Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Dominik Langenegger
Patrik Bichsel

Design and Implementation of a Protocol-Aware Connection Tracker Framework

Semester Thesis SA-2006-20
April 2006 to November 2006

Tutor: Bernhard Tellebach
Co-Tutor: Daniela Brauckhoff
Supervisor: Bernhard Plattner

Acknowledgment

It has been an interesting task to get to know some of the available technology which can be used in order to construct a flexible framework. We enjoyed the challenge of designing a framework which can be easily extended and clearly we very much enjoyed the insight in state of the art protocols of the Internet.
Special thanks to Bernhard Tellenbach and Daniela Brauckhoff who supported us as we were working on the thesis. Additionally we want to express our gratitude to the head of the Communication Systems Group, Prof. Dr. B. Plattner, for providing the opportunity of our research leading to this thesis.

Zurich, November 17, 2006

Dominik Langenegger, Patrik Bichsel

Abstract

The NoAH project is aiming towards information gathering from Internet cyberattacks. It works using honey pots which attract attackers by pretending to contain valuable information. If a honey pot gets attacked it should be able to log the activities of the attacker and hence recognise the scheme that might be used. From those information an attack pattern could be extracted in order to prevent this kind of attack in the future. The aim of NoAH is automated attack signature generation and the detection of attacks upon those signatures.
Existing attacks partially work by sending packets that result in crashing the attacked system. This mechanism could be mitigated when the combination of packets with the according vulnerable states would be known. Therefore real time state information can be exploited to protect a system by not accepting certain packets when the host is in a vulnerable state. Consequently this would lead to a protection mechanism that is placed on a very low layer.
This thesis is about designing and implementing an extendible and scalable framework. The latter is used for tracking the state of several protocols by observing the network traffic between a target host and several alleged attackers. The main advantage of the system should be the real-time protocol state tracking which is not provided by other open source software. The basis for state tracking is provided by the current framework however the necessary extensions are not yet deployed. Also the proper design of the output and the interface to the attacked host has to be enhanced.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This documentation is seen as an addition to the documentation provided along with the source code of the framework that has been implemented. Chapter 2 provides an overview over the theoretical aspects which form the basis of the project. However it is assumed that the reader has a basic understanding of Internet network technology. This means that only aspects which have an influence on the project have been taken into account.
As there are many techniques used, which may not all be known in detail by the reader, Chap. 3 gives an overview and explains the most fundamental functions that have been used. The idea behind this chapter is to provide a basic understanding. If a part of the project needs to be extended the sources of this information can be consulted to gain a deeper insight.
The next chapter (Chap. 4) explains on a high level of abstraction how the tracker framework is designed. Several aspects are examined in more detail and should make a look into the source code unnecessary. Therefore the project can be continued without deep understanding of all parts of it. The design part directly leads to Chap. 5 where the performance of the implementation is measured in different environments. The clues taken from those measurements are examined in detail in Chap. 6. Chapter 7 shows the direction in which the project might be continued and concludes the thesis.

Chapter 2

Theoretical Background

2.1 Network Layer

There exist several protocols working at the network layer. At the current state of development of the Internet, the most important for higher layer protocols is the Internet Protocol (IP). Consequently only this protocol is considered in the current implementation of the framework wherefore only IP is taken into account at the theoretical discussion.

2.1.1 Internet Protocol (IP)

IP has the ability to address hosts over several, possibly different, physical networks. Consequently its main service is the globally unique addressing scheme. Unlike the global uniqueness of the Ethernet addressing this scheme is hierarchical and therefore it provides scalability. The service of IP is transparent to the upper layer protocols and it is a best effort service. This means that no guarantees about the arrival of packets can be given. Hence duplication, loss or corruption of packets and reordering of packets can arise. Furthermore no time constraints whatsoever can be met.

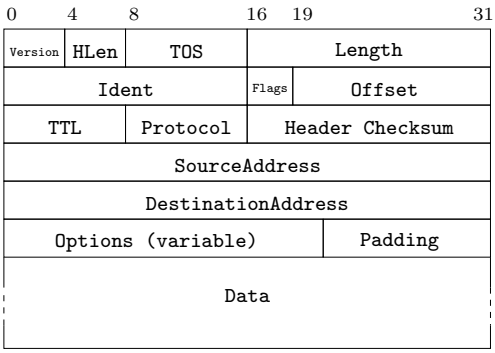

Figure 2.1: IP-Header

The differences of physical networks which have to be traversed, imply that a packet may have to be transferred over networks using different maximal transmission units (MTU). In such a case the packet is fragmented which means that the original packet is divided into several packets in order to get packets which are smaller or equal to the MTU of the network. Thereby the "more fragments" (`MF`) bit in the `Flags` field of the header is set in all but the last packet. In order to reassemble the packets correctly an identifier is necessary. This is implemented by the identification number (`Ident`) which is the same in all fragmented packets. Subsequently the identification number has to be unique for the pair of a source and destination IP address. Additionally the `Offset` field provides the offset of the packet relative to the beginning of the latter. This offset is given in bytes and is set to zero if the packet is not fragmented. The packet can be reassembled at the IP layer which is done at the end-point of communication and not at an intermediate node. [1]

Nowadays fragmentation is not used often as most applications use MTU path discovery and set the "don't fragment" (`DF`) flag which is the last bit of the `Flags` field. This forces the connection point where fragmentation should take place to send an error message back to the origin of the packet. In that message the MTU of the subsequent network is indicated. Hence the original packets are built not larger than the size indicated.
The header of the IP protocol in version 4 is shown in figure 2.1. Version 6 is not considered as it is not widely implemented so far. Most of the newly introduced fields allow to reject corrupted packets or implement a certain quality of service enhancement. As they are not of further interest for the thesis, they are not explained in detail. Another enhancement of version 6 is the extension of the address numbers to 128 bits.

2.2 Transport Layer

The transport layer offers and end-to-end service but there are different possibilities of what functionalities are provided to upper layers. The two most commonly used protocols user datagram protocol (UDP) and transmission control protocol (TCP) provide entirely different services wherefore a short description of their most important properties are given in the following.

2.2.1 User Datagram Protocol (UDP)

Designed for services where the transmission time is more important than the order of the data frames or the lack of such a data frame, UDP extends the network with few functionality. Mainly the possibility that several processes of a system can communicate at the same time with the same remote system and even the same process on the latter is provided. This property is realised by using ports which, in combination with the IP address, identify unambiguously a certain process on a system. Hence UDP offers a demultiplexing service without other functions which enhances the speed as the datagrams remain short and their interpretation is simple.
Additionally the header of UDP provides a checksum which is built over the UDP header, pseudo-header and payload. The pseudo-header consists of the source IP address (`SourceAddress`), destination IP address (`DestinationAddress`) and the protocol number (`Protocol`) of the IP header and the `Length` field of the UDP header. This is done to protect against packets which have been misrouted.

