ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Dominik Uebersax

Implementation of an Anomaly Injection Framework

Diploma Thesis DIP-2008
March 2008 to July 2008

Tutor: Daniela Brauckhoff
Co-Tutor: Arno Wagner
Supervisor: Prof. Bernhard Plattner

Abstract

In order to answer several remaining open questions in the area of flow-based anomaly detection, network traffic traces containing anomalies with selected characteristics are a prerequisite. In this project the implementation of an anomaly injection framework called FLAME is presented. The framework combines the controllability offered by simulation with the realism provided by real traffic, by injecting hand-crafted anomalies into a given background traffic trace.

Zusammenfassung

Auf dem Gebiet der Anomaly Detection gibt es immernoch etliche offene Fragen, die man zu loesen versucht mit Hilfe von Netzwerk-Traffic-Traces, welche Anomalien mit ausgewaehlten Charakteristiken enthalten. In dieser Arbeit wird die Implementation eines Frameworks namens FLAME vorgestellt. Das Framework kombiniert die Kontrolle welche man ueber simuliertem Netzwerktraffic hat mit dem Realismus von reelem Traffic, indem es von Hand konfigurierbare Anomalien in gegebenen Hindergrund-Traffic einspeist.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network anomaly detection systems today are deployed in major backbone infrastructures. Detection systems that work on the flow level are used for a low footprint detection. To create a tool to test these detection system, this framework for the generation of anomaly traffic traces was designed and implemented in parallel with the paper by Brauckhoff et al [3] describing the approach.

1.1 Motivation

Up to now there were generally two approaches to test anomaly detection systems. The first one is to use existing traces of known attacks and anomalies in network traffic. The drawback of this approach is that the data is either very short and static or the data is anonymized or sampled. Both scenarios are not optimal to thoroughly test detection systems. Other approaches involve the simulation and generation of complete traffic traces on the packet as well as on the flow level. This approach has the drawback that even the background traffic is less realistic and is therefore less optimal for testing. The approach taken here takes the best from the previous approaches by combining the usage of collected traffic, which doesn't contain any anomalies as background traffic and injecting simulated anomaly traffic into those traces. We believe that this gives a much better result in testing and developing anomaly detection systems.
The framework will also serve as a general tool when working with NetFlow data. It should be easy to implement new components or modify existing ones and use them in combination with the components implemented in this project.

1.2 The Task

The task of the this project was to implement a first prototype of an anomaly injection system which runs on Unix-like environments and is able to inject handcrafted anomalies into existing flow traces. The actual project assignment was divided into three major parts. The first part was to analyze various kind of anomalies and to find and extract features to describe them. The main part then was the design and implementation of the framework and all its components. The last step of the assignment was to evaluate the implemented framework using real data from the SWITCH [1] network.

1.3 Related Work

This work was very much inspired by the paper on packet trace manipulation from Rupp et al [10]. The authors present several basic manipulation operations for packet traces, namely, merging, adapting, stretching or compressing, moving, and duplicating. The main difference is that they work with packet traces, while we rely on flow traces. Moreover, they do not provide methods for generating anomalous traffic, but rely on existing packet traces that already contain attack traffic. In contrast, the focus of this work is to simulate the anomalous traffic since this gives us full control about the characteristics of the anomalous traffic.

Another of inspiration was the work by Mirkovic et al [8] on modeling denial of service attacks and countermeasures. We will try to explore the knowledge about DoS attack characteristics gained in this work. This approach differs from that work in two ways: first we use an existing trace as background traffic, and second we do not restrict ourself to denial of service attacks.

Trident, developed by Sommers et al [12] is another tool for generating malicious and benign IP packet traffic traces. Benign traffic is generated using application-specific automata and a tool called Harpoon [11], which was developed earlier by the authors and by the work of Vishwanath et al [14] who describe the creation of realistic traffic on the packet level. Attack traffic is generated with MACE [12], which was extended to support 21 different attacks (e.g., Welchia, teardrop, synflood) for this work. Trident focuses on the evaluation of packet-level NIDS such as Bro, while our goal is to evaluate flow-based anomaly detection systems. In contrast to NIDS, these systems do not rely on individual packet header or payload characteristics but on link- or host-level statistics such as flow or packet counts.

