A Software System for Packet Trace Customization with Application to NIDS Evaluation

Diplomarbeit

Angefertigt unter der Leitung von
Prof. Ph.D. Anja Feldmann
am Fachbereich Informatik
der Universität des Saarlandes

von

Andy Rupp

Saarbrücken, 28. Februar 2004
Hiermit erkläre ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Andy Rupp

Saarbrücken, 28. Februar 2004
## Contents

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Structure of this Work ...................................... 2

2 Background 3
   2.1 Intrusion Detection .......................................... 3
   2.2 Evaluating Network Intrusion Detection
     Systems ........................................................... 5
     2.2.1 Criteria .................................................... 5
     2.2.2 Approaches ............................................... 8
   2.3 Related Tools for Trace Processing .......................... 11

3 Basic Trace Manipulating Operations 14
   3.1 Motivation: Usage of Traffic Traces in NIDS Tests .......... 14
     3.1.1 Interweavement of Attack and Background Traffic Traces 14
     3.1.2 Varying the Characteristics of Traffic Traces ............ 16
   3.2 The Operations ............................................... 19
     3.2.1 General Considerations when Applying Trace Manipula-
           tions ......................................................... 19
     3.2.2 Adapting Traces to a Particular Environment ............. 22
     3.2.3 Merging .................................................... 24
     3.2.4 Stretching and Compressing ................................. 26
     3.2.5 Removing .................................................. 32
     3.2.6 Moving ..................................................... 33
   3.3 Summary and Future Work ..................................... 33

4 An Advanced Pipes and Filters Architecture 34
   4.1 Motivation and Goals ......................................... 34
   4.2 Design Overview .............................................. 36
   4.3 The Component Interface ...................................... 39
   4.4 Inter-Component Communication ............................... 42
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Example Attack Trace</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Example Network Environment</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Adapted Attack Trace</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>RTO Calculation According to RFC 2988</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>RTO Calculation According to “TCP/IP Illustrated”</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Example Traces for Scaling and RTO Calculation</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Advanced Pipes and Filters Architecture Overview</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Role of “Pipes and Filters” Components within an Example Filter System</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>The Component Interface</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>I/O Interface Classes</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>The PluginRegistry</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>The MergeNetworkController</td>
<td>46</td>
</tr>
<tr>
<td>4.7</td>
<td>Topology of an Example MergeNetwork</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Configuration of an Example MergeNetwork</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Interaction of Architecture Parts during the Creation of a Component</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Interaction of Architecture Parts during the Connection of Components</td>
<td>53</td>
</tr>
<tr>
<td>4.11</td>
<td>MergeNetwork Computation Round 1</td>
<td>54</td>
</tr>
<tr>
<td>4.12</td>
<td>MergeNetwork Computation Round 2</td>
<td>54</td>
</tr>
<tr>
<td>4.13</td>
<td>MergeNetwork Computation Round 3</td>
<td>55</td>
</tr>
<tr>
<td>4.14</td>
<td>MergeNetwork Computation Round 4</td>
<td>55</td>
</tr>
<tr>
<td>4.15</td>
<td>Integration of a Scheduler/Computation-Event-Manager</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>The TTM Plugin Package</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>The PCapPacket Datatype</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>The PCapFileSource Plugin</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>Configuration of a PCapFileSource Instance</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>The PCapFileSink Plugin</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>Configuration of a PCapFileSink Instance</td>
<td>68</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

5.7 The PCapPacketSorter Plugin . . . . . . . . . . . . . . . . . . . . . 69
5.8 Configuration of a PCapPacketSorter Instance . . . . . . . . . . 69
5.9 The PushPullPipe Plugin . . . . . . . . . . . . . . . . . . . . . . . 70
5.10 Configuration of a PushPullPipe Instance . . . . . . . . . . . . . 70
5.11 The IPSpoofer Plugin . . . . . . . . . . . . . . . . . . . . . . . . 71
5.12 Configuration of an IPSpoofer Instance . . . . . . . . . . . . . . 72
5.13 The Common Structure of a Spoofing Rule . . . . . . . . . . . . . 74
5.14 Two Attack Traces . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.15 Section of Background Traffic Trace bgt_1 . . . . . . . . . . . . . 76
5.16 Configuration: Interweavement of Background and Attack Traffic 77
5.17 Section of the Interweaved Background and Attack Traffic . . . . 78
# List of Tables

2.1 NIDS Capacity Verification Metrics .................................. 7  
3.1 Scaling and RTO Calculation ........................................... 31  
4.1 MergeNetworkComponent Methods ..................................... 40  
4.2 ConnectionSide Methods ............................................... 42  
4.3 Input/OutputSide Methods .............................................. 43  
4.4 PluginInterface Methods ............................................... 45  
4.5 ComponentContainer Methods ......................................... 47  
4.6 MergeNetworkController Methods ..................................... 48
Chapter 1

Introduction

1.1 Motivation

For several years already, commercial, military and academic organizations are very interested in deploying intrusion detection technology to protect their networks and hosts. Larger organizations even consider these systems besides firewalls as part of their standard security equipment. Since there exist a variety of systems using different approaches to intrusion detection, each having its strengths and weaknesses, customers have a strong demand for quantitative tests in order to compare systems and assess which is most suitable for them. Of course, also researchers need such evaluations for locating weak points in current intrusion detection systems and to focus their improvement efforts.

However, no scientifically accepted evaluation methodology exists at present and the results of previous testings are limited in their expressiveness and partially flawed or biased [34]. This is due to the difficulties in defining meaningful metrics, but particularly because of the complexity and expense of designing and implementing appropriate tests: The prevalent testing approach attempts to simulate realistic conditions within a testbed by generating network traffic which should have the typical characteristics of a particular network environment and by actually launching attacks against testbed hosts. Since this approach is very expensive and time-consuming, researchers [26] have proposed to make use of prerecorded normal and attack traffic (so-called packet traces or traffic traces).

The motivation behind this work is to build a basis for further efforts towards a trace-aided evaluation methodology. Therefore one part of this thesis identifies and discusses a set of basic trace manipulations which are necessary to customize traces for evaluating intrusion detection systems. The main part of this thesis concerns the design and implementation of a flexible and expandable software system that should enable the realization of arbitrary trace customizations. Trace
manipulating operations can be implemented as small independent components (i.e., plugins) of an abstract architecture which allows a freely chosen order in which they are applied. In this way users can perform complex customizations of traffic traces. Moreover, input and output operations can also be realized as plugins. Thus it is easy to enhance the software system in a way that it is able to, e.g., retrieve traffic traces from a database, perform arbitrary manipulations and send the resultant traffic directly onto a network. These and other features make the developed software system an ideal tool to experiment with and improve trace-aided evaluation.

1.2 Structure of this Work

Chapter 2 of this thesis presents an overview of approaches to intrusion detection, intrusion detection system evaluation and prior as well as recent projects that are related to our software system. In Chapter 3, we identify and discuss basic trace manipulating operations that we consider useful in order to customize traces for evaluating intrusion detection systems. Chapter 4 covers the design and implementation of the first part of our software system for trace customizations, i.e., a flexible and extensible architecture for multipurpose filter systems. Finally, Chapter 5 presents the second part of this software, i.e., a plugin package for trace manipulations, and an exemplary application of the software.
Chapter 2

Background

This chapter gives an introduction to the field of intrusion detection in Section 2.1 and discusses difficulties of evaluating network intrusion detection systems in Section 2.2. Furthermore, Section 2.3 presents an overview of tools and projects that are related to our future goal of trace-aided evaluation and our software system.

2.1 Intrusion Detection

Heady et al. [15] define an intrusion as “any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource”. Attacks are concrete instances of intrusions. Usually both terms are used synonymously.

The purpose of an intrusion detection system (IDS) is to detect, identify and respond to this form of malicious activity targeted at computing and networking resources. An IDS can be classified by its monitoring scope and type of data that it examines in its attempt to detect attacks. A host-based IDS (HIDS) typically monitors a single host or a small set of hosts and analyzes information residing on that host(s), like application or operating system log files. Within this thesis we are only interested in network(-based) IDSs (NIDSs) which observe the network traffic going to and coming from the monitored systems.

A detailed discussion of the strengths and weaknesses of both approaches can be found in [34]. One advantage of the network-based approach is that some number of hosts can be monitored simultaneously and therefore it enables the detection of attacks targeted at multiple hosts. It has the disadvantage that attacks that do not traverse the monitored network segment are not detectable. This includes attacks launched at the system console of an host, attacks originating from and directed at hosts belonging to the trusted local area network or attacks whose traffic takes an alternate (i.e., unobserved) route to the attacked host. To over-
come this disadvantage an NIDS can have multiple sensors which collect traffic at different points of a network. Moreover, the analysis of the collected data can be decentralized to distribute the increased computational load. It is typical for sensing and analyzing to occur on the same platform. Most (network) intrusion detection systems are based on one (or both) of the following two major detection principles:

**Anomaly-based** detection assumes that all malicious activities are different from normal activities and therefore observable as anomalies. To identify these deviations an anomaly-based IDS must know the profile of normal activity within the environment it should be deployed. Normally, this type of IDS is able to “learn” the necessary characteristics by itself from benign training data which is typical for the environment. Of course, in doing so one risks that the IDS also learns to accept attacks as normal activities. Hence, the differentiation and characterization of normal and anomalous activity is the subject of ongoing research.

**Misuse-based** detection attempts to find known patterns or signatures of attacks within the gathered data. The signatures needed for this are mostly created manually as new types of intrusions become known and are then inserted into the signature database of the IDS. This is similar to the principles used by anti-virus software. However, the specification of good attack signatures is a difficult and time consuming task. An ideal signature is abstract enough to still match (slight) variations of an attack but not too general so that it also matches benign activity. It should be pointed out that most commercial IDSs use misuse-based detection mechanisms.

If we compare both detection approaches, we can generally say that misuse-based systems are less susceptible to false alarms than anomaly-based IDSs. This is because anomaly-based systems detect per definition unusual activity. Yet, unusual activity is not necessarily an attack in all cases. For the same reason it is difficult for this type of systems to precisely identify a detected attack (i.e., label it with a name and provide additional information). Since signatures correspond to concrete attacks, these systems have it easier to identify attacks than the other systems. On the other hand by knowing the characteristics of regular activity an anomaly-based IDS has the possibility of detecting truly novel attack types. To combine the advantages of both approaches hybrid systems have been developed. A survey of existing systems and a more detailed introduction to intrusion detection can be found in [4] and [25].
2.2 Evaluating Network Intrusion Detection Systems

This section presents some measurable characteristics of network intrusion detection systems and discusses approaches to collect such measurements. Thereby we follow and extend the discussion in [26].

2.2.1 Criteria

In order to determine and compare the quality and performance of different intrusion detection systems we are interested in meaningful evaluation criteria. Since no standard set of criteria or metrics has been defined so far, this section summarizes and discusses measurements which have been suggested by several evaluation approaches [26]:

- **Coverage**
  One of the basic characteristics an NIDS evaluation should determine is the maximal subset of all known attacks detectable by a specific NIDS under ideal conditions. For a misuse-based NIDS this set could be determined by using the inherent mapping between signatures and attacks. But since there exist no standardized naming scheme for attacks, it is difficult to compare the coverages if they are derived from the signature databases of various NIDSs. For non-misuse-based systems, we have to test or analyze which attacks out of the set of all known attacks are detectable by a particular methodology. However, this is not feasible due to the huge number of known (and unknown) attacks. Moreover, there exist no database containing all attacks known to the security community. Therefore the set of all known attacks should be reduced to a smaller representative subset by selecting one or more attacks from each class of an equivalence partitioning of the set [33]. Ideally, the equivalence relation should be chosen in a way that the NIDS can detect either all attacks belonging to a class or none of them. Unfortunately, the classes set up by existing classification approaches do not fulfill this property\(^1\). However, they could be useful to approximate the attack coverage of an NIDS.

- **Probability of False Alarms**
  Another important characteristic [5] is the probability of false alarms (aka false positives) generated by an NIDS in a specific environment during a particular period of time. False positives are alerts caused by normal non-malicious background traffic. Generally, it is not easy to determine which

---

\(^1\)Current classification attempts do not even define mutual exclusive classes.
aspects of the traffic are responsible for false positives generated by a particular NIDS. The false positive rate that is measured within the scope of an NIDS evaluation is dependent on the traffic characteristics of the test environment. Since there exist no standard network environments which can be used as references, it is difficult to derive from an evaluation which rates can be expected in a different environment.

- **Probability of Detection**
  Another important NIDS characteristic is the probability of attack detection in a specific environment during a particular period of time. This measurement is the opposite of the so called **false negative** rate, i.e., the rate of missed attacks. The expressiveness of a measured detection rate is restricted because it depends on the set of attacks chosen for the test and the traffic characteristics of the test environment. Moreover, the sensitivity of an NIDS can usually be adjusted to increase (decrease) the probability of detection which on the other hand also increases (decreases) the probability of false alarms. Thus, the false positive and detection rate of an NIDS have to be measured using the same configuration.

- **Ability to Identify an Attack**
  The ability of an NIDS to label an attack with a common name, an attack category and additional information (e.g., IP(s) of attacking host(s)) is a valuable feature for further analysis and counter measures. However, the absence of a common naming scheme and attack classification makes it difficult to verify the correctness of attack identifications.

- **Ability to Determine Attack Success**
  The ability of an NIDS to distinguish between failed and successful attacks allows to prioritize alerts. Busy analysts can focus their work on those high priority alerts and hopefully prevent further damages.

- **Ability to Detect Novel Attacks**
  This measurement determines whether an NIDS is able to detect attacks that are unknown to the security community. As we know from the previous section this is not applicable to misuse-based systems.

- **Ability to Correlate Events**
  The ability of an NIDS to correlate attack events is important for the detection of complex attacks that involve multiple steps. Events can be generated by different sources like, e.g., multiple NIDS sensors, routers and firewalls.

- **Resistance to Attacks Directed at the NIDS**
  This measurement determines whether an NIDS is resistant to attacks targeted at the NIDS itself (e.g., Denial of Service attacks).
### Table 2.1: NIDS Capacity Verification Metrics

<table>
<thead>
<tr>
<th>Test metric</th>
<th>Affected bottlenecks</th>
<th>Used Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packets per sec.</td>
<td>Packet Capture Archit., Packet Flow Archit.</td>
<td>CPU cycles, network interface bandwidth, memory bus bandwidth</td>
</tr>
<tr>
<td>Bytes per sec. (average packet size)</td>
<td>Packet Capture Archit., Packet Flow Archit.</td>
<td>CPU cycles, network interface bandwidth, memory bus bandwidth</td>
</tr>
<tr>
<td>Protocol mix</td>
<td>Packet Analyzer</td>
<td>CPU cycles, memory bus bandwidth</td>
</tr>
<tr>
<td>Number of unique hosts</td>
<td>State Database</td>
<td>Memory size, CPU cycles, memory bus bandwidth</td>
</tr>
<tr>
<td>Number of new connections per sec.</td>
<td>State Database</td>
<td>CPU cycles, memory bus bandwidth</td>
</tr>
<tr>
<td>Number of concurrent connections</td>
<td>State Database</td>
<td>Memory size, CPU cycles, memory bus bandwidth</td>
</tr>
<tr>
<td>Alarms per sec.</td>
<td>Alarm Reporting Engine</td>
<td>Memory size, CPU cycles, memory bus bandwidth</td>
</tr>
</tbody>
</table>

- **Resistance to NIDS Evasion Techniques**
  This measurement determines whether an NIDS is able to detect stealthy versions of attacks. [32, 45] present techniques that have been used to evade NIDSs. Simple techniques are, for example, IP fragmentation and URL encoding.

- **Capacity Verification**
  In order to detect attacks, NIDSs perform deep packet inspection\(^2\) and keep track of the state of observed packet flows. Since this task can be very time and resource consuming, it is important to measure the ability of an NIDS to capture (i.e., record packets), process and perform at the same level of accuracy under a given network load as it does on a quiescent network. Mike Hall and Kevin Wiley [13] have identified typical bottlenecks of NIDS architectures and the appropriate traffic parameters (metrics) to stress them. Following major stress points within NIDS architectures have been located by them:

  - The **Packet Capture Architecture** is the hardware/software part of an NIDS architecture that is responsible for packet capturing.

---

\(^2\)NIDSs analyze the full content of a packet including its headers and application-layer payload.
– The Packet Flow Architecture is the overall architecture for data flow within an NIDS and includes the Packet Capture Architecture.

– The Packet Analyzer is the part of an NIDS that is responsible for protocol decoding and payload inspection (i.e., deep packet inspection).

– The State Database of an NIDS is responsible for storing and retrieving information that are necessary to perform state tracking. Typical information of this kind are, e.g., the source and destination of captured packets, the state of TCP connections and the type and number of packets going to or coming from observed hosts.

– The Alarm Reporting Engine is responsible for the generation and logging of alarm events.

Table 2.1 shows which of these bottlenecks and involved computing resources are mainly affected by a specific test metric. Mike Hall and Kevin Wiley hope that by profiling their own networks, customers can use the results of standardized capacity tests to assess whether the performance of an NIDS is suitable for their environment and satisfies their requirements.

Due to complexity concerns existing NIDS testing efforts only focus on the evaluation of a small subset of the above criteria. The measurements that have received the most attention have been the detection rate, the false positive rate and the resistance to evasion techniques.

### 2.2.2 Approaches

Although there is a great demand for quantitative NIDS evaluations, at present no commonly accepted evaluation methodology exists. This is due to the difficulties in defining meaningful criteria, but particularly because of the complexity and expense in designing and implementing tests to measure these criteria. As we have seen in the previous section, for most of these measurements one need to provide an NIDS with attacks and appropriate background traffic. Therefore the main issues one have to cope with in order to evaluate NIDSs are:

- How to test an NIDS with attacks?
- How to obtain appropriate background traffic?

In the following we discuss approaches to both issues and difficulties that are concerned with these approaches.

