

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Diplomarbeit

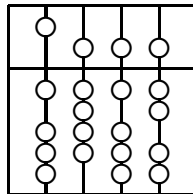
High-Performance Packet Recording for Network Intrusion Detection

Stefan Kornexl

Aufgabensteller: Prof. Anja Feldmann, Ph.D.

Betreuer: Dipl.-Inf. Holger Dreger
Dipl.-Inf. Robin Sommer

Abgabedatum: 17.01.2005



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

München, 17.01.2005

(Stefan Kornexl)

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Structure of Thesis | 2 |
| 2 | Background | 3 |
| 2.1 | Intrusion Detection | 3 |
| 2.2 | Need for Packet Recording | 5 |
| 2.3 | Related Work | 6 |
| 2.4 | The Open Source NIDS Bro | 7 |
| 3 | Understanding the Workload | 9 |
| 3.1 | Environments | 9 |
| 3.2 | Traffic Data Volume | 11 |
| 3.3 | Volume Analysis Based on Connection Level | 12 |
| 3.3.1 | Data Sets | 12 |
| 3.3.2 | Procedure | 14 |
| 3.3.3 | Results | 16 |
| 3.3.4 | Connection Size Cut-Off | 18 |
| 3.4 | Conclusion | 19 |
| 4 | Design | 21 |
| 4.1 | Requirements Review | 21 |
| 4.2 | System Architecture | 22 |

| | | |
|----------|---|-----------|
| 4.3 | Packet Capture | 23 |
| 4.4 | Classification | 25 |
| 4.5 | Storage Containers | 27 |
| 4.5.1 | Structure | 27 |
| 4.5.2 | Storage Format | 28 |
| 4.5.3 | Memory Cache | 29 |
| 4.5.4 | Disk Cache | 29 |
| 4.5.5 | Connection Tracking | 30 |
| 4.6 | Indexing | 32 |
| 4.7 | Processing Queries | 34 |
| 4.8 | User Interface | 36 |
| 5 | Implementation, Status and Experiences | 38 |
| 5.1 | Implementation | 38 |
| 5.1.1 | Programming Paradigm and Language | 38 |
| 5.1.2 | Operating System | 39 |
| 5.1.3 | Technical Structure and Concurrency | 40 |
| 5.1.4 | Status of Implementation | 42 |
| 5.2 | Evaluation | 43 |
| 6 | Conclusion | 46 |
| 6.1 | Summary | 46 |
| 6.2 | Future Work | 46 |
| | Bibliography | 48 |

Chapter 1

Introduction

1.1 Motivation

Today, hosts that are connected to the Internet are steadily targeted by attacks that aim to subvert the integrity of the systems and data. Information theft is a growing problem. Organizations suffer from financial or other loss due to attacks on computer systems.

Strategies to improve security have become crucial. Among other techniques, Network Intrusion Detection, i.e. the monitoring of a computer network for signs of events relevant to security, is an important approach towards security. Network Intrusion Detection Systems (NIDSs) use various ways to examine traffic captured from the network for anomalies. Such an anomaly generates an alarm by the NIDS. This may lead to a number of actions. A message may be generated and sent to security personnel. A firewall may be modified to protect a network against further possibly malicious access from outside. And the NIDS may record a summary of the event, including time of day, IP addresses involved, etc.

Practical use of NIDSs for security monitoring has shown that there is a need for as much information as possible about the security-related events that have occurred. The availability of packet-by-packet records of the network traffic related to a given attack would allow for its in-depth scrutiny and exact documentation of an incident for purposes of forensics. Additionally, the NIDS itself may take advantage of the possibility to get the network data stream from a point further back in time. This might be useful for analysis of past events that the NIDS has not considered “interesting” until more recently seen traffic hinted at their relevance. Thus, the availability of packet records is considered a big benefit for network security monitoring.

Unfortunately, there is no system that aims at the provision of such packet-by-packet records in the intrusion detection context, and we do not know of research work being done in this particular area. The objectives of this project are to design and implement a packet recording system that is able to store network traffic and to answer common kinds of queries for stored data. It should be able to operate at today's network speeds. This requires a set of techniques and careful design that mitigate the load on the recording system. We have to analyze the nature of the data to be recorded and devise mechanisms that allow to store different kinds of traffic for different periods of time. Then, the system needs to provide for retrieval of selected subsets of the stored data.

In a sense, if the Intrusion Detection System compares to the sensors and control devices of a burglar alarm system, the subject of this work are the closed-circuit television cameras, the choice of their placement considering their range of view, their control by the alarm system and the automated archival and retrieval of the interesting episodes on the recording tapes.

In this thesis, we describe the design and implementation of a packet recording system allowing for classification of traffic, class differentiated storage in RAM and on disk for user configurable periods of time, and indexing and retrieval of stored packet data. We designed an interface to the Open-Source NIDS Bro and give examples of configuration of the prototypic implementation to run in a production environment on a fairly high-loaded 1 Gbit/s Internet link of a large university network.

1.2 Structure of Thesis

This thesis is structured as follows. In § 2 we give an overview of intrusion detection in general and present concrete systems; we illustrate the problem addressed in this project and point at some related work. § 3 deals with the characteristics of the environments that form the premises for further work. The design we devise for a solution to the problem is presented in § 4. We describe an implementation of that design and results of its use in § 5. Finally, § 6 summarizes the results and suggests future issues for this matter.

Chapter 2

Background

2.1 Intrusion Detection

The term *intrusion detection* refers to the task to detect attacks against computers and computer networks. An Intrusion Detection System (IDS) aims to detect anomalous activity on a host or network that hints at attacks. There are different kinds of intrusion detection. IDSs running on a host to detect anomalous activity on that host are called host-based IDSs, whereas IDSs that operate on a network and monitor network traffic to detect anomalous activity on the network are called network-based IDSs.

Host based intrusion detection requires a lot of complexity, since it has to be provided for in every host that has to be monitored. It has its own advantages, for example more insight into traffic if it is being encrypted. Network based intrusion detection on the other hand is being widely deployed as an integral part of security measures in production networks today and is being actively researched. Network IDS (NIDS) has a number of advantages. The possibility to place it at a central point in a big network, where it can monitor all the traffic entering the network, allows to cover all the hosts in that network without the cost of deploying intrusion detection mechanisms on all the hosts. A NIDS monitors the live network traffic for attack detection. Once suspicious activity is detected, the record of this is stored in the NIDS and can not be altered by a successful attacker. On a host based IDS, on the contrary, the IDS and its records may be modified by an attacker who has compromised the host in order to remove evidence of the attack. Another advantage is the NIDS's independence from the hosts to monitor. Its operation continues properly, even if a host has been successfully compromised; in that case the further correct operation of a host based IDS can not be guaranteed. Combinations of host based and network based systems are also being studied.

For this work we focus on network based IDS. NIDSs are placed at a point in the network and equipped with network interfaces that allow them to capture all traffic that is subject to monitoring. Monitoring aims to detect anomalous activity on the network. Network intrusion detection approaches can be divided in two categories.

The *pattern matching* (or *signature based*) approach detects anomalies on the network by comparing the data stream to a set of signatures. Signatures are developed independently from the IDS itself, either by the vendor or by the user (or community of users), as new types of attacks are identified. Traffic is considered to be of interest if it matches a signature. This can be a string signature that matches the payload of packets, such as suspicious user account names like “root” in the *telnet* or *ftp* applications or suspicious commands in an *smtp* session. A drawback of signature based intrusion detection is that *false positives*, i.e. alerts that do not base on a real security incident but are due to an error are quite prevalent. This is because legitimate traffic at many points matches the signatures by coincident. Even worse is the fact that a signature based IDS is only able to detect attacks that are covered by signatures. A newly devised attack will not be detected by the signature based IDS unless a signature for the new attack is added to the IDS. So a signature based IDS is not able to detect previously unidentified types of attacks. The quality of signature based IDS depends on the quality of the signatures. If they are not crafted carefully enough, the number of false positives, as well as of *false negatives*—i.e. an attack that is not detected and signaled—rises.

Another approach to detect attacks is a policy based strategy. It involves the IDS to track detailed information about the activity on the network. This allows for more complex examination of the traffic than simple signature matching, and enables the user to express more sophisticated rules that decide what traffic is to be considered suspicious. Using more generic rules that do not depend on the signature of a concrete attack, it is possible to detect previously unknown attacks.

Whenever an IDS detects an attack or activity suspected to be an attack, it presents the information about it in some way. This includes a record in a log file or an alert that may be posted to security personnel. The IDS may even respond to the attack by blocking further access to the protected network for remote hosts suspected of malicious activity. Such a mechanism is termed a *reactive firewall*.

A number of vendors offer IDSs. For research purposes, however, open-source systems are of much more value since their principle of operation is known and they offer rich experimentation, development and evaluation possibilities. There are a few open-source NIDSs, which we will present in the following paragraphs.

Snort was designed to be a lightweight, easy-to-use and cheap system for intrusion detection for small to medium size networks, as an alternative to commercial

systems [Roe99]. It only provides pattern matching detection. The number of signatures that are provided for the use with snort is rather big and gives a wide coverage of common attacks. The user community of snort is fairly large and a great number of signature rules are made available in a timely fashion as new attacks are identified. However, snort suffers the inherent drawbacks of signature based intrusion detection as mentioned above.

Bro [Pax99] features a sophisticated policy based approach and additionally allows for signature based matching. Bro offers much more potential than snort does, however it is in less widespread use. At the time of this writing, Bro is mostly known and being used in the research community. A considerable effort is being put in the development of Bro towards a system that is better documented and easier to use with less expertise. We decide to use Bro for our work because it offers great flexibility and strength and it is open source. Besides it is being co-developed by our research group and run in our network. We will give an overview of Bro in § 2.4.

An IDS may generate an alarm and a log file entry when it detects activity on the network that is deemed relevant from the security perspective. This prompts security personnel that possibly malicious activity took place. However, the record of that event is not as comprehensive as users of IDSs typically want them to be. When a security incident happened, the information about its occurrence and a few general data from an IDS log file like type of attack and the IP address of the suspected attacker are not sufficient for in-depth security monitoring. We consider the availability of a trace of the packets on the network that were related to the attack priceless. Unfortunately, in general they are not available because there are no systems that provide for the recording of packets and are able to extract the relevant data from the records according to intrusion detection requirements.

2.2 Need for Packet Recording

Our experience of the use of IDSs to detect network intruders in productive networks has shown that in the case of a relevant security incident—for example a successful attack—there is a high demand in as much information about the past incident as possible. We want to know exactly about “when, how, who, what”. If an attack on a host has been successful, we most certainly are interested not only in **how** the attack was carried out but also in **what** happened after the attacker gained control over the machine and the data stored on it. The exact record may even be important for the prosecution of an attacker. In this context the maintenance and use of such records is called forensics.

Another use case for a packet recording system involves the IDS itself. [SP04] hints at the possibility of an IDS making use of the availability of past traffic for augmented possibilities of analysis. The idea that the recording system is able to provide the IDS with traffic from the past—whereas the IDS experiences a temporal order on the packets during regular operation—leads to the nickname “time machine” in their work.