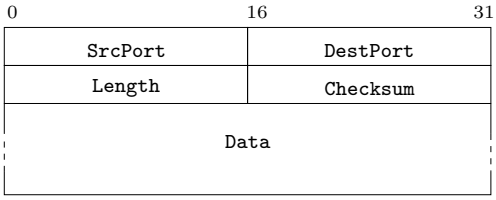

Figure 2.2: UDP header

Figure 2.2 shows the UDP header in detail. The port number of the sender and the receiver has a length of 16 bits. The source port is optional as well as the checksum. Important to notice is that an all zero checksum is transmitted only if the checksum is not computed. If the computed checksum is all zero then its one complement (all ones) is transmitted.
The length in the UDP header indicates the size of the UDP payload. It is given in a 16 bit field and it is specified including the length of the UDP header. Subsequently a maximal payload length (including the IP header) of $2^{16} - 8$ bytes is possible for a UDP packet. [2]

2.2.2 Transmission Control Protocol (TCP)

The transmission control protocol has been developed to guarantee a reliable connection between processes on different hosts. This means that the delivery of a packet is guaranteed and the sequence of arrival on the upper layer is the same as it has been at the sender. Additionally it provides flow- and congestion control. These properties stand in contrast to the attributes of UDP but they are desirable for a variety of reasons even though they have drawbacks. Especially the implementation becomes much more sophisticated which slows the information flow down.

Additionally there is much more overhead data and retransmissions cause some packets taking much longer to transfer. The purpose of this discussion is mainly to provide an insight on the properties that are meaningful in conjunction with the tracker framework.

TCP Header

The header of the TCP protocol consists of the same fields as the UDP header but additional fields are required in order to provide the desired service to upper layers. Figure 2.3 shows the available fields in more detail.

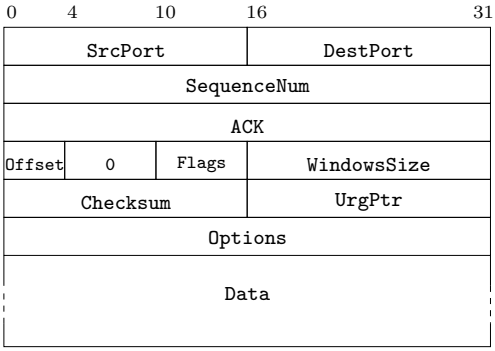

Figure 2.3: TCP header

The source port (`SrcPort`) and destination port (`DestPort`) are used in the same way as seen in UDP. In conjunction with the IP addresses of host and server, they unambiguously identify a socket. The latter identifies one specific connection between any two processes running on hosts connected through the Internet. [3] [4]
To guarantee the correct packet order at the receiver and to be able to identify duplicated packets, the protocol has to introduce some sort of numbering of the packets. This is done using the `SequenceNum` field of the header. Assigning a sequence number to each octet of bits assures a continuous numbering. The reason for the numbering of bytes is that the upper layer considers TCP to provide the transmission of a stream. Hence it does not have to deliver the data as packets. To make the acknowledgment process easier they have cumulative meaning. It is evident to see that the numbering space is finite which is a problem that increases gradually with the increasing speed of Internet connections.
The options are at the end of the header and can begin at any octet boundary. The `Offset` indicates the offset caused by the options where the difference between used option bits and offset is padded (i.e. set to zero). [5]

Connection states

Tracking the state of the TCP protocol on the monitored host needs basically the `SYN`, `ACK` and `RESET` flags. The theoretical discussion is hence concentrating on the states, the transitions and the sequences resulting in transitions. The transitions and states can be best shown in a figure (see Fig. 2.4).
The transmission control block (TCB) serves to administer a connection. The current state and all necessary information is stored in it. This implies that from the states in Fig. 2.4 all but the ”closed” state have an associated TCB, which makes the latter state somewhat theoretical.
The passive open indicates a host that is willing to accept connections (in the following called server) whereas the active open is used by a host to initiate a connection itself. An apparent problem is when many hosts want to initiate connections where one server has to allocate memory for many TCBs. This can easily exhaust the available memory of the server.
A short description of the states is given in the following:

closed: just a theoretical state

listen: means that a host is accepting connections from remote hosts on a specific port

SYN sent: when a host is waiting for a matching connection request after having sent a connection request

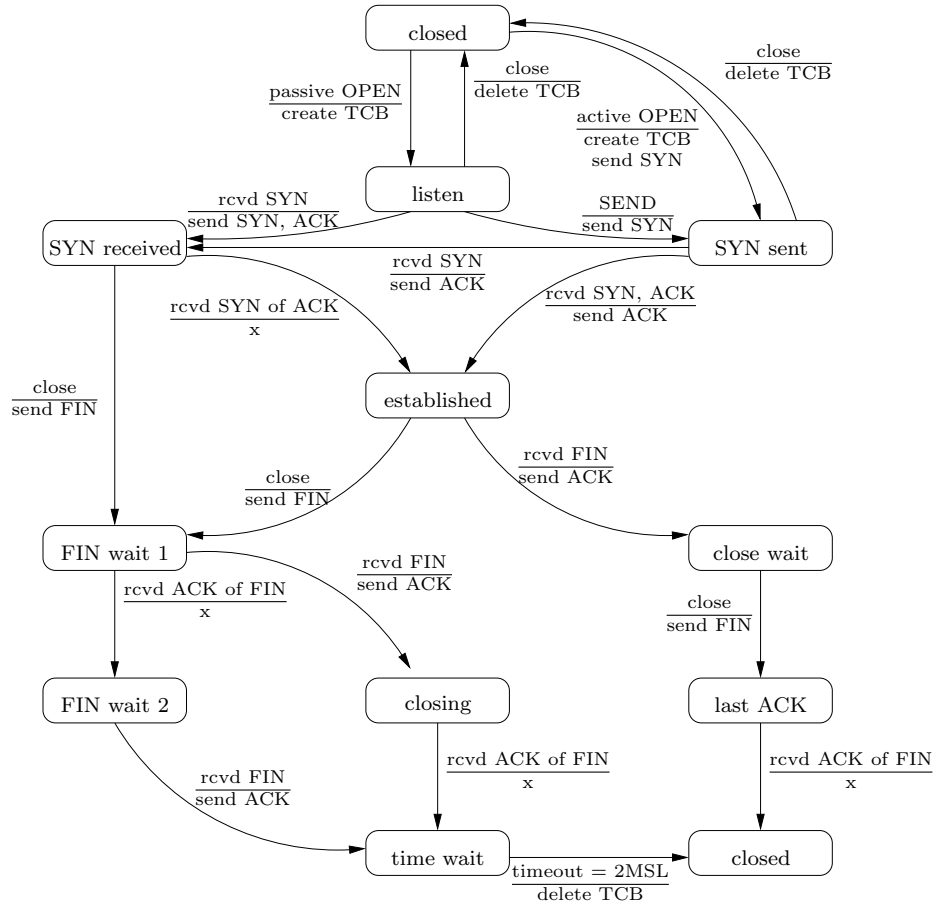

Figure 2.4: Connectionstates of the TCP protocol

SYN received: represents waiting for the connection acknowledgment belonging to the connection request when a connection request has been sent or one has been received

established: in this state a connection is successfully opened and data can be transferred