To get an idea on how anomaly detection mechanisms work, several papers were consulted. There is the work of Myung-Sup et al [4] which presents an anomaly detection algorithm using flow data. The work of Gerhard Muenz and Georg Cale is on a real-time analysis of flow data for anomaly detection and described in [9] and also helped me to gain inside into anomaly detection. In the last paper by by Soule et al [13] filtering and statistical methods were used as an anomaly detection approach.

Several other sources were used as inspiration for the first part of this work, where traffic and anomaly features were listed and defined to create a usable set of description attributes for any anomaly. Most notably there was the work from Lakhina et al [7] for mining anomalies using traffic feature distributions. Two papers mainly contributed to the anomalies list in chapter 2. First the work of Paul Barford and David Plonka [2] which is called "Characteristics of Network Traffic Anomalies" and the very similar work by Anukool et al [5] named "Characteristics of Network-Wide Anomalies in Traffic Flows".

1.4 Overview

Chapter 2 is all about the definition and modeling of network anomalies and network traffic in general. The following chapter (3) continues with traffic generation theory and proposes a model to create realistic results. The next chapter (4) presents the overall application design and briefly describes all involved components. All components and their implementation are described in more detail in chapter 5. The results and an evaluation of the project are presented in the 6th chapter. In the 7th and last chapter various ideas for future work are proposed and a conclusion about the project is given.

Chapter 2

Anomalies

In the first section of this chapter the term anomaly is defined. The next section gives an overview over different anomalies that can surface when analyzing Netflow data. In the subsequent section a set of parameters is defined, which enables one to define all kind of anomalies by assigning different values to the parameters. The next section shows an approach to describe insertion and deletion actions on the flow level and continues by using them to model some anomalies.

2.1 What is an Anomaly

Before we start to define various kinds of anomalies we need a general definition of such an anomaly. We define the term as follows: We speak of an anomaly when something happens on the network that results in a significant change of the normal network traffic. An anomaly does always have a reason: Either something goes wrong, something is changed or someone tries to maliciously do harm.

2.2 List of Anomalies

The following tables contain a categorized list of anomalies that can occur in networks and netflow data analysis.

Operational Anomalies	Description
Outages	Outages can occur at various levels in a network. For example a single server or a router can have an outage. When outages occur all traffic from and to the device disappears. There are various reasons for outages, it is for example possible that the device itself has a failure or that the physical connection between two points gets damaged.
Ingress shift	Ingress shifts can occur for example when there are changes in routing policies or when clients change addresses of services. There are other type of intended change which result in ingress shifts. The traffic affected by such changes does not disappear but shifts. It is also possible that the traffic will be routed through another router and it will appear like an outage on the monitored device.
Export failure	Netflow data is exported as UDP packets which can get lost in between the router and the collector. It is also possible that the export did somehow fail on the router itself.
Table contention	The router which is exporting the netflow data has a table containing all active flows which have not reached one of their timeouts. The space of that table is limited and when there arrive new flows while the table is full, the new flows will be dropped. The netflow export will then be missing that data.
Plateau behavior	It is possible that a plateau behavior occurs on a link when environmental limits are reached. This mean that no more traffic can fit on that specific line and a plateau like graph will arise when plotting traffic for that link.