Bro, as well as snort, is able to write packets to a trace file on disk. In Bro, for example, the user may specify that for a part of the traffic either no information, or the headers of the packets or the whole packets will be logged. However, packets are written to disk only after they are identified by the detection mechanism. This is not sufficient because it does not give access to the interesting portion of traffic. So there is a need for a system that enables the access to past network traffic by storing it in some cache.

Such a caching system needs to allow a certain degree of flexibility. For example, we want it to be able to incorporate run-time configuration commands. This may be leveraged for longer term storage of traffic subsets that have been identified as belonging to an attack by the IDS. If the IDS detects an attack, it could send a command to the recording system to alter the storage rules accordingly. We envision a system that cooperates with the IDS.

2.3 Related Work

To our knowledge, there is no existing project dealing with the objectives of this work. Neither did we find any publications on this subject. However, there are some loosely related commercial products we took a look at and want to summarize here.

A commercial product that is related to the subject of this project is the McAfee Security Forensics appliance [McA04]. It features massive storage and claims to be able to write sustained Gbit line speed to its disks. The user has access to the stored data by a GUI. The appliance is based on a proprietary hardened Linux operating system. This product follows what we call a “brute force” approach. It is not designed to use sophisticated methods to store traffic in an intelligent way that allows for a more lightweight solution, but seems to be relatively inflexible. And, due to the commercial nature, it is “closed source”, so there is no insight into its operation and no way to modify it according to individual site’s needs. There is no communication interface to an IDS, it is only designed to be a stand-alone system.

A similar product is the Niksun NetDetector [Nik04]. It, too, comes as an appliance and it is based on FreeBSD. Likewise, it lacks openness, flexibility and integration with intrusion detection.

There are numerous reports about research in the intrusion detection field. However, we did not find any that touches the subject of this work. Sometimes caching is an issue, but it is not put in the same light as the project at hand does. Our goals are clearly more comprehensive than any of the related work we know of.

There are a few attempts to provide solutions for network security forensics. To summarize, we believe that the idea as expressed for the problem of this project is rather new.

2.4 The Open Source NIDS Bro

Bro [Pax99] is a NIDS that was developed by Vern Paxson at the Lawrence Berkeley National Laboratory (LBNL) and International Computer Science Institute (ICSI), Berkeley, CA. It is being actively enhanced by a number of researchers worldwide.

Bro uses its own script language to describe how it should behave. This configuration has to be adapted for the site that Bro is to run at to reflect the site's security policy.

Figure 2.1 illustrates Bro's system design with respect to the flow of data. Packets captured from the network are processed by an event engine which performs protocol analysis on them and generates a higher-level stream of *events*. Events are information about activity on the network that is more abstract than the actual packets' contents. These are handed to the policy script interpreter which executes the user-supplied policy scripts. They contain *handlers* for the events that, upon their call, do the analysis on a higher level and initiate further action, for example generate notifications. The details of how this analysis is performed is specified by the user through the policy scripts; for example, threshold levels for some kind of activity to raise an alarm are defined in policy scripts.

This design aims at effective processing of the data stream from the network. The event engine processes the greatest volume of data, applying relatively inexpensive operations to the single packets. This part of the system is written in C++ for maximum performance. The event stream generated by the event engine is reduced in volume, so the policy script interpreter has to process less volume and is thus able to perform more complex operations. These are written in the Bro script

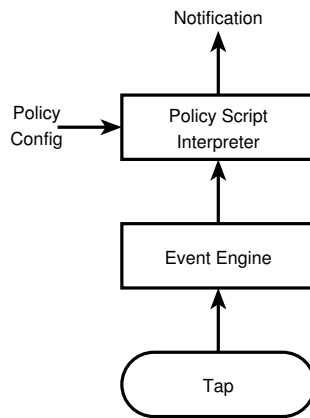


Figure 2.1: Data flow in Bro

language for maximum flexibility; the policy script language allows the user to define any aspect of the analysis. This design approach promotes a clear separation of the policy configuration from the core mechanisms of analysis.

Further action following the detection of suspicious activity is performed in the policy script interpreter. This includes the creation of new events. Events may be sent over a network by Bro's communication protocol. This feature is of special interest for our project and we will make use of it for communication of the recording system and Bro (see § 4.8).

Chapter 3

Understanding the Workload

Given the goal to run in today's high-speed environments, the packet recording system needs to be carefully designed not to exceed today's computer hardware limits. The amount of data itself is a significant problem; it easily exceeds what can be handled using commodity hardware.

To get a feeling for the nature of the traffic we have to deal with, and to get ideas how to solve this problem, we need to analyze the workload traffic. This chapter describes the environments we used to gather sample traffic data and presents general considerations on traffic volume as well as the methods and results of our analysis. Finally, we draw a few conclusions for the feasibility of our intention and for design issues.

3.1 Environments

The prototype implementations in this work were to run on the link from MWN (Münchener Wissenschaftsnetz) to the Internet. We analyzed traffic data from two additional sites, LBL and NERSC, to get an image of the needs of other sites. All of these sites are running the Bro Intrusion Detection System. Following will be a short description of the sites.

MWN

The Münchener Wissenschaftsnetz (Munich Scientific Network), in Munich, Germany, connects all of the Munich universities and a number of scientific institutions and provides connectivity to the Internet by a 1 Gbit/s link. It comprises

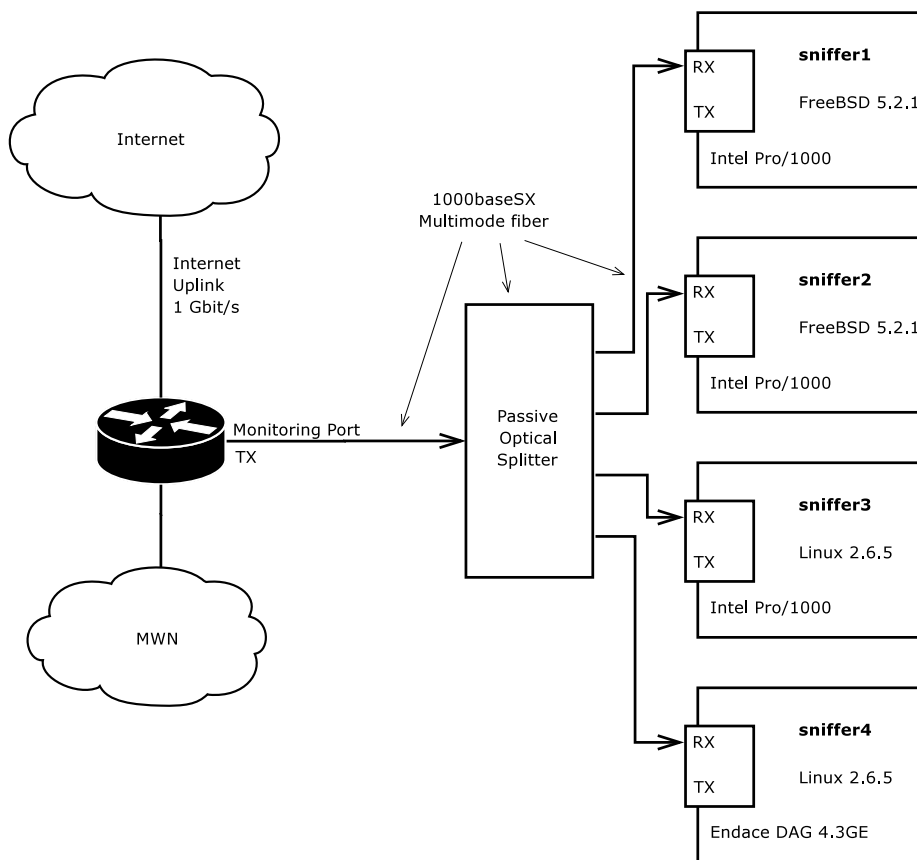


Figure 3.1: MWN sniffer lab setup

around 50,000 hosts and 65,000 users. On average during a weekday, the load on the Internet link is around 200 Mbit/s and about 44,000 packets per second are transmitted. During busy hours, it is typically loaded above 350 Mbit/s hourly average, with more than 70,000 packets per second.

Among others, the Technische Universität München (TUM) and its departments are connected to the MWN. The TUM Department of Computer Science runs a very popular FTP server, *ftp.leo.org*. It offers a wide range of popular software, including download mirrors of Linux distributions and it generates roughly 0.5-1 TB of traffic a day from MWN to the Internet.

The Computer Science Department of TUM is running a few sniffer hosts on the MWN Internet up-link. In this sniffer lab, all of the sniffer hosts get the same packets from a monitoring port of the border router through an optical splitter device (see Figure 3.1).

LBL

Lawrence Berkeley National Laboratory in Berkeley, California, USA is a U.S. Department of Energy (DOE) National Laboratory managed by the University of California. It conducts research in a wide range of fields.

In the Berkeley Lab network, there are about 6,000 hosts and 4,000 users. LBNL is connected to the Internet by a Gbit/s up-link, with an average load of 320 Mbit/s and 37,000 packets per second.

NERSC

The National Energy Research Scientific Computing Center, Oakland, California, USA is one of the National User Facilities of LBNL. It is a scientific computing facility providing supercomputing resources to researchers around the US.

There are around 600 hosts and more than 2000 computational scientists working with the facilities. The nature of this site's traffic is quite different from the two other's. The presence of supercomputers in the network leads us to suspect large file transfers and less traffic due to interactive applications like WWW.

On average, the Gbit/s NERSC link carries 260 Mbit/s and 43,000 packets per second.

3.2 Traffic Data Volume

The goal for this work is to maximize the chance that an “interesting” part of the past traffic will be available in the system's cache for as long a time as possible. Because of the limitations due to limited available storage space, it will be necessary to reduce the amount of data to store in some way.

Considering a 1 Gbit/s link which would be loaded at 30% on average, we would get 37.5 MB per second by sniffing on that link, or 135 GB per hour, 3.24 TB per day.

A “brute-force” approach to the problem of traffic data recording would be to store every packet in a cache, without distinguishing packets. When the cache runs out of space, it evicts oldest packets first to be able to store new ones. We consider commodity PC hardware today to be typically equipped with the orders of magnitude of 2GB RAM and 500GB disk space. It seems that the “brute-force” approach—while it is easy to implement and ensures that no part of the traffic is lost—is not very promising, since the strategy as shown would give us a cache lag

of at most a few hours. We aim to enlarge that period of time for traffic that is considered “interesting”.

In addition, it might not be possible to write that big amount of data to disk sufficiently quickly. If a 1 Gbit/s link is loaded at 75%, it transmits 93.75 MB/s. It may be that certain hardware—and software parts like drivers, input/output parts of the operating system—is not able to cope with such a high volume of data to be written to disk. Uncontrollable loss would be the result of this failure. Here again, we aim to find a viable solution by reducing the data stream.

In order to be able to find out how to accomplish these goals, we need to gain more fine-grained insight into traffic and of what it consists.