\(^{3}\)“appropriate” usually means realistic and non-malicious.
How to test an NIDS with attacks?

To tackle this problem two approaches have been proposed:

- **Using Attack Scripts**
  A common approach is to create a testbed network with attacker and victim hosts and place the NIDS into this network. For that purpose attack scripts (aka *exploits*) and appropriate victim software must be collected, installed and customized. This is an expensive and very time-consuming task\(^4\) which hampers progress in the field of NIDS evaluation. Although exploits are widely available on the Internet, it takes time to find those which are suitable for a particular testing environment. This is due to the fact that each attack script runs on a particular operating system and works only against a particular version of a victim software which must be configured in a particular way. According to [26], experiences have shown that if a suitable exploit has been found it roughly takes one person-week to review (and correct) the code, test it, determine where the attack leaves evidence, automate it and embed it into a testing environment. Traditionally, **expect** [10] has been used to automate attacks which is a common tool for automating interactive applications. A newer tool called **Thor** [23] has been developed especially for NIDS evaluation purposes and is able to automate attacks, apply evasion techniques and collect generated alerts. Apart from the problems concerning attack scripts the acquisition of victim software can also be difficult. For example, the special version of a vulnerable software which is required by an attack may not be (easily) obtainable from the vendor.

- **Using Attack Traces**
  Mell et al. [26] propose to analyze whether the use of attack traces is an alternative to launching attack scripts on a testbed. Attack traces contain the traffic which flows between the attacking and the victim host(s) when the attack is launched. Because this is exactly what an NIDS can observe, an attack instance can be specified by its traffic trace. In the scope of an evaluation, attack traces can either be replayed separately or they can be merged with recorded background traffic. Then the resulting trace can be replayed. We analyze the latter approach in Section 3.1.1. If there exists an extensive database of attack traces, the costly task of collecting and customizing attack scripts and victim software for each individual testbed can be omitted or at least alleviated. Moreover, such a database is strongly demanded for related tasks such as the development of attack signatures. Since a distributed generation of attack traces is possible, e.g., by recording attack traffic in different testbeds or by extracting it from captured real-world traffic, the development of such a database should not be that difficult.

\(^4\)Within the scope of an earlier project we assured ourselves of this fact.
How to obtain appropriate background traffic?

Three different approaches have been proposed regarding the task of choosing background traffic:

- **Generating Background Traffic**
  The most common approach is to generate background traffic in the testbed network that is used to launch the attacks. Within the scope of the most extensive IDS evaluation efforts performed by MIT Lincoln Laboratory [21, 22], one has tried to recreate the real-world network traffic produced by a small Air Force base. Therefore complex software automata have been developed in order to "simulate hundreds of programmers, secretaries, managers and other types of users running common UNIX application programs and some Windows NT programs". Moreover, custom Linux kernel modifications have been applied to allow a small number of actual hosts to appear as thousands of hosts with different IP addresses. Unfortunately, neither the original software nor the resulting Lincoln Adaptable Real-Time Information Assurance Test-bed (LARIAT) [35] is publicly available. Other NIDS evaluation efforts, like those of the NSS group [28], use commercial traffic generators to generate background traffic in their testbed network. The advantage of this common approach is that the generated background traffic can be recorded and distributed freely since it contains no private or sensitive data. Moreover, the traffic does not contain malicious activity because it has been generated artificially. Another advantage is that NIDS tests involving a traffic generator are easily repeatable by just reusing the same configurations for the generators. However, this approach has also some major drawbacks. Modelling and simulating the traffic characteristics of a specific environment (or generally real-world traffic) is a very costly and time-consuming task. Furthermore, commercial traffic generators which facilitate traffic generation are every expensive. Moreover, these generators are fairly restricted in terms of the type and characteristics of traffic they are able to produce. Finally, since traffic is generated artificially it is likely to have some unrealistic proper-ties, e.g., with regard to payload content.

- **Using Real Traffic**
  Another less common approach is to replay or mirror traffic from a real network onto the isolated testbed network where the attacks are launched. For example, Neohapsis Laboratories have mirrored live traffic from De-Paul University in Chicago to perform their most recent IDS evaluation [36]. Generally, this is an optimal approach to determine detection rates within a particular environment because real traffic is used as background
CHAPTER 2. BACKGROUND

load. Moreover, the detection rate of an NIDS can be measured at different levels of activity since the traffic load varies over time. However, there may be problems if a small set of victim hosts are used for the sole purpose of being attacked [26]. Anomalies-based systems may learn that all packets destined for these special hosts are malicious. In this way detection rates can be artificially increased because this is not a realistic situation.\(^5\)

The major drawback of using real traffic for NIDS evaluations is that we cannot guarantee that it contains no (unidentified) attacks. Even if we use recorded traffic instead of directly mirroring live traffic, it is not feasible to identify all malicious activity within these large traces (by manual analysis). As a consequence a determination of false positive rates is difficult when using real-world traffic.

This approach also suffers from the general problem of accurately replaying traffic traces (at high speeds). Since many NIDSs base their decisions and actions on whether certain events occur within certain periods of time, it is important to preserve the inter-arrival times of the recorded packets (given by their timestamps) when replaying.

Another disadvantage of using real traffic for testing is that it is problematic to make the test data publicly available for verification or reuse due to privacy concerns.

- **Using Sanitized Traffic**

  In order to tackle privacy problems concerned with sharing real traffic datasets, researchers have proposed to sanitize traces by removing all sensitive information. Since NIDSs perform deep packet inspection, sanitizing cannot imply simply striping off packet payloads and anonymize packet headers. Therefore Ruoming Pang and Vern Paxson have developed a tool [29] that allows anonymization of both packet headers and payloads. However, poor sanitization may remove or forge too much of the packet contents and thereby make traffic traces look unrealistic. The other more serious risk is that sanitization may fail to garble all sensitive data.

2.3 Related Tools for Trace Processing

Traffic traces have been a valuable source for research and analysis in various areas of computer networking. The discussion of Section 2.2 shows that they can also be useful tools to overcome the drawbacks of the most common NIDS evaluation methodology. For that purpose network traffic must be recorded, certain trace modifications must be applied (e.g., merging, sanitizing) and traffic traces

\(^5\)Except for the hosts of so called honeynets.
must be replayed onto a (testbed) network. The following tools implement some of these operations:

- **tcpdump** is the most commonly used tool for network monitoring, packet capturing and protocol analyzing. *tcpdump* is based on the Packet Capture Library (*libpcap*) which has been developed along with it. This library provides a system-independent interface for user-level packet capturing, enables the use of kernel-space packet filtering mechanisms (BPF [24]) and defines a standard format for trace files (*pcap* files). Because of these features all listed tools and several network intrusion detection systems (e.g., Bro [30]) are build on top of *libpcap*. Further information about *tcpdump* and *libpcap* can be found at [39].

- **Ethereal** is a comprehensive GUI-based network protocol analyzer. It can examine (and capture) packets from a live network or from trace files of various formats. Users can interactively browse through the captured data and view summary and detail information about each packet. More about Ethereal can be found at [9].

- **tcpslice** is a simple tool written in C that can be used to extract portions of *pcap* files. In addition, it allows to glue several files together. The current version of *tcpslice* can be found at [44].

- **bro-anonymizer** is a recent extension to the network intrusion detection system Bro. It has been presented at SIGCOMM 2003 after the completion of our filter architecture. This tool uses Bro’s application parsers and policy language to anonymize (i.e., remove sensitive information) traffic traces. Also it allows users to write scripts in order to realize other kinds of trace transformations. Details about this anonymizer can be found in [29].

- **NetDuDE** (NETwork DUmp data Displayer and Editor) is a GUI-based tool that allows a user to edit packets contained in a *pcap* file interactively. Moreover, *NetDuDE* allows to write plugins for protocol analysis and packet processing, like, e.g., a checksum fixer plugin which is already included. This tool has been primarily developed to enable the manual introduction of anomalies within traffic traces in order to test a traffic normalizer element (*norm*) for network intrusion detection systems [14]. *NetDuDE* has been extended in several directions, since the version we have reviewed (an early release of Version 0.4.0). For example, *libnetdude* has been developed which is a C-library of functions and data structures for the manipulation of *pcap* files. More information about *libnetdude* and *netdude* can be found at [27].
• tcpdump is a well-known packet replay engine based on libpcap and libnet [20]. It offers as basic operation to replay multiple pcap files back-to-back at the speed at which they were recorded. Alternatively, a speed rate can be specified which tcpdump tries to maintain during replaying. Moreover, it enables the selective sending or dropping of packets and some packet modifications on-the-fly, like randomizing IP addresses or rewriting the destination MAC of each packet. Since the version of tcpdump that we reviewed before starting the development of our architecture (v1.4.beta2), some other useful features like, e.g., an IP range remapping mechanism have been added. More details about this tool can be found on its website [43].

• TCPivo is a new packet replay engine which has been presented at a workshop during SIGCOMM 2003 [11]. Unlike tcpdump much effort has been spent in its design and implementation to address the problems of accurate replay and high throughput. By employing novel mechanisms for managing trace files and low-overhead timers, TCPivo is able to accurately replay network traces at high-speed on commodity hardware. TCPivo’s source code\textsuperscript{6} and the necessary kernel patches can be downloaded at [41].

\textsuperscript{6}currently a 92 MB sized tar-file.
Chapter 3

Basic Trace Manipulating Operations

Current NIDS testing efforts suffer from difficulties in collecting and customizing attack scripts and victim software. In addition, the generation of appropriate background traffic is also a very costly and time-consuming task. To tackle these problems researchers have suggested to utilize attack and (sanitized) background traffic traces [26]. Though this approach would have several advantages, it has not been explored in more detail yet. This chapter advances the idea of trace-aided NIDS evaluation by discussing basic trace manipulating operations which are motivated by exemplary uses of traces in NIDS tests.

Section 3.1 presents two example scenarios of when and how traces can be used to evaluate certain criteria of NIDSs. These scenarios suggest a couple of basic trace operations that can realize the required customizations. Section 3.2 discusses the identified operations in more detail.

3.1 Motivation: Usage of Traffic Traces in NIDS Tests

This section presents two example scenarios of how traces can be used in order to determine the false positive/negative rates and capacity of an NIDS.

3.1.1 Interweavement of Attack and Background Traffic Traces

The most obvious use of traces within the scope of an NIDS evaluation (which has already been proposed) is to determine the false positive/negative rates in
a particular environment. These figures are classical performance metrics that are included in nearly all testing efforts. To create an appropriate test trace for replaying we need a trace of non-malicious background traffic and a set of attack traces. In the ideal case, the background traffic is real traffic from the network environment the NIDS is tested in. Moreover, if we want to test the detection range (i.e., the coverage) of an NIDS the set of attack traces should ideally equal a representative cross section of all possible attacks.

Let us assume that we have some set of attack traces that we want to interweave with a non-malicious background traffic trace. If the background traffic and the attack traffic have been collected at different environments it might be necessary to adapt (Section 3.2.2) the attack traffic to the network environment where the background traffic has been recorded. The purpose of this adaption is that attack traffic should not stick out from the background traffic. Rather an attack should look like it is directed against or originates from a system in the local network environment implied by the background traffic. A basic adaption may even be required to actually enable an NIDS to detect the injected attacks. This is because the NIDS might be aware of the properties (policy) of the local environment and, as a result, ignore any packets that do not match these properties. A more detailed discussion of the conditions and how to realize the adaption is given in Section 3.2.2.

After adapting a trace file we can merge (Section 3.2.3) the prepared attack traces with the background traffic trace, i.e., we inject them at certain points into the trace. One should keep sufficient spaces between the injection points to simplify the correlation of alerts with attacks later on. Moreover, one may try to make the test trace appear more realistic by simulating some kind of interaction between the attack traffic and the background traffic. For example, the occurrence of regular traffic from and to a specific FTP server needs to be avoided, e.g., by removing (Section 3.2.5) this traffic, after injecting an attack which normally crashes this server. Otherwise, an NIDS that is able to detect a successful attack (by passive means) might get confused.

If the preparation of the test trace is completed, one can replay it and calculate the false positive and negative rates from the output of the NIDS. At the same time, one can also determine the ability of an NIDS to identify these attacks. If the replay engine accurately meets the original packet timestamps, one can additionally measure the delay between the occurrence of an attack on the wire and the generation of the corresponding alert.

It is recommended to repeat these measurements under several stressful conditions. For that purpose one has to vary certain characteristics of the background traffic trace, as described in the next section, and then inject the same attack traces.

Finally, it should be pointed out that this testing approach has some limitations: First, it seems to be difficult to properly test an NIDS that performs some
type of active monitoring, e.g., an NIDS that queries a host that has been attacked in order to detect attack success. Furthermore, it may be difficult to generate appropriate traces for evaluating the detection capabilities of an NIDS that has multiple sensors distributed over the monitored network.

3.1.2 Varying the Characteristics of Traffic Traces

The great advantage of a trace-aided evaluation approach is that it is easy to create traffic with different characteristics by reusing and manipulating available traffic traces. The ability to vary traffic characteristics is essential to test how an NIDS performs under different (stressful) conditions.

Mike Hall and Kevin Wiley [13] have identified typical bottlenecks of NIDS architectures and the corresponding traffic characteristics (parameters) to stress them (cp. Section 2.2.1). This section presents some ideas about how to adjust these parameters using simple trace manipulating operations. Each method attempts to only change one parameter without affecting the others. This is important for benchmarking each bottleneck accurately. However, it is only possible to a certain extent since some parameters are closely related by nature.

Most of the ideas described in the following rely on the existence of a database containing real (sanitized) and synthetic traffic traces. Ideally, this database should be publicly available as demanded in [26]. Just as discussed in Section 3.1.1 it is useful to adapt the traces to the particular environment before using them. The adjustment of most parameters described in the following involves the removal or adding of TCP, UDP or ICMP packet flows. Such a flow, as we use the term it in this thesis, can be defined as the sequence of packets exchanged between two endpoints during a certain period of time. The exact semantics of “endpoint” and “period of time” depends on the context, i.e., the protocol we are interested in. How we interpret this definition with respect to the protocols TCP, UDP and ICMP is shown in Section 3.2.1.

Protocol Mix

The protocol composition of the monitored traffic has a great effect on the load of the packet analyzer of an NIDS. If the goal is to modify multiple characteristics this one should be adjusted first because it highly affects the others.

We suggest to achieve the desired protocol mix by merging flows (possibly from different trace files) that belong to specific application-layer protocols into a given trace file or removing them from it. To keep traces realistic it should be

\[1\] For the purpose of capacity verification it may not be necessary that the available traces are free of attacks.
avoided to add the same flow several times. However, if the number of adequate flows in the database is insufficient, one can duplicate and manipulate flows in a way such that they look (to a certain extend) different.

In this fashion, one can enrich the protocol mix of a synthetically generated long-time trace and therefore make it realistic\(^2\).

Subsequent adjustments to other parameters that involve the removal or addition of flows should try to keep the protocol mix. This can be done by selecting the right mix of flows.

**Average Packet Size**

The packet size also affects the time that an NIDS needs to perform its analysis. The average packet size has to be adjusted along with the protocol mix, e.g., by choosing flows with appropriate packet sizes. In this way we can also try to keep this trace characteristic if we want to adjust other parameters by adding or removing certain flows.

**Number of Packets/Bytes per Second**

The number of packets and bytes per second affects the performance of the packet capture/flow architecture of an NIDS. They are closely related and cannot be changed independently without altering the average packet size, too.

A simple way to increase/decrease the average value of both parameters is to compress/stretch (Section 3.2.4) the whole trace. By using this method the number of new/concurrent TCP connections per second is somewhat affected. Moreover, the length of the generated trace compared to the original trace directly corresponds to the stretching/compressing factor. The modification of the length of a trace is not always a tolerable side effect. Other difficulties about stretching or compressing traffic traces can be found in Section 3.2.4. As an alternative we can also alter these parameters by adding or removing flows at appropriate areas of a trace.

Apart from adjusting the average number of packets/bytes per second a modification of the mean deviation of the average might be a reasonable goal. For example, one might want to establish a consistent work load for the NIDS regarding the packet/bytes it has to capture per second. In this case we can move (Section 3.2.6) flows from areas above the average (e.g., bursts) to areas that fall below the average value. Of course, the applicability of this idea must be carefully examined.

\(^2\) Usually this kind of trace file only contains traffic belonging to one or a few application-layer protocols due to the restricted modelling capabilities of the traffic generator.
Adjustments of other parameters that involve the removing or adding of flows affects the number of packets/bytes per second. Sometimes this side effect can be limited by choosing sparse flows.

**Number of Unique Hosts**

This parameter directly affects the size and thus the performance of a database inside an NIDS which tracks the state of monitored traffic flows. The number of unique hosts that occur within a traffic trace can be adjusted by remapping IP addresses. That means, if one wants to decrease the number of unique hosts the corresponding set of IPs is remapped to a smaller set. To increase this parameter, an IP address that is involved in multiple TCP/UDP/ICMP flows as source or destination address is replaced in each flow by a new unique address. The advantage of this method is that the other parameters are not affected. The number of unique hosts can be further increased by adding flows from and to hosts whose IPs are not already contained in the base trace. However, in doing so other traffic characteristics are somewhat changed.

To preserve the number of unique hosts when other parameters are adjusted, e.g., by adding flows, the IP addresses of these flows have to be made consistent to the ones already used in the trace.

**Number of Concurrent TCP Connections**

The number of concurrent TCP connections usually affects the performance of the state database. This parameter is closely related to the average connection duration of a traffic trace. Often the number of simultaneous connections is high/low for traces that mostly consist of long-lived/short-lived connections. Accordingly, a first approach for adjusting this parameter is to stretch or compress the TCP connection flows contained in the trace. Of course, the average packets/bytes per second is changed as well. Section 3.2.4 describes this operation in more detail.