Table 2.1: Operational Anomalies

Abuse Anomalies	
DoS	A Denial of Service attack tries to overwhelm a victim by flooding it with requests. There are various kinds of DoS attacks. Some try to abuse implementation weaknesses such as the SYN-flood attack, others just flood the line to the victim so that legitimate requests can no longer arrive. DoS attacks are nicely described in [8]
DDoS	Distributed Denial of Service attacks take this to the next level by starting a coordinated attack from a big number of different hosts, overwhelming their victim. DDoS attacks can succeed where normal DoS attacks fail for example due to a really high bandwidth connection of the victim.
Scans	Scans are anomalies that often precede an actual attack and are used to get an overview over potential victims and vulnerabilities. Network scans scan whole subnets or other IP ranges and are usually limited to a specific port. While port scans are normally only to one or few hosts scanning a wide range of ports.
Worms	There are various kinds of worms. They can implement features of the above abuse anomalies as well as other behavior. For a taxonomy of worms see [15]

Table 2.2: Abuse Anomalies

Other	
Flash crowds	Flash crowds are similar to DDoS attacks because they generate a large amount of incoming requests on a web server for example. But unlike a DDoS attack flash crowds include only legitimate requests from real clients. A flash crowd can for example occur when a web site is highly advertised during a popular event or when a long awaited software version is released. Flash crowds typically start with a sharp peak in requests which gradually decrease over time.
Alpha events	We speak of an alpha event if a high (noticeable) amount of traffic is flowing between few hosts, for example due to large file transfers.

Table 2.3: Other Anomalies

2.3 Parameterizing Anomalies

As seen in the previous section there are a lot of different anomalies and before we can begin to model them, we have to define general traffic attributes. The goal of this section is to extract a number of such attributes which should enable one to describe any anomaly. Using, amongst others, ideas from [7], [11] and [14] table 2.4 was created showing the 7 main attributes which were chosen for the characterization.

Attribute	**Description**
Rate	The rate at which the anomaly occurs. For example constant rate from beginning to end. The rate is defined by a set of probability distributions which are further discussed in the next chapter
Duration	The points in time where the anomaly starts and ends.
Protocol	The protocol of the anomaly traffic. For example TCP, UDP or ICMP.
Topology	The topology describes the source and destination points involved in the anomaly traffic. This can be a big list of sources to one host or from one source to many destination hosts. Any other combinations are thinkable.
Ports	The source and destination ports of the anomaly traffic. This can be many to one, one to many or any other kind of combination.
Type	There are two major types: Either the anomaly inserts traffic or it deletes traffic.
Level	This is not really an attribute, but one needs to be clear on what level the traffic is generated or analyzed. We work either on the packet level or on the flow level.

Table 2.4: Attribute descriptions

2.4 Modeling Anomalies

This section is all about modeling anomalies. This is done on the flow level and using the parameterization defined in the last section. As seen in the introduction a flow is described by a fixed five-tuple of attributes (source/destination IPs and ports, and the protocol). We first define general insertion and deletion actions using flows. And in the next step some anomalies are modeled with these general insertion and deletion actions using pseudo code. The modeling is very much simplified and we will discuss more complex traffic generation in more detail in the next chapter where we will see that most parameters used in this section to describe the traffic will actually be picked from distributions.

2.4.1 General Insertion and Deletion Actions

All attributes are starting with a $-sign representing variables, which will have to be defined for a concrete definition of the anomaly traffic. The flow-defining parameters are omitted in this first definition.

```
Insert $rate flows between $start and $end
```

This insertion action is defined as general as possible. The $rate only defines the load (number of bytes) and the inter arrival times (time between packet arrivals), since we only look at single flows. The arrival times distribution of various flows is to be included when combining various insertions.

```
Delete flows between $start and $end
```

To form an anomaly by deleting certain flows, we will have to match all flows against a list of masks and delete the matches.