3.3 Volume Analysis Based on Connection Level

The approach to store all data in a cache seems not to be the most elegant way, as we discussed in the previous section. We aim to find a solution that discards some of the traffic in favor of the more efficient handling of the remaining, based on a sensible decision.

We conducted an analysis of the traffic by looking at statistics that were generated by instances of Bro running at the three sites MWN, LBNL and NERSC. We have access to log files from all three sites containing traffic volume statistics and use them to gain insight into the traffic, with the granularity of connections in mind.

3.3.1 Data Sets

Bro can be configured to generate *connection logs* during its normal operation. Every session that Bro sees on the network will then be logged to connection log files.

The latest version of the policy script that enables this logging generates ASCII log files consisting of one summary line for each connection, with 13 fields in every line. The fields, in the order as written to the file, contain the information as shown in table 3.1.¹

Note that we only have summary information about the connections, so an analysis can only be an estimation of the real situation. For example, we do not have

¹Some of the files to our availability were written by older versions of Bro in a slightly different format. Particularly, there was a notion of local host, remote host and locally or remotely initiated connection. The new format stores originator and responder for a more clear output. The old formats were converted to the new ones before being analyzed.

| <i>Field name</i> | <i>Description</i> | |
|-------------------|---|------------|
| start | timestamp of first packet for this connection | T_0 |
| duration | duration of connection | ΔT |
| orig_h | connection originator's address | |
| resp_h | connection responder's address | |
| service | the connection's service (e.g. smtp, http, IRC, other) | |
| orig_p | port number of the originator | |
| resp_p | port number of the responder | |
| proto | protocol (TCP or UDP) | |
| o_bytes | number of bytes sent by originator | s_o |
| r_bytes | number of bytes sent by responder | s_r |
| state | for TCP, gives the connection's final state (such as normal establishment and termination [SF], or rejection [REJ]) | |
| flags | indicate if connection was initiated locally or remotely | |
| addl | some additional information, may be empty | |

Table 3.1: Bro connection log file structure

information about when actually data is transmitted, we only have the total number of bytes transmitted by the originator and the responder of the connection, respectively (s_o and s_r).

There are some pitfalls to watch out as to the data in the connection summary logs. Due to reasons internal to Bro which we will not further discuss here, sometimes there are weird numbers, for example there might be a connection with a negative duration, or transmitting at an unrealistic bandwidth rate. For our analysis, we take care of these and simply skip them in order not to let them spoil our results. We consider the damage to a realistic estimation by these weird connections a much greater threat than the error introduced by omitting them. (Most are very small ones in terms of data transmitted, anyway.)

There is a problem with UDP traffic as to our analysis of the connection summaries. Although there is no UDP connection state, Bro has a concept of UDP sessions. UDP packets are considered to be either requests or replies to a request, and a connection summary can be written accordingly. However, the UDP analyzer in Bro's event engine exhausts state space very quickly due to a lack of a proper "connection" termination concept, which is inherently not provided in UDP. As a result of this, the UDP analyzer is usually not activated in Bro configurations. As to the connection log files we have at hand, only the NERSC logs do include UDP summaries. We have to take this fact into account when looking at the results of the load evaluation.

3.3.2 Procedure

We want to take a look at the connections we see on the networks. We are interested in the data volume a recording system would have to keep over time. We want to look at a method that allows to lower the amount of data and experiment with the parameters that determine the decision of what data to store.

For the procedure of this analysis we assume a cache that can keep any amount of data, structured in network packets. Each of this packet belongs to a connection. A connection is defined by its parameters, the five tuple of transport protocol, source and destination addresses and source and destination transport protocol port. We simulate the cache by calculating the volume of data in the cache over time using the connection summary log files. First, we try to find out what happens if we store all connections and keep them in the cache for a certain period of time T_e (*eviction time*) after they were closed.

The idea for the computation of the volume of data to keep in the cache over time is to run over the connection's summary data as shown in table 3.1. For the analysis as described in the following, we need **start**, **duration**, **o_bytes** and **r_bytes** for each connection.

For every connection, we assume that its total amount of data it transmits in both directions during its lifetime ($s := s_o + s_r$) is evenly distributed over the connection lifetime. We do not separate the directions; both the originator's and the responder's data will be stored in the same cache and be subject to the same eviction policy. Thus the input for our analysis are tuples of:

$$(T_0, \Delta T, s) \tag{3.1}$$

At the start of every connection, we know when it will terminate and we assume it will fill the cache at a constant rate of $\frac{s}{\Delta T} \text{Bytes}/s$.

The model for the cache's simulation is quite simple. We use the following state variables:

- τ last timestamp processed
- V current volume in cache
- ρ current rate of cache growth
- n_a number of currently active connections
- n_c number of connections currently in cache

Initially, all state variables are zero.

We traverse the list of connections to successively update the state variables (the model for the cache). This goes in three steps. First, starting from the original connection log file, we extract a tuple (3.1) for each connection and generate an “event” file. An event for the cache model represents a change in the parameters of cache growth ρ or absolute volume V at a point in time. It is defined as

$$(t, \Delta\rho, \Delta V, \Delta n_a, \Delta n_c)$$

where t is the time this event takes place, $\Delta\rho$ gives the change of cache fill rate at the event’s point in time, ΔV is the change of absolute data volume in the cache, Δn_a and Δn_c are the changes in numbers of connections that are active and stored in the cache, respectively.

For every input tuple (3.1) representing a single connection, three event tuples are generated:

1. $(T_0, \frac{s}{\Delta T}, 0, 1, 1)$

This means, at the beginning of the connection the fill (growth) rate for the cache is adjusted to reflect the new connection that is starting, so the connection’s average rate $\frac{s}{\Delta T}$ is added to ρ . No change in absolute volume takes place, and the numbers of active and cached connections (n_a and n_c) both are incremented by one.

2. $(T_0 + \Delta T, -\frac{s}{\Delta T}, 0, -1, 0)$

At the connection’s end ($T_0 + \Delta T$), the fill rate is decremented by the connection’s average rate. Again, there is no change in cache’s absolute size. The number of active connections is decremented by one to reflect the end of this connection, the number of cached connections does not change.

3. $(T_0 + \Delta T + T_e, 0, -s, 0, -1)$

At time $T_0 + \Delta T + T_e$, the cache time T_e has passed since the end of the connection. We decided to clear the cache from the connection at this point in time, so we subtract s from the cache’s absolute volume V , and decrement the number of connections in cache n_c by one. No other changes take place with this event.

After generating an appropriate *event file*, in the next step all of these event tuples are ordered by t so we have a temporally ordered list of events. Third, this ordered list is processed to successively adjust the values of the cache model’s state variables. In every step one event is read and processed by modifying the state variables according to the following rules:

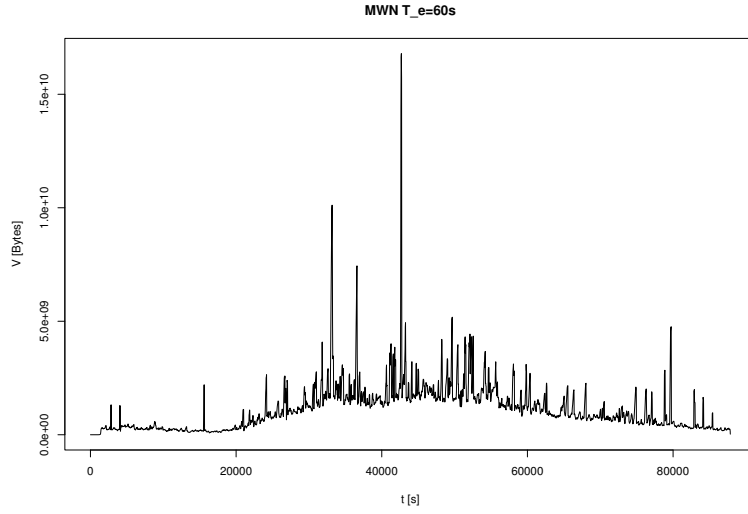


Figure 3.2: MWN plot for $T_e = 60$ s

$$\begin{aligned}
 \tau' &= t \\
 V' &= V + \rho \cdot (\tau' - \tau) + \Delta V \\
 \rho' &= \rho + \Delta\rho \\
 n'_a &= n_a + \Delta n_a \\
 n'_c &= n_c + \Delta n_c
 \end{aligned}$$

We write the values of the state variables to a file in every step. So we have the functions of them over time and are able to analyze the results of this analysis.

3.3.3 Results

The results of such an analysis for a day of connection logs for MWN (Mon, 10/25/2004) with $T_e = 60$ s, $T_e = 3600$ s and $T_e = 86400$ s (connections are evicted one minute, one hour or one day after their ending) are shown in Figure 3.2, Figure 3.3 and Figure 3.4.

For $T_e = 60$ s we see that there is a “basic load” of a storage volume of a few Gigabytes throughout the day. But there are a few peaks that let the cache grow to over 10 or 15 GB. With $T_e = 3600$ s, these peaks are not visible any more. Using this eviction timeout the cache size grows to the order of magnitude of 100 GB. This fits with our estimations in § 3.2. If we set the cache timeout to $T_e = 86400$ s

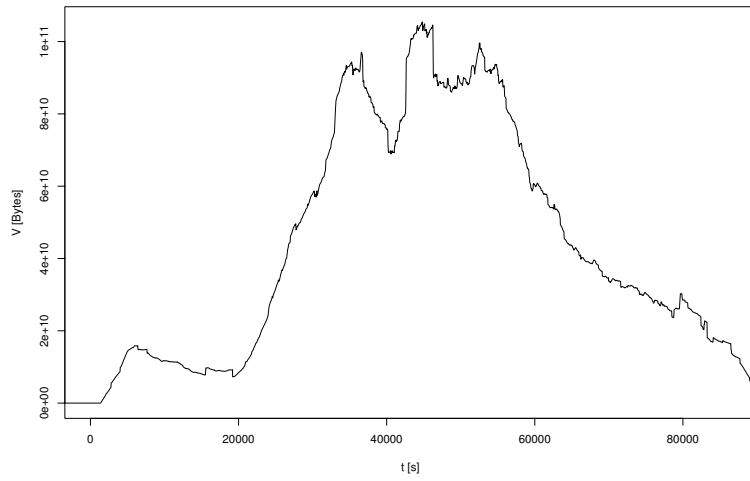


Figure 3.3: MWN plot for $T_e = 3600$ s

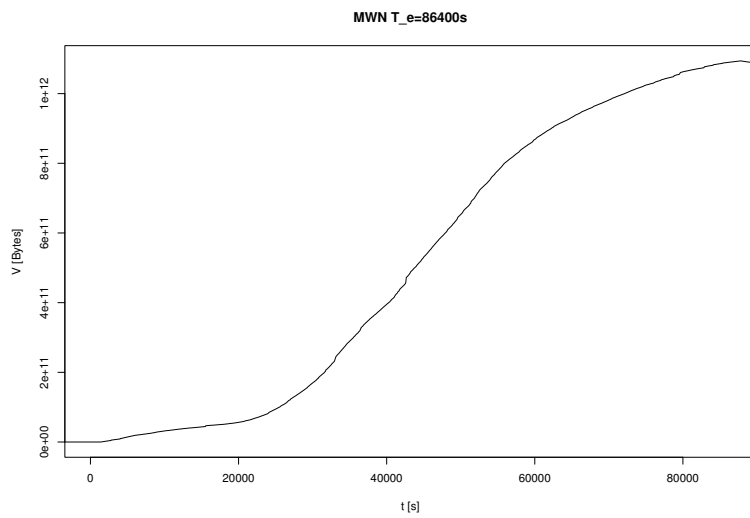


Figure 3.4: MWN plot for $T_e = 86400$ s

(24 hours), we reach the Terabyte scale by the end of the day, and the connections are still accumulating in cache.