We can further increase/decrease the number of concurrent connections by adding/removing sparse long-lived TCP connection flows. Appropriate candidates are, e.g., idle TELNET and SSH sessions or HTTP sessions that use persistent connections to transmit a few small objects. IP remapping helps here to preserve the number of unique hosts.

The number of concurrent connections can be preserved by selecting only short-lived flows for addition or removal when modifying other parameters.
Number of New TCP Connections per Second

The number of new TCP connections per second affects the number of insert operations of the state database. Thus, this parameter may have a great impact on the performance of the NIDS.

One approach to adjust this parameter is by removing or adding TCP flows at stress points. Stress points may be caused by HTTP sessions that create a large number of new (non-persistent) connections within a short period of time to transfer embedded objects. Therefore, our first attempt to adjust this target parameter consists in adding/removing several non-persistent connections or whole HTTP sessions. However, the protocol mix is affected thereby. Yet, it is possible to reestablish the desired mix afterwards by the addition or removal of packet flows belonging to other application-layer protocols.

Again the TCP connection flows that we add or remove should be selected appropriately (i.e., sparse, short-lived, ...) to preserve the other traffic characteristics.

3.2 The Operations

The previous sections discussed several options for customizing traffic traces to evaluate certain aspects of network intrusion detection systems. This section discusses the basic trace manipulating operations introduced so far in more detail. In particular, we analyze whether these operations generate any irregularities that cannot occur in real traffic and if these artifacts are recognizable by an NIDS. We start our discussion with some general considerations when applying trace manipulations.

3.2.1 General Considerations when Applying Trace Manipulations

Section 3.1 motivates the ability to add, move, remove or alter packet flows in trace files. In order to be able to do so, it is crucial to have the means of identifying flows within these trace files. In this section we discuss what information are provided by trace files and if these information are sufficient for identifying TCP, UDP or ICMP flows. Moreover, we consider the general problem of avoiding artifacts when applying trace manipulations.
Identifying Flows in Traffic Traces

As stated in Section 2.3, there exist several tools, like `tcpdump`, that are able to capture, decode and store packets that have been received on a computer’s network interface(s). By listening to an interface in “promiscuous mode” such a tool can record all traffic that is originated in, passing or sent to the network segment the listening host is connected to. In addition to watching the output of a capture tool in real time, e.g., for the purpose of network diagnostics, trace files can be generated that contain all data captured from the network.

Taking the output of `tcpdump` as an example, the trace file is actually constructed by the underlying `libpcap` library. Hence, every application that is able to understand this format can access and work with the data. For each packet seen on the network, the `libpcap` trace file format stores the following information:

- the actual data, i.e., any TCP/IP header information and payload beginning at the link layer.\(^4\)
- meta information like, e.g., packet timestamp, listening interface, link layer protocol, (real) packet size, (captured) packet size.

Taking the flow definition from Section 3.1.2 as a reference, we are able to verify whether trace files provide all information we need to identify flows and flow boundaries. The definition states that a flow is a sequence of packets between endpoints during a certain period of time. Let us discuss how to concretize the notions of “endpoint” and “period of time”.

In the case of a TCP or UDP (connection) flow an endpoint can be regarded equivalent to the tuple (IP address, port number) which is TCP’s and UDP’s way of addressing a specific application process on a specific host. Thus, the endpoints of a TCP/UDP flow can be uniquely identified by the 4-tuple that consists of

- source IP address and source port
- destination IP address and destination port

With ICMP, as the third flow type we are interested in, we are dealing with the network layer which unambiguously identifies its protocol entities and endpoints by IP addresses.

As we can see, by containing the protocol headers for IP and TCP/UDP respectively, trace files provide all information necessary to determine the endpoints of a

---

\(^3\)In fact, most capture tools enable the user to define filter criteria that restrict the amount of captured traffic.

\(^4\)As stated before, a “snap length” can be defined which limits the amount of collected data per packet.
flow. However, to definitely identify a flow as mandated by its definition, we must have a clear understanding of when the flow begins and when it ends. In the case of TCP, this is easy because TCP is a connection oriented protocol that explicitly signals the establishment and termination of a connection.  

UDP and ICMP, on the other hand, provide no simple means for deciding whether a packet still belongs to a flow or whether the flow has already been finished. Rather, we have to consult heuristics that inspect the application protocol in use. This protocol can be as simple as DNS which is in most cases based on a request/response pair that can be easily matched by spotting the endpoints.

On the opposite side of the spectrum, there are also quite complex protocols, e.g., real time protocols, that are based on UDP. Determining the temporal boundaries of these flows is only possible after a careful analysis of the carried application protocol.

Avoiding Artifacts

According to the Webster’s Online Dictionary an artifact is “something characteristic of or resulting from a human institution or activity”. In our special case we are working with traffic flows that shall be injected to a certain environment, i.e., trace file. In this context, we can define an artifact as any manipulation to the trace file that could be interpreted by the NIDS being tested in an unintended way.

Intuitively, this definition makes sense. However, the term “unintended” cannot be generally defined in a clear way. By adding a simple attack trace, for example, no statement is made about the actual attack vector, i.e., what details within the trace file really constitute the attack. That is, at least in theory each detail of an attack trace might contribute to the attack. Thus, it is not self-evident what parts of a trace file make up the attack and should therefore not be modified and which parts are allowed to be adapted.

Moreover, changing the characteristics of a trace file, as suggested in Section 3.1.2, without appropriate considerations can rapidly make the traffic look artificial and suspicious. This is even worse when the traffic is presented to an NIDS.

For this reason, the modifications that are applied to a trace file should be as minimal as possible and handled with great care. Thus, in the following sections we identify which operations are possible and what artifacts might result when these operations are applied to traffic traces.

---

5However, things can become rather tricky when we are facing packet drops, e.g., the “last ACK”, or other ambiguities that result from unknown OS-related endpoint semantics or network phenomena.
3.2.2 Adapting Traces to a Particular Environment

If traces are recorded in different environments, Section 3.1.1 and Section 3.1.2 suggest to first adapt them before applying further operations. The goal of this adaption is to make them look like as if they are from the same environment. Accordingly, the resulting trace can be expected to be more realistic. A motley mix of unadapted traces can be problematic if we test an NIDS that expects (e.g., via training) some specific environment. This applies especially for anomaly-based or other kinds of NIDS which collect (or must/should be configured with) information about their environment. They can be mislead to discard attack packets or generate false alarms just because of traffic that looks strange with respect to the expected environment.

Let us consider an example to find out which sort of adaptions should be performed. Suppose we want to adapt the attack trace depicted in Figure 3.1 to some background traffic trace that has been recorded at the network environment in Figure 3.2. The attack trace has been recorded at a different environment. It contains the network packets that have been exchanged between the attacking host with IP 192.168.0.1 and MAC 0:4:76:8b:1:8f and the victim FTP server with IP 192.168.0.2 and MAC 0:90:f5:7:a:e9.

Since the attack should appear to be directed at the FTP server of our local environment shown in Figure 3.2, we should first replace the server’s IP address (192.168.0.2) within the attack trace with the IP of our local server (134.96.65.27). Probably we should also change the private IP address of the attacker (192.168.0.1) to some public IP like, e.g., 131.159.14.1. If the background traffic has been captured at a router port like in our example environment, the port’s MAC address (0:0:0:0:0:a) is contained in all incoming packets (i.e., packets destined for the local network) as source address and in all outgoing packets as destination address. So if we do not want the attack trace to stick out within the background traffic, we should modify its Ethernet addresses as well. To complete the adaption we also adopt the MAC address of the local server.
FTP server’s Ethernet device (0:0:0:0:0:0b). The last modification is especially urgent if the background traffic already contains regular traffic from and to this server. Figure 3.3 shows the adapted attack trace. It has to be pointed out that other environments require other and possibly additional customizations of protocol parameters, e.g., UDP/TCP port numbers or TTLs.

However, one should be aware that so far the adaption only provides a minimal base. The data stream generated by some application layer protocols may also need adaptions along with the underlying UDP/TCP flow. Let us consider an FTP session as an example.

The FTP PORT command message, which initiates a new data connection between an FTP client and server, contains the IP address of the client and a port number for the new connection. So if we replace the IP of an FTP client in a TCP control connection flow, we should do the same for a transmitted PORT command message.

FTP also serves as an example for complications that result from the interdependence of flows. As indicated in the previous paragraph, FTP establishes two separate TCP connections for transferring control and bulk data. There exists a strong dependency between these flows, because the appearance of the data connection, i.e., IP addresses and port numbers, is mandated by a PORT command within the control channel. As a result, adjustments that are made to the control connection have to propagate to the data connection likewise.

Problems like this one are by no means restricted to FTP. ARP\(^6\) request/reply packets also have to be modified along with the replacement of related IP or MAC addresses.

---

\(^6\)Address Resolution Protocol. Used for the mapping between IP and MAC addresses.
addresses. Another example is RTP (Real Time Protocol) that announces port numbers in much the same way as FTP.

However, the degree of required adaption depends on the knowledge of an NIDS about its environment, the degree of its verification of whether the monitored traffic matches its knowledge and the degree of its sensitivity against irregularities it detects. Further work is needed to explore what level is sufficient for most NIDSs.

### 3.2.3 Merging

Most trace customizations considered in Section 3.1 involve merging some traffic traces. This section presents a description of the merging operation and discusses some of its difficulties.

Let us suppose that we have a set of different source traces that have to be merged. To preserve the inter-packet times of interdependent traffic they should be merged according to their packet timestamps (i.e., chronologically). Yet, the time periods over which the traces have been recorded do rarely overlap. Therefore a chronological merge using their absolute packet timestamps would merely result in a concatenation. If this is not desired we need to compute relative timestamps for the packets of each trace. Here, the first packet of each trace respectively serves as the reference point. After that, the first packet of each trace is labelled with timestamp 0. Then we can choose one of the traces (base trace) and inject the remaining traces at appropriate points into the selected trace.

Technically the injection of a trace at a certain point means adding a constant value to each timestamp of its packets, before merging it chronologically with the base trace. Depending on the traces, the right choice of injection times can make a difference from a statistical point of view. However, the injection points are sometimes fixed by higher level methods that make use of this operation (e.g., attacks should probably be injected at predefined points of the background traffic trace).

---

**Figure 3.3: Adapted Attack Trace**

<table>
<thead>
<tr>
<th>Src. MAC</th>
<th>Dst. MAC</th>
<th>Src. IP</th>
<th>Port</th>
<th>Dst. IP</th>
<th>Port</th>
<th>Prot. Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:0:0:0:0:a</td>
<td>0:0:0:0:0:b</td>
<td>131.159.14.1.32858</td>
<td>&gt; 134.96.65.27.21</td>
<td>S 2403014348:2403014348</td>
<td>(0)</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:b</td>
<td>0:0:0:0:0:a</td>
<td>134.96.65.27.21</td>
<td>&gt; 131.159.14.1.32858</td>
<td>S 605233624:605233624</td>
<td>(0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ack 2403014349</td>
</tr>
<tr>
<td>0:0:0:0:0:a</td>
<td>0:0:0:0:0:b</td>
<td>131.159.14.1.32858</td>
<td>&gt; 131.159.14.1.32858</td>
<td>.</td>
<td>ack 1</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:a</td>
<td>0:0:0:0:0:b</td>
<td>131.159.14.1.32858</td>
<td>&gt; 131.159.14.1.32858</td>
<td>. 1:1449 (1448)</td>
<td>ack 1</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:b</td>
<td>0:0:0:0:0:a</td>
<td>134.96.65.27.21</td>
<td>&gt; 131.159.14.1.32858</td>
<td>.</td>
<td>ack 2049</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:b</td>
<td>0:0:0:0:0:a</td>
<td>134.96.65.27.21</td>
<td>&gt; 131.159.14.1.32858</td>
<td>. 2049:2051 (2)</td>
<td>ack 1</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:a</td>
<td>0:0:0:0:0:b</td>
<td>131.159.14.1.32858</td>
<td>&gt; 131.159.14.1.32858</td>
<td>. 1:32 (31)</td>
<td>ack 2051</td>
<td></td>
</tr>
<tr>
<td>0:0:0:0:0:a</td>
<td>0:0:0:0:0:b</td>
<td>131.159.14.1.32858</td>
<td>&gt; 131.159.14.1.32858</td>
<td>.</td>
<td>ack 32</td>
<td></td>
</tr>
</tbody>
</table>
The remainder of this section addresses a general problem of merging traces. We want to generate traces that can be replayed accurately. Therefore we need to take the characteristics of the replay device into account. For example, the replay device enforces a minimal delay between the start times of two successive packet transmissions. For a 10BaseT half-duplex link this delay equals the transmission delay of the first packet plus 96 bit times (due to Ethernet’s CSMA/CD protocol). Hence, we must keep minimal distances between adjacent packet timestamps because these timestamps indicate when packets should be on the wire.

We talk about a **timestamp collision** if an inter-packet gap falls below this minimal boundary. The existence of a sequence of timestamp collisions within a generated trace can highly distort the inter-packet times seen on the link during replaying, which affects the inter-packet times of the interdependent traffic we like to preserve. Unfortunately, with our technique timestamp collisions are quite probable and become even more probable the larger the number of traces to merge is.

What can be done to avoid timestamp collisions? The minimization of collisions using “good” base traces and appropriate injection times seems to be a hard problem. Moreover, in most cases these are fixed and cannot be chosen freely. Rather a heuristic method is needed that eliminates collisions afterwards. Such a method can use any of the following operations:

- Move single packets, TCP connection or UDP flows (cp. Section 3.2.6). The moving operation is especially useful if injection times are somewhat flexible.
- Remove single packets, TCP connection or UDP flows (cp. Section 3.2.5). The removing operation is especially useful if it is not necessary to retain all additional traffic.
- Stretch/Compress TCP connection or UDP flows (cp. Section 3.2.4). These operations are especially useful if injection times should be kept, yet the inter-packet gaps of injected flows are already too small.

However, it is still difficult to decide when which operation yields the best results. Moreover, we will see that is difficult to identify suitable packets or flows for such an operation. The application of an operation can create undesirable artifacts that result from dependencies between the selected packet/flow and other traffic. Thus, further work is needed to develop a complete algorithm for eliminating timestamp collisions.
3.2.4 Stretching and Compressing

Section 3.1.2 and Section 3.2.3 motivate the stretching or compressing of whole traffic traces or single flows. The goal of these operations is the modification of the duration and/or the inter-packet gaps of a trace or flow.

A simple approach to implement this is to multiply the relative timestamps of all packets of a trace/flow by a constant scaling factor $s$. If we choose $s < 1$ (scale down) the trace/flow will be compressed. If we choose $s > 1$ (scale up) the trace/flow will be stretched.

Now let us discuss this simple method.

General Problems when Scaling Traces

Because the scaling down operation decreases inter-packet gaps, we should choose the scaling factor in such a way that timestamp collisions are avoided (cp. Section 3.2.3). However, this is difficult because the occurrence of collisions depends on the inter-packet gaps of the trace.

Kamath et al. [19] have shown that the scaling down operation is not suitable for generating realistic high bandwidth traces from low bandwidth traces: Inter-packet gaps can be affected by multiple factors such as transmission delays, propagation delays, queuing delays, protocol characteristics, processing delays at servers and user thinking times at clients. If we scale a trace, we multiply all inter-packet gaps with the same constant $s$. Thus, we decrease not only transmission delays, what is okay when one assumes faster links, but also other delays that contribute to a packet’s timestamp. This is problematic from a statistical point of view. “The presence of a faster link does not imply that the propagation delay would change or that a user would read a web page faster” [19]. Analog problems apply to the scaling up operation.

Scaling TCP/UDP Flows

The previous paragraph deals with general problems concerning the scaling of traces. Now we discuss which specific difficulties arise when TCP connection or UDP flows get scaled. Of course, such flows can be scaled separately or along with the trace file they are contained in. In both cases we must take interdependent flows into account. For example, in the case of TCP one may destroy the correspondence between a control message flow and the data flow when one scales only the control connection of the FTP session. This cannot happen if one scales a whole trace containing an FTP session.

However, even if one scales a whole trace there can be problems concerning the maximal time gaps between interdependent packets or flows which are stipulated by specific protocols and that must not be exceeded (i.e., transport-layer
or application-layer timeouts). The next paragraphs deal with this issue in the context of analyzing the effects of scaling on certain TCP characteristics.

First we consider the problems with regard to TCP’s acknowledgement generation strategy (consult [37] for an introduction to this strategy). ACKs may be delayed by a connection side’s TCP implementation. But according to RFC 1122 [6] an ACK should not be excessively delayed. In particular the delay must be less than 500 ms. If one scales up a TCP connection flow, the time interval between the occurrence (i.e., the timestamp) of a data packet and its corresponding acknowledgement may exceed this upper bound. Yet, such an artifact can only be found if we pretend that the generated trace has been recorded at the sender side of this ACK. Otherwise queuing and propagation delays could have contributed to the delay.

However, even if one does not know where the trace comes from the following may look odd: The reason why ACKs are delayed is that a receiver does not need to acknowledge every data packet separately. It can generate a cumulative ACK for two packets if another (in-order) packet arrives during the hold back time. When the time gap between the two data packets is stretched too strongly (\(\gg 500 \text{ ms}\)), then it is abnormal to see a cumulative ACK for them. This is especially improbable if we pretend that the scaled trace has been recorded at the sender side.

Next let us consider what happens concerning TCP’s ACK generation strategy when one scales a connection down. Suppose we generate a compressed connection that contains two separately acknowledged data packets whose time gap falls below the 500 ms boundary. Usually this cannot be treated as artifact: The gap seen at the receiver side of the data packets may have been larger due to queuing delays on the remaining path. Moreover, the generation of delayed cumulative ACKs is not mandatory. Only if one knows that a connection side supports delayed cumulative ACKs this case may look irregular.

