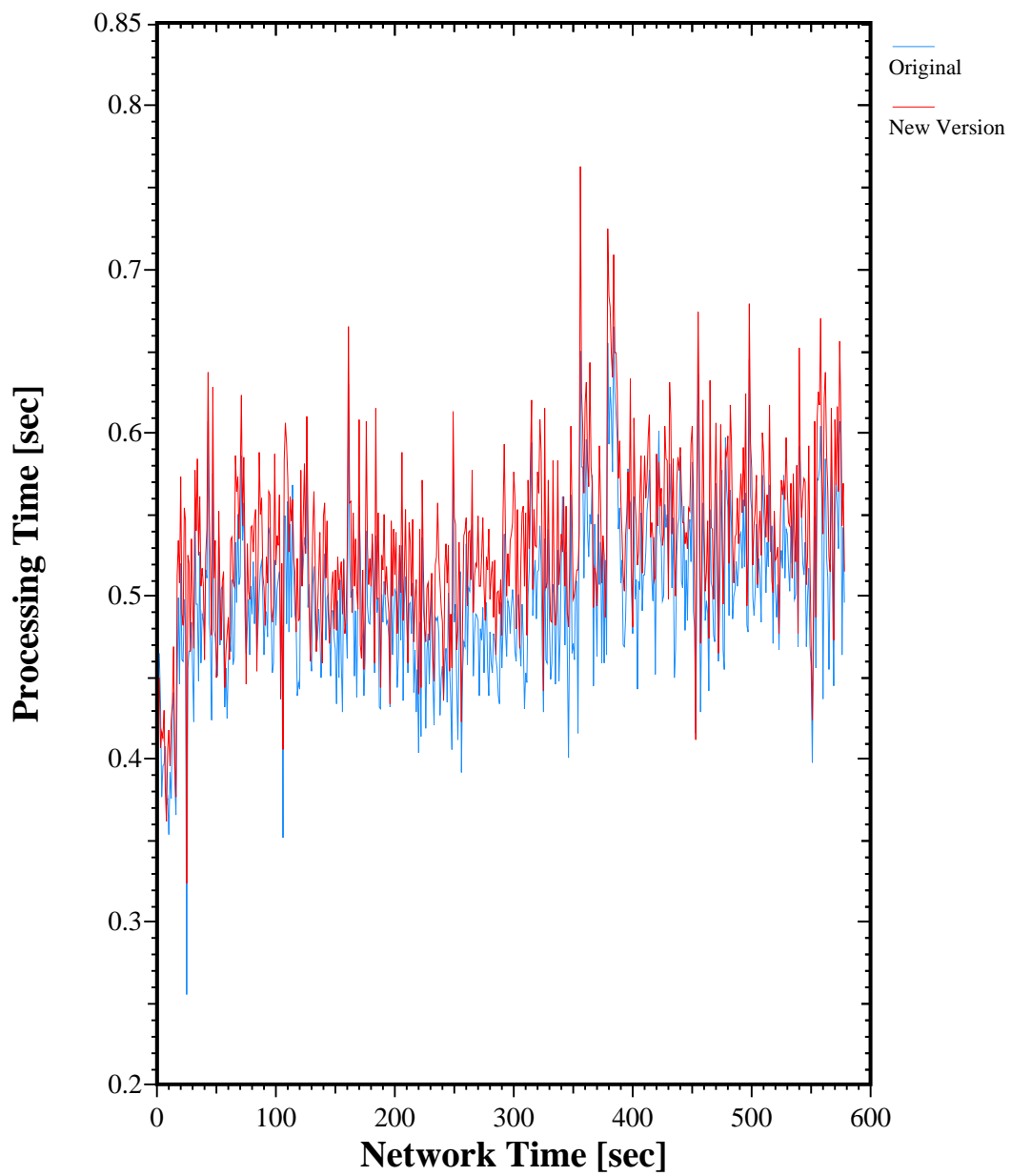


Figure 5.9: Results of Performance Test 2

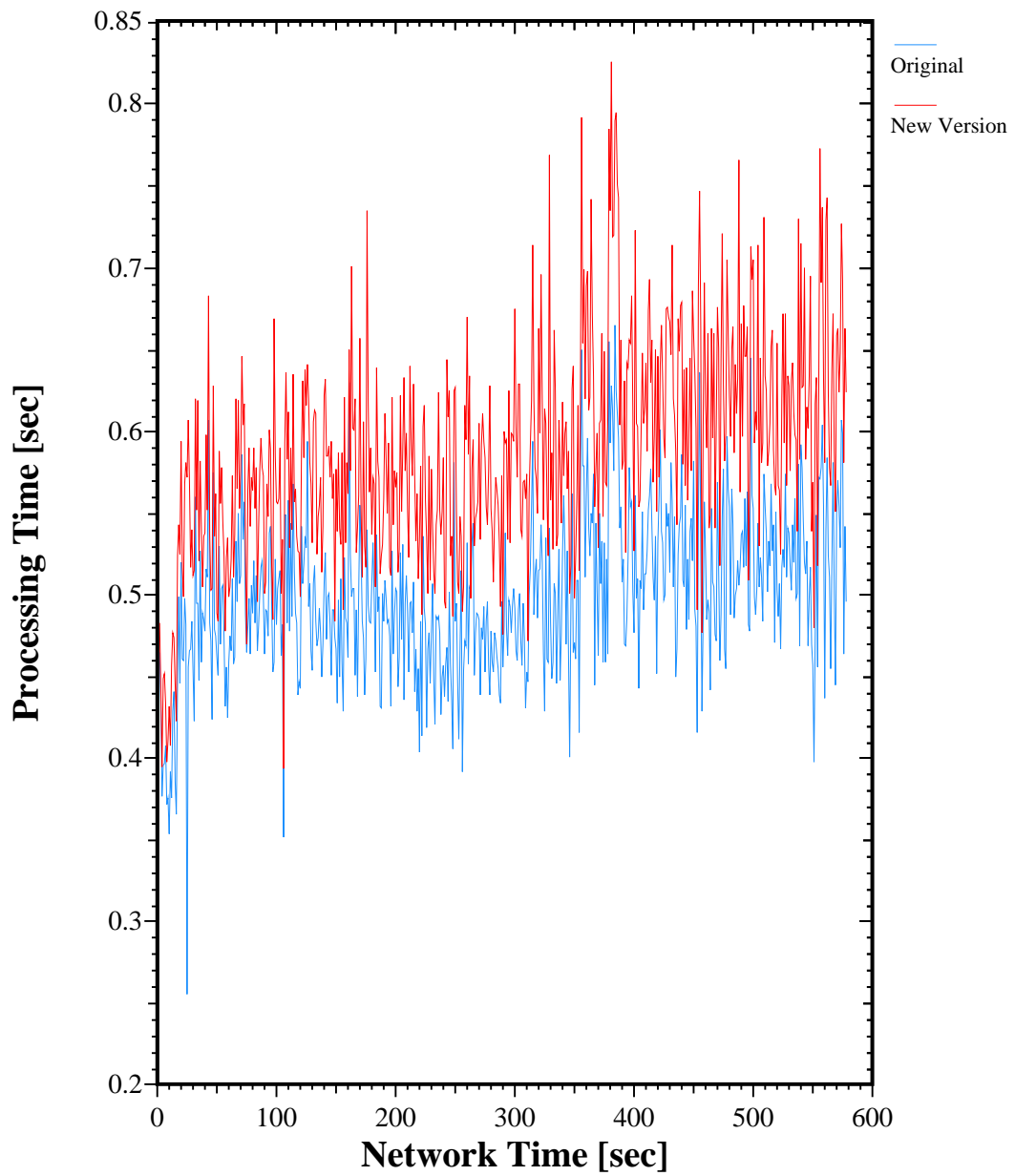


Figure 5.10: Results of Performance Test 3

# Chapter VI

## Conclusion

In this chapter we summarize our work and draw the conclusion from the results. After that we give an outlook on future work.

### 6.1 Summary

This work focus on the problem that a NIDS has to classify network connections by their application layer protocol to perform detailed protocol analysis. As this is not laid down typical, NIDSs use a static protocol detection by port numbers. But this is not a reliable technique for choosing the correct application layer protocol. This may lead to false detections or even to omission of analysis. The consequence may be undetected attacks.

For this we have presented a new design for a NIDS that allows to use a combination of multiple protocol detection methods. Consequently, our architecture has to support dynamic changes in which analyzing modules are applied to a connection. We accomplish this by associating a dynamic analyzer tree with each connection. For the classification of the application layer protocol we have developed two components. The first is called ALS and stores the configuration of all analyzers. The second component named ALSA is inserted in the analyzer tree and has the task to identify the application layer protocol of the connection by using the mentioned detection methods. Additionally, the ALSA buffers the data stream in order to deliver the whole connection content to the suitable analyzers. The combination of those techniques yield an architecture for dynamic protocol analysis for NIDSs.