3.3.4 Connection Size Cut-Off

We want to reduce the load on a caching system. From Figure 3.2 we can deduce that there are a few connections that are very large in size. A lot of these large connections are from FTP traffic from *ftp.leo.org* downloads. We can argue that it is very unlikely that data from these large connections is interesting in terms of security monitoring. Most of the data from these large FTP data transfers is static and repeats over and over again when large files from the FTP server are downloaded by clients from all over the Internet. On the other hand, these connections take away very much of the resources of the recording system. So the ratio of cost to value is very bad. If there is some interesting data in a large connection, the chances are that this interesting part will be found at the beginning of the large connection, because it is most likely there that the relevant information from the protocol such as headers can be extracted.

As an alternative policy for the cache system we could decide not to store more than s_{max} bytes per connection. Since it is not known from the start of a connection how fast it will grow in size and its total size is not known, the system will have to start storing the connection's data until a decision can be made according to a "large connection policy". When the connection grows to s_{max} , the system could either drop the data stored up to that time. This would prevent us from monitoring large connections altogether. Or the system could stop recording when the cut-off limit s_{max} is reached, but keep the recorded data. The beginning of large connections might be interesting, so we decide to keep the beginning of large connections. This also helps in the storage system's design. It would be considerably restricted by the requirement to be able to evict single connections from the cache structures.

So now we alter the simulation to reflect this new storage policy. We recognize that the limit s_{max} is reached at the point in time $s = s_o + s_r$ grows to this limit, so we will simulate this exact behavior. To do so we generate event tuples from the connection log files like in the previous section, with the new tuples:

$$\begin{aligned} & (T_0, \frac{s}{\Delta T}, 0, 1, 1) \\ & (T_0 + \Delta T \cdot \frac{s_{max}}{s}, -\frac{s}{\Delta T}, 0, -1, 0) \\ & (T_0 + \Delta T \cdot \frac{s_{max}}{s} + T_e, 0, -s_{max}, 0, -1) \end{aligned}$$

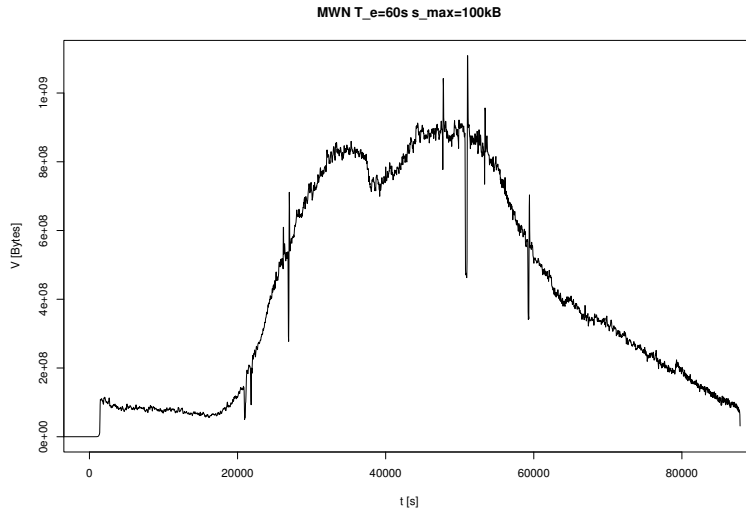


Figure 3.5: MWN plot for $T_e = 60$ s, $s_{max} = 100$ kB

Figure 3.5 shows the space usage graph for MWN with eviction timeout $T_e = 60$ s and connection size cut-off $s_{max} = 100$ kB. The peaks we saw in the same simulation without a cut-off are gone. Space needs do not exceed 1 GB. Even if we set the size cut-off limit to 10 MB ($s_{max} = 10$ MB) and store the connections for one hour after their ending ($T_e = 3600$ s), the overall space usage behavior seems to be manageable in real systems. The data from the other sites show similar characteristics, with the NERSC simulation especially benefiting from the cut-off of large connections, which matches our expectation due to their traffic's nature.

3.4 Conclusion

The ambitions of this project are faced by a substantial load to be processed. With data rates of a few hundred Mbit/s, the disk space available in today's commodity computers is exhausted in a few hours.

We looked at some characteristics of real traffic at different real sites. A brute force approach without insight into traffic's composition is not very promising. There are some very large connections using very much of the available storage resources and contributing a very small amount to the goal to cache interesting data. We discussed the principle of a size cut-off and simulated the behavior of a caching system implementing it. Cutting off large connections is a big relief on the load for the caching system. At the same time, we think the chances of a loss of relevant information due to the cut-off are fairly small. So we will use this

principle for our design.

The results of the cut-off simulation were very encouraging. We did not have an idea of how this would behave prior to these calculations. But the results show that a *time machine* for an IDS can be realized with fairly small amounts of system memory.

For this analysis, we did not look into the structure of traffic in terms of applications (protocols, ports). Clearly this will be a further distinction that can be leveraged by the system to optimize the cache strategy. We will look into this when discussing the design decisions in the next chapter.

Chapter 4

Design

The previous chapter showed that for real traffic, a recording system as targeted by this project is practicable. We worked out a fairly easy way to reduce traffic in a reasonable way, the connection cut-off.

This chapter will present the design of our system. First, we give an overview of the requirements for the design, then we describe the individual parts of our design.

4.1 Requirements Review

The recording system to develop should be able to store traffic from the network in some cache at high performance, i.e. there should be no loss of data due to lack of efficiency in the recording mechanisms. We want to extract specific subsets of data from the system's cache, for this we have to specify the selection. This is what we call a query. Query parameters, i.e. the specification of the data subset we are interested in, can be time ranges or protocol information (such as IP addresses or port numbers).

A query based on a time range might be specified if the querying instance knows when in time the interesting portion of traffic was on the network. This might be the case when a concrete security incident has to be analyzed and evidence of it has to be extracted from the recording system.

Queries may also be expressed using information given in the packets' headers. This is an essential feature since it allows to search for a concrete subset of the packets. For example a query might request all packets sent from host *A* to host *B*.

In general, we consider querying the system a less performance-critical aspect than the high-performance operation of the capture and storage parts of the system. A query might take a few seconds, since “after the fact” analysis does not depend on the timeliness of its input. So we give priority to capture and storage and take special care of the structures and mechanisms employed there. However, a query has to finish in a reasonable time, of course. We will see that provisions have to be made for this, too.

4.2 System Architecture

We devise a stand-alone recording system that operates separate from the Intrusion Detection System itself but is able to communicate with it for control and data transfer purposes. This gives the advantage of avoiding any impact on the IDS’s performance. If the recording system runs on a different host than the IDS, it does not compete for resources with the IDS. This is important because IDSs themselves are quite demanding in resources like CPU time and system memory, as well as we expect our system to stress available resources to a certain degree in order to cope with the fairly high load we anticipate.

Another advantage of this architectural choice is that the recording system can run on its own and be used for slightly different purposes than we address in this work. It might be used as a forensics tool without any connection to an IDS. If an attack or otherwise interesting event is detected in some way, e.g. manual log file review or an obvious damage, the recording system can be queried for the records to gain packet-by-packet insight into what happened.

We use the strategy worked out in the previous chapter, connection cut-off in our design. As an additional way to alleviate space usage and allow potentially “interesting” portions of the traffic to stay in the system cache for a longer time, we distinguish between different parts of traffic and treat different parts in different ways. We call these distinct parts *classes*, the process of separating traffic to different classes *classification*; this is done according to *classification rules*. A class defines the parameters that specify the details of how to store the associated traffic, including the connection cut-off size; all these parameters are called the *storage policy*.

The recording system consists of the following components (for a diagram see Figure 4.1). The packet capture module receives network packets from the interface to the operating system or specialized capture hardware and delivers the packets to the classification module. Classification allows for different treatment of packets as they are associated with a certain class of packets according to defined properties. The storage containers handle the incoming packets and store

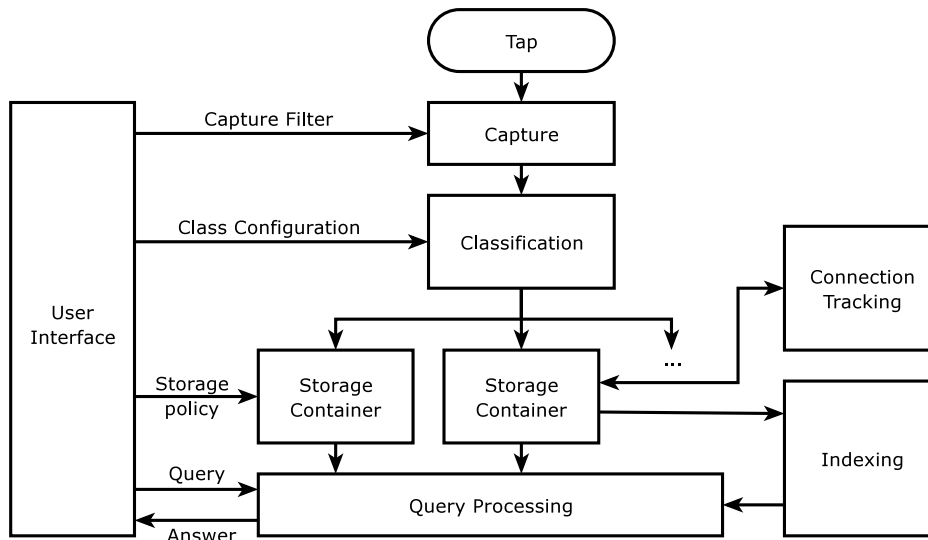


Figure 4.1: System architecture

them in memory, on disk and evict them later, according to the storage policy defined by their respective class. Indexing provides mechanisms for quick access to subsets of packets that are stored anywhere in the system. The query module accepts queries for stored data, retrieves it from the storage containers and returns it. Finally there is a module that communicates with its environment to offer the functionality of configuration and queries, this is the user interface to the system.

In the following sections we will describe each module in detail.

4.3 Packet Capture

The system we devise has to run on a computer with a network interface to the network from which data has to be cached. Such a computer system is called a *packet capture* system or *sniffer*. The sniffer’s network interface card (NIC) that is connected to the network to observe is put in a special operation mode by the operating system, such that all packets that are observed by that NIC are accessible by the packet capture application (i.e. our recording system). The NIC’s connection to the measurement network is called *tap*.