Finally, we consider TCP’s timeout and retransmission strategy (see [37]). TCP sets a timeout for each data packet it transmits to detect packet loss. Because the network load changes during the connection, the time interval between the transmission of a packet and the reception of its acknowledgement varies as well. This time interval is called \textit{round-trip time (RTT)}. A TCP connection side should always adjust the timeout value to the current degree of congestion within the network. Thus, TCP frequently measures the RTT and takes this sample to compute the retransmission timeout value (\textit{RTO}).

What we are interested in is whether the scaling of a connection modifies the RTTs in a way that we expect to see retransmissions at different points if this would be a real connection. But to decide whether a packet would be subject to a timeout and needs to be retransmitted, one must know which RTO the sender uses. There are several pieces of information necessary to reconstruct the RTOs of a connection side:
CHAPTER 3. BASIC TRACE MANIPULATING OPERATIONS

Initial RTO:
\[ RTO_0 = 3 \text{ seconds} \]

RTO calculation when the first RTT measurement \( R_1 \) is made:
\[
\begin{align*}
SRTT_1 &= R_1, \\
RTTVAR_1 &= \frac{SRTT_1}{2}, \\
RTO_1 &= SRTT_1 + \max(G, 4 \cdot RTTVAR_1)
\end{align*}
\]

RTO calculation for subsequent RTT measurements \( R_i \) (\( i > 1 \)):
\[
\begin{align*}
RTTVAR_i &= (1 - \beta) \cdot RTTVAR_{i-1} + \beta \cdot |SRTT_{i-1} - R_i|, \\
SRTT_i &= (1 - \alpha) \cdot SRTT_{i-1} + \alpha \cdot R_i, \\
RTO_i &= SRTT_i + \max(G, 4 \cdot RTTVAR_i)
\end{align*}
\]

Figure 3.4: RTO Calculation According to RFC 2988

1. Details of the RTO calculation algorithm that is used.

2. The measured RTTs.

RFC 2988 [31] deals with the computation of TCP’s retransmission timer. It specifies the equations in Figure 3.4. \( SRTT_i \) is the smoothed round-trip time which is an weighted average of the \( i \) measured RTT values. \( RTTVAR_i \) is the smoothed mean deviation of the \( i \) RTT samples from its average \( SRTT_i \) (weighted average of the round-trip time variations). The weights \( \alpha \) and \( \beta \) should be set to \( \frac{1}{8} \) and \( \frac{1}{4} \) respectively, but these values are not mandatory. \( G \) is the clock granularity (consult [37] for details) used for RTT measurements and the retransmission timer. No requirements are specified for \( G \), especially no fixed value. Until a first RTT measurement can be made, the initial RTO \( (RTO_0) \) should be set to 3 seconds. This value is also optional. Finally, RFC 2988 suggests to introduce a minimal RTO \( (RTO_{\text{min}}) \) of 1 second. Additionally, a maximum value may be placed on the RTO \( (RTO_{\text{max}}) \) provided it is at least 60 seconds. In “TCP/IP Illustrated” [37] we can find a slightly different version of the described RTO calculation algorithm (Figure 3.5).

Therefore, in summary one can say that RFC 2988 does not stipulate a mandatory algorithm. In addition, there exist versions of the basic algorithm that (slightly) differ from the one in RFC 2988. Moreover, it is also presumable that some TCP implementations do not comply with this basic algorithm at all.

Yet, these details matter. Table 3.1 demonstrates that it highly depends on the assumed calculation details whether a scaled connection (or even a genuine connection) looks faked. For that purpose, the connections shown in Figure 3.6 have been scaled. After the scaling, we have tried to reconstruct the RTT samples \( (R_i) \) and the RTO values \( (RTO_i) \) at the client side of both connections. For
CHAPTER 3. BASIC TRACE MANIPULATING OPERATIONS

Initial RTO:
\[ RTO_0 = 6 \text{ seconds} \]

RTO calculation when the first RTT measurement is \( R_1 \) made:
\[
\begin{align*}
SRTT_1 &= R_1 + \frac{1}{4}, \\
RTTVar_1 &= \frac{SRTT_1}{2}, \\
RTO_1 &= SRTT_1 + 4 \cdot RTTVar_1
\end{align*}
\]

RTO calculation for subsequent RTT measurements \( R_i \) \((i > 1)\):
\[
\begin{align*}
RTTVar_i &= (1 - \beta) \cdot RTTVar_{i-1} + \beta \cdot |SRTT_{i-1} - R_i|, \\
SRTT_i &= (1 - \alpha) \cdot SRTT_{i-1} + \alpha \cdot R_i, \\
RTO_i &= SRTT_i + 4 \cdot RTTVar_i
\end{align*}
\]

Figure 3.5: RTO Calculation According to “TCP/IP Illustrated”

each client side three potential RTO calculation configurations (i.e., \( RFC_1 \), \( RFC_2 \) and Stevens described in Table 3.1) are considered that differ slightly from each other. Points within the table where a measured RTT exceeds its respective RTO (i.e., \( R_i \geq RTO_i \)) are marked with a special symbol (\( \ddagger \)). Since no connection contains a retransmission these points have to be considered artifacts. Yet, it all depends on the assumed RTO calculation details whether such irregularities occur in a (scaled) connection. Thus, the maximal \( s \) that can be used to scale a certain connection without creating artifacts depends on these details.

To calculate the RTOs in our example we have simply assumed that the client has measured the RTT samples specified in Table 3.1 \( (R_i, i = 1, \ldots, 4) \). But strictly speaking it is not that easy to determine which values are measured. Even if one knows that the considered trace has been recorded directly at this side one needs further details: Traditionally, TCP implementations take one RTT measurement within a window, but is is possible to take multiple measurements in parallel [31]. Accordingly it is difficult to infer from a trace which RTTs are involved in which RTO calculation. Supposed we know the packets from which RTT samples are taken. Then we still have to know the precision of the timer to estimate the values measured by the connection side. If a side uses a “heartbeat” timer (see [38] for details) an estimation may differ by \( G \) seconds from the actual value.

It is improbable that an NIDS is aware of such details of the RTO calculation mechanisms applied by the observed clients and servers. Moreover, it cannot directly measure the RTTs as seen at the endpoints because it is placed somewhere on the path between client and server. An NIDS may only apply an “RTT estimation in the middle” techniques, like the one described in [17]. Accordingly, we
believe that scaled TCP connection flows only look conspicuous in exceptional cases (concerning TCP characteristics). Probably a gap between a SYN and the corresponding SYNACK that exceeds the initial (fixed) RTO \((RTO_0)\) by far\(^7\), needs to be identified as artifact.

It is sometimes possible to circumvent these difficulties and avoid scaling, if only the duration of a TCP connection needs to be increased. For example, a persistent connection of an HTTP session can be stretched by just increasing user thinking times, i.e., the delays between HTTP requests.

However, through further investigations we hope to find boundaries for \(s\) so that scaling does not create an artificial packet timeout. Further tests have to show how probable the artifacts discussed in this section are, and more importantly how critical they are for the evaluation of NIDS.

\(^7\) 6 seconds
### Table 3.1: Scaling and RTO Calculation

Three different configurations have been used to compute the RTOs:

- **RFC1**: Based on the algorithm in RFC 2988 with $G = 0.001 \text{sec}$, $RTO_{min} = 0 \text{sec}$.
- **RFC2**: Based on the algorithm in RFC 2988 with $G = 0.01 \text{sec}$, $RTO_{min} = 1 \text{sec}$.
- **Stevens**: Based on the algorithm in "TCP/IP Illustrated" with $G = 0.5 \text{sec}$, $RTO_{min} = 0 \text{sec}$.

<table>
<thead>
<tr>
<th>s</th>
<th>$RTT_1$</th>
<th>$R_1RTO_0^{RFC1}$</th>
<th>$R_1RTO_0^{RFC2}$</th>
<th>$R_1RTO_0^{Stevens}$</th>
<th>$RTT_2$</th>
<th>$R_2RTO_1^{RFC1}$</th>
<th>$R_2RTO_1^{RFC2}$</th>
<th>$R_2RTO_1^{Stevens}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.018711</td>
<td>.019/3</td>
<td>.02/3</td>
<td>0/6</td>
<td>.018937</td>
<td>.019/0.057</td>
<td>.02/1</td>
<td>0/1.5</td>
</tr>
<tr>
<td>10</td>
<td>.18711</td>
<td>.187/3</td>
<td>.19/3</td>
<td>0/6</td>
<td>.189370</td>
<td>.189/0.561</td>
<td>.19/1</td>
<td>0/1.5</td>
</tr>
<tr>
<td>15</td>
<td>.280665</td>
<td>.281/3</td>
<td>.28/3</td>
<td>0.5/6</td>
<td>.284055</td>
<td>.284/0.843</td>
<td>.28/1</td>
<td>0.5/3</td>
</tr>
<tr>
<td>30</td>
<td>.56133</td>
<td>.561/3</td>
<td>.56/3</td>
<td>0.5/6</td>
<td>.56811</td>
<td>.568/1.683</td>
<td>.57/1.68</td>
<td>0.5/3</td>
</tr>
<tr>
<td>40</td>
<td>.74844</td>
<td>.748/3</td>
<td>.75/3</td>
<td>0.5/6</td>
<td>.75748</td>
<td>.757/2.244</td>
<td>.76/2.25</td>
<td>1/3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>$RTT_3$</th>
<th>$R_3RTO_2^{RFC1}$</th>
<th>$R_3RTO_2^{RFC2}$</th>
<th>$R_3RTO_2^{Stevens}$</th>
<th>$RTT_4$</th>
<th>$R_4RTO_3^{RFC1}$</th>
<th>$R_4RTO_3^{RFC2}$</th>
<th>$R_4RTO_3^{Stevens}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.072301</td>
<td>.072/0.48</td>
<td>.07/1</td>
<td>0/1.5</td>
<td>.019384</td>
<td>-</td>
<td>.02/1</td>
<td>0/2</td>
</tr>
<tr>
<td>10</td>
<td>.72301</td>
<td>.723/0.47</td>
<td>.72/1</td>
<td>0/1.5</td>
<td>.19384</td>
<td>-</td>
<td>.19/1</td>
<td>0/1.5</td>
</tr>
<tr>
<td>15</td>
<td>1.084515</td>
<td>1.085/0.706</td>
<td>1.08/1</td>
<td>1/3</td>
<td>.29076</td>
<td>-</td>
<td>-</td>
<td>0.5/2.5</td>
</tr>
<tr>
<td>30</td>
<td>2.16903</td>
<td>2.169/1.41</td>
<td>2.17/1.41</td>
<td>2/3</td>
<td>.58152</td>
<td>-</td>
<td>-</td>
<td>0.5/3.5</td>
</tr>
<tr>
<td>40</td>
<td>2.89204</td>
<td>2.89/1.88</td>
<td>2.89/1.89</td>
<td>3/2.5</td>
<td>.77536</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>$RTT_5$</th>
<th>$R_5RTO_4^{RFC1}$</th>
<th>$R_5RTO_4^{RFC2}$</th>
<th>$R_5RTO_4^{Stevens}$</th>
<th>$RTT_6$</th>
<th>$R_6RTO_5^{RFC1}$</th>
<th>$R_6RTO_5^{RFC2}$</th>
<th>$R_6RTO_5^{Stevens}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.041284</td>
<td>.041/3</td>
<td>.04/3</td>
<td>0/6</td>
<td>.063921</td>
<td>.064/0.123</td>
<td>.06/1</td>
<td>0/1.5</td>
</tr>
<tr>
<td>50</td>
<td>2.0642</td>
<td>2.064/3</td>
<td>2.06/3</td>
<td>2/6</td>
<td>3.19605</td>
<td>3.196/6.192</td>
<td>3.2/6.18</td>
<td>3/7.5</td>
</tr>
<tr>
<td>130</td>
<td>5.36692</td>
<td>5.367/3</td>
<td>5.37/3</td>
<td>5.5/6.0</td>
<td>8.30973</td>
<td>-</td>
<td>-</td>
<td>8.5/18.0</td>
</tr>
<tr>
<td>160</td>
<td>6.60544</td>
<td>6.605/3</td>
<td>6.6/3</td>
<td>6.5/6</td>
<td>10.22736</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>$RTT_7$</th>
<th>$R_7RTO_6^{RFC1}$</th>
<th>$R_7RTO_6^{RFC2}$</th>
<th>$R_7RTO_6^{Stevens}$</th>
<th>$RTT_8$</th>
<th>$R_8RTO_7^{RFC1}$</th>
<th>$R_8RTO_7^{RFC2}$</th>
<th>$R_8RTO_7^{Stevens}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.124114</td>
<td>.124/128</td>
<td>.12/1</td>
<td>0/1.5</td>
<td>.083432</td>
<td>.083/197</td>
<td>.08/1</td>
<td>0/2</td>
</tr>
<tr>
<td>70</td>
<td>8.68798</td>
<td>8.688/9.007</td>
<td>8.69/9</td>
<td>8.5/10</td>
<td>5.84024</td>
<td>5.84/13.827</td>
<td>5.84/13.83</td>
<td>6/14</td>
</tr>
<tr>
<td>130</td>
<td>16.13482</td>
<td>-</td>
<td>16/18</td>
<td>10.84616</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11/26</td>
</tr>
<tr>
<td>160</td>
<td>19.85824</td>
<td>-</td>
<td>-</td>
<td>13.34912</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
3.2.5 Removing

Section 3.2.3 suggests the application of the removal operation for single packets, TCP connections, or UDP flows to eliminate timestamp collisions. Generally, the removal of single packets from a trace file is a critical operation: The local context contains almost always enough information to figure out when a packet is missing. This must normally be treated as an artifact.

Let us consider two examples. Suppose we remove a packet from a recorded TCP connection. The resulting gap is mostly detectable by tracking the sequence numbers of the TCP stream. If it is assumed that all packet losses appearing in the traffic trace are also seen by the respective endpoints then the gap within the connection capture must be considered unrealistic. That means, the respective endpoint should have reacted with a retransmission that would be contained in the trace.

Under certain conditions, a gap within an UDP flow might look like an artifact as well. Although UDP is not a reliable transport protocol and therefore one cannot expect a retransmission, the application layer protocol lying on top of UDP may miss a packet and therefore reveal an irregularity. For example, normally we do not see a DNS response without a preceding query.

There exist environments where traffic with such artifacts is considered real. The traffic seen at a mirror port of a switch or router is such an example. Since only a copy of the real traffic is observed at a mirror port, dropped packets on these networks are not recognized as losses by the endpoints. Since the real traffic still contains these packets, we cannot expect any reactions by the endpoints. Packet drops occur at the mirror port if the queue overruns, e.g., when the port’s output capacity is exceeded. If one assumes an environment of this kind it might be reasonable to remove single packets once the capacity of the bottleneck within the trace is reached (cp. Section 3.2.3).

The removal of all packets belonging to a TCP connection flow is a less critical operation. However, one must pay attention to the application layer protocol using the TCP connection. If this protocol uses multiple (simultaneous) TCP connections which are interdependent, the lack of some connection possibly looks unrealistic. A missing control connection within the scope of an FTP session is a concrete example. But generally it will not be so clearly recognizable if one, e.g., removes a nonpersistent connection from an HTTP session. To decide whether one can filter out a UDP flow, the protocol lying on top should be considered as well.

Further research is required to find filtering rules which minimize the probability of creating artifacts.

\footnote{There exist few cases where other protocol characteristics must be consulted.}
3.2.6 Moving

Among other things this operation can be used to correct timestamp collisions. It raises similar difficulties as the removing operation in the previous section. If we consider the operation’s application to a single packet, we can conclude the following. The order of interdependent packets should never be changed by moving single packets. That means, e.g., a TCP packet containing data must not occur within a trace after the appearance of the corresponding acknowledgement. If one wants to move a packet of an UDP flow one has to look at the application layer protocol to identify possible dependencies to other packets. For example, a DNS response should not appear before its query.

Even if we adhere to this rough rule of thumb, we may run into similar problems as those with TCP’s RTOs and other (fixed) time boundaries, as discussed in Section 3.2.4. But as we have also seen, exceedances of these timeouts can rarely be assessed as artifacts. Therefore we could use the heuristic that it is okay to move a packet within the time frame determined by its (interdependent) predecessor and successor. In order to accomplish an accurate replaying of packets, minimal inter-packet gaps must be retained as well.

Section 3.2.5 has shown that dependencies to other traffic flows must be analyzed. For the moving operation this is also required: The control connection of an FTP session cannot be moved arbitrarily. A FTP session whose stream of control message does not correspond to its data stream certainly looks strange. However, dependencies are not always easily recognizable.

We can conclude that more experience is required here as well to minimize the generation of artifacts.

3.3 Summary and Future Work

This chapter has advanced the idea of trace-aided evaluation. We discuss how certain aspects of an NIDS can be evaluated using traffic traces and suitable customizations. In order to realize these customizations a set of basic trace operations is proposed. Finally, we discuss these operations in more detail particularly with regard to whether they generate any trace artifacts which can be problematic for NIDS tests.

Several questions and problems remain open which require actual experiments besides further theoretical analysis. In addition, the trace-aided evaluation approach is easily extendable. For example, the resistance of an NIDS against (simple) evasion techniques can be tested using attack traces which have been manipulated by certain (more complex) operations. To allow tackling of the open problems and aid further development of this approach we have designed and implemented a flexible and expandable software system which is described in the next two chapters.
Chapter 4

An Advanced Pipes and Filters Architecture

4.1 Motivation and Goals

The previous chapter has identified a set of basic trace manipulating operations that are necessary to realize certain customizations. Trace manipulations can be seen as some sort of filter operations on a specific type of data, i.e., traffic traces. A particular customization can be realized by using a particular system of filters, i.e., a specific interconnection of certain filters. To provide a means for the required trace customizations, we need a software system that

- implements filter operations,
- builds and manages systems of filters for trace customizations,
- supplies the components of the filter system with input,
- outputs the resulting traces.