FIN wait 1: after having sent a termination request or when waiting for a connection termination request

FIN wait 2: when waiting for a connection termination request from the remote host

close wait: in this state the host is waiting for a connection termination request from the local user

closing: means that the host is waiting for a connection termination request acknowledgment from the remote communication partner

last ACK: is waiting for an acknowledgment of the connection termination request which has previously been sent to the remote communication partner and includes an acknowledgment of its connection termination request

time wait: when waiting for enough time to pass in order to be sure that the remote communication partner received the acknowledgment of its connection termination request

The notation of the transitions implies that when the action above the separating line happens, then the action below the line is executed. The action above can be initiated by the user (e.g. `close`) or by the remote host (e.g. `recv SYN`).
From the given figure the tree way handshake and other fundamental properties of the TCP protocol can be derived. For completeness sake the flow- and congestion control mechanism of TCP should be described but as at the current state of the project these do not come into play, they are omitted.

Chapter 3

Techniques

3.1 XML parsing

A property that a framework with a wide utilisation range should fulfil is configurability. At the same time it should not be complicated to adjust the configuration for the needed purpose. In addition to the public accessible parsing programs, we have been equipped with a lightweight XML parser by Bernhard Tellenbach. For those reasons we decided to use a configuration file written in XML.
The parser has not been examined in detail nevertheless the main functionalities are given here. It parses a file and imports the nodes in a DOM tree with the root element called "XMLNode". The tree resulting from the XML file is then imported into a more appropriate data structure.

3.2 Description of pcap

The purpose of the pcap library is to capture received packets and pass them in a utilisable form back to its caller. Before the capture begins there can be used several options to filter the result. For example the sniffing device, IP of the host to be monitored or the network mask of a monitored group of hosts can be specified. Even a port on the monitored host can be specified. Other programs like "ethereal" or "tcpdump" using the library have also been a decision enhancing factor in favour of the pcap library.
The data received from pcap is structured into header and packet information. When a packet is received pcap generates a header with physical information before passing the packet on. This header is structured in the following way:

```
struct pcap_pkthdr {
    struct timeval ts;    /* time stamp */
    bpf_u_int32 caplen;   /* length of portion present */
    bpf_u_int32 len;      /* length this packet (off wire) */
};
```

The first element contains the time at which this packet has been captured. The second and third are both length' of the packet where the difference is that len might be larger than caplen when the packet size exceeds the snaplen argument of the pcap_open_live() routine which is described below in further detail.
The packet information is handed over via a u_char pointer. This pointer is directed to the begin of the captured packet. It can be examined best when using a structure with field length according to the definition (Chap. 2). The used structures are given in the netinet directory. Using those structures every field of the respective header can be accessed easily. Still carefulness is not obsolete as the bit ordering has to be transformed from network bit order to host bit order.

3.2.1 Functions of the pcap library

The library offers a wide variety of functions where only the ones that have been used are taken into account.

```
int pcap_lookupnet(char *device, bpf_u_int32 *net, bpf_u_int32 *mask, char *errbuf);
```

This function is used to determine the network address and the subnet mask for a device given by ∗dev. Those values are returned using ∗net and ∗mask respectively. In case of successful execution the function returns 0 and -1 otherwise. In the latter case ∗errbuf is populated with an error description.

pcap_t ∗pcap_open_live(**char** ∗device, **int** snaplen, **int** promisc, **int** to_ms, **char** ∗errbuf);

When willing to capture packets from a device this function must be called. The packet capture descriptor which is returned, is needed for all other functions dealing with capturing packets. The device is the name of the network device (e.g. eth0). Snaplen specifies the maximum number of bytes that are captured per packet and promisc being 1 indicates that the device is put into promiscuous mode. The integer to_ms specifies the read timeout in milliseconds. The last argument again serves to hold an error description when the routine fails. In that case the returned pointer is set to NULL.

int pcap_datalink(pcap_t ∗p);

The data link layer type is returned when this function is called. It uses the packet capture descriptor from pcap_open_live() as input argument. In case of Ethernet the return value is set to IFT_ETHER.

int pcap_compile(pcap_t ∗p, **struct** bpf_program ∗fp, **char** ∗str, **int** opt, bpf_u_int32 mask);

This function serves to compile an expression into a filter which can be used to select packets at the layer of the pcap library. The arguments needed are the device descriptor ∗p as always, a pointer where the resulting filter can be placed in (∗fp) and the filter given as expression at location ∗str. After all there can be specified whether there should be optimisation performed on the resulting filter or not and the network mask (mask) needs to be specified.

int pcap_setfilter (pcap_t ∗p, **struct** bpf_program ∗fp);

Here the previously complied filter and the capture device descriptor are used to make the filter active when capturing new packets.

int pcap_loop(pcap_t ∗p, **int** cnt, pcap_handler callback, u_char ∗user);

This routine reads and processes packets. Once again the capturing device descriptor is used as an input argument. The capturing is stopped when cnt packets have been captured except when cnt is set to -1 which indicates that it should loop forever. Callback specifies the function that should be called in order to further process the packets. This function has to be of the following structure:

void user_routine(u_char ∗user, **struct** pcap_pkthdr ∗phdr, u_char ∗pdata)

The last argument of pcap_loop() corresponds to the first argument of the callback function. This means that data can be passed from the pcap_loop() function to the user_routine() using this pointer.

The ∗phdr points to the header that is generated by the pcap library and ∗pdata points to the packet data.

void pcap_breakloop(pcap_t ∗p)

With this routine it is possible to end the pcap_loop() routine. It uses the capture device descriptor created by the pcap_open_live() function. As the loop routine might be blocked inside the user_routine() at the time this function is called, the termination might only be executed after the next packet arrives.

3.3 Dynamically Linked Libraries (DLL)

The extendability of the framework is guaranteed by the use of dynamically linked libraries. For creating such libraries programmers have to write plug-ins as external C-functions and compile them in a specific way. To use functions of a library in a program one has to load the library first and then functions of this library can be called. The implementation is done in the dlfcn.h header file and the dl library. [6]

The extendability comes at the price that the program is slowed down at startup because of the loading of those libraries but when only a few and small libraries are used it is almost not noticeable.