The sniffer’s operating system provides a high-level interface to the packet capture mechanisms in the operating system’s kernel. On UNIX systems, on which we developed and tested implementations of this design, the *libpcap* library [Law04] is such an application programming interface (API) for use by packet capture ap-

plications. In our implementation, we use *libpcap* as the interface at which our system receives packets from the network. Of course, there are other possible ways for the packet flow from the physical network to the application. For example, there is specialized hardware like the Endace DAG card [End04a], which is designed for packet capture purposes only (as opposed to ordinary network interface cards). The DAG card can be accessed by *libpcap*, or by a DAG specific library (or API).

Essentially the functionality of packet capture interfaces (APIs) is to deliver all packets received at the NIC to the application. Every packet is completed with additional *meta information*, including a timestamp value that indicates the time the packet arrived at the NIC. Other such meta information on a packet is its total length and the length that was captured from the network (which may be less than the total length if it was cut in the middle due to capture length configuration). The format of the meta information as passed to the application depends on the packet capture API that is used. *libpcap* has its own packet header format. The Endace DAG API, on the other hand, uses the Extensible Record Format (ERF) [End04b]. In both cases, the packet data itself is given to the capture application in raw format (byte string), beginning with the link layer header. The capture module passes the meta information in the capture API's format and the packet data in raw format on to the classification module.

At this point in the system, performance problems may impact the integrity of the captured data. If an application using a capture API is not able to process packets as quickly as they are delivered by it, the capture API buffers the packets from the network until they can be dispatched to the application. If this packet buffer runs out of free space, further packets arriving on the network will be dropped. This is a situation that clearly is to avoid, since lost packets may mean that relevant information our system aims to save is lost.

Using ordinary NICs and *libpcap*, the packet timestamps are of limited accuracy. Due to the interaction of NIC hardware buffers, the computer hardware and the operating system, the timestamp value given by *libpcap* is not exactly the time the packet arrived at the NIC's network connection. Specialized hardware like the DAG card can give the timestamp of packets much more accurately because they implement special hardware timers that yield the timestamp information for single packets. In our application, however, we do not care about slight deviations in the exact timestamp. In spite of inaccuracies, the packets and reported timestamps are in temporal order, so we do not have to worry about out-of-order packets. We do not consider the effect of timestamp inaccuracies negative for security monitoring. Packet delays occur in packet-switched networks anyway, so the exact timestamp of malicious packets as seen at one point in the network does not necessarily tell about the exact timing of malicious activity itself.

libpcap offers a feature to specify a filter against which packets from the tap are matched. Only matching packets are forwarded to the capture application if such a filter is set. Our application allows the user to configure such a filter. This enables the user to activate preliminary filtering at a very low level (in the operating system kernel). Then, packets not matching the preliminary filter are not passed on to the recording system at all. This could be leveraged as a performance enhancement if for example a defined IP subnet is not to be processed at all. The filtering mechanism *libpcap* offers is the Berkeley Packet Filter (BPF) [MJ93]. BPF can process arbitrarily complex filter rules. These rules are presented as a *human readable filter* in the format described in [JLM01b]. An example for a filter in human readable string format is `'not net 1.2.3.0/24'`. This string is compiled to a *BPF program* which is used to test packets against the filter criterion. The example matches IP packets that do not originate from and are not addressed to the network 1.2.3.0/24.

4.4 Classification

The next module gets its input, a stream of packets, from the capture module. We treat portions of traffic differently than others. For this we establish a classification module that separates incoming traffic into classes according to a given configuration.

For classification, there needs to be a set of rules that determine how to separate the traffic into classes. Later these classes are assigned properties that determine the parameters for the storage mechanisms. Separation is done packet by packet as the packets are handed over by the capture module.

A way to examine packets and match them against given criteria is to use the Berkeley Packet Filter (BPF) filtering mechanisms. We decide to use this framework for our classification module because it is powerful, easy to use and accessible via the *libpcap* library we already use for packet capture. *libpcap* provides a function that compiles strings to BPF programs, as well as a function that takes a BPF program and a packet and returns whether the packet matches the filter rule or not.

An example for a rule in the BPF human readable string format is `'src net 1.2.3.0/24 and dst net 2.2.3.0/24'`. This example matches IP packets that originate from the network 1.2.3.0/24 and are addressed to 2.2.3.0/24.

We set up a number of rules and assign each rule a class that the matching packets are to belong to. Arbitrary rules can be given by the configuration. Note that a single packet might match a number of rules. Since we want to avoid to store packets

in duplicates in containers for more than one class, we introduce the concept of *precedence*. Every rule has to be assigned an integer that indicates its precedence. When a number of rules match, the respective packet is passed to the class given by the rule with the highest precedence number.

The IDS (or another user of the system) is able to configure rules and classes during the operation of capture and storage (i.e. during run-time). This enables the definition of temporary classes for parts of the traffic the IDS wants to watch more closely. We gave examples for this application in § 2.2. The mechanisms involved in configuration will be described in § 4.8.

The IDS might create a temporary class for example because it wants a part of the traffic to be stored longer than the currently active class configuration would define. This might be initiated because a subset of the traffic on the network is considered as potentially relevant for forensics. The IDS creates a temporary class matching the desired traffic subset at the point in time it recognizes that this subset has to be kept. However, at that point in time part of this traffic subset may already have been classified belonging to some other class according to the filter rules prior to the definition of the new temporary class. Thus this older portion of traffic was stored in the other class' container. This implies that this portion will be treated according to the policy of the old class. The reason for the creation of the temporary class is to define a specific storage policy for parts of the traffic. So in order to keep all parts of a traffic subset in the temporary class that the user wishes to, including the older portions of matching traffic, we allow a *pre-fill query* to be specified with the creation of a new class. This query is executed on all traffic stored in the system and its results pre-fill the storage container of the newly created class. The packets found as a result to this pre-fill query are marked as *duplicated* in the originating class' storage container as they are duplicated to the new class' storage. The duplicated flag is a field of the storage's internal header (as described in 4.5.2). This prevents later queries from rendering duplicate packets. Such duplicates would be hard to detect during query processing without knowledge of their origin in a pre-filled class.

Classes may be deactivated by dynamic configuration, too. This does not mean that the contents in their storage containers are deleted, but rather that their rule in the classification module is removed and the packets that were matched by it are handled by the remaining classes according to their rules, if any of them matches. (If none matches, the traffic in question will be discarded again.) A record of the existence of this class from the time of its creation till its deactivation is kept, because for any point in time, we need to know what classes were active at that point; this is important for later queries (see § 4.7).

4.5 Storage Containers

The classification module divided the incoming stream of packets into classes. Now we describe the design of the storage mechanisms that make use of the distinction by classes and of the cut-off mechanism devised in § 3, and we describe the format and data structures for the actual data storage.

4.5.1 Structure

The classification module as seen in § 4.4 divides the incoming traffic in a number of classes. The storage module treats each of these classes according to the *storage policy* that is specified for the individual class. We associate each class with a *storage container*. This container takes the packets that are matched by the classification module to be in its class. Every storage container is independent from the others in its operation. It has its own space and own storage policy, which consists of parameters that define the behavior of the storage container.

The system stores incoming packets in system memory (RAM) as well as on disk. This comes naturally, since when packets are delivered by the operating system's interface from the network tap (or some other source, as mentioned in § 4.3) to the capture application, they are first stored in system memory. There we can process, copy and move packets very quickly in a system memory cache. Eventually the packets will have to be evicted from the system memory cache. Then they may be written to a disk cache.

The storage policy for a class defines how much memory space is reserved for the associated storage container, it defines the size of both the system memory and disk caches. We express the amount of space allocated for a cache in terms of data volume (i.e., bytes). Another idea for cache size specification could be to give time ranges (e.g. class A is assigned 60 min worth of storage). This comes to mind since the container (and its caches in RAM and on disk) fills and empties as data comes in and is evicted. So the cache size could be given as the period of time it takes a cache to “refill” (or the lifetime of a packet in a cache). But since we know neither the overall rate of data nor the rates the individual classes experience, the total volume of data in the cache can not be controlled if cache size is defined by a period of time. We have limited space, so we have to be able to ensure an upper limit of space usage that accommodates the parameters of all present classes.

4.5.2 Storage Format

We need to define the format in which to store the packet data and associated information in our caches. The packet data is the essential payload our system targets, including protocol headers and the packet's actual payload data. Meta information, including a timestamp to represent the time the packet was received by the packet source, completes this data we receive from the capture module.

Information on each packet with respect to the system's internal management is stored in an *internal header*. As of the design at hand, there is one flag in that header, the duplicated flag. It indicates packets that are obsolete because they were transferred to another class' storage container. This happens with the dynamic creation of new classes as described in § 4.4. The internal header is the first of the headers at the place of storage for each packet.

Of the meta information that is rendered by the capture module, especially the timestamps are of interest. They are important in order to be able to find packets by the time of day they entered the system. Additionally, we want to be able to reconstruct network traffic from the past ("forensics"). A series of packets including protocol headers and payload, with meta information associated with each packet, is called a *packet trace*. Packet traces are the basic material dealt with in incident forensics. So we need to store the timestamps of the individual packets.

As discussed in § 4.3, the input from the capture module to the storage module consists of the packet data and meta information. This meta data is in a special header format, e.g. the pcap packet header. We store the meta information in the format it is rendered by the capture module and the originating packet source, e.g. in pcap packet header format if *libpcap* is used. We note that this meta information format is interchangeable, so as an alternative it might be in ERF format, for example if the Endace DAG API is used by the capture module. Of course, the actual implementation has to consider the length of the meta header and address the fields therein according to the chosen format. We store the meta information as a header appended to the internal header, in front of the packet data.

In the following discussion of caches, we refer to a "packet" in a cache as the concatenation of the internal header, meta information header and the packet data, the latter consisting of the protocol headers and eventually the packet's payload.

4.5.3 Memory Cache

As mentioned above, packets first enter the system memory cache. We know the amount of data this cache has to keep, it is of a fixed size. The memory cache has to be able to insert new packets and remove the oldest packets. Packets are of varying size. So the basic operation is like a FIFO buffer.

We choose to design this memory FIFO as one block of memory (*buffer*) allocated once by the operating system. This buffer is used as a *ring buffer* storing packets. Notice we have to keep track of where the first (oldest) valid packet is located. Additionally, we maintain a pointer to the current write position, which is after the last written packet. Insertion starts at the begin of the initially empty buffer. When the end of the buffer is reached, some packet will not fit in the remaining space. Then the write operation “wraps” over the end of the buffer to its beginning. There, packets have to be removed (evicted) to clear enough space for the new packet. Whenever packets have to be evicted from the memory cache, the system has to take care they are properly moved to the disk cache according to the policy; however, a policy might specify that for a class only a memory cache is used, in which case the packets to be evicted from the memory cache are discarded. Since packets vary in size, more than one packet may have to be removed when one new packet is about to enter the cache. As the write position advances again, every time a new packet is inserted, one or more of the packets ahead of the write position have to be removed.