Before we can think about concrete solutions, these rough requirements must be refined and more elaborated goals must be stated:

- **Modularity**
  The software system should provide the structural facilities to encapsulate filter operations as small separate components that can be easily verified. Small and correct components increase the chance of reusability and allow the incremental development of complex filter systems.
• **Flexibility**
Users of the software system should be able to exchange filter components easily. To facilitate the creation of different filter systems with the same building blocks the coupling between filter components should be reduced to a minimum. Moreover, the design should support the development of filter components with multiple input and output channels.
A system that is flexible in this sense can be reused to solve related problems. This means in particular, if our filter system design is flexible enough then we can realize various trace customizations using a small set of filter components.

• **Ease Of Use**
There should be a convenient way for users to specify, configure and control a desired system of filters. Potential users are not supposed to be experienced programmers. Therefore, they should not be obliged to dig into countless lines of code to build a custom filter system.

• **Economic Resource Usage**
Both the environment and the filters should handle their memory and disk usage with care because large amounts of data should be processible. Especially we want to be able to merge large traffic traces.

• **Abstraction**
It would be highly desirable not to let the filter management impose any limitations on the kind of data processed by a filter system nor the way this processing is performed. In doing so our software system is prepared for various tasks (in the field of NIDS evaluation) besides mere trace manipulation.

• **Interactivity**
The software system should be extensible in a way that a user can interactively influence the behaviour of the controlled filter system during realtime data processing.
An enhancement of this kind could be, e.g., the facility to exchange, reconfigure or reorder filters at runtime. The fulfillment of this design goal is reasonable if we think of a scenario where we replay merged traffic in realtime to the network and want to modify certain characteristics of the traffic (e.g., the protocol mix) or take some other actions (e.g., launch attacks, fragment packets) at the push of a button.

• **Integratability**
It should be easily feasible to integrate the core of our software system into a future all-in-one NIDS evaluation platform as demanded in [3].
Before trying to reinvent the wheel, it is always worthwhile to check whether existing software already meets our requirements or can at least be used as a starting point. At the time of this work all related tools (cp. Section 2.3) miss our requirements and are, additionally, lacking essential filter operations (e.g., a more natural way to remap IPs). Thus, the use and extension of a third party tool has been prevented.

Intuitively, another approach could be to realize filter operations as UNIX filter programs and to build filter systems by using named pipes (aka fifos). However, it might be hard to implement and debug such a multi-threaded filter system. Moreover, to interconnect and setup filters manually (using `mkfifo`) is an inconvenient task and can be complicated if the filter system becomes a complex graph structure. Furthermore, this approach might not be well-suited when we consider the interactivity goals described above.

Therefore, we decided to develop a new software system from scratch. The following sections describe the design of the “Advanced Pipes and Filters Architecture” which is responsible for the management and control of filter systems. The concrete components of our filter systems for trace customization are described in the next chapter.

### 4.2 Design Overview

Due to the requirements our software system should meet, an implementation as a flat, procedural program must be abandoned a priori. Rather, our goals motivate to follow an object-oriented paradigm.

In order to realize the previously described goals and to assure a certain degree of quality we have developed this architecture using several methods and tools of software engineering. We have created an object-oriented design which is influenced by some architecture and design patterns. Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” [12], i.e., a design pattern describes the core of a solution to a common well-known problem. An architecture pattern describes a problem at a higher abstraction level and a respective solution that affects the whole structure of a software system whereas the impact of a design pattern is localized more strictly.

The developed architecture is mainly based on the Pipes and Filters architecture pattern described in [7] and the Pluggable Component architecture pattern presented in [46]. The Pipes and Filters pattern addresses some of our modularity and flexibility goals. The Pluggable Component pattern addresses the desired runtime properties and describes a flexible way of component configuration. Additionally, several design patterns like Facade and Iterator [12] have been employed.
They are described in conjunction with the architecture parts where they are applied. Of course, the patterns have been customized, elaborated and extended to build a working architecture. Figure 4.1 shows which areas of the resulting architecture are most affected by the use of a particular pattern.

Now we sketch the main parts of the object-oriented design by considering Figure 4.1. The architecture distinguishes between different types of data processing components, in the following simply called components:

- **DataSources** provide a filter system with input data and do not process any data from other components.

- **DataSinks** save the modified data and do not provide any data to other components.

- **Filters** realize some data manipulating operations. They process data from components, i.e., “predecessors”, and provide the results of these operations to other components, i.e., “successors”.

![Figure 4.1: Advanced Pipes and Filters Architecture Overview](image-url)
CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

Figure 4.2: Role of “Pipes and Filters” Components within an Example Filter System

- **Pipes** are data buffering components. They can be used as optional interconnections between DataSources, DataSinks, Filters and even other Pipes.

Figure 4.2 sketches which concrete roles these components could play within an example filter system for the purpose of trace customization.

The abstract base class **MergeNetworkComponent** defines a common interface all components have to implement (see Section 4.3). The framework also specifies two data exchange mechanisms and an inter-component communication mechanism. Section 4.4 describes this architecture part in more detail. Concrete components can be developed independently from the core system and are integrated by the **PluginRegistry** at runtime. This class is responsible for the management of available plugins. More details concerning the plugin mechanism can be found in Section 4.5. From the connection of several components a graph or network-like structure evolves. In the following we will call this structure **MergeNetwork** or simply network (instead of filter system). The **ComponentContainer** stores the MergeNetwork and offers efficient methods to browse through it. The most important part of the architecture constitutes the **MergeNetworkController**. It provides all necessary methods to create and control the MergeNetwork. Thus, it becomes an interface to the core functionality of the architecture. Section 4.6 describes the MergeNetwork and this controller in more detail. To provide users with a comfortable, though powerful way to make specifications about the desired MergeNetwork, a special language and the appropriate interpreter (**MNCLInterpreter**) are also part of the framework. It is discussed in detail in Section 4.7.
4.3 The Component Interface

We start the detailed description of our architecture with the interface that offers a uniform way of component control. As we can see in Figure 4.3, there exist several types of components that differ in the role they play in the MergeNetwork. The number of input interfaces (InputSides) and output interfaces (OutputSides) to other components is associated with this role. A DataSource supplies the MergeNetwork with data. Therefore, it can have an arbitrary number of OutputSides but no InputSide. Realization of data modifying operations within DataSources is not prohibited. A DataSink saves data that has been modified by the MergeNetwork. Therefore, it can have an arbitrary number of InputSides but no OutputSides. Realization of data modifying operations within DataSinks is not prohibited either. An IntermediateComponent can have both an arbitrary number of InputSides and an arbitrary number of OutputSides. It represents an abstraction for components which are no end points of the MergeNetwork like Pipes and Filters. Pipes are simple data buffers. Among other things they can be used as converters between components with incompatible data transfer mechanisms (see Section 4.4). Filters implement actual data modifying operations, whereby modifying can mean to remove, to alter or to enrich data.
## CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getID()</code></td>
<td>returns the unique ID of a component instance inside a MergeNetwork</td>
</tr>
<tr>
<td><code>activate()</code></td>
<td>activates the (data modifying) function of a component</td>
</tr>
<tr>
<td><code>deactivate()</code></td>
<td>deactivates the (data modifying) function of a component</td>
</tr>
<tr>
<td><code>configured()</code></td>
<td>returns whether a component has been configured correctly</td>
</tr>
<tr>
<td><code>setConfiguration()</code></td>
<td>configures a component</td>
</tr>
<tr>
<td><code>getConfiguration()</code></td>
<td>returns the current component configuration</td>
</tr>
<tr>
<td><code>getInputSide()</code></td>
<td>returns a certain InputSide interface</td>
</tr>
<tr>
<td><code>getOutputSide()</code></td>
<td>returns a certain OutputSide interface</td>
</tr>
<tr>
<td><code>getComponentSpec()</code></td>
<td>returns the component’s specification</td>
</tr>
<tr>
<td><code>ready()</code></td>
<td>returns whether a component is ready to compute()</td>
</tr>
<tr>
<td><code>compute()</code></td>
<td>triggers the component’s data processing</td>
</tr>
<tr>
<td><code>notifyMNCtrl()</code></td>
<td>is solely used by the component itself to send messages to the MergeNetworkController</td>
</tr>
</tbody>
</table>

Table 4.1: MergeNetworkComponent Methods

DataSource, DataSink, IntermediateComponent, Pipe and Filter are abstract base classes that all share the common interface given by MergeNetworkComponent. The methods defined by this interface are described in Table 4.1. With the exception of the last one, their tasks are quite obvious. A component can use `notifyMNCtrl()` to report certain events to the MergeNetworkController. Errors must be reported in this way to let the MergeNetworkController decide about adequate reactions. Moreover, central collection and logging of other status messages is enabled.\(^2\)

Within the architecture, reports are encapsulated as Notification objects of different types. A textual description of the event can also be attached. Currently the following Notification types exist:

- **CONFIG_ERROR**
  
  A component uses this type to report a fatal error during its configuration.

---

\(^{1}\) This property depends on the concrete component, but mostly this includes that all Input/OutputSides are connected and the component’s configuration is correct.

\(^{2}\) This ability is especially interesting for interactive systems.
A Notification of this type indicates a non-fatal problem during a component’s configuration.

- **COMPUTATION_ERROR**
  A component uses this type to report a fatal error during its computation process.

- **COMPUTATION_WARNING**
  A Notification of this type indicates a non-fatal problem during a component’s computation process.

- **EODD**
  EODD means “End Of Data Delivery”. Using this Notification type a component can signal the MergeNetworkController that it will not supply the MergeNetwork with any more data.

The Class MergeNetworkComponent is associated with two other classes, as we can see in Figure 4.3. A **ComponentConfiguration** object stores a component’s configuration as name/value pairs (of type string/string). We cannot use different types of configuration parameters here because we want to treat components uniformly. Moreover, the system does not even know which parameter types a particular configuration consists of.
Each component owns a **ComponentSpecification** which includes a unique typename for the component, the maincategory (DataSource, DataSink or IntermediateComponent) it belongs to and a textual description.

This section specifies a common shape for all components destined for our architecture. An abstraction of this form establishes the basis for a plugin mechanism which allows us to develop and integrate new filters without extending or recompiling the environment.

### 4.4 Inter-Component Communication

From the previous section we know how a single component can be controlled. However, to build a MergeNetwork we must also define how components can be interconnected and how they can exchange data. In the following we will cover these issues. The framework supports two types of data transfer mechanisms (see Figure 4.4). To implement them the architecture defines two special input interfaces, i.e., **InputSidePull** and **InputSidePush**, and their compatible counterparts, i.e., **OutputSidePull** and **OutputSidePush**. If a component uses an InputSidePull then input data must be explicitly requested from the other component’s OutputSidePull. Otherwise, if a component uses an InputSidePush data arrives without demand. In both cases the component can only receive new data after it has emptied the data buffer of its input interface (using `getBuffer()` in Table 4.3), else further input is blocked. The main reason why we have decided to support two transfer mechanism types is that we suppose that some sorts of data manipulating operations can be implemented more easily or efficiently using either the pull mechanism or the push mechanism or a combination of both to transfer input and output data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connect()</code></td>
<td>establishes a connection to another compatible ConnectionSide</td>
</tr>
<tr>
<td><code>disconnect()</code></td>
<td>closes an existing connection</td>
</tr>
<tr>
<td><code>connected()</code></td>
<td>returns whether the ConnectionSide is connected</td>
</tr>
<tr>
<td><code>getConnectionSideSpec()</code></td>
<td>returns the specification of a ConnectionSide</td>
</tr>
<tr>
<td><code>checkCompability()</code></td>
<td>static method that verifies whether two ConnectionSides are compatible</td>
</tr>
</tbody>
</table>

**Table 4.2: ConnectionSide Methods**

\(^3\)e.g., “PcapFileSource” for a DataSource that supplies the MergeNetwork with data from a trace file in pcap format.
CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input()</td>
<td>is invoked by an OutputSide's output() method to transfer data (of type XFerDatatype)</td>
</tr>
<tr>
<td>getBuffer()</td>
<td>returns the received input data (of type XFerDatatype)</td>
</tr>
<tr>
<td>sendCtrlMsg()</td>
<td>is invoked by a component to send a CtrlMessage</td>
</tr>
<tr>
<td>rcvCtrlMsg()</td>
<td>is invoked by an OutputSide's sendCtrlMsg() method to transfer a CtrlMessage</td>
</tr>
<tr>
<td>getCtrlMsgs()</td>
<td>returns the received CtrlMessages</td>
</tr>
</tbody>
</table>

(a) InputSide＜XFerDatatype＞ Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output()</td>
<td>is invoked by a component to send data (of type XFerDatatype)</td>
</tr>
<tr>
<td>sendCtrlMsg()</td>
<td>is invoked by a component to send a CtrlMessage</td>
</tr>
<tr>
<td>rcvCtrlMsg()</td>
<td>is invoked by an InputSide's sendCtrlMsg() method to transfer a CtrlMessage</td>
</tr>
<tr>
<td>getCtrlMsgs()</td>
<td>returns the received CtrlMessages</td>
</tr>
</tbody>
</table>

(b) OutputSide＜XFerDatatype＞ Methods

Table 4.3: Input/OutputSide Methods

Additionally, Figure 4.4 shows that the architecture not only distinguishes between transfer mechanisms but also between transfer data types. Thus, InputSides and OutputSides, the base classes for input respective output interfaces, are defined as template classes with the used data type as parameter (i.e., XFerDatatype). The reason for this is to avoid the introduction of a general base class for data and resultant downcasts. Thereby we obtain an indefinite number of InputSide and OutputSide types that can be used any component. On the other hand we know from the previous section that it is necessary to have a common interface for all components and thus it is not possible to define getInputSide() and getOutputSide() methods for each type.

To solve this problem, methods used to (dis-)connect sides have been separated from methods responsible for inter-component communication. The latter are encapsulated in a abstract base class named ConnectionSide. Additionally, this class specifies a method to assure that two ConnectionSides are compatible with regard to their transfer mechanisms and data types. In other words, an InputSide can only be connected to an OutputSide if both use the same transfer
mechanism and exchange data of the same type. Class ConnectionSideSpecification stores the information necessary for this compatibility check. Table 4.2 presents the methods defined by the ConnectionSide interface.

The methods of the classes InputSide and OutputSide are shown in Table 4.3. As we can see, these interfaces provide methods to exchange control messages. They are represented within the architecture by their own class called CtrlMessage. There exist different types of CtrlMessages whereby the most important are the following:

- **DATA_REQUEST**
  This type is used by a component within the scope of the pull mechanism to request data from a predecessor.

- **EODD**
  This type has the same semantics as the homonymous Notification type. Thus, a component can signal a successor that it will not transfer any more data.

In summary, this section shows how component instances can interact and exchange data within our framework.

### 4.5 The Plugin Mechanism

Components destined for the Advanced Pipes and Filters Architecture are realized as plugins. Thus, we gain the advantage that they can be developed separately by third parties and integrated into the system at runtime. As a consequence, the system does not know anything special about the plugins. To control an instance of a component, the common abstract base class MergeNetworkComponent exists.

But how can our architecture create a component instance without knowing the name of its constructor? This is a point where the chosen implementation language has influenced the architecture design: C++ does not provide a direct plugin mechanism. Instead one has to revert to the concept of loading shared libraries using UNIX system functions (dlopen(), dlclose()). With their help it
is possible to access C functions of a shared library just by knowing their function names. More information about how to load shared libraries can be found in [8]. The way our architecture utilizes this concept for its plugin mechanism is based on the articles [1, 16].

Now we have a look at the resultant mechanism. A plugin implements a concrete component. To manage instances of this component the framework expects the plugin to bring along an implementation of the PluginInterface (PIF) described in Table 4.4. Apart from that, the plugin must define the C function getPIF() to make its PluginInterface accessible. Now a user still needs a possibility to select which instances of all the available plugins should be created. One solution is to define a mapping between PluginInterfaces and well-known keys (e.g., the class names). The PluginRegistry class (Figure 4.5) offers this mapping service. Method registerPlugins() loads all available plugins inside a certain directory and stores their PIFs using the typenames (ComponentSpecification members) as keys. Access to PIFs for DataSources, DataSinks and IntermediateComponents, as required by the four MergeNetwork-Controller methods also shown in Figure 4.5, is granted by getSourcePIF(), getSinkPIF() and getIComponentPIF(). The final result is that one can use the well-known typenames as a part of the configuration language (Section 4.7) to specify which components participate in a MergeNetwork.

### 4.6 The MergeNetwork and its Controller

Component instances are stored in a special type of container. The ComponentContainer (Figure 4.6) implements a graph structure where nodes are used to encapsulate components as contents. Stored components are accessible via iterators.

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createInstance()</td>
<td>by stating an ID, a configuration and a messenger⁴, this method creates a new component instance on the heap and returns a pointer to it</td>
</tr>
<tr>
<td>destroyInstance()</td>
<td>by stating a pointer to a component, this method destroys the component object on the heap⁵</td>
</tr>
<tr>
<td>getComponentSpec()</td>
<td>returns the component’s specification without creating a component instance</td>
</tr>
</tbody>
</table>

Table 4.4: PluginInterface Methods

---

⁴The messenger is used by notifyMNCtrl() to deliver Notifications to the controller.
⁵We must not call delete from outside, because this operator may be overloaded.
These iterators can also be used to traverse the MergeNetwork or as a position statement inside the graph structure. More about the common iterator principle can be found in [12]. Our iterator type is especially inspired by the well-known C++ STL\(^6\) iterator implementation as described in [18]. Table 4.5 presents the most important methods of the ComponentContainer.

Figure 4.6 shows the MergeNetworkController that acts as a connector between the core system and the user interface, represented by the MNCLInterpreter. Therefore, it is some form of facade, described in [12]. With the aid of the PluginRegistry this class realizes common use cases of the MergeNetwork.

Operations that modify the MergeNetwork in some way are implemented as methods of the controller, whereas non-modifying component operations are directly available via a restricted version of the ComponentContainer iterators. This approach is reasonable to protect the MergeNetwork from modifications from elsewhere and to gain a central entity that is aware of the network’s integrity.