3.4 Threads

With regard to multi-core architecture it can be a benefit to run independent computations in different threads. The analysis of connection states of different connections is such an independent computation and therefore if it reveals that state tracking is very performance consuming it is best to divide this burden on several cores.
On UNIX-systems one can access thread programming through an interface specified by the IEEE POSIX 1003.1c standard (1995). The implementation of this standard is called pthreads and defines a set of C language programming types and procedure calls. It is implemented with a pthread.h header file and the thread library pthread. [7]

3.5 Message Queues

Communication between threads can be done in different ways. One can use pipes, shared memory, message passing and several other techniques. For the tracker framework message queues seemed to be the most suitable.
The basic idea is that processes or threads can exchange information via access to a common system message queue. The sending process places its message in the queue and then it can be read by another process. Each message consists of a type and a message field. The type field helps receiver processes to selectively pick messages by a certain type from the queue. Unlike other mechanisms, like the file byte-stream data flow of pipes, messages in message queues have an explicit length.
Sending and receiving messages can either be done in blocking or non-blocking operations. In blocking operations the sending process cannot continue until the message has been transfered. A receiver in blocking operations has to wait until a message of the desired type appears in the queue. Non-blocking message passing allows for asynchronous message transfer. A process in non-blocking operation is not blocked by sending or receiving a message. To use message queues in a C program one has to include the sys/ipc.h and sys/msg.h header files. [8]

Chapter 4

Design

At first a rough overview of the design is given to illustrate the basic ideas. The framework captures packets that are exchanged between the monitored host and any other host. The details about which packets to capture and several other configuration aspects are defined in the configuration file which is loaded when the framework starts. After the framework has received the packet it passes it on to the appropriate plug-in. When a plug-in receives a packet it starts the analysis of the protocol it was programmed for and reports the result. After having finished its analysis the packet is passed on to the sub-plug-in if the latter exists. If a plug-in has no successor for a given payload, it tells the framework that the handling of the current packet has finished.
On a closer look as given in Fig. 4.1 we can spot the basic components:

- Configuration file
- Dispatcher component
- State tracker threads [1...n]
- Protocol libraries (plug-ins)
- Reporting component

The design and the functionality of those components are described in the following sections.

4.1 Configuration File

The arguments for the program are not specified on the command line. Instead they can be declared in XML-syntax in the configuration file. A central point is the declaration of the IP address of the monitored host. Through this the tracker framework knows which packet to capture on the link because we only want to track connection states of packets going to or coming from the monitored host.
Another fundamental configuration concerns the linkage of the plug-ins among themselves and to the framework. A list with one entry per plug-in is created. Each entry points to a structure wherein plug-in specific information is stored. This information is about the location of the library which implements the plug-in and also about the sub-plug-ins to which the concerning plug-in can pass packages for further analysis. The relationship between the plug-ins is illustrated in Fig. 4.2.

4.2 Dispatcher

The dispatcher is called whenever a packet according to the filter criteria of `libpcap` is received. The task is the detection of the connection, this specific packet belongs to. Therefore it first detects if the network layer protocol is IP. If that is the case, the source and the destination IP address are read. Additionally the `offset` field of the header is analysed in order to detect fragmented packets. Whenever the `offset` is zero the existence of the protocol of the transport