2.4.2 DDoS

To model a distributed denial of service attack, we need to perform a number of insertion actions. There will be traffic from many hosts (real or spoofed) to a single host and single port (usually). To model the attack more realistically the $start and $end times need to be randomized a bit, since not all attacking hosts will start and stop at exactly the same moment. There are many variants of DDoS attacks using different protocols so I won't specify it here. The following model is a simple DDoS attack from a list of hosts to a single host and port. All attackers start at approximately the same time.
Of course also the rate of each flow would have to be randomized a bit, since not all attackers posses equally good connections.
I didn't incorporate the flows coming from the victim, since this depends greatly on the attack.

```
// $start: Start time of the attack
// $end: End time of the attack
// $hosts: List of hosts participating in the attack
// $server: The victim host
// $port: The port connected to on the victim
// rand(): Small random time value

foreach (host in $hosts to $server:$port) {
  Insert constant rate flow between $start+rand() and $end+rand()
}
```

2.4.3 Flash Crowd

A flash crowd is quite similar to a DDos attack, but it normally originates from a majority of known hosts or host groups, whereas a DDos attack originates at a big number of different and mostly unknown host groups. Flash crowds are also characterized by a sharp request peak at the beginning (not quite as sharp as an attack) and a gradual drop off in requests over time. To model the flash crowd we can't just insert all the requests with the mentioned rate, but the distribution of simultaneous different requests has to display the overall rate of the event. There is also no protocol specified here.
What we do is to pick a number of hosts from a host list, which simultaneously connect to a server on a specific port. The number of hosts is picked dependent on the time of the distribution (e.g. p(t)) assigned to the event. For each inserted request flow, there is also an answer flow back to the requesting host.

```
// $start:  Start time of the anomaly
// $end:  End time of the anomaly
// $hosts:  List of hosts connection to the server
// $server: The victim host
// $port: The port connected to on the victim
// $peak: The max number of arriving connections at a time intervall
// $rtt:  Round Trip Time from router to victim and back
// dist(t): Distribution of connection arrivals

for (t from  $start to $end) {
  select (dist(t) * $peak) #hosts from $hosts to $server:$port {
    insert constant rate flows between t and t+$time
    insert reverse flows between t+$rtt and t+$time+$rtt
  }
}
```

2.4.4 Server Outage

If there is a server outage somewhere, all traffic from the server will immediately be dropped. Packets to the server will remain unanswered and also disappear within a small delay.

```
// $start:  Start time of the outage
// $end:  End time of the outage
// $mask: mask to select all flows involving the server
// $delay:  Delay until traffic to the server stops
// rand():  Randomization for delay

for (any flows matching $mask) {
  for (flows to server) {
    delete flows between $start+$delay+rand() and $end
  }
  for (flows from server) {
    delete flows between $start and $end
  }
}
```

2.4.5 Ingress Shift

An ingress shift occurs if a party shifts traffic from one ingress point to another. So there will be a decrease in traffic in one group of flows (which involve the old ingress point) and a spike for new/other flows (which involve the new ingress point). Since this is a permanent change, there is only a start time. If the shift is only for one service, a port would have to be specified too. To model the change, insertion and deletion actions are combined accordingly. To implement the change the existing flows would just have to be modifiied to reflect the change (replace destination and source addresses respecively). The following pseudo code models an ingress shift for a server or service change.

```
// $start:  Start time of the change
// $mask: mask for selecting hosts, ports and protocol implementing
    the changes

for (flows matching $old_ingress_point AND $mask) {
  for (flows from $old_ingress_point) {
    delete flows beginning at $start
    insert same flows from $new_ingress_point beginning at $start
  }
  for (flows to $old_ingress_point) {
    delete flows beginning at $start
    insert same to from $new_ingress_point beginning at $start
  }
}
```

Chapter 3

Traffic Generation: Theory

To actually be able to implement the modeling of anomalies as discussed in the previous chapter, we now need to focus on the generation of the actual traffic. Various approaches have been taken in this field, ranging from packet based traffic generation using the actual IP stack of the Operating System as described by Sommers et al in [11] to traffic generation on the flow level as described in a previous semester thesis by REF [SA Ref 8]. The approach taken in this project is very closely related to the latter, with the difference that we don't need to model a whole set of traffic to simulate a complete trace going through a router, but only to inject, delete or modify certain flows to simulate anomalies. Another fundamental difference is that the traffic generated on the packet level is only simulated in software, there are no actual packets created using the IP Stack of the operating system.