An alternative to one big buffer block would be to individually allocate memory space for every single packet or a number of smaller blocks each taking a number of packets and to link these multiple buffers in some structure. This would result in a lot of allocation and deallocation operations by the operating system. Experience has shown that the operating system is not in all cases able to ensure maximum performance if a very large number of such operations takes place.

To keep track of where to find packets by timestamps, we maintain a variable to represent the timestamp of the oldest packet in the memory cache. This enables us to find the place where to look for packets during querying (see § 4.7).

4.5.4 Disk Cache

In addition to system memory, we need to make use of disk space for storage since the few Gigabytes of system memory typically available will be exhausted fairly quickly. Disk space, on the other hand, ranges in the hundreds of Gigabytes, or even a few Terabytes. This enables our system to store traffic for a fairly long

period of time. This section describes the structure of the packet cache on disk and the mechanisms involved in operating it.

A storage container's disk cache is, like its system memory cache (see previous section), basically a FIFO cache: packets enter the cache, stay there for as long as its size and the insertion rate permit, and are eventually evicted. Storage on disk is structured by the operating system using a file system. File systems provide an entity *file* and offer mechanisms to read, create, append, move and delete files. For a storage container's disk cache we decide to use files of fairly small size (a few Megabytes) and keep a number of them at a time. The concatenated contents of the individual files is the timely ordered series of the packets in the disk cache. The files are named in a way that allows to associate them to a class (by a *class name*). Files in a class are given ascending numbers as they are created one after another. Again, as in the system memory cache, we do not cut packets in the middle but store them in a file in whole. The eviction of the oldest packets in the disk cache is done on the granularity of files, so we simply delete the oldest file if the storage container size is reached or exceeded.

Another possible way to organize packet storage on disk would be to bypass file systems and store packet data in raw disk partitions. If done correctly, this might improve input/output performance for ring buffer functionality. However, this would call for a number of provisions for the organization of the data. We did not experience performance bottlenecks using a file system in the proposed way, so we refrained from this raw partition approach.

As in the system memory cache, we maintain information on where in time the disk cache ends (i.e. the timestamp of the oldest packet). This marks the point in time for which the oldest packets are available in this storage container, since after eviction from the disk cache they are removed entirely from the system. In the disk cache's file structure, we keep track of the timestamps for every individual file. This gives us a fast way to find the file where to look for packets during querying (see § 4.7).

4.5.5 Connection Tracking

We looked at the traffic volume to expect in § 3 and argued a cut-off of large connections is a suitable method to reduce the load to a more manageable level while keeping the more promising part of the data. The storage module implements this cut-off. For this it needs to be able to operate at the granularity of connections, i.e. it needs to track connections, assign incoming packets to a connection and update information about how much data a single connection has transmitted. So we need to maintain the state of connections. We call this connection tracking.

Connection tracking needs to maintain a set of connections. If a new connection starts, it has to be added to this set. Without appropriate precaution this set would grow very fast. Connections that have ended have to be removed from the set that is kept in memory. Additionally, we have to keep in mind that due to the great amount of scanning activity in today's network traffic, the number of connections is by far greater than the number of connections due to regular network usage. The connections we see on the network due to scanning activity are very short and usually consist of a single packet or a few packets generated by the scanner, and possibly as a response from the scanned hosts. So both a horizontal scan, i.e. a scan over a number of host addresses and a vertical scan over the ports of a single host appear as a number of distinct connections according to the definition of a connection as the 5-tuple of host addresses, transport layer protocol and transport layer port addresses. So there is a need to manage the connection space in a way that prevents the available space from exhausting. For this, we propose the removal of connections for which there has not been any packet on the network for a certain period of time (time-out). Since we need to look up and update a connection for every packet entering the system and possibly insert a new connection to the set, the according operations have to be efficient in terms of computation time.

To summarize, we need to provide space-efficient storage of connections and time-efficient access and lookup, insertion, time-out and deletion mechanisms. Therefore we devise the use of a hash for efficient access by a search key, in combination with a queue for an efficient time-out mechanism. The hash maps the search key, the connection parameters 5-tuple, to a structure containing the connection parameters and associated information like number of bytes transmitted by that connection and last packet's timestamp. This structure is kept in a queue and points back to the hash entry for efficient in-parallel maintenance of the queue and the hash.

When a new packet for a connection arrives, the according connection structure is updated with the new timestamp and the number of transmitted bytes is increased. Then the connection structure moves to the beginning of the queue. If there is no connection for the new packet, a new connection structure is inserted at the beginning of the queue. The pointers in the hash table are updated accordingly. This maintains a temporal order on the queue so that the entry at the end is the oldest connection, i.e. the one for which there was no packet for the longest period of time. So for inactivity time-out, at each new packet arrival we look at the last entry in the queue and evaluate the last packet's timestamp. If it is older than the time-out value it is removed from the queue and from the hash.

Using these data structures and operations we can keep track of the connections and their transmitted number of bytes and can decide to stop recording packets

from a connection when the cut-off threshold is reached.

4.6 Indexing

We allow queries for protocol header information (see § 4.1). Keep in mind that our system stores a very high amount of packets in the caches, especially on disk. So a linear search, i.e. the traversal of the packets in the caches and successive comparison of the header information to the search key would not be viable. The system needs to provide specially designed mechanisms in order to be able to answer such queries. We devise *indexes* that maintain information on where to find certain packets based on header values.

We use the term *index* for the redundant information stored and the algorithms working on them to quickly locate packets matching a certain value in their protocol headers (in a certain protocol field). When searching for a certain value in this field, we call this the *search key*. We devise indexes that map search keys to time intervals. An index for a certain protocol field (e.g. source IP address in the IP header) gives the time ranges in which packets matching a search key (an IP address in this example) have been received and stored by the system. When processing an actual query, the query module then scans linearly through the intervals it gets from the index. We allow this scan to take a little bit of time, as we decided not emphasize on query performance (we allow queries to take a few seconds).

One single index does the job for a certain protocol header field (thus for a certain search key). So we might have an number of indexes, each one for a certain header field. How many indexes we use and what fields to index depends on what fields we expect to be useful to index because we expect queries on them and on how much resources the index information and its maintenance uses up. We will discuss a few experiences concerning indexing performance in § 5.2.

Assume an index I_f on protocol field f . The index mechanism we devise maintains a set of all values of f in known packets. For each value the index I_f keeps a list of time intervals in which packets with the according value in f were received. The intervals are “closed” after time ΔT passed without receiving a packet matching the according value in its field f .

For each active index I_f we examine every incoming packet P and look at the value of f_P . If this value is in the set of values for I_f we look up the time intervals for f_P . The latest interval $[T_1, T_2]$ is still “active” if for the current packet’s timestamp t_P holds $t_P \leq T_2 + \Delta T$. In this case, the interval is updated to $[T_1, t_P]$. Else—the existing interval is not “active” any more—a new interval $[t_P, t_P]$ is inserted. If f_P is not in the set of values for I_f we insert the new value and an associated new

time interval $[t_P, t_P]$. Since in one index I_f we have to look up by f every time a packet arrives, this has to be implemented in an efficient way.

Using this strategy, the index builds a directory of all values and a list of intervals for each value. We store the index in system memory and eventually move the index entries to disk, in accordance to the actual content the index entry points at. Thus, intervals $[T_1, T_2]$ are evicted from RAM and moved to disk as the according content (packets with $t_P \geq T_2$) is moved to disk. Note that this mechanism requires efficient access to the oldest intervals.

Considering computation-time efficiency, when processing the incoming packets, we desire to have fast access to the structures storing the time intervals by the search key value. With respect to the eviction of intervals to disk as the according packets move from a system memory cache to a disk cache, we need an efficient way to find the intervals that are below a threshold timestamp value (the systems memory's oldest packet's timestamp) and move them to a disk index.

We meet these requirements by using a hash data structure for access to the intervals $[T_1, T_2]$ by the search key (see Figure 4.2). The search key is hashed and the hash table points to a structure containing the boundaries of the most recently created interval. These structures are organized in a queue. Each structure points back to the hash entry for easy in-parallel maintenance of the hash and queue contents. Every time an interval is updated, the structure moves to the beginning of the queue, so the queue is in temporal order of update times. This allows efficient access to the oldest intervals. Each interval structure possibly points to a structure containing the preceding interval for the same search key, forming a linked list of all the intervals that are stored for a search key value.

In order to estimate how much space an index would need we consider to two dimensions in which the index space grows. One is number of search keys, the other the number of intervals for a given search key. We construct a bad case for both and consider their ramifications.

Assume an index on the TCP destination port field. If there is an episode of network traffic that contains one packet for every possible destination port, like during a scan, the index triggers the creation of an interval for every search key value. Assume the implementation uses a floating point timestamp representation using 8 bytes for a timestamp. For a protocol field of 16 bits like the TCP destination port field, this means that the index grows by approximately 1 MB.

An index on an IP address field is a similar case where to expect problems during a scan. Consider a scan from outside to the internal network. We estimate the typical size of an internal network 2^{17} to 2^{18} addresses, as this is the size of the MWN, one of the large research environments we target with this project. A thorough scan of the internal network increases the size of the index for an IP address

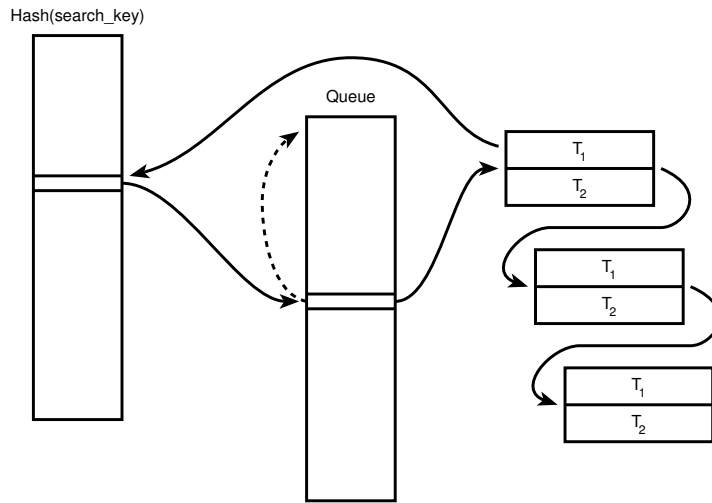


Figure 4.2: Index Data Structure

field by 2 to 4 MB for the estimated network size. For considerations of scans on a larger scale, like from the inside network to the total IP space (2^{32} addresses), we have to keep in mind the time this takes and that as the scan proceeds, the resulting traffic is very likely to be evicted from system memory. Thus the according index intervals will be moved to disk. So we do not fear that indexing is in danger of suffering from scanning activity.

Now assume an index with $\Delta T = 1s$, and a network traffic that contains one packet matching a search key per second. There would be a new interval every second. Indexing 24 hours of such a traffic with the constructed index would require approximately 1.3 MB.

For both dimensions of index growth, the constructed “bad cases” promise a moderate impact on index space usage. Therefore we expect the indexing mechanism we described can be used on realistic traffic. We will discuss similar experiences with respect to index space considerations we gathered with the prototypic implementation in § 5.2.