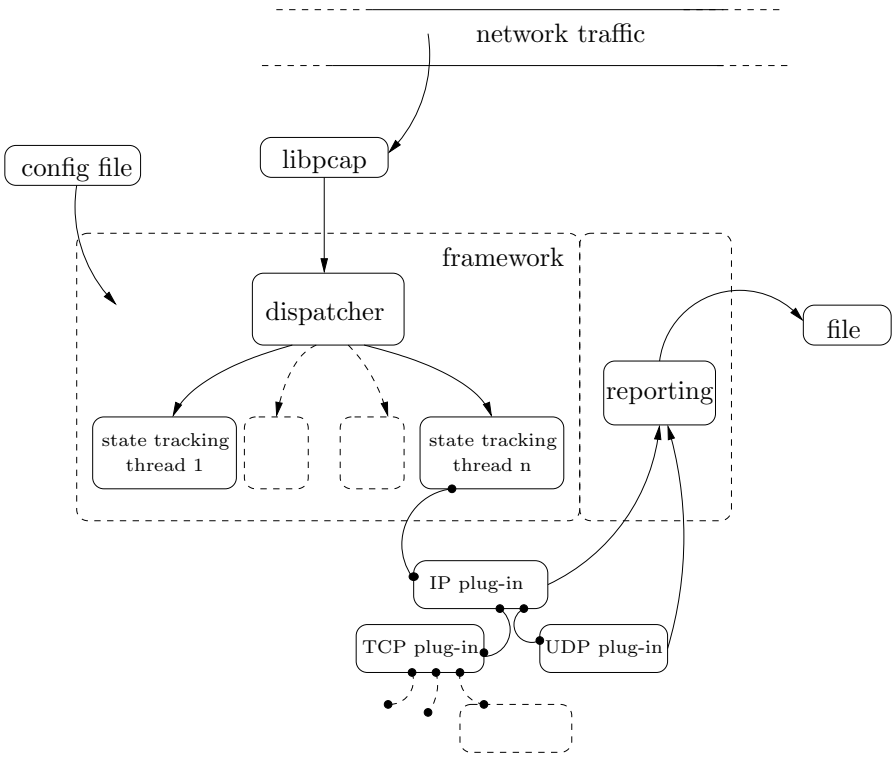

Figure 4.1: Detailed design of the tracker framework

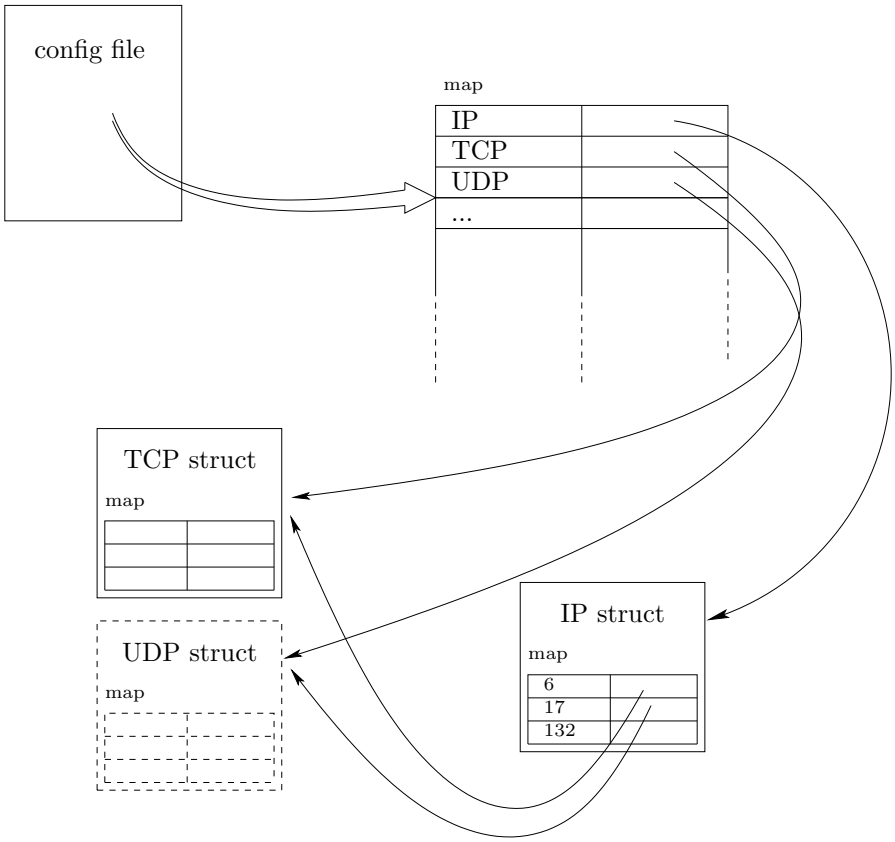

Figure 4.2: Illustration of the import of the plug-in relationship

layer protocol is guaranteed. First the case of not fragmented packets is considered which can then be extended onto the more involved case of fragmented packets.

As the source and destination address are not uniquely identifying a connection, the source and destination port of the transport protocol have to be considered as well. This is at the current state of development done for both TCP and UDP but no other transport protocols. After having unambiguously identified the connection it is assigned a key consisting of the source and destination IP address and the source and destination port. This so called socket key serves the identification of all packets belonging to a transport layer protocol connection. If the socket key is already mapped onto a certain state tracker thread, the packet is sent to the latter. If no thread is associated with the connection it chooses randomly one of the state tracker threads, adds the association to the connection list and sends the packet to the associated thread.
Having a closer look at fragmented packets reveals that a packet with the `offset` flag set to zero could also be the first fragment of a fragmented TCP/UDP packet. Therefore the `MF` bit of the IP header is read after the analysis of the transport layer protocol. If it is set, a mechanism has to guarantee that further fragments are associated with the one which just arrived. These packets can be recognised as their identification number (`Ident`) is identical to the identification number of the first fragment. Consequently a second key consisting of the source and destination IP address and the identification number is built. This key will be named IP-key and as it will allow to reassemble fragmented packets it only will be generated for fragmented packets.
As soon as a packet having a non-zero offset arrives the map with all IP-keys can be consulted in order to find the connection details which have to be associated with the arrived packet. Obviously the problem of unordered arrival of the fragments has to be faced. This is solved straight forward as the IP-key is entered into the corresponding map and the connection details are left unchanged. As soon as the first fragment arrives this data can be collected and will be entered into the designated variables.
All the described mechanisms do not touch the problem of the correct reassembly of fragmented packets. Therefore a means should make sure that a packet is processed only if all fragments arrived. This is involved as the mechanism has to be stable under the condition of duplicates arriving and all fragments arriving at random order. These requirements are fulfilled using a list where every fragment enters its starting and its ending point. As soon as at least the first and the last fragment of an IP packet arrived, the list is checked. This means that the end of one entry has to be identical with the beginning of the following entry. As soon as this attribute holds from the beginning to the end of the list, it is processed and the entry in the IP-key map is deleted.
Further arriving packets will provoke a new entry in the IP-key map. Such entries will be deleted after the expiration of the timeout which can be specified. This behaviour corresponds to the RFC 791 and should therefore be identical to the behaviour of the monitored host. Although when a packet arrives shortly before or after the expiration of the timeout, the difference of arrival between the monitored host and the tracking host might cause unequal behaviour. This is not taken into consideration. [1]

4.3 State Tracking Threads

The main purpose of the state tracking threads is the support of parallelisation. On a multi core architecture several state tracker threads can operate in parallel. The main drawback of the usage of the latter is that the same thread must get all packets belonging to a certain connection. Otherwise the state of a stateful protocol can not be observed correctly. This forces the dispatcher to recognise that a packet from a host A to the monitored host belongs to the same connection as a packet in the opposite direction. This difficulty is considered when building the socket key as described in Sect. 4.2.
A state tracker thread gets a message in its incoming message queue right after the dispatcher has identified the connection that a captured packet belongs to. The packet is taken out from the message queue and then analysed by all plug-ins which are loaded and which are associated with the captured packet. When it has passed all analysis the packet is removed and the state tracker thread is ready for processing a new packet from its incoming message queue.

4.4 Plug-ins

At the current state of the program it supports only the Internet Protocol on the network layer. Therefore all packages that reach the state tracker thread are passed to the IP plug-in. The IP

plug-in analyses the packet and sends the results of that analysis to the reporting thread by putting them into the outgoing message queue of the state tracker thread. Thereafter the packet is passed to the next plug-in which starts the analysis it has been programmed for and then forwards the packet as well.
When a plug-in has no successor plug-in then it has to inform the reporting thread about this fact. This is done by placing a well defined message in the outgoing message queue.

4.5 Reporting

The major task of the reporting thread is to provide one message buffer for each state tracker thread. Whenever a message sent by a plug-in is received the reporting thread looks to which state tracker thread that message belongs to and appends it to the appropriate buffer. Messages that indicate the end of the analysis by all implemented plug-ins cause the content of the corresponding buffer to be copied into a much larger buffer, the main buffer of the reporting thread. Afterwards the small buffer is ready for collecting data of a new packet again. The main buffer collects information packet by packet and writes it periodically out into a file. The idea behind this implementation is that file access is time intensive and should therefore be minimised.

Chapter 5

Measurement Results

At the current state of the project no meaningful tests concerning the overall performance can be made. Nevertheless the resource consumption on a standard machine helps to determine whether the framework itself might be fast enough or not.