3.1 Traffic Generation Overview

To create realistic netflow records we are using a layered approach. In a first step IP packet headers are generated which in a next step are collected and processed by a class simulating an actual router. The flow records are created and output by the router simulator. Flow records that are output have the following basic attributes:

Per flow settings
- Source address
- Destination address
- Source port
- Destination port
- Protocoll
- Start time
- End time
- Packet count
- Total size in bytes

3.2 Packet Generation

This section describes how the traffic is generated on the packet level. These packets are collected by the class simulating the router as mentioned in the last section. To generate realistic traffic we need several distributions to model the desired network traffic. [11] proposes the following 3 main distributions: *1. file size, 2. inter connection times, 3. active sessions*. We take a similar approach here and need the following distributions:

1. Distribution of packet sizes
2. Distribution of packet inter-arrival times
3. Distribution of inter-connection times
4. Distribution of session lenghts
5. Distribution of connection arrivals or active connections
6. Distribution of Round Trip Times

The file sizes used in [11] are indirectly modeled via the packet sizes, inter-arrival times and session lengths. Distributions 1 and 2 need to be available for both directions, since in most cases there will be a flow in each direction (e.g. TCP where every packet has to be acknowledged). The RTT distribution models the latency between the requests and answers as observed on the router. Furthermore we need distributions for source and destination IPs, as well as for ports:

7. Source/destination IPs distribution
8. Source/destination ports distribution

Using these 8 distributions the main packet generator (a script) creates packet generators which simulate traffic between 2 hosts. All packet generators are created and instantiated before the simulation can start. The basic packet generators are created with values picked from the global distributions for start and end time, the source/destination hosts and ports, the session length, as well as distributions for the packet sizes and packet inter-arrival times. All packets generated by the packet generators are passed to the flow generator.
Differences between UDP, TCP and possibly ICMP traffic are modeled by choosing appropriate distributions. For example UDP traffic is mainly dependent on the application that generates it (UDP doesn't have any flow or congestion control) and therefore produces quite different traffic than TCP.
The more complex the anomaly one wants to simulate is, the more complex the main script will be which creates all the configurations of the packet generators and spawns them.

```
1. for (t from  $start to $end) {
2.  select (dist(t) * $peak) #hosts from $hosts to $server:$port {
3.    insert constant rate flows between t and t+$time
4.    insert reverse flows between t+$rtt and t+$time+$rtt
5.  }
6. }
```

If we look at the above example from the last chapter (2.4.3) where a flash crowd was modeled using basic insertion actions, we can now describe it more precisely using these distributions:

Line 1: The start end end times of the anomaly are modeled using the distribution of active connections.

Line 2: The amount of active connections is maintained in an internal list in the generator and adheres to the distribution of active connections. The connections themselves are selected from the source/destination IPs and ports distributions. Same connections can appear again after a time picked from the inter-connection times distribution.

Line 3: This line stands for one basic packet generator. The start end end times of the session are picked from the session length distribution, the rate represents the inter-arrival times and packet sizes which are passed to the basic packet generator as distributions.

Line 4: The traffic in the opposite direction is directly created by the same basic packet generator using the roundtrip time and the packet size and inter-arrival time distributions for the reverse direction.

Having a traffic generator which can be fully configured to create various kind of traffic is one thing, but choosing reasonable distributions is a prerequisite to actually create realistic traffic. It is not part of the scope of this work to propose how these distributions are gained or chosen. This framework only provides the means of using and implementing them into the traffic generation.

3.3 Flow Generation

The flow generator receives settings to simulate an actual router. These settings include for example active and inactive timeouts. The Netflow generator maintains a hashtable of active connections. Arriving packets are added to an already active session or a new session is created. When a session times out (due to one of the timeout parameters) a netflow record is created in the internal format and emitted by the Netflow generator. Further details follow in chapters 4 and 5.