4.7 Processing Queries

We need to find algorithms that address the problem of finding and yielding specified subsets of data stored in the system’s cache.

Our packet recording system stores a great amount of data in its cache, with an especially vast volume of data in its disk cache. That volume is estimated to

typically be in the hundreds of Gigabytes for the kind of hardware we considered for our system. So it is not trivial to extract the wanted information in reasonable time for all the queries we considered in § 4.1.

For the case of a query by time, the task to find the right place to look for the matching subset of packets is fairly easy. This is because the packets in the caches are in temporal order. We leverage this order for efficient access.

We assume a query that asks for all packets in a range of time and that specifies this interval by a start and end time value. All queries to the system relate to all places of storage, system and disk memory. Thus the result could be in a system memory cache, in a disk cache, part in memory and/or part on disk, or it is not in the system at all. While recording, the system keeps track of the timestamps of the oldest packets in system memory caches and disk caches, respectively, i.e. we know the time boundaries of system memory cache and disk cache as well as disk cache and eviction for each class. The number of classes is a relatively low number, so it is viable to sequentially query every storage container (i.e., every class).

If we have to look up the results of a query by time in a system memory cache, we can use *binary search* to quickly locate the packet nearest to a given timestamp to search for. Binary search repeatedly divides the search interval in half. If the search key is less than the value of the item in the middle, it narrows the search to the lower half, otherwise to the upper half. It continues until the search key is found or the interval is empty, meaning the search was without result. In both cases, success and no result, the run time is $O(\log N)$, where N is the number of elements in the search space, since the size of the search interval is divided by 2 in every step, so there are around $\log_2 N$ steps.

As we have seen, the disk caches are structured in files that are relatively small in size. For every file we keep, we know the timestamps of the chronologically first and last packets in that file. So for a given query, we can narrow the location of packets for a given range of time to a set of files. Since the single files are fairly small, searching for a timestamp in them is fairly easy and comparable to the search for a timestamp in system memory cache described above. Again, we use binary search for this task. Due to the small file size we consider the time needed for seeking in the files negligible in comparison to the order of speed we allow for query processing.

If there is a query by protocol information, we can make use of the indexes we introduced in § 4.6. An example for such a query is “get all packets with source IP address A ”. Since we established indexes on protocol fields, we can process such a query by looking up the relevant index and proceeding with queries for the resulting intervals, like shown in the previous case with a query by time. This

results in a range of packets that contain packets matching the query specification; the index helped to find the approximate location of the packets we search for. Now the query continues by a linear search for the exact match in these ranges. Note that this linear search might take a few seconds, depending on the size of the time ranges that have to be searched.

For more complex queries we allow combinations of the mentioned basic query types to form logically *anded* constructs. An example for such a more complex query is “get all packets with source IP address *A* that were received between 1:30 AM and 2:30 AM”.

The fact that each basic part of this composed query yields time intervals allows it to be easily processed by generating the intersection of the intervals resulting from the basic queries. Logic *and* is easily manageable in this way, as is a logic *or* by generating the union of resulting intervals. As with the basic query case, the next step is to linearly scan through the obtained (and combined) intervals.

Whenever processing queries, we need to look at the internal header’s *duplicated flag* as discussed in § 4.5.2. When a packet is duplicated, we must not include it in the query result, because it will be rendered by the query in another class.

4.8 User Interface

The system needs to have an interface to the outside world to receive configuration settings and updates, to accept queries and to transmit query result data to the user. A user may be a real person working with some interface or an IDS that communicates with the recording system and makes use of it. We choose a message passing solution for communication for these purposes.

Most of the modules of our system require a set of parameters, or configuration for the control of their operation. The right configuration depends on the environment in which to run the system. Each site will have different requirements and goals, so the configuration needs to be adaptable. Some kind of configuration control is needed.

The capture module may be supplied a BPF filter for preliminary filtering. We allow this filter to be changed during run-time. Similar filtering is done in the classification module, using the same filtering mechanism as the capture module does. Here, configuration data is a bit more complex and requires the use of a list of rules assigned a precedence number and a class name. Again, this configuration state may be updated during run-time.

Each class in the system needs the parameters defining a storage policy. These are a class name, the size of memory and disk caches and the connection cut-off

size. Classes may be added and deactivated at run-time. When adding a class, the user may specify a pre-fill query. An example for this is for the IDS to define a temporary class and storage policy like described in § 4.4.

We devise messages that transmit the according requests for configuration updates. Every possible change to the configuration has its own message type and the according message is in a defined format that contains all the information associated with the configuration update.

Finally we need to have an interface to the system to request queries and to get back the results. The query request is a message that contains the query specification. The request may define where the results should be sent. They may go back to the requesting user via the message passing system according to a defined format. Or they may be saved to a disk of the recording system. The latter option may be interesting for forensics, for example if a user wants to archive a portion of the stored traffic.

The configuration, control and query messages can be exchanged with the system over different ways. For a real person working with a console, communication can be directly to the system via its standard input and output. An IDS would need some kind of network protocol, since we allow for the recording system to run on a different host than the IDS.

Bro is able to receive and transmit the *events* that represent its internal messages via the Bro Communication Protocol [ref]. The Bro data structures permit to build messages as depicted in this section, and there are data types for the transmission of binary query result data. A suitable interface to Bro is the *Broccoli Bro Client Communications Library* [Kre04], an API to the Bro Communication Protocol. It allows to build applications that can communicate with running instances of Bro and exchange messages via the Bro Communication Protocol.

Chapter 5

Implementation, Status and Experiences

We implemented the design depicted in the previous chapter in a prototype. The prototype was run in a highly loaded environment and tested to confirm that the project's goals are feasible. This chapter will describe the implementation and show the experiences gathered running it.

5.1 Implementation

The prototype that was developed in the scope of this project very closely follows the design from the previous chapter. We illustrate a number of decisions made and the essential points that were raised during implementation.

5.1.1 Programming Paradigm and Language

For our implementation we chose an object-oriented approach and used C++ as programming language. The object-oriented decision was motivated by the data structures that are instantiated dynamically. They can be nicely represented by classes and instantiations thereof. This promotes a clearly structured implementation.

Our implementation certainly requires a high degree of efficiency. The choice for C++ as an object-oriented language was motivated by the fact that—according to experience—its programs are as fast as C programs if they are properly designed. There are highly optimizing compilers available that produce efficient native code.

We have not experienced any performance problems due to the choice of C++ for our implementation.

We make use of C++ templates for the implementation of classes representing the data structures we use for connection tracking and indexing. Templates allow to write classes for data structures with place-holders for the actual data types. Instantiation of the template classes takes place when they are used in some part of the source code with a concrete type. Then the compiler takes care of the source code modification during compilation of template code. This makes reusing code more easy and increases the efficiency of programming when abstract data structures are used; again the resulting code is more manageable and easier to understand.

We also use a template for a generic index class that can be applied to arbitrary protocol header fields; it implements generic operations for the maintenance of the index data structures. This generic class uses a place-holder (i.e. template parameter) for the concrete data type to index. This data type is a class that has to be defined for the concrete protocol field, it is derived from an abstract index field class. When a new field is to be indexed, the modifications to the source code are fairly moderate; we simply create a new index field class and instantiate a new index with this new type as template parameter.

5.1.2 Operating System

Both FreeBSD and Linux are primary candidates for capture systems (sniffers) in security monitoring endeavors. It is worth noting that the commercial products we hinted at in § 2.3 are also based on these open source operating systems. The Bro IDS for example is mainly run and developed on these systems, too. We gave an overview of the Munich sniffer lab on the MWN in § 3.1. These systems are FreeBSD and Linux systems. (They are also used to run, develop and test Bro.) The prototype implementation has to run on these hosts for testing purposes. So we developed and tested our implementation on these two operating systems.

From the programming perspective, both are very similar. The system interfaces as used by our implementation like the *libpcap* library are nearly identical. This allows to develop and maintain source code that compiles and runs on both systems without any important extra effort.

However, there are a few noticeable differences when it comes to capture performance. We noticed that the rate of packet drops, and therefore the overall performance quality, considerably depends on the operating system in combination with the used network interface card (NIC), the NIC driver and some settings in

the operating systems. For example, on a FreeBSD system with the Syskonnect SK-9843 gigabit Ethernet adapter and the closed-source driver shipped with that NIC, the performance was much poorer than that of the same adapter on a Linux system (where there is an open-source driver in the kernel distribution). On the other hand, Linux and FreeBSD performed comparably with some other NIC and driver combinations.

There are a few patches for operating system kernels which aim to improve capture performance. On Linux, for example, there is the RING kernel patch [Der03]. It optimizes the packet capture mechanism in the kernel and aims to improve the performance during sniffing, thus to lower packet drop rates. We experimented with the RING patch. It did dramatically improve the performance, beyond the maximum performance we experienced on FreeBSD, but it also led to problems during the tests. The patched Linux systems crashed very often.

We did not examine the interaction of all these factors in depth. There is no clearly winning operating system. Presently, work is conducted at the Network Architectures Research Unit of Technische Universität München that tries to find out how to maximize performance in capture systems, carefully considering the mentioned aspects.

5.1.3 Technical Structure and Concurrency

The design of our system provides for two basic functionalities. As for the *storage* part, packets enter the system via the capture module and are processed by the indexing, classification, connection tracking and storage subsystems. All this computation is done sequentially for every single packet that arrives. The remaining part of the system is the *query* subsystem. It carries out a query every time the user interface passes on query request. By design, the task of consuming packets from the tap and the query task are independent. They have to be implemented in a way that prevents them from impacting each other. We decided that the capture task has a higher priority than the query task, because we want to avoid loss of packets during capturing. It is especially important that a running query does not impact the performance of the capture task. Note that most of the time a query takes is needed for the search in caches, which we allowed to be a rather slow process. We described this strategy in § 4.7.

In the implementation, we decide to make the two parts *storage* and *query* run as distinct flows of control in the program. This provides the concurrency we want to have for the both subsystems. A common way to implement this is to use threads for each of the flows of control that run in parallel. Threads are encapsulations of flows of control in a program. A multi-threaded program has several threads

running through different paths in its code "simultaneously". All threads share the same data; there is only one data space per multi-threaded program. Thus the use of threads, or *multi-threading*, is an adequate way to implement a system that consists of subtasks that run in parallel and cooperate on the same task. Compared to distinct processes that would run independently and communicate with each other, threads have the advantage that their shared data makes them rather easy to use for the programmer. Furthermore, on some systems a context switch between heavyweight processes is more expensive in terms of computation time than the execution of two threads.

Our implementation uses the POSIX thread model that is provided to the programmer by the *libpthread* library. We use two threads in our program. The first one is the *main thread* that was started with the main program. At initialization, this launches the *storage thread*, which in turn calls the *libpcap* `pcap_loop()` function that collects and processes packets [JLM01a]. Whenever a packet arrives, a callback function that was specified with the call of `pcap_loop()` is invoked and takes care of the further processing of the packet. All this makes up the *storage* subsystem and runs in the capture thread. At the same time the main thread continues to run in a loop that waits for queries and processes them as they are entered in the system. Thus the tasks of the *query* subsystem take place in an other thread than the *storage* subsystem.