5.1 Low Load Measurements

In a first step the performance with low network traffic has been measured. This is sensible as the NoAH project at the current state does not intend to use higher data rates.
The measurements have been made using a desktop computer (Intel Celeron 3.06GHz, 512MB RAM) running a Debian Linux distribution with the tracker framework running. Additionally KSysGuard has been running to log the CPU usage and the packets received. A file of approximately 50MB was transfered from an IBM ThinkPad T43p (Intel Pentium Mobile 2.1GHz, 1GB RAM) to another computer equipped with an Intel Celeron 3.06GHz (512MB RAM) using FTP as application layer protocol. The CPU usage and RAM utilisation of both client and server have been watched during the transfer and they have never been exhausted. All computers have been connected via a hub (Lantronix LTR8T) having a maximum speed of 10MBit/s.
As a single core machine has been in use only one thread was used for state tracking. Running the tests on a machine having several cores and consequently using more state tracking threads has not been possible due to the lack of corresponding hardware.
We chose to investigate the CPU usage as it seemed to be a sensible metric for the resource consumption of the tracker framework. No other processes need significant amount of CPU time as long as the user is inactive. The latter has obviously been assured in order to make the measurements meaningful and reproducible.
As already mentioned a file of 50MB has been transfered from the server to the client. Thereby the CPU load as well as the number of packets have been logged on the computer running the tracker framework. The results of these measurements can be seen in Fig. 5.1. The correlation between the CPU usage and the network load is clearly visible. Furthermore the used bandwidth has a direct influence on the CPU usage. From Fig. 5.2 can be seen that this dependence seems to be linear. This is according to the expected complexity which is proportional to the number of packets. This might change as soon as more elements are implemented. The increased system CPU load during data transfer is due to the fact that the arriving packets have to be processed from the network card into the memory. There they can be fetched by libpcap in order to hand them over to the tracker framework.
In addition to the comparison of network load and CPU consumption the number of packets which have been captured by the framework have been counted. It is necessary to do so as an incomplete trace of packets leads to fundamentally different results at the stage where protocol state tracking is implemented. The number of packets arriving inside the tracker framework has been compared with the number of packets that have been seen by the server and the one that have been seen by the client. As the framework was monitoring the server one might assume that the number of packets seen by the server is equal to the number of captured packets. This is not the case due to packets not using IP as network layer protocol (e.g. ARP) which are filtered at the level of the libpcap library. Consequently those packets can not be measured with the actual configuration of the framework. The results of the measurements are presented in Tab.

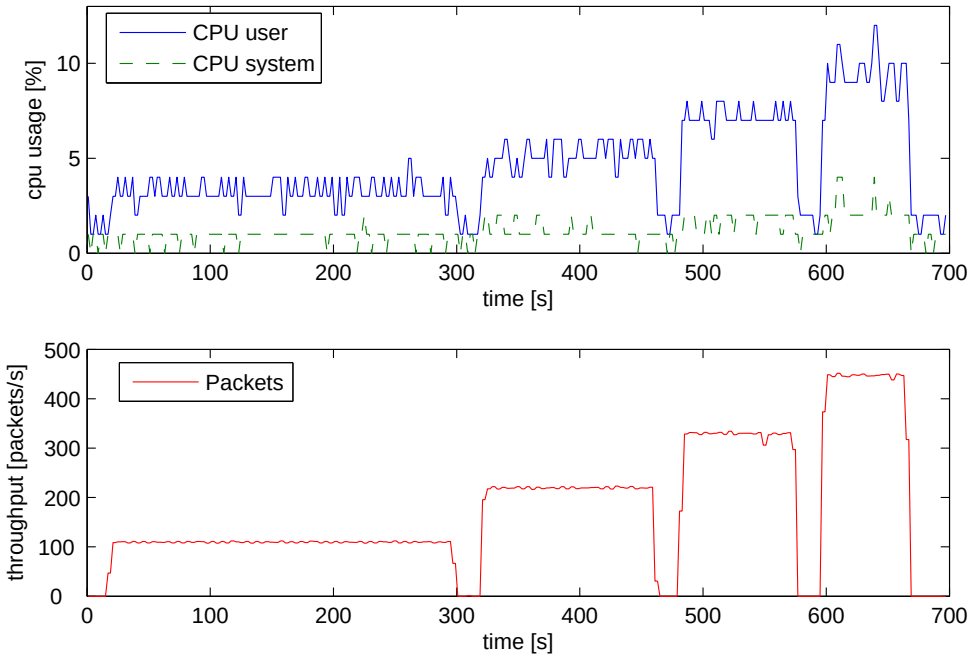

Figure 5.1: Measured CPU load at 2, 4, 6 and 8 MBit/s network load

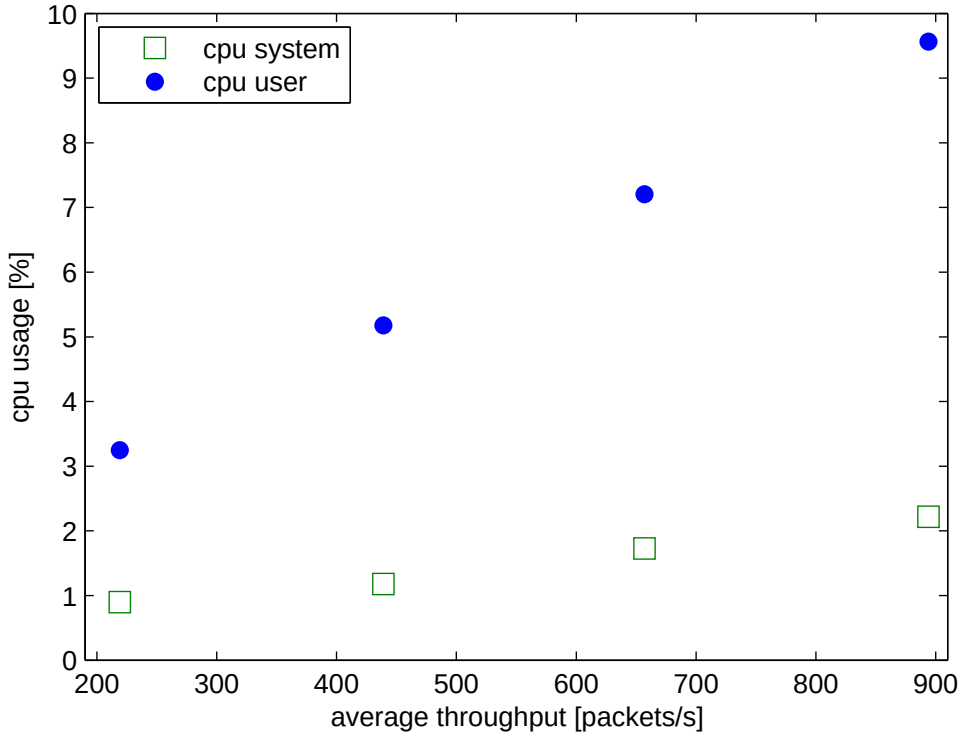

Figure 5.2: Influence of data rate on CPU load of user and system

5.1. During these measurements the bandwidth has been limited such that the CPU usage has never been at its limit.

The same tests have been run using ethereal instead of the framework for counting the packets. As ethereal uses libpacp as well it should have a comparable number of packets which are not

series	*counted by framework*	*counted by server*	*difference*
1.	126461	126473	12
2.	146191	146201	10
3.	140999	141007	8

Table 5.1: Comparison of the number of packets counted with the framework and at the network card at low data rates.

seen. The results of these measurements are gathered in Tab. 5.2. The percentage of not received packets using the tracker framework accounts to 0.0073% whereas ethereal does not receive 0.0075% of the packets.

series	*counted by ethereal*	*counted by server*	*difference*
1.	108737	108747	10
2.	119072	119079	7

Table 5.2: Comparing the number of packets counted with ethereal to the network card at low data rates.