Table 4.6 lists all methods offered by the controller together with a brief description. At this point we take a look at the most prevalent operations that simplify the use of our core system. By stating a \texttt{typename} the MergeNetworkController can create new component instances using the PluginRegistry and store them inside the ComponentContainer. A component instance can be configured

---

\(^6\)STL is an abbreviation for the Standard Template Library.
CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newNode()</td>
<td>creates a new graph node and stores a component instance as contents (returns the node’s position)</td>
</tr>
<tr>
<td>setEdge()</td>
<td>connects two nodes</td>
</tr>
<tr>
<td>getSourcePosList()</td>
<td>returns the positions (iterators) of all sources</td>
</tr>
<tr>
<td>getSinkPosList()</td>
<td>returns the positions (iterators) of all sinks</td>
</tr>
</tbody>
</table>

Table 4.5: ComponentContainer Methods

by its creation or subsequently with the aid of its unique component ID or position. Moreover, we can connect\(^7\) two components if we know their IDs or positions and specify the desired InputSide/OutputSide pair. As soon as the successive assembling of the MergeNetwork has finished it can take up its actual work. For that purpose the controller provides its \texttt{compute()} method. This method triggers the computation of the components in a suitable order until all components have sent a Notification of type \texttt{EODD} (see also Section 4.3).

The efficiency of data processing is obviously influenced by the inter-component transfer mechanisms along the paths of a MergeNetwork and the chosen trigger sequence. We have developed the following trigger strategy:

- The triggering process is divided into \textbf{computation rounds}.
- During a computation round all components will be invoked exactly once.
- Components that have sent an \texttt{EODD} Notification will not be triggered anymore.
- Components on “push paths” should always be triggered in a top-down manner.
- Components on “pull paths” should be triggered in an alternating order, i.e., first in a bottom-up and then in a top-down manner.

The sample computation in Section 4.8 applies this strategy. The sole purpose of the last two issues of this strategy is to decrease the number of computation rounds. These optimizations are not implemented yet.

Now that we have finished the survey of the core system, we can study the user interface in the next section.

\(^7\)Of course, only if they are compatible.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newSource()</td>
<td>by stating a typename, this method creates a new instance of the resp. DataSource</td>
</tr>
<tr>
<td>newSink()</td>
<td>by stating a typename, this method creates a new instance of the resp. DataSink</td>
</tr>
<tr>
<td>newIComponent()</td>
<td>by stating a typename, this method creates a new instance of the resp. IntermediateComponent</td>
</tr>
<tr>
<td>configureComponent()</td>
<td>by stating an ID (or position), this method configures the resp. component</td>
</tr>
<tr>
<td>activateComponent()</td>
<td>by stating an ID (or position), this method activates the resp. component</td>
</tr>
<tr>
<td>deactivateComponent()</td>
<td>by stating an ID (or position), this method deactivates the resp. component</td>
</tr>
<tr>
<td>connectComponents()</td>
<td>by stating two IDs (or positions) and I/O interfaces, this method connects the resp. components</td>
</tr>
<tr>
<td>getComponentSpec()</td>
<td>by stating its typename, this method returns a component’s specification</td>
</tr>
<tr>
<td>getMNPosition()</td>
<td>by stating its typename, this method returns a component’s position within the MergeNetwork</td>
</tr>
<tr>
<td>handleNotification()</td>
<td>handles Notifications sent by components</td>
</tr>
<tr>
<td>ready()</td>
<td>returns whether the MergeNetwork is ready to compute</td>
</tr>
<tr>
<td>compute()</td>
<td>starts the computation of the MergeNetwork</td>
</tr>
</tbody>
</table>

Table 4.6: MergeNetworkController Methods

### 4.7 The Configuration Language and its Interpreter

From Section 4.6 we know which class of the architecture is responsible for the creation and management of a MergeNetwork. Now the user needs a convenient way to interact with the MergeNetworkController. For this reason the framework defines a special language to describe the required MergeNetwork and a respective interpreter. Method interpretMNC() of class MNCLInterpreter can interpret configuration files with the following structure:

```plaintext
Components {  
   # component declarations
}

Connections {  
   # connection declarations
}
```
Inside the **Components** section we declare and configure all components a MergeNetwork should consist of. The order of declarations does not matter. A component declaration looks like this:

```plaintext
<component’s maincategory> <componentID> {

type <typename>; 

Config {
  # Configuration parameters
  <name> = <value>; 
  <name> = <value>; 
  # ...
}
}
```

The special parts (<...>) of this declaration have the following semantics:

- **<component’s maincategory>** is the maincategory the component belongs to, i.e., DataSource (abbr. Source), DataSink (abbr. Sink) or IntermediateComponent (abbr. IComponent).
- **<componentID>** assigns a unique ID to the component within the MergeNetwork.
- **<typename>** is the component’s typename (cp. Section 4.3).
- **<name> = <value>** assigns a value to a component’s configuration parameter.

The **Connections** section specifies how declared components should be interconnected. To connect OutputSide number 0 and InputSide number 1 of two components with component IDs CompA respective CompB, one writes the following line:

```plaintext
CompA[0] -> CompB[1];
```

The order of connection declarations is irrelevant.

Now that the description of the individual architecture parts is completed, we consider their dynamic behaviour within the scope of an example in the next section.
The previous sections have described the constituent architecture parts and their responsibilities. This section depicts the interactions of these parts and the inter-component communication, by considering the construction and computation of an example MergeNetwork. Let us assume that we want to construct the MergeNetwork illustrated in Figure 4.7 which consists of the following components:

- A source \( S_01 \) of type XSource that supplies the MergeNetwork with data of type X from a file. This component has one OutputSidePush interface.
- A source \( S_02 \) of type YSource that supplies the MergeNetwork with data of type Y from a file. This component has one OutputSidePull interface.
- A pipe \( P_1 \) of type PushPullPipe that acts as a transfer-mechanism converter between source \( S_01 \) and filter \( M_1 \). This component has one InputSidePush and one OutputSidePull interface.
A filter $F_1$ of type Y2XConverter that converts data of type $Y$ to data of type $X$. This component has one InputSidePull and one OutputSidePull interface.

A data merging filter $M_1$ of type XMerger. This component has two InputSidePull interfaces and one OutputSidePush interface. $M_1$ waits until it has received one unit of data from each InputSide. Then it decides randomly which data unit will be forwarded and which will be buffered. If one InputSide runs out of data, $M_1$ acts like a simple pipe and forwards all data from the other InputSide.

A filter $F_2$ of type X2YConverter that converts data of type $X$ to data of type $Y$. This component has one InputSidePush and one OutputSidePush interface.

A sink $Si_1$ of type YSink that stores data of type $Y$ in a file. This component has one InputSidePush interface.
CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

Figure 4.8 presents an appropriate configuration for our target network. To realize the specified network, the MNCLInterpreter uses the MergeNetworkController which in turn collaborates with the rest of the architecture.

The sequence diagram in Figure 4.9 describes how the respective parts interact to create instance $S_{o1}$ of the XSource component. Figure 4.10 shows the interaction steps required to connect components $S_{o1}$ and $P_1$. The creation and interconnection of other component instances follows these examples.

After the MergeNetwork has been built and configured, its controller starts the data processing, i.e., it triggers the components’ `compute()` methods in a certain order. We discuss now the data flow during the computation.

Suppose the given input files `input.x` and `input.y` contain only one packet of data respectively. Additionally, we assume that the MergeNetworkController calculates the trigger sequences that are given in the following diagrams (using some implementation of the strategy in Section 4.6). In this way the resulting data processing finishes after four computation rounds of the MergeNetwork. Remember that a computation round is the period of time within which all components are invoked exactly once.
Figure 4.11: MergeNetwork Computation Round 1.
Trigger Sequence: M1, F1, So2, So1, P1, F2, Si1

The first computation round (Figure 4.11) starts with the invocation of M1’s `compute()` method. During this method call, M1 sends a CtrlMessage of type `DATA_REQUEST` (abbr. DR) to pipe P1 (1) and filter F1 (2). F1’s data buffer is also empty at this time, therefore, it sends a data request to source So2 (3). So2 reacts by transmitting a piece of data (4). Then So1’s data processing is triggered by the controller. Thereupon, the source sends a data packet to its successor P1 (5). Because P1 had received a DR CtrlMessage from M1 (1) it can forward the data during the next computation step (6). After the two idle components F2 and Si1 have been invoked, the first round is completed.

Figure 4.12: MergeNetwork Computation Round 2.
Trigger Sequence: So2, F1, M1, So1, P1, F2, Si1

Figure 4.12 illustrates the steps of the second round. A slightly modified trigger sequence is applied here. As first step, the MergeNetworkController calls So2’s `compute()` method but there is nothing to do for this component. After that, F1 converts the data received in the course of the first round and transfers it to M1 (1). In the next step M1 decides to forward the piece of data from P1 (2) and requests another one (3). When the MergeNetworkController triggers So1, the source cannot supply the network with any more data. So1 informs the controller about this fact by sending a Notification of type `EODD`. For the same reason
it sends an EODD CtrlMessage to P1 (4). P1’s single input source has run out of data, so it cannot fulfill M1’s data request. Therefore, it does the same as So1 one step before (5). Finally, F2 converts the data from M1 and forwards it to sink Si1 (6) which stores it.

![Figure 4.13: MergeNetwork Computation Round 3.
Trigger Sequence: M1, F1, So2, F2, Si1](image)

In round three (Figure 4.13), the controller uses the trigger sequence from round one but it has excluded the two components that have finished their jobs (So1, P1). At the beginning of this round M1 recognizes that it will not receive any more input from P1. From now on M1 behaves like a pipe. It transfers the buffered data to F2 (1) and sends a request for more data to F1 (2). F1 forwards this request to So2 (3) but So2 observes that it cannot fulfill any more requests. Thus, it takes the same actions (4) as So1 during the previous round. After that, F2 takes its turn. This component has received data from M1 which it sends to Si1 (5). In the last step of the third round, sink Si1 saves the data packet.

![Figure 4.14: MergeNetwork Computation Round 4.
Trigger Sequence: F1, M1, F2, Si1](image)

In this example the final computation round (Figure 4.14) only serves the purpose of propagating the “End Of Data Delivery” through the remaining MergeNetwork (1-3). After step three all components have sent an EODD Notification to the MergeNetworkController. Thus, the whole computation is done.
4.9 MergeNetwork Modifications at Computation Time

The preceding sections describe the design of architecture parts actually implemented. In this section we outline architecture extensions to support interactive data processing. To realize these basic ideas, certainly one has to refine them. Generally, we suppose that the controller’s trigger algorithm runs as separate process or thread. Based on this assumption, we discuss how the following short list of basic operations can be performed during the computation of a MergeNetwork:

1. Stop and resume the computation of a network.
2. Activate or deactivate a component.
3. Reconfigure a component.
4. Exchange a component.
5. Abort the computation of a network without losing half-processed data.

A simple approach to realize the first two fundamental operations is to extend the MergeNetworkController by the methods `stopComputation()` and `resumeComputation()`. `stopComputation()` forces the controller to stop triggering the components’ `compute()` methods as soon as possible and keep a pointer to the component whose turn is next. `resumeComputation()` tells the controller to continue the execution of its trigger algorithm at the breakpoint.

Next, we want to discuss list items 2-4. We start by answering the question in which state the network must be to execute one of these component operations without affecting the ability of the MergeNetwork to process data. Basically, it is not a good idea to perform an operation that modifies the state of a component while it is computing. During a computation break most components should be in a state that allows their de/activation or reconfiguration. Nevertheless some components may have special configuration parameters which cannot be changed at any arbitrary break. In this case, we expect the component to delay the substitution of these parameter values and perform the task on its own as soon as possible.

To safely exchange a component during a break, further conditions must be satisfied. A component cannot be removed if it still holds buffered data, because the data would be lost. A similar problem raises if a component awaits requested data from a predecessor or has to fulfill an outstanding request from a successor.
We violate\(^8\) the current inter-component communication protocol if we replace the busy component by a new one. An InputSidePull should not receive unrequested data and an OutputSidePull can only send data if it has been requested before. This kind of difficulties can be eliminated by extending the MergeNetworkComponent interface with the method `suspend()`. By calling this method a component is told to reach an idle state as soon as possible.

To prevent a component from receiving unwanted data via its InputSidePush interface or an unwanted request via its OutputSidePull interface, one could provide both interface types with an additional method `lock()` (and its counterpart `unlock()`). Because `suspend()` is an operation that modifies the state of a component and we want to be compliant with the approach in Section 4.6, this method should only be accessible using the controller.

Thus, the controller is provided with a homonymous method that performs the following steps: By stating an component ID or position the controller stops the computation of the network (`stopComputation()`) and calls the component’s `suspend()` method. Then it checks whether the controller has received a Notification of type `SUSPENDED`\(^9\). If this is not the case, it signals the trigger algorithm to proceed `resumeComputation()` and waits until the right Notification arrives. As a last step the method interrupts the whole computation again and returns. At this point the old component can be safely removed and replaced by the new one. The pure exchange of a component does not affect the transfer mechanisms along the respective paths and so the old trigger sequences can be reused. The controller’s `suspend()` method is also convenient if we want to abort the data processing without loosing half-processed data existing in the buffers of the network. For that purpose we suspend one component after another in a breadth-first manner starting at the sources. Applying this order, we avoid deadlocks.

Now we know how network modifications at computation time could be implemented within our architecture. Complex operations which involve multiple steps (like the exchange of a component) can be encapsulated as separate objects. But we are still missing a facility for users to initiate them. A text based or graphical interface might be the right user device for this task. Generally, such interfaces can be integrated into the architecture by using the Model-View-Controller pattern described in [7]. This pattern divides an interactive application into three parts. The model contains the data that needs to be managed and represents the core functionality of the application. Within our architecture it is embodied by the MergeNetworkController. Views like, e.g., a window or console that displays Notifications, provide users with information about the model at runtime. A con-

\(^8\)However, these problems could also be fixed by extending the communication protocol.

\(^9\)We have to introduce this Notification type.
CHAPTER 4. AN ADVANCED PIPES AND FILTERS ARCHITECTURE

Figure 4.15: Integration of a Scheduler/Computation-Event-Manager

troller handles user input and thereby it changes the model. The conjunction of a
view and a controller is what Buschmann refers to as a user interface. To ensure
the consistency between a user interface and the model, a “change-propagation
mechanism” must be defined. In our case, this can be the forwarding of Notifica-
tions to user interfaces. Within this chapter a controller like the MNCLInterpreter
is already called a user interface. However, this interface does not allow to per-
form interactive operations at computation time.

The remainder of this section describes how user interfaces like a scheduler or
a computation event handler can be integrated into our architecture. A scheduler
executes operations at predefined times during the computation process. A com-
putation event handler responds to events that occur during data processing by
modifying the network. An event could be the reception of a special Notification
type. The development of sensor components that can be placed into the network
to signal custom events is also conceivable. A simple use case of the computa-
tion event handler would be the exchange of a source when it runs out of data.

Figure 4.15 shows how this kind of interfaces fits into our design. The schedule
of operations that modify a MergeNetwork, respectively the rules destined for the
event handler must be defined in advance. This can be done by extending the
configuration language and its interpreter (see Section 4.7).

This section has sketched ideas to support interactive operations that modify a
MergeNetwork while it is processing data. Thereby it also shows the flexibility of
the architecture design.

---

10 Therefore, it does not need to be informed about changes at computation time.
11 i.e., before the computation starts.
4.10 Summary and Future Work

This chapter has presented the parts of an architecture that serves as a flexible and extensible framework for multipurpose filter systems. The framework allows the development of dedicated components by third parties and offers the possibility to integrate them into the system at runtime. Another feature is the ability to build filter systems for various tasks just by modifying the structure of an existing MergeNetwork. The final section shows how the architecture could be extended to support interactive data processing. Besides spending time on extending the architecture there is still room for improvements within the existing parts. The implemented architecture nevertheless constitutes a well-suited fundament to develop filter systems tailored for the customization of traffic traces. The source code\footnote{It should be pointed out that not all of the design refactoring measures that have been performed while writing this chapter have also devolved to the source code. Thus, there currently exist some minor deviations between the architecture described here and the actual implementation that will, however, be corrected in the near future.} of this architecture can be found at [2].

The next chapter describes the second part of our software system for packet trace customization, i.e., a set of plugins for the presented architecture that perform suitable trace manipulations.
Chapter 5

A Plugin Package for Trace Manipulations

In Chapter 3, we identify several basic trace manipulating operations to customize traces for evaluating intrusion detection systems. Chapter 4 describes the design of a flexible and extensible architecture that offers a suitable basis for the implementation of these filter operations. Finally, this Chapter presents the TTM\(^1\) plugin package (Figure 5.1) we have developed in order to realize such trace manipulations and to test the correctness of the architecture’s implementation. Due to time restrictions only a prototypical plugin development was feasible and not all operations described in Section 3.2 are realized. Nevertheless, the existent plugins constitute a valuable starting point to experiment with and eventually realize complex trace customizations.

\(^1\)TTM is an abbreviation for Traffic Trace Manipulation.

![Figure 5.1: The TTM Plugin Package](image-url)
CHAPTER 5. A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

5.1 Overview

All plugins in the package (Figure 5.1) operate on the PCapPacket data type which is described in Section 5.2. PCapPacket objects encapsulate network packet data given by traffic traces in pcap format (see Section 2.3) and additional information about each packet like, e.g., a summary about the TCP, UDP or ICMP flow the packet belongs to. Within this chapter we use the same “definition” for the term flow as given in Section 3.2.1.

An instance of the PCapFileSource plugin reads pcap files and user-supplied flow summary files in order to provide the MergeNetwork with PCapPackets generated from the extracted data. Details about this component can be found in Section 5.3. Trace files in pcap format can be created using tools like tcpdump [39] which are based on the Packet Capture Library (libpcap). Appropriate flow summary files can be created using tools like tcpreduce [42].