Chapter 4

Framework Design

This chapter describes the application design, design choices and problems encountered during the design phase. The first section shows an overview over the main components of the framework including a short description of each one and a description of the overall data flow. The following section describes how the components are decoupled from each other and how the inter process communication and data flow works inside the framework. The data format for the communication is defined in the last section of this chapter.

4.1 Overview

The anomaly injection framework consists of 5 major components which are very loosely coupled to allow maximum flexibility. Figure 4.1 shows these components in a sample data flow arrangement. The first component is the **NetflowReader**, it takes as input a file that contains a flow trace and outputs Netflow Packet in the internal format. From now on we call those packets NFPs. The next component on the upper data flow lane is the **NetflowDeleter**. It takes as input NFPs and deletes certain records according to its configuration. Starting in the lower lane is the TrafficGenerator which is actually a component split into various sub-components. The **TrafficGenerator** creates network traffic in the form of IP packet headers using **PacketGenerators**. These packet headers are passed to the **NetflowGenerator** which then outputs NFPs as usual. Two NFP streams can be merged into a new NFP stream by the **NetflowMerger**. The final component in this setup is the **NetflowWriter** which again takes NFPs and writes them back to disk as a netflow trace. Components can be freely rearranged according to their interfaces to handle different tasks.

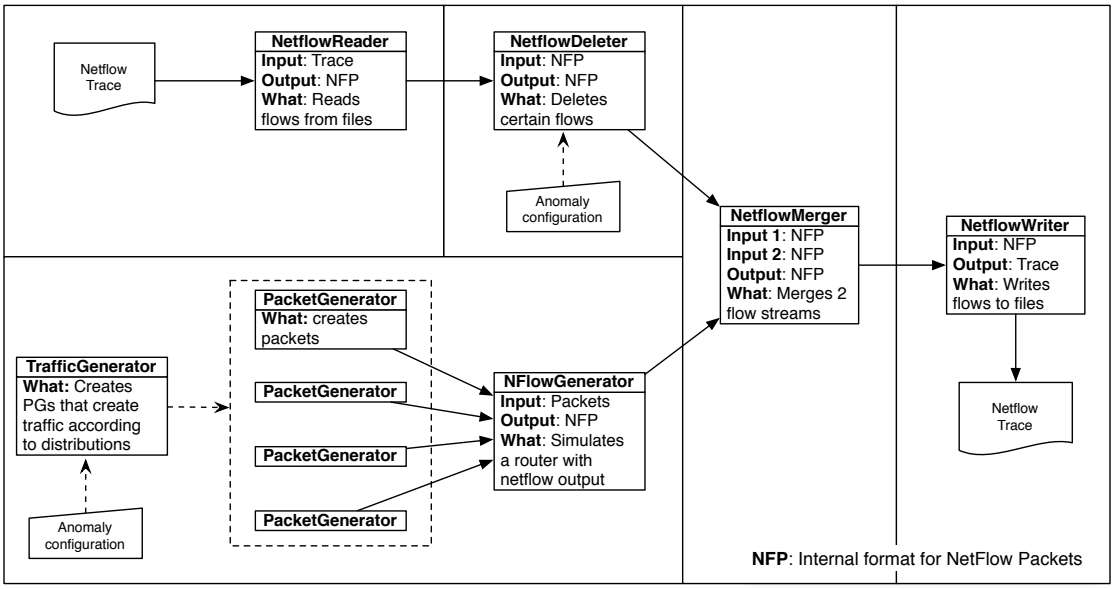

Figure 4.1: Framework Overview

4.2 Inter Process Communication

As previously mentioned a loose coupling allows for maximum flexibility and easy extensibility of the components and the framework as a whole. Therefore a format for IPC had to be defined which all components have to use for communication. For that reason each component implements interfaces to receive and send records in that defined format. In the C++ part of the framework this interface is implemented in the FlowForwarder class.
This approach allows us to rearrange the components in any order or insert new ones into the data flow. It furthermore enables the framework to work with components written in different programming languages or even scripting languages, since some languages are better fit for some tasks.
To setup a working data pipeline using components of the framework, a shell script is written which connects them to each other and starts the simulations.
Another characteristics of this approach is that the chain of components is processed in a top down approach. This means that data needed by a component is evaluated in a lazy way. The last component "requests" new data from the previous component in the chain which is waiting for that request. The request is propagated backwards through the component chain until every component has everything it needs. We almost automatically get this behavior by connecting the components using named pipes (the above statement is not completely accurate, since the pipe written to by the previous component has a buffer of a certain size which will be filled even if the data hasn't been read by the next component yet). Furthermore a certain degree of parallelism of the components is achieved by this approach since all components run as different processes and can run on different processors or processing cores.