5.2 High Load Measurements

The setup for high throughput measurements has been entirely different. This was necessary as the hubs only worked up to a speed of 10MBit/s. Not only the hub was replaced but also the two desktop machines. Instead there was an IBM ThinkPad T42 (Intel Pentium Mobile 1.8GHz, 1GB RAM) running Debian Linux for monitoring purposes and as client served an IBM ThinkPad X40 (Intel Pentium Mobile 1.4GHz, 1GB RAM) running Windows XP. The server still was the IBM ThinkPad T43p but those computers have been connected using a managed switch (Zyxel ES-2108 Dimension).
Figure 5.3 shows that the findings are very similar to the low throughput case. The increase in number of arriving packets lets the CPU usage increase linearly.

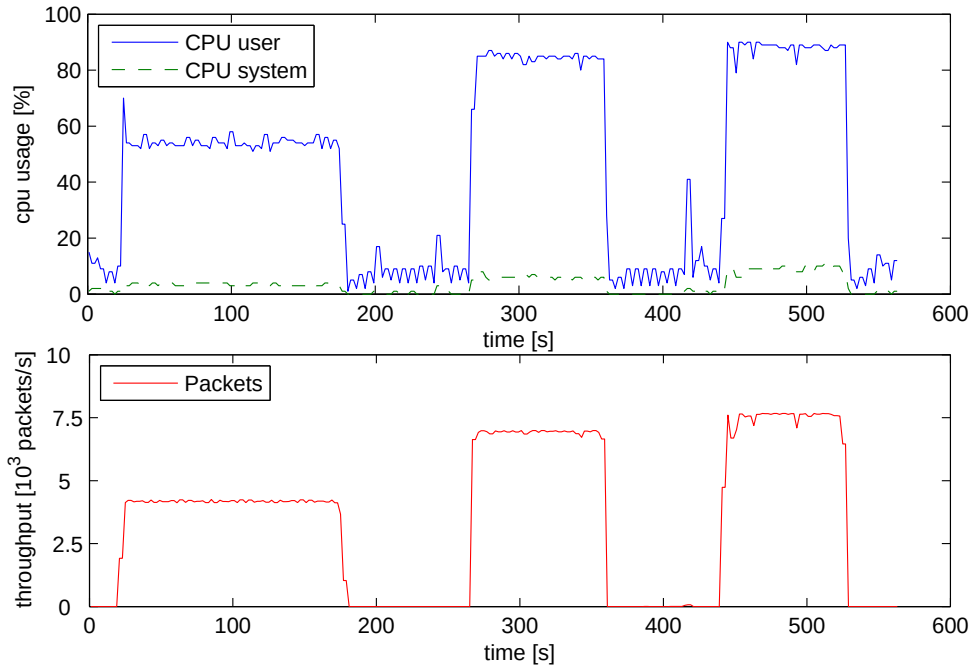

Figure 5.3: Comparison between CPU consumption and network load.

In order to be able to track the state of the monitored host, it is mandatory as mentioned before that all packets which arrive at the latter also are captured by the computer running the tracker framework. Therefore the number of packets have been counted using the same setup as given above. The results, which are given in Tab. 5.3, clearly show that the number of packets not seen inside the framework can not be due to packets using another protocol than IP. The percentage of unseen packages lies between 48% and 53%. Clearly this result would have to be confirmed in order to acheive reliable numbers but it is obvious that state tracking in this configuration would not be possible.

speed	*counted by framework*	*counted by server*	*difference*
3550kBit/s	486715	1041127	554412
4680kBit/s	491073	1044375	553302
5070kBit/s	543003	1040835	497832

Table 5.3: Comparison of the number of packets counted with the framework and at the network card at high data rates.

As the setup between low data rate and high data rate measurements has changed significantly it appears to be necessary to count the packet numbers using the high data rate equipment but limiting the data rate. The limit should be somewhat below the rates that could be considered in the low data rate setting. Doing this and using the tracker framework as well as ethereal to count the packages leads to the values given in Tab. 5.4.

speed	*counted by framework*	*counted by server*	*difference*
893kBit/s	734214	1015793	281579

speed	*counted by ethereal*	*counted by server*	*difference*
893kBit/s	772877	1005505	232628

Table 5.4: Packet counts using low data rates in high performance setting.

Even at this rate more than 20 percent of the packets can not be counted within the tracker framework. Obviously ethereal experiences the same problem which leads to the assumption that the difference is not due to the implementation of the tracker framework. This result is at least partially due to libpcap which is not able to deliver all packets when using data rates higher than 10MBits/s. [9]
Another cause could be the used hardware. As already mentioned the host running the tracker framework as well as the switch have been exchanged. The switch used one 100MBit/s port for monitoring while the communicating hosts were using two 100MBit/s ports. The possible peak rate for the monitoring port could possibly reach 200MBit/s which is clearly not supported.

Chapter 6

Summary

The measurements clearly show that the implemented framework running on an average computer is able to capture all packets. This is evident when a protocol state tracker should be implemented upon that framework. As soon as the network load gets higher there occur problems with packets that are not reaching the inside of the program. This is clearly not tolerable but there is a high probability that libpcap is the cause of these losses. This statement is supported by reference measurements with ethereal. One possible solution to this problem can be found in [9].
Different hardware should be used in further measurements to eliminate the possibility of malfunctioning hardware. More extensive testing would lead to more data and more reliable information about the possible cause of the huge packet loss.
The CPU load of the program is not even close to be exhausted when running at a speed where no packets get lost. This means that further processing is not considered to be a problem. It has to be verified whether packets get lost when they arrive at a time when the CPU is exhausted. If this assumption can be proved to be wrong there will even be less problems. Upcoming hardware even is mostly equipped with several cores which could yield to better performance due to the architecture of the framework which is designed to exploit multi threading.
The multi threaded approach at first seems to guarantee higher performance when using appropriate hardware. At second thought the dispatching thread, deciding to which thread a connection should be assigned to, has a delicate role. In addition to the complexity of the task it is time consuming and a packet might get lost at this point when the libpcap buffer is overwritten. This assumption however has not been investigated. Moreover the design is not regarded as the packet has to be examined in more detail before being able to pass it on.
It should be taken into consideration that not only the threads lead to a certain overhead but similarly the dynamically linkable libraries and the message queues do. The latter are necessary because of the multi threading approach. Measurements with a framework not using any of those techniques would be interesting to draw a conclusion about the impact of the whole overhead. A lightweight approach without exploiting multiple threads for state tracking is assumed to have better performance when the monitored host is running at low network speed. Certainly the actual approach has its strength when the framework runs at very high speed using a multi core computer. Another sensible variant would be a framework using threads and analysing the content of network and transport layer before assigning a packet to a thread. In comparison to the given approach this would mean that the IP and TCP/UDP plug-ins would be integrated before the dispatcher. Subsequently the content of these headers would not have to be analysed twice. The plug-ins could be used for state tracking of application layer protocols.