We have implemented a prototype of our design for the open source NIDS Bro. It contains the new components as the structure for the analyzers already. The ALSA is equipped with protocol detection by port, signatures and customized decision functions of analyzers. In addition we have adapted the IRC analyzer

to integrate into the new architecture. This enables us to analyze complete IRC connections identified by signatures.

The tests have demonstrated that the design is practicable. The protocol detection tests yield very good results for IRC by signatures. On the other hand, the dynamic approach needs significantly more resources. But when using the static protocol detection the performance loss is hardly noticeable. Another advantage of the application of this architecture is the support for a future implementation of application layer tunnel analysis.

Some extensions that would round off our prototype are mentioned in the following paragraph.

## 6.2 Future Work

There are a lot of following issues that improve the new architecture of Bro to become more effective.

Up to this time, the ALSA is only implemented to handle complete TCP connections. This means that tests with partial connections as the IRC analyzer have to follow. In the context of partial connections another problem may appear. If segments of TCP do not contain complete lines or commands but parts of them, the activation of an analyzer in the middle of a connection may fail if it is too restrictive. This fact should be reminded when adapting analyzers.

The tests have demonstrated that buffering of network traffic is not as expensive as expected. This leads to two different improvements. The first is to use a sliding window buffer. This means, that we still have 4 KB of buffer but do not completely delete it when filled up. Only the oldest data is discarded and overwritten with new data. As soon as the first byte of a connection is lost, we only instantiate new analyzers that are able to handle partial connections. The second approach is the idea of dynamic buffer sizes. If there is an analyzer already inspecting the traffic, buffering might be useless. On the other hand, if there is no analyzer at all we use a larger buffer for this connection. The idea of buffering connections in this way was used in another project at our research unit as well. It is called Time Machine [42] and aims to replay connections for Bro some day.

After these problems have been solved it is useful to modify the protocol analyzers. In most cases this will not be much work. The adaption to the classes and interfaces should be easy, enabling analyzers to accept partial connections is more difficult. For FTP and IRC there is additional work to do. Both will use the prediction table of the ALS for FTP-DATA respectively DCC connections.

The most complex part of work is to find suitable signatures. These require a lot of testing with real traffic.

The need of analyzing tunneled traffic becomes more and more important as usage increases. The ALSA is ready for detecting encapsulated protocols although untested due to the missing of analyzers. The main problem is that we are able to analyze only tunnels based on application layer encapsulation. This requires some protocol analyzers, like e.g. the HTTP analyzer, to produce a correct output which can be delivered to the analyzer of the encapsulated protocol. Tunnel analysis for the lower layers require the extraction of an autonomous TCP and IP analyzer.

A further issue is the idea of applying this approach to UDP connections as these suffer from the same problem of protocol classification by ports as TCP connections. Therefore, it might be useful for UDP as well. But as UDP is a stateless protocol it might require another strategy for buffering.

We have presented design and implementation of a dynamical approach for protocol analysis with enhanced protocol detection methods. This solution is no more only a future vision but is realizable with today's computers. Summarizing we can say that this work has pointed out a new path to more sophisticated Network Intrusion Detection Systems.

# Bibliography

- [1] Steven J. Scott,  
Threat Management Systems - The State of Intrusion Detection, 2002,  
<http://www.superhac.com/docs/threatmanagement.pdf>.
- [2] NetOptics,  
Deploying Network Taps with Intrusion Detection Systems,  
<http://www.netoptics.com/products/pdf/Taps-and-IDSs.pdf>.
- [3] Brian Laing,  
HOW TO GUIDE: Implementing a Network Based Intrusion Detection System, Internet Security Systems,  
<http://www.snort.org/docs/iss-placement.pdf>.
- [4] Martin Roesch,  
Snort: Lightweight Intrusion Detection for Networks,  
In *Proc. 13th Systems Administration Conference - LISA '99*, pages 229-238  
1999.
- [5] Official Snort Website,  
<http://www.snort.org>.
- [6] Writing Snort Rules,  
[http://packetstormsecurity.nl/papers/IDS/snort\\_rules.htm](http://packetstormsecurity.nl/papers/IDS/snort_rules.htm).
- [7] Vern Paxson,  
Bro: A System for Detecting Network Intruders in Real-Time  
*Computer Networks (Amsterdam, Netherlands: 1999)*,31(23-24):2435-2463,  
1999,  
<http://www.icir.org/vern/papers/bro-CN99.html>.
- [8] Robin Sommer,  
Bro: An Open Source Network Intrusion Detection System,  
DFN-Arbeitstagung über Kommunikationsnetze, 2003,  
<http://net.in.tum.de/~robin/papers/dfntag03.ps>.