The counterpart of the PCapFileSource plugin is the PCapFileSink plugin presented in Section 5.4. This component extracts the pcap packet data from incoming PCapPacket objects and stores it in a file in pcap format.

An instance of the IPSpoof plugin is able to forge the IPs and ports of UDP, TCP and ICMP flows according to a set of user-supplied spoofing rules. This plugin is described in Section 5.7.

Traces can be merged using the PCapPacketSorter plugin. An instance of this component sorts PCapPackets received from an arbitrary number of InputSidePull interfaces chronologically. A PCapPacketSorter instance uses an InputSide type (pull) that is incompatible to the OutputSide types of PCapFileSource, IPSpoof and other PCapPacketSorter instances. In order to connect one of its InputSides to the OutputSide of one of these components, a PushPullPipe must be interposed which acts as transfer mechanism converter. This incompatibility has been generated deliberately to verify further parts of our architecture. Details about the PCapPacketSorter and PushPullPipes can be found in Section 5.5 and Section 5.6 respectively.

Finally, an exemplary application of the plugin package is presented in Section 5.8. The plugins are used to interweave some attack traces with a background traffic trace (cp. Section 3.1.1).

5.2 The PCapPacket Datatype

The PCapPacket class shown in Figure 5.2 is the XferDatatype used by all plugins in this package. A PCapPacket object encapsulates the raw data of a network packet (i.e., a simple u_char*), the pcap header for this packet (pcap_pkthdr) and additional information about both the trace (TraceInfo) and the TCP, UDP or ICMP flow (FlowSummary) the packet belongs to.
Next we consider the responsibilities and details of these auxiliary classes. Only the pcap pkthdr structure (Figure 5.2) is part of the Packet Capture Library. 

libpcap provides for every captured network packet the following information within this structure:

- **ts** is the packet’s (absolute) timestamp in seconds since January 1, 1970, 00:00:00 GMT and microseconds since the beginning of that second.

- **caplen** indicates how many bytes of the packet have been captured. Note that libpcap can be told to capture only the first $n$ bytes of a packet. Thus, it is also the length of the raw packet returned by `getPacket()` of class PCapPacket. For our purposes `caplen` should equal the actual packet length.

- **len** is the actual length of the packet (off wire).

Information about the traces is important especially for the PCapFileSink plugin. This data is needed to create a pcap file header. Therefore a pointer to the respective TraceInfo object is added to each PCapPacket which includes the following details about a trace:

- **traceid** is a unique ID for the trace within the MergeNetwork.

- **linktype** indicates the data link type of all packets belonging to the trace.
CHAPTER 5. A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

- **snaplen** specifies the maximum number of bytes a captured packet of this trace can consist of. Thus, it defines an upper bound for **caplen**.

- **pcap_version_major** is the major number of the **libpcap** version used to capture the trace.

- **pcap_version_minor** is the minor number of the **libpcap** version used to capture the trace.

A FlowSummary object provides trace manipulating components with valuable information about the (bidirectional) TCP, UDP or ICMP flow a packet belongs to.\(^2\) This information is, e.g., used by an IPSpoofer instance to determine whether a packet belongs to a flow whose packets should be spoofed\(^3\) according to an user-supplied spoofing rule. Moreover, flow summaries will also facilitate future plugins which adjust traffic characteristics by (re)moving or scaling certain flows (cp. Section 3.1.2). A FlowSummary object contains the following data:

- **flowid** is an unique ID for the flow within the MergeNetwork.

- **endpoints** is a pointer to an object that identifies the flow’s endpoints.

- **starttime** equals the timestamp of the flow’s first packet.

- **duration** equals the time difference between the timestamps of the flow’s last and first packet.

- **bytes** equals the sum of the lengths (**len**) of all packets (in both directions) belonging to the flow.

- **packets** equals the number of all packets (in both directions) belonging to the flow.

The class **Endpoints** encapsulates information about the source and the destination of a TCP, UDP or ICMP flow (see also Section 3.2.1). The source of a flow is the endpoint that has initiated the flow (the client). In general, the first packet belonging to the flow has been sent by this side. The destination of a flow is the responding side (the server). An Endpoints object keeps the following information:

- **srcip/dstip** specifies the IP address of the flow’s source/destination.

---

\(^2\)If a packet does not belong to a TCP, UDP or ICMP flow (e.g., an ARP packet) no FlowSummary is provided and **getFlowSummary()** returns a null pointer.

\(^3\)i.e., the IPs and ports of the packets should be forged.
• srcport/dstport specifies the port number used by the flow’s source/destination. Note that in the case of an ICMP flow no port numbers are included.

Thus, the description of our basic datatype is complete. Now we consider how it is processed by the components of the TTM package.

### 5.3 The PCapFileSource Plugin

This plugin is essential for traffic trace manipulation. An instance of the PCapFileSource plugin gathers data from user-supplied pcap files and flow summary files in order to provide the MergeNetwork with PCapPackets assembled from this data. The PCapFileSource class and its associated classes are shown in Figure 5.3. It gets connected to the MergeNetwork using an OutputSidePush<PCapPacket> interface.

Let us consider the component’s computation process under the assumption that it has been configured correctly and is ready to compute (cp. Table 4.1). If the component has not been activated, it does nothing when its compute() method is called. Provided that the component is activated when its compute() method is triggered, it tries to output the PCapPacket which it has assembled during the last computation round\(^4\). If this is not possible (because, e.g., the successor’s InputSide blocks further input) the component’s compute() method returns immediately.

Otherwise it creates a new PCapPacket as follows: It reads the next raw packet (u_char*) and the corresponding pcap packet header (pcap_pkthdr*) from

\(^4\)The first PCapPacket is assembled during configuration time. The reason for this “preprocessing” is implementation specific.
a pcap trace file and encapsulates this data in a PCapPacket object. Then the packet’s timestamp located in the pcap packet header is manipulated according to the component’s configuration (see below). If the packet belongs to a TCP, UDP or ICMP flow and a summary for this flow is available in the FlowSummaryContainer, it is retrieved using getFlowSumm() and attached to the PCapPacket object. This container provides a mapping between the 5-tuple consisting of the packet’s timestamp, client IP, server IP, client port and server port and the respective FlowSummary object. The summaries are parsed from a file (see below) and inserted (insertFlowSumm()) into the container when the respective pcap file is opened. flowIDs are supplemented by the PCapFileSource component.

If the container does not include a corresponding summary for a TCP, UDP or ICMP packet a dummy FlowSummary (containing minimal information) is created, added to the PCapPacket and inserted into the container. The packet’s timestamp is thereby taken as starttime and the Endpoints object is created based on the packet’s network headers.

Finally, characteristics about the source trace (TraceInfo) are added to the PCapPacket object. They are gathered from the header of the current pcap file and augmented with a unique traceID generated by the component. This completes the assembling of a PCapPacket and thus it is buffered for output.

The behaviour of the component can be customized through its configuration. The configuration parameters offered by a PCapFileSource are shown in Figure 5.4. More than one pcap file can serve as input. Trace files in pcap format can be created using tools like tcpdump [39]. It is recommended (but not mandatory) to provide each trace with a separate file which contains appropriate flow summaries. A summary file for input trace <file> is expected to have the name <file>.fsumm. A line of a flow summary file must have the format specified below:

\[
\text{<starttime> <duration> <srcport> <dstport> <bytes> <packets> <srcip> <dstip>}
\]

Because of the timestamp modifications described below, the given <starttime> of each flow summary entry is automatically adapted by the component. Summary files can be created using tools like tcpreduce [42].

Optionally a bpf filter expression and a timestamp adjustment value can be specified for each of the input files. Through the use of a filter expression certain packets can be selected from a trace file and pushed into the MergeNetwork. Otherwise all packets of the trace are supplied. If no specific filter expression is given for a file the value of the default_filter parameter is taken, if it is available. More about bpf filter expressions can be found in [40]. In the following, we refer to the stream of packets selected from trace number \(i\) as \(pstream_i\).

The starttime assigns an absolute timestamp to the first packet of \(pstream_i\). Of course, the timestamps of the remaining packets are adjusted in
Source <componentID> {

type "PCapFileSource";

Config {
    # Mandatory configuration parameter
    pcap_src_list = "<file1> [filter_expression] [starttime]
                     [, <file2> [filter_expression] [inter-pstream_gap]
                       [, ...] ... "];

    # Optional configuration parameters
    [default_ips_gap = <inter-pstream_gap>;
    [default_filter = <filter_expression>;

    }
}

Figure 5.4: Configuration of a PCapFileSource Instance

a way that preserves the inter-packet gaps of \textit{pstream}_{1}. The \texttt{starttime} expression can have one of the following forms:

- \([s]\), where \(s\) is either a time value in seconds and fractions of a second (up to microsecond resolution) or a time string in the format \texttt{MM/DD/YY HH:MM:SS}.

- \([a, b]\), where \(a\) and \(b\) are time values that specify an time interval from which \(s\) is chosen randomly.

If no \texttt{starttime} has been defined, the original timestamps of \textit{pstream}_{1} are kept.

The inter-pstream gap value for file \(i\) \((i > 1)\) of \texttt{pcap_src_list} defines the time gap between the timestamps of the last packet of \textit{pstream}_{i-1} and the first packet of \textit{pstream}_{i}. The packet timestamps of \textit{pstream}_{i} are adjusted accordingly. Note that these adjustments do not destroy the inter-packet gaps of the packets of this stream. Each inter-pstream gap expression can have one of the following forms:

- \([c]\), where \(c\) defines the time gap in seconds and fractions of a second (up to microsecond resolution).

- \([a, b]\), where \(a\) and \(b\) are time values that specify an time interval from which \(c\) is chosen randomly.
CHAPTER 5. A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

If no specific inter-pstream gap is given for a file, the value of default_ips_gap is used if available. If an interval is specified here, a separate c is chosen for each file where the default time gap is applied. If no default_ips_gap is defined, a value of 0 is assumed. Finally, it should be pointed out that we obtain a stream (consisting of (pstream)1...n) with relative packet timestamps starting from time 0 by assigning a value of 0 to starttime. This is a basic feature for the application of the merging operation described in Section 3.2.3.

A PCapFileSource instance is configured if at least one pcap file is provided in pcap Src_list which is accessible for reading. It is ready to compute if it is configured and its OutputSide has been correctly connected.

5.4 The PCapFileSink Plugin

Just like the PCapFileSource this simple plugin is essential. A PCapFileSink instance extracts the pcap packet data from incoming PCapPacket objects and stores it in a specified file in pcap format. The PCapFileSink can be connected to the MergeNetwork by the use of one InputSidePush<PCapPacket> (Figure 5.5).

Suppose now that the component has been configured correctly and is ready to compute. If the component is not activated it does nothing when its compute() method is called. In this case input from its predecessor is automatically blocked by its InputSidePush<PCapPacket>. If it is activated and a PCapPacket has been received it extracts the packet data and the corresponding pcap packet header from this object. If a starttime (see Figure 5.6) has been specified at configuration time the packet’s timestamp is adjusted. Otherwise it is left untouched. Moreover, it is checked whether the type of the link the packet has been recorded at is the same as the one of the packets already stored. Because normally a pcap file should only contain packets from the same linktype, a Notification of type COMPUTATION WARNING (Section 4.3) is sent to the MergeNetworkController, if a violation occurs. Also in this case the raw packet and its pcap packet header are stored in the destination file.

The component’s configuration parameters are shown in Figure 5.6. At least the name of the pcap output file must be specified (parameter pcap_dst_file). Additionally, a starttime can be stated just as in the configuration of the...
PCapFileSource component. In this case the relative timestamp of each incoming packet is calculated using the timestamp of the first packet as reference point. Then the value of parameter \textit{starttime} is added. A \textit{starttime} expression must have the following form:

- \textit{starttime} = s; where s is either a time value in seconds and fractions of a second (up to microsecond resolution) or a time string in the format MM/DD/YY HH:MM:SS.

A PCapFileSink is configured once the specified destination file could be opened for writing. Once its InputSide is connected, it is ready for computing.

```
Sink <componentID> {
    type "PCapFileSink";

    Config {
        # Mandatory configuration parameter
        pcap_dst_file = <file>;

        # Optional configuration parameter
        [starttime = <starttime>;
        ]
    }
}
```

Figure 5.6: Configuration of a PCapFileSink Instance

### 5.5 The PCapPacketSorter Plugin

The interweavement of traffic traces as described in Section 3.2.3 is implemented by the PCapPacketSorter plugin in cooperation with the PCapFileSource plugin. A PCapPacketSorter instance sorts PCapPackets from multiple InputSides into an ascending order based on their timestamps. It requires that the stream of packets arriving at the same InputSide is already sorted chronologically. This is, e.g., the case if packets arrive in the same order as they are pushed into the MergeNetwork by a PCapFileSource instance. A PCapPacketSorter provides an arbitrary number of InputSidePull<PCapPacket> interfaces and one OutputSidePush<PCapPacket> interface to connect it to the MergeNetwork (Figure 5.7).

Suppose now that the component has been correctly configured and is ready to compute. If the component is deactivated it does nothing when its \texttt{compute()}
method is triggered. Otherwise the component requests data from every predecessor when `compute()` is called for the first time. After that the method returns and the component does nothing until all packets have arrived. This is necessary since a PCapPacket from each InputSide must be available to decide which one to forward. It is understood that the component must keep track of its predecessors to not wait in vain for a packet from a predecessor which has run out of data. If all expected packets have been received the one with the lowest timestamp is put into the output stream and another packet is requested from the respective predecessor component. The PCapPacketSorter follows this strategy until its last predecessor runs out of data.

We can see from Figure 5.8 that a PCapPacketSorter instance does not need to be configured. It is ready to compute if its OutputSide and at least two of its InputSides have been connected to the MergeNetwork. The maximal number of InputSides that can be created is only limited by memory constraints.

```plaintext
IComponent <componentID> {
    type "PCapPacketSorter";
    # No specific configuration is needed
}
```

Figure 5.8: Configuration of a PCapPacketSorter Instance

5.6 The PushPullPipe Plugin

A PushPullPipe simply acts as a transfer mechanism converter between the `InputSidePull<PCapPacket>` interface of a PCapPacketSorter instance and the `OutputSidePush<PCapPacket>` interface of a PCapFileSource, an IPSpoofor or another PCapPacketSorter instance. Thus, a PushPullPipe instance can be connected to the MergeNetwork using one `InputSidePush<PCapPacket>` and one `OutputSidePull<PCapPacket>` interface (Figure 5.9). It buffers incoming PCapPackets and forwards them in a FIFO manner. If the pipe is deactivated it does nothing and input from its predecessor is blocked automatically by its InputSide.
A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

Figure 5.9: The PushPullPipe Plugin

Figure 5.10 shows the configuration of a PushPullPipe instance. There are no mandatory configuration parameters but one can optionally specify the maximal number of packets that can be stored in the buffer \( \text{max\_buffer\_size} \). If the buffer has reached the specified size, further input is blocked. This option is especially useful if memory is a constraint. The component is ready to compute if its InputSide and its OutputSide are connected.

Finally, it should be pointed out that the component pays no attention to the type of data (PCapPacket) it buffers. It could be reused within MergeNetworks which process different types of data. To make this possible further work on the framework is required.

```plaintext
IComponent <componentID> {
    type "PushPullPipe";
    Config {
        # Optional configuration parameter
        [max_buffer_size = <size_in_packets>];
    }
}
```

Figure 5.10: Configuration of a PushPullPipe Instance

5.7 The IPSpoofer Plugin

An instance of this plugin can be used to perform a basic adaption of traces to a desired environment (Section 3.2.2) or to adjust the number of unique hosts (Section 3.1.2) that appear within a generated trace. The IPSpoofer class and its associated classes are shown in Figure 5.11. An instance forges the IP addresses and port numbers of packets belonging to a TCP, UDP or ICMP flow according to a set of user-supplied spoofing rules. The manipulation of MAC addresses is not implemented yet. The component can be connected to the MergeNetwork using one InputSidePush<PCapPacket> and one OutputSidePush<PCapPacket> interface.
CHAPTER 5. A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

Figure 5.11: The IPSpoofer Plugin

Let us consider the component’s computation process. Suppose that it has been configured and is ready to compute. If the component is deactivated when its `compute()` method is triggered it simply forwards received packets without modifying them. Otherwise it first verifies whether a packet has an IP header. If the packet does not contain an IP header it is of no interest and forwarded immediately.

In the case of an IP packet the component checks whether already a spoofing entry exists for the flow the packet belongs to. The required information about this flow is given by the PCapPacket’s FlowSummary (cp. Section 5.2). A spoofing entry consists of the flow’s real EndPoints and the spoofed (i.e., forged) EndPoints. Its retrieval (`find()`) from a SEContainer is a fast operation because a hash map has been chosen as the underlying data structure. If no entry can be found for the flow one is derived from the set of spoofing rules (see below) that has been specified by a user at configuration time. The SRInterpreter class is responsible for the storage (`addRule()`) and generation (`deriveSpoofing()`) of spoofing entries. If the flow does not match the pattern of any rule a special entry is returned indicating that the flow’s packets should be left untouched. Each generated entry is stored (`insert()`) in the SEContainer using the respective `flowID` as key value. It is removed (`erase()`) again after the last packet of the flow has been spoofed.

If a packet’s IP addresses and port numbers are replaced according to a spoofing entry the IP and TCP/UDP checksum is recalculated. The FlowSummary object belonging to the PCapPacket object is updated as well. Finally, the component tries to transmit the spoofed packet to its successor.
IComponent <componentID> {

    type "IPSpoofer";

    Config {
        # Mandatory configuration parameter

        # Spoofing rules can be specified either directly ...
        spoofing_rules = "<pattern> <spoofing_type> <spoofing>
                         [, <pattern> <spoofing_type> <spoofing>
                         [, ...] ... ]";

        # ...or within a separate file.
        sr_file = <file>;
    }
}

Figure 5.12: Configuration of an IPSpoofer Instance

Next we consider the component’s configuration shown in Figure 5.12. Spoofing rules can be specified either directly as the value of parameter spoofing_rules or within a separate file by using parameter sr_file. It is mandatory to choose one of these possibilities. Spoofing rules are comma-separated. The right hand side of a rule (<spoofing>) specifies the modifications which should be applied to a flow’s packets if this flow matches the pattern specified on the left hand side (<pattern>) of the rule. The order in which the rules are entered is important because only the modifications implied by the first matching rule take effect.