Chapter 7

Outlook

The first thing that needs to be done is testing the fragmentation part within the current framework. To complete this task successfully it is evident to read into the code as well as to have enough hardware that can be configured freely. Two methods that seem equally time intensive are the construction of packets or the setup of an environment using past technology. Most current software exploits the `DF` flag and avoids fragmentation in the first place.
The most important thing that has to be done is the implementation of the TCP plug-in with a protocol state machine. When implementing this state machine the one from `iptables` might be of interest. To consider is the fact that the tracking host does not have the same timing as the monitored host. Hence it does not know the exact arrival time and when it comes close to timeouts there are "theoretical" states reached. This means that the tracking host can only decide about the state of the monitored host when the response has been seen. Obviously this leads to extensions that have to be made when starting from the `iptables` state machine.
After the implementation and testing of the TCP state machine the testing of the latter is evident as it forms the basis for many application layer protocols. Consecutively development of further plug-ins can be considered, always keeping an eye on performance.
Another enhancement that could be made, as soon as performance is a problem, is the development of a technique to assign the connections to certain threads. When doing this, fundamental properties of Ethernet traffic (such as long-range dependence) need to be considered. Possibly even changing an assigned connection to another thread can be done. This would however need a substantial understanding of the current implementation.
On the measurement side it would be interesting to compare how a computer with several cores can exploit the current implementation. Especially interesting would be to use more than two state tracking threads and offer enough cores for all threads to run. This scenario however only makes sense if there are many different TCP connections as otherwise all packets have to be dealt within the same thread.

Appendix A

Timetable

Week	*Task*
1	Read: IP, TCP, UDP, Linux Sockets;
2	Configuration of the system: Linux, NXServer, Eclipse, SVN
3	Further reading: C++, Doxygen, Ethereal,
4	TCPDump, libpcap, Metasploit
5	Feasibility study: Plug-In architecture, XML, packet capturing (libpcap)
6	create design
7	define interface (dispatcher to plug-in, local host to remote host)
8	implement framework
9	
10	implement tracker
11	
12	test, last improvements
13	
14	presentation

Appendix B

Problem Statement

B.1 Introduction

Today intrusion prevention systems (IPS) are widely deployed to protect IT infrastructures from malicious activities. Most IPSes use signatures, which describe some characteristic of a malicious activity, to detect and prevent these activities. Basically, signatures can operate on two levels, the network level and the host-level.
Network-level signatures are much easier to deploy and to distribute than host-level signatures, and thus widely used today. However, the drawback of network-level signatures is the high rate of false positives they cause. This is due to the fact, that e.g. malicious byte patterns also frequently occur in legitimate traffic.

Protocol Awareness

A way to reduce the false positive rate of network-level signatures is to implement protocol awareness. This approach is widely used in commercial applications. Also some open-source IPS allow for protocol awareness since they support regular expressions [10]. Protocol awareness can provide a lot of useful information (e.g. protocol states, field lengths, and field types) for generating more accurate and meaningful signatures. The goal of this work is to provide information about protocol states for individual connections to ameliorate signatures generated by a honeypot system.

NoAH and Argos

NoAH is a European project which has the goal to develop an infrastructure for security monitoring based on honeypot technology. Honeypots will be used as early-warning systems capable of detecting attacks at the early stages of their infestation. In NoAH, honeypots are normal PC's running OSes such as Windows or Linux, and services like webservers or ftp-servers. The trick is that the OS runs within a containment environment called Argos [1] [12]. This containment environment serves mainly two purposes: First it contains the effects of an attack, and second it provides some information about an attack for signature generation. However, for generating more accurate signatures it would be helpful to have a tool that tracks the protocol state of all connections to the honeypot and provides this information to Argos.

B.2 The Task

A protocol-state tracker will be developed and tested. This includes tracking protocol states for TCP, and at least one additional application layer protocol (e.g. DCE/RPC). The task is split into the following subtasks:

1. **Understanding TCP and DEC/RCP protocol state machines:**
   Study the relevant RCFs to get an understanding for the protocols that are to be tracked.

[1] Argos runs the operating system together with its services inside the x86 emulator QEMU [11] in order to supervise their execution.

2. **Design and Implementation of the tracker framework:**
   This includes the following parts: a connection tracker module for TCP, a connection tracker module for one application layer protocol, and a simple user interface. All design decissions should be taken with extensibility, scalability, and usability in mind.

3. **Testing and Evaluation:**
   Test the developed prototype for correct operation. This includes testing the prototype under high loads. Moreover, a demonstration with a real exploit (malicious payload removed of course) will serve as proof-of-concept.

B.3 Deliverables

During this thesis the following deliverables will be produced:

1. Brief description of TCP and application layer protocol states
2. Detailed documentation of the tracker design
3. A running prototype of the tracker framework including a TCP and application protocol tracker module, and a simple user interface

Further optional components are:

- Implementation of additional application protocol tracker modules

Supervisors

Bernhard Tellenbach, tellenbach@tik.ee.ethz.ch +41 44 632 70 06, ETZ G97
Daniela Brauckhoff, brauckhoff@tik.ee.ethz.ch, +41 44 632 70 50, ETZ G97

Bibliography

[1] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[2] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[3] Andrew M. Rudoff W. Richard Stevens, Bill Fenner. *UNIX Network Programming: The Sockets Network API*, volume 1. Addison-Wesley, third edition, 2004.

[4] Thomas F. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. Charles River Media, INC., first edition, 2004.

[5] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[6] D. A. Wheeler. Program library howto, dynamically loaded (dl) libraries, 2006. [Online; accessed 22-November-2006].

[7] B. Blaise. POSIX threads programming, 2006. [Online; accessed 25-November-2006].

[8] D. Marshall. Programming in C, UNIX system calls and subroutines using C, 2006. [Online; accessed 25-November-2006].

[9] L. Deri. Improving passive packet capture: Beyond device polling, 2006. [Online; accessed 24-November-2006].

[10] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[11] Fabrice Bellard. Qemu. http://fabrice.bellard.free.fr/qemu/, 2005.

[12] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.