- [9] Robin Sommer, Vern Paxson,  
Enhancing Byte-Level: Network Intrusion Detection Signatures with Context  
<http://www.icir.org/vern/papers/sig-ccs03.pdf>.
- [10] Official Bro Website,  
<http://www.bro-ids.org>.
- [11] Bro User Manual,  
<http://www.bro-ids.org/Bro-user-manual/index.html>.
- [12] Bro Reference Manual,  
<http://www.bro-ids.org/Bro-reference-manual/index.html>.
- [13] Andrew S. Tanenbaum,  
Computernetzwerke, 3. Auflage, Pearson Studium, 2000.
- [14] James F. Kurose, Keith W. Ross,  
Computer Networking, 3rd edition, Addison-Wesley, 2005.
- [15] Internet Assigned Numbers Authority, Port Numbers,  
<http://www.iana.org/assignments/port-numbers>.
- [16] W. Townsley et al.,  
Layer Two Tunneling Protocol "L2TP", RFC-2661  
August 1999.
- [17] W. Simpson,  
IP in IP Tunneling, RFC-1853  
October 1995.
- [18] T. Dierks et al.,  
The TLS Protocol Version 1.0, RFC-2246,  
January 1999.
- [19] S. Kent,  
Security Architecture for the Internet Protocol, RFC-2401,  
November 1998.
- [20] Generic Routing Encapsulation (GRE) - Cisco Systems,  
[http://www.cisco.com/en/US/tech/tk827/tk369/tk287/  
tsd\\_technology\\_support\\_sub-protocol\\_home.html](http://www.cisco.com/en/US/tech/tk827/tk369/tk287/tsd_technology_support_sub-protocol_home.html).
- [21] SSL 3.0 Specification,  
<http://wp.netscape.com/eng/ssl3/>.

- [22] Application Layer Packet Classifier for Linux,  
<http://17-filter.sourceforge.net>.
- [23] Protocol Definitions for L7-Filter,  
<http://17-filter.sourceforge.net/protocols>.
- [24] Sebastian Zander et al.,  
Self-learning IP Traffic Classification based on Statistical Flow Characteristics, Passive and Active Measurement 2005 in Boston,  
<http://www.pam2005.org/PDF/34310328.pdf>.
- [25] Official TCPDUMP and LIBPCAP Website,  
<http://www.tcpdump.org>.
- [26] Steven McCanne and Van Jacobsen,  
The BSD Packet Filter: A New Architecture for User-level Packet Capture,  
December 1992,  
<http://www.tcpdump.org/papers/bpf-usenix93.pdf>.
- [27] J. Reynolds et al.,  
Assigned Numbers, RFC-1700,  
October 1994.
- [28] J. Oikarinen, D. Reed,  
Internet Relay Chat Protocol, RFC-1459,  
May 1993.
- [29] C. Kalt,  
Internet Relay Chat: Architecture, RFC-2810,  
April 2000.
- [30] C. Kalt,  
Internet Relay Chat: Channel Management, RFC-2811,  
April 2000.
- [31] C. Kalt,  
Internet Relay Chat: Client Protocol, RFC-2812,  
April 2000.
- [32] C. Kalt,  
Internet Relay Chat: Server Protocol, RFC-2813,  
April 2000.
- [33] Website fore IRCnet,  
<http://www.ircnet.org>.

- [34] Website for QuakeNet,  
<http://www.quakenet.org>.
- [35] Website for irc.leo.org,  
<http://irc.leo.org>.
- [36] Roland Gruber,  
Implementation of an IRC Protocol Analyzer for the Network Intrusion De-  
tection System Bro, Lehrstuhl VIII, Technische Universität München,  
<http://www.net.in.tum.de/teaching/projects/topics.html>.
- [37] P. Deutsch and J.-L. Gailly,  
ZLIB Compressed Data Format Specification Version 3.3, RFC-2813,  
May 1996.
- [38] Official ZLIB Website,  
<http://www.gzip.org/zlib>.
- [39] Overview of the MWN,  
<http://www.lrz-muenchen.de/services/netz/mhn-ueberblick/>.
- [40] Overview of the DFN,  
<http://www.dfn.de>.
- [41] Server-Map of the TU Ilmenau,  
[http://irc.tu-ilmenau.de/all\\_servers/](http://irc.tu-ilmenau.de/all_servers/).
- [42] Stefan Kornexl, High-Performance Packet Recording for Network Intrusion  
Detection, January 2005  
<http://www.net.in.tum.de/teaching/projects/docs/kornexl.pdf>.