For correct behavior of the program, concurrently running subtasks need to take special precautions when accessing shared data. Data may be incorrect at the time a thread accesses it because it is being modified by another thread. The parts of the code where data accessed by another thread may be incorrect are called *critical sections*. A multi-threaded program has to avoid such situations, as they may lead to incorrect results, and may lock up or crash the program. A way to provide for the safety of multi-threaded programs is to deny other threads the access to inconsistent data during a critical section. In our implementation, we do so by locking data structures that may be concurrently accessed by the storage and query threads.

These locks protect the index structures and the cache begin and end timestamps for memory caches and disk files. For the cache contents, too, some strategy ensuring integrity is needed; if for example a query reads in a memory cache (which we recall is a ring-buffer, see § 4.5.3), the contents may be overwritten by the storage thread while reading. Unfortunately, it is not viable to lock an entire cache (memory or disk) for writing during a query, since this would block the storage subsystem for a substantial period of time and lead to packet loss. So we devise a special way to ensure data integrity during querying. In the case a query is about to start a search for packets in a memory cache, it sets a mark at the point where it reads packets. The storage thread may continue to write in the memory

cache; however, if it reaches a mark previously set by the query thread, it has to notify it about the ongoing write operation. Then the query aborts its search in the memory cache, because the memory cache's contents have become invalid with respect to the current query. Note that the content that has been overwritten in the memory cache has moved to the disk cache in the meantime. So the query simply continues by examining the according disk cache. For searches in the disk cache, there is no concurrency problem, since we organize the disk cache in files. The contents of a disk cache file do not change after it is filled. So the query thread can safely read files on disk.

Note that this access strategy includes that a query does lock data structures and block the storage thread. But the period of time this lock has to be established is rather short. It encompasses the time it takes to lookup a relatively small number of timestamp values and index intervals. This is very short compared to the time the actual search takes.

The notification of the query thread by the storage thread is implemented by a registration mechanism. A query object sets a mark in the memory cache object and signs up for notification of the write pointer reaching that mark. This leads to the query object being listed as a handler for that event in the memory cache object. If the event takes place, a flag variable in the query object is set and the loop performing the linear search breaks. Then the query continues as depicted in the previous paragraph.

5.1.4 Status of Implementation

In the scope of this project we built a prototype of an implementation for the recording system. The goal for the prototype was to show that storage of packets as depicted by the design in § 4 is practicable for the kind of networks and load analyzed in § 3. The prototype includes functionality for capture, storage, indexing, connection tracking and a basic query interface.

The configuration for traffic classification and storage parameters is done by changes to the source code. There is no dynamic configuration of classes. Indexes may be added by simple adaptations in the source code. We implemented classes supporting the indexing of IP addresses and TCP/UDP port numbers. In both cases, source and destination addresses (ports) have their own class with the definition of where in the packet to obtain the respective key. They are integrated in a class hierarchy, derived from a general IP address or port class, respectively, which implements the indexing of both source and destination address or port.

We did not implement a complete communication interface like that depicted in § 4.8. The interface to our prototype consists of a simple command line dialog

running on the program's standard input and output. Using this command line interface (CLI) the user is able to check the status and some statistics of the system. A preliminary filter may be set and changed during run-time. Queries may be entered on the CLI in a fixed format, together with a file name. The output of query results is limited to writing a packet trace to a new file on disk by the name supplied with the query request.

The prototype writes statistics about number of packets and bytes received, data rate and packet drops to a log file in given time intervals. This helps with the evaluation of the system performance on real-world traffic.

5.2 Evaluation

We ran our implementation on two of the machines in the sniffer lab that monitors the traffic on the MWN (see Figure 3.1), i.e. on real network traffic from a productive network. We described the MWN, its size and load characteristics in § 3.1. This section will present the qualitative results of the test runs.

The used sniffer host machines are systems with two AMD Opteron 1.8 GHz processors, with 2 GB main memory and 500 GB available disk space. The network interface to the measurement point is an Intel Pro/1000 XF card. We tested on both FreeBSD and Linux. During the tests that led to the experiences we will describe in the following paragraphs, we did not use special optimizations in the system's kernels, settings or capture mechanisms like those mentioned in § 5.1.2. Using the standard setups and configurations we did not see any significant difference in the performance of our program when run on either operating system (OS). We interpret this by suggesting that (i) the packet capture performance of both OSs does not differ as much as to make a big difference and (ii) that the major part of computation time (CPU time) is spent in the execution of the algorithms we implemented for the storage subsystem rather than the capture mechanism, and the execution time does not vary much between the OSs when the same C++ code is compiled on either with the same compiler optimization settings.

We ran the capture system in November 2004 on the MWN sniffer lab. On weekdays (Monday to Friday) we experienced a data volume of 2.1 TB per day average. During busy hours the data rate typically was above 350 Mbit/s hourly average, with more than 70,000 packets per second. Our configuration for the recording system set a connection cut-off size of 100 kB. We instantiated a number of indexes. It was obvious from the study of CPU load numbers and packet drop rates that the indexing subsystem creates a lot of load, strongly depending on the number of indexes that have to be kept. To get an idea of how much the influence of

number of indexes is, we tried two configurations. One establishes two indexes: source and destination IP address. The other uses source and destination IP address, source and destination port and general IP address. A general IP index keeps entries for both source and destination IP address. This creates a higher load than a specialized index, since it has to insert two keys for every packet. The indexes had a granularity of $\Delta T = 60$ s. Memory allocation for the system memory caches was such that the total system memory need for caching was 1 GB. We configured three simple classes, one storing all TCP (system memory cache 700 MB), the other all UDP (200 MB), and the third all other traffic (100 MB). Disk cache was allocated to a total of 500 GB: 350 MB for TCP, 100 MB for UDP and 50 MB for the “other” class.

When watching the performance of the recording system we looked at the percentage of packet drops—i.e. the number of packets dropped by the total number of packets that arrived at the tap—for a number indicating how good the system was able to cope the load. Unfortunately, during the time of testing at periods other capture applications ran concurrently on the same hosts. This led to generally higher packet drops. So it was difficult to obtain exact data. However, we tried to estimate some data representing performance of the system by repeating the experiments and gaining a feeling for the response in different situations.

The first qualitative conclusion was that the prototype implementation ran smoothly after removing bugs that typically were due to simple unthoughtfulness during programming. It did not crash or run out of memory due to possible memory leaks. While we can not prove that the design which is the basis for our prototype actually has promoted a solid implementation of a recording system, we can argue that it did at least not lead to a bad implementation.

With the set of two indexes, we typically had packet drops of under 0.5%, that could grow to around 1.5% at peak data rates. When we started the system with four active caches (with the general IP index actually indexing two search keys) the packet drops grew slightly to around 3% at peak load periods, but did not increase noticeably at lower load.

A query took anything from roughly 5 to 20 seconds, depending on the situation. We do not have exact numbers for query duration at hand. Of course it is possible to express queries that run for a far longer time, either because a lot of data is to be extracted or a very long period of time has to be searched. However it is important to note that execution of a number of successive queries did not have an influence on the performance of the storage subsystem as measured by the packet drop rate. Obviously enough, the separation of storage and query subtasks for concurrency was an important decision in this context. It would be a very bad idea to pause the storage task for that many seconds—nearly all packets during that time the query

takes would be lost.

We tried to estimate the system memory needs of the system in excess of the (fixed) cache sizes. These would include all the internal data structures, especially the index structures of the memory cache index. While we can not say how much memory the index structures use, we can say that all the system memory used additionally to the total cache size was typically around 100 MB and did never grow over 200 MB. This is a strong hint that our assumption that the indexes would not “explode”—use up all available memory and eventually crash the system—is valid.

Chapter 6

Conclusion

6.1 Summary

In this work we addressed an issue for intrusion detection we consider a hot topic at the time of this writing. There is no existing system that allows the systematical recording of network traffic and retrieval of specified traces for security monitoring. We showed how such a system integrates with intrusion detection and pointed out its benefits. Then we analyzed the workload such a system has to cope with and derived a strategy to reduce the volume of data. We concluded that from a theoretical point of view, a recording system is viable and devised a design of such a system. It features a clear structure, flexible configuration and powerful data handling designed for large volume.

While we did not implement all parts of the system we devised a design for, the prototype and the functionality it covers allowed us to show that the assumptions and design issues are valid and that the recording system will be a useful tool for practical use.

6.2 Future Work

There are a number of open issues in this field of work. There is no working implementation of a communication interface. We will have to implement such a subsystem. Furthermore, we need to “teach” Bro how to use the recording system. I.e., we have to develop policy scripts that implement the configuration and query communication protocol sketched in § 4.8.

When there is a working interaction between Bro and the recording system, we can start to gather more extended experience with the deployment of the system and evaluation of its usefulness and performance.

We believe that, starting from the work done in this project, we can build such a system that will be ready for productive use and will be a valuable tool for security monitoring.

Bibliography

- [Der03] Luca Deri. PF_RING: Passive packet capture on linux at high speeds. http://www.ntop.org/PF_RING.html, April 2003.
- [End04a] Endace Measurement Systems. Endace DAG Network Monitoring Interface Cards. <http://www.endace.com/networkMCards.htm>, December 2004.
- [End04b] Endace Measurement Systems. Extensible Record Format. <http://www.endace.com/support/EndaceRecordFormat.pdf>, October 2004.
- [JLM01a] Van Jacobson, Craig Leres, and Steven McCanne. *pcap - Packet Capture library*. Lawrence Berkeley National Laboratory, Berkeley, CA, January 2001. pcap Manual Page.
- [JLM01b] Van Jacobson, Craig Leres, and Steven McCanne. *tcpdump - dump traffic on a network*. Lawrence Berkeley National Laboratory, Berkeley, CA, 2001. tcpdump Manual Page.
- [Kre04] Christian Kreibich. Broccoli: The Bro Client Communications Library. <http://www.cl.cam.ac.uk/~cpk25/broccoli/manual/>, 2004.
- [Law04] Lawrence Berkeley National Laboratory. tcpdump and libpcap. <http://www.tcpdump.org/>, January 2004.
- [McA04] McAfee. McAfee Security Forensics. http://www.mcafeesecurity.com/us/_tier2/products/_media/mcafee/ds_secur%ityforensics_lo.pdf, December 2004.
- [MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter 1993 Conference*, pages 259–270, 1993.

- [Nik04] Niksun. NIKSUN NetDetector. http://www.niksun.com/Products_NetDetector.htm, December 2004.
- [Pax99] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, December 1999.
- [Roe99] Martin Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proc. 13th Systems Administration Conference - LISA '99*, pages 229–238, 1999.
- [SP04] Robin Sommer and Vern Paxson. Exploiting Independent State for Network Intrusion Detection. Technical Report TUM-INFO-11-I0420-0/1.-FI, Technische Universität München, 2004.