Figure 5.13 presents the common structure of a spoofing rule. The <src_side_pattern> and <dst_side_pattern> define which criteria a flow’s source and destination Endpoint must fulfill to be spoofed. It is possible to specify a network and a port pattern for each Endpoint. A <network_pattern> has one of the following forms:

- host A.B.C.D, where A.B.C.D is an IPv4 address.

- net A.B.C.D/n, where A.B.C.D/n specifies the network in CIDR notation (e.g., 134.96.68.0/24). Note that host A.B.C.D is equivalent to net A.B.C.D/32.

A flow’s source (destination) Endpoint matches the respective <network_pattern> if its source (destination) IP belongs to the specified network. This is the case if the leftmost \( n \) bits of the IP address and the
leftmost \( n \) bits of \( A.B.C.D \) are identical. If no \(<\text{network_pattern}>\) has been stated for a source (destination) Endpoint every \( srcip (dstip) \) matches.

The application of a spoofing rule to packet flows from or to matching hosts can be further restricted by specifying a \(<\text{port_pattern}>\). Such a pattern has one of following forms:

- \( \text{port } n \), where \( n \) is a port number (i.e., a number between 0 and 65535).
- \( \text{port } [a, b] \), where \( a \) and \( b \) are port numbers which define a range. A flow’s \( srcport (dstport) \) matches this pattern if the following applies:
  \[ a \leq srcport (dstport) \leq b. \]

If no \(<\text{port_pattern}>\) is included within a \( \text{src_side_pattern} \) (\( \text{dst_side_pattern} \)) every \( srcport (dstport) \) matches. In this case even a non-existent port of an ICMP flow matches.

The \( \text{src_side_spoofing} \) and \( \text{dst_side_spoofing} \) define which modifications are applied to the packets of a flow if this flow matches the left hand side of a rule. A flow’s \( srcip \) or \( dstip \) can be forged using one of the following \(<\text{network_spoofing}>\) statements:

- \( \text{host } A.B.C.D \), where \( A.B.C.D \) is an IPv4 address. This statement replaces the respective IP address with \( A.B.C.D \).
- \( \text{net } A.B.C.D/n \), where \( A.B.C.D/n \) specifies a network in CIDR notation. This statement replaces the respective IP address with an IP randomly chosen from network \( A.B.C.D/n \).
- \( \text{host } * \) This statement replaces the respective IP address with an IP randomly chosen.

If the \(<\text{network_spoofing}>\) statement is omitted in a \( \text{src_side_spoofing} \) or \( \text{dst_side_spoofing} \) the respective IP address is kept.

The \(<\text{spoofing_type}>\) of a rule specifies how multiple flows from or to the same host are spoofed:

- \( \rightarrow \) enforces a well-defined mapping of \( srcip \) and \( dstip \) addresses within the scope of a rule. Supposed there exist multiple flows with the same \( srcip (dstip) \) that match this rule, then the same spoofing IP is assigned to the \( srcip (dstip) \) field of each flow.
# Spoofing types

\[ <\text{src\_side\_pattern}> <\text{dst\_side\_pattern}> \rightarrow <\text{src\_side\_spoofing}> <\text{dst\_side\_spoofing}> \]
\[ <\text{src\_side\_pattern}> <\text{dst\_side\_pattern}> \rightarrow* <\text{src\_side\_spoofing}> <\text{dst\_side\_spoofing}> \]

# Patterns

\[ <\text{src\_side\_pattern}> ::= [\text{src} <\text{network\_pattern}>] [\text{src} <\text{port\_pattern}>] \]
\[ <\text{dst\_side\_pattern}> ::= [\text{dst} <\text{network\_pattern}>] [\text{dst} <\text{port\_pattern}>] \]

# Spoofing actions

\[ <\text{src\_side\_spoofing}> ::= [\text{src} <\text{network\_spoofing}>] [\text{src} <\text{port\_spoofing}>] \]
\[ <\text{dst\_side\_spoofing}> ::= [\text{src} <\text{network\_spoofing}>] [\text{src} <\text{port\_spoofing}>] \]

Figure 5.13: The Common Structure of a Spoofing Rule

- \( \rightarrow^* \) enforces that a new spoofing IP is chosen for each flow that matches this rule. Thus the same srcip(dstip) can be mapped to different spoofing IPs. This type of spoofing is useful to increase the number of unique hosts within a traffic trace (cp. Section 3.1.2).

Forging of port numbers is also possible. A flow’s srcport(dstport) can be forged using one of the following <port spoofing> statements:

- \( \text{port } n \), where \( n \) is a port number.
  This statement assigns \( n \) as srcport(dstport) of every matching flow.

- \( \text{port } [a, b] \), where \( a \) and \( b \) are port numbers which define a range.
  By the use of this statement \( n \) is chosen randomly from this range and assigned as srcport(dstport) of a matching flow. Thereby a new spoofing port is chosen for each flow.

- \( \text{port } * \)
  This statement is equivalent to \( \text{port } [0, 65535] \).

If the <port spoofing> statement is omitted in a <src side spoofing> or <dst side spoofing> the respective port number is kept.

An IPSpoofer instance is configured if at least one spoofing rule is specified by a user. Once its InputSide and OutputSide are connected to the MergeNetwork, it is ready to compute.
5.8 Application: Interweavement of Attack and Background Traffic Traces

In this section we present a simple application of the plugin package and thus also of our architecture. Let us use the existent plugins to interweave two attack traces with a background traffic trace as described in Section 3.1.1. Of course, a test trace containing only two attacks is not suitable to determine the false positive/negative rates of an NIDS but it shows the principle and simplifies the example.

Figure 5.14 depicts the attack traffic traces that should be interweaved with our background traffic. Both traces have been recorded within our testbed (192.168.0.0/24) by launching attack scripts against none vulnerable victim software. Thus neither attack was successful. The first trace pftp\_dos (a) contains the network packets that have been exchanged between the attacking host 192.168.0.1 and the victim host 192.168.0.2 while executing an attack script that tries to exploit a vulnerability (buffer overflow) in the PowerFTP server software (v2.03). The second trace iis\_dt (b) contains the network packets that have been exchanged between the attacking host 192.168.0.2 and the victim host 192.168.0.1 while executing an attack script that tries to exploit...
a vulnerability (directory traversal bug) in Microsoft’s IIS HTTP server software (v5.0).

In order to interweave these attack traces with some background traffic trace, we propose in Section 3.1.1 to not just merge the traces. Rather two additional manipulations are recommended: The attack traffic should be adapted to the background traffic and some interaction between attack and background traffic should be simulated. Since both attacks have not been successful, the simulation of interaction is not mandatory.

If we want to adapt the attack traffic to the background traffic we need some information about the network environment where the background traffic has been collected. Let us suppose we own a background traffic trace \( bgt_1 \) that has been recorded somewhere at the network 134.96.223.0/24. This trace mainly contains traffic from and to the local HTTP server 134.96.223.242 and FTP server 134.96.223.215. Thus, we pretend that the hosts 134.96.223.242 and 134.96.223.215 are the victims of our attacks. Additionally, we want to inject the attack traces within the small section of this trace that is depicted in

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Prot.</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>107439658.056150</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>58357 &gt; www [SYN] Seq=4259045563 Ack=0</td>
</tr>
<tr>
<td>107439658.060607</td>
<td>134.96.223.215</td>
<td>66.104.13.105</td>
<td>FTP</td>
<td>Response:</td>
</tr>
<tr>
<td>107439658.061260</td>
<td>134.96.223.215</td>
<td>66.104.13.105</td>
<td>TCP</td>
<td>FTP Data: 471 bytes</td>
</tr>
<tr>
<td>107439658.061268</td>
<td>134.96.223.215</td>
<td>66.104.13.105</td>
<td>TCP</td>
<td>30456 &gt; 46606 [FIN, ACK] Seq=4269971195 Ack=4251019494</td>
</tr>
<tr>
<td>107439658.064009</td>
<td>66.104.13.105</td>
<td>134.96.223.215</td>
<td>FTP</td>
<td>Request: TYPE I</td>
</tr>
<tr>
<td>107439658.076677</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>www &gt; 58357 [SYNC, ACK] Seq=2735428675 Ack=4259045564</td>
</tr>
<tr>
<td>107439658.077513</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>58357 &gt; www [ACK] Seq=4259045564 Ack=2735428676</td>
</tr>
<tr>
<td>107439658.077885</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>GET /8_walFF2J1A3ZQQG/html.jsp?target=firmenportr ait.html HTTP/1.1</td>
</tr>
<tr>
<td>107439658.081128</td>
<td>134.96.223.215</td>
<td>66.104.13.105</td>
<td>TCP</td>
<td>30456 &gt; 46606 [ACK] Seq=4269971196 Ack=4251019495</td>
</tr>
<tr>
<td>107439658.082387</td>
<td>134.96.223.215</td>
<td>66.104.13.105</td>
<td>FTP</td>
<td>Response: SIZE diff-1.3.28i-1.3.99i.gz</td>
</tr>
<tr>
<td>107439658.083009</td>
<td>66.104.13.105</td>
<td>134.96.223.215</td>
<td>FTP</td>
<td>Request: SIZE diff-1.3.28i-1.3.99i.gz</td>
</tr>
<tr>
<td>107439658.098236</td>
<td>134.96.223.18</td>
<td>209.89.161.91</td>
<td>TCP</td>
<td>2086 &gt; 38740 [PSH, ACK] Seq=4138065718 Ack=3907551068</td>
</tr>
<tr>
<td>107439658.098626</td>
<td>134.96.223.242</td>
<td>131.152.123.21</td>
<td>TCP</td>
<td>www &gt; 58357 [ACK] Seq=2735428676 Ack=4259046074</td>
</tr>
<tr>
<td>107439658.098865</td>
<td>209.89.161.91</td>
<td>134.96.223.18</td>
<td>TCP</td>
<td>38740 &gt; 2086 [ACK] Seq=3907551068 Ack=4138065730</td>
</tr>
</tbody>
</table>
```

Figure 5.15: Section of Background Traffic Trace \( bgt_1 \)
 CHAPTER 5. A PLUGIN PACKAGE FOR TRACE MANIPULATIONS

Components {
  # Background Traffic Source
  Source bgt_source {
    type PCapFileSource;
    Config {
      pcap_src_list = "bgt_1";
    }
  }
}

# Attack Traffic Source
Source attack_source {
  type PCapFileSource;
  Config {
    pcap_src_list = "pftp_dos [1074339658.060563],
           iis_dt [0.009, 0.019];
  }
}

IComponent spoofer {
  type IPSpoofer;
  Config {
    spoofing_rules =
       *dst port 80 -> src host 131.152.123.21
        dst host 134.96.223.242,
       dst port 21 -> src host 134.96.223.105
        dst host 134.96.223.215*;
  }
}

IComponent psorter { type PCapPacketSorter; }

Sink sink {
  type PCapFileSink;
  Config {
    pcap_dst_file = "bgt_attack_mix";
  }
}

# Pipe between bgt_source and psorter
IComponent pipe1 { type PushPullPipe; }

# Pipe between spoofer and psorter
IComponent pipe2 { type PushPullPipe; }

Connections {
  bgt_source[0] -> pipe1[0];
  attack_source[0] -> spoofer[0];
  spoofer[0] -> pipe2[0];
  pipe1[0] -> psorter[0];
  pipe2[0] -> psorter[1];
  psorter[0] -> sink[0];
}

Figure 5.16: Configuration: Interweavement of Background and Attack Traffic

Figure 5.15. To further hide the attacks within the background traffic we can manipulate them such that they seem to originate from the hosts 131.152.123.21 and 66.104.13.105 which have ongoing HTTP and FTP sessions at this time.

Figure 5.16 presents a configuration for our architecture that creates and configures a MergeNetwork for our purposes. By running our software system with this configuration the attack traces pftp_dos and iis_dt are adapted as described and then merged into the background traffic trace bgt_1. As injection point for the first attack trace time 1074339658.060563 is chosen (i.e., right after the first packet of the background traffic section depicted in Figure 5.15). Then the second attack trace is injected 0.009 seconds till 0.019 seconds after the last packet of the first attack trace. Figure 5.17 shows the respective section of trace bgt_attack_mix that results from an actual run of our software system. Here the attack trace iis_dt has been injected 0.018602 seconds after the last packet of trace pftp_dos.

---

6This may not be true if the NIDS has collected certain characteristics of the hosts from their normal traffic and strongly responds to deviations.
<table>
<thead>
<tr>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Prot. Info</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1074339658.056150</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>58357 &gt; www [SYN] Seq=4259045563 Ack=0</td>
</tr>
<tr>
<td>1074339658.060563</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>TCP</td>
<td>32858 &gt; ftp [SYN] Seq=2403014348 Ack=0</td>
</tr>
<tr>
<td>1074339658.060589</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>Response: 150 About to open data connection.</td>
</tr>
<tr>
<td>1074339658.060607</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>Response: 32858 [SYN, ACK] Seq=605233624 Ack=2403014349</td>
</tr>
<tr>
<td>1074339658.060589</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>32858 &gt; ftp [ACK] Seq=2403014349 Ack=605233625</td>
</tr>
<tr>
<td>1074339658.061008</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>Data: 471 bytes</td>
</tr>
<tr>
<td>1074339658.061008</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>FTP Response: 226 File transfer complete.</td>
</tr>
<tr>
<td>1074339658.061008</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>220 Personal FTP Server ready</td>
</tr>
<tr>
<td>1074339658.061008</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>30456 &gt; 46606 [FIN, ACK] Seq=4269971195 Ack=4251019494</td>
</tr>
<tr>
<td>1074339658.061008</td>
<td>131.152.123.21</td>
<td>134.96.223.242</td>
<td>FTP</td>
<td>Request: Type I</td>
</tr>
</tbody>
</table>

Figure 5.17: Section of the Interweaved Background and Attack Traffic
5.9 Summary and Future Work

This chapter has presented the implemented part of a plugin package for the manipulation of traffic traces in pcap format. As we have seen in Section 5.8, using the existent plugins we are already able to create test traces for the determination of false positive/negative rates (cp. Section 3.1.1). The source code of the described plugins can be found at [2].

In a next step we plan to implement and experiment with a component which tries to eliminate timestamp collisions by scaling flows and by moving or removing single packets/flows (cp. Section 3.2.3). Due to complexity concerns and to comply with our modularity and flexibility goals, it may be reasonable to split up this component into two components: One component performs the analysis, decides which operation are applied to which flow/packet and informs a second component about these decisions which actually enforces these operations. By this means we can easily reuse the second component for other purposes because it implements basic trace manipulations.

Another idea is to implement a source plugin which is able to retrieve traffic traces from a database according to some sort of descriptions. In the simplest case this can be, e.g., the names of attacks or an attack category.

Also we take into account to work together with the developers of tcpreplay [43] in order to realize the functionality of tcpreplay as plugins of our architecture.
Chapter 6

Conclusion

This work covers the development of a software system for packet trace customization with application to NIDS evaluation. By considering concrete scenarios of how traces could aid in evaluating false positive/negative rates and the capacity of an NIDS, we have identified a set of trace manipulating operations that are necessary to realize required trace customizations. Then we have discussed whether and in which cases these operations generate any irregularities that cannot occur in real-world traffic and if these artifacts could be problematic for NIDS testing.

The major part of this work has dealt with the design and implementation of a flexible and expandable software system for trace customization. The core of this software system is an abstract architecture that is able to manage and control filter systems for arbitrary types of data. This architecture allows to realize trace manipulating operations as small independent components that can be integrated into the architecture at runtime. By means of a convenient configuration language users can specify which and how instances of these components should be interconnected to build a filter system, i.e., the types of operations and the order in which they are applied can be chosen freely. This enables the composition of more complex trace customizations. Moreover, the architecture allows users to supply own component plugins to extend the system’s functionality in terms of data acquisition, manipulation and output.

Within the scope of this work a basic set of plugins for trace manipulations has already been developed. This package consists of

- the PCapFileSource plugin whose instances can supply a filter system with network packets from pcap files, adjust packet timestamps and map packets to flows on the basis of user-supplied flow specifications.

- the PCapFileSink plugin whose instances write packets that have been manipulated by a filter system back to a pcap file.
• the IPSpoof plugin whose instances can forge the IPs and ports of packet flows according to a set of user-supplied spoofing rules that have a bpf-like syntax.

• the PCapPacketSorter plugin whose instances merge packets from multiple input channels chronologically.

First experiments demonstrated that the architecture with the currently implemented set of plugins indeed is a valuable tool for flexible, fine-grained trace customization.

However, several questions and problems remain open which are concerned with the issue of artifacts. Experience still has to show to what degree these artifacts deteriorate the applicability of trace-aided evaluation. Further experiments and theoretical analyses are required to determine whether certain kinds of artifacts must be avoided to properly test an NIDS and if the avoidance of these artifacts is technically feasible.

In order to contribute to the development of trace-aided evaluation, our software system needs to be extended along with the identification of new usage scenarios and adherent trace manipulations. Besides the creation of new input, output and manipulation plugins, it would be worthwhile to develop a higher level language for trace customization on top of our architecture and plugins. Finally, we hope that our software system will be elaborated enough to become a part of a future all-in-one NIDS evaluation platform as demanded in [3].
Bibliography


URL: http://www.net.informatik.tu-muenchen.de/~rdc/.


URL: http://www.ethereal.com/.

URL: http://expect.nist.gov/.


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