4.3 Internal Netflow Packet Format

The format for the internal netflow packets is defined as follows:
First there is a header describing the packet and defining the number of records it contains. The header is followed by the amount of packets as described in the header. The data structure of the records as shown below are defined in the netflowvxplusplus framework and used unaltered in this project.
An NFP is at least 76 bytes long (the header + one record) and is 1468 bytes long when it contains 30 records.

```
struct NetflowHeader {
  // Header Version Field, must be 5, 9 or 254 (generic)
  uint16_t version;
  // Number of flows following this header or the number of flowsets if v9
  uint16_t count;
  // The time since the device was started, in milliseconds
  uint32_t sysUptime;
  // The current time of the device, in seconds since 1970
  uint32_t unixSecs;
  // Only if v5: Current time of the device, in nanoseconds since 1970
  uint32_t unixNSecs;
  union {
    // Only if version is 5: Sequence counter value of total flows seen
    uint32_t flowSequence;
    // Only if version is 9: Sequence number of this paket
    uint32_t packageSequence;
  };
  // Only if version is 5: Type of device
  uint8_t engineType;
  // Only if version is 5: Slot number of the flow-switching engine
  uint8_t engineId;
  // Only if version is 9: Device identification
  uint32_t sourceId;
};
```

Listing 4.1: NFR Header: 28 bytes

```
struct NetflowRecord {
  // Source IPV4 address
  uint32_t addr;
  // Destination IPV4 address
  uint32_t dstAddr;
  // Nexthop IPV4 address
  uint32_t nextHop;
  // SNMP input device index
  uint16_t input;
  // SNMP output device index
  uint16_t output;
  // Number of pakets in this flow
  uint32_t dPkts;
  // Number of bytes in this flow
  uint32_t dOctets;
  // Sysuptime value at start of this flow
  uint32_t first;
  // Sysuptime value at end of this flow
  uint32_t last;
  // Layer4 source port
  uint16_t port;
  // Layer4 destination port
  uint16_t dstPort;
  // Unused padding
  uint8_t pad1;
  // TCP flags
  uint8_t tcpFlags;
  // IP protocol number
  uint8_t prot;
  // Type of service
  uint8_t tos;
  // Source BGP autonomous system number
  uint16_t as;
  // Destination BGP autonomous system number
  uint16_t dstAs;
  // Source IPV4 mask (subnet mask in slash notation)
  uint8_t mask;
  // Destination IPV4 mask (subnet mask in slash notation)
  uint8_t dstMask;
  // Unused padding
  uint16_t pad2;
};
```

Listing 4.2: NFR Record: 48 bytes

Chapter 5

Implementation Details

This chapter contains implementation details of the components of the injection framework. It is important to know that the system is for testing purposes and therefore the code does not contain much error checking. If the components are not used as intended, unexpected behavior and errors may occur.

5.1 FlowForwarder

The FlowForwarder class is a helper class which defines interfaces which use named pipes to read and write NFPs (as defined in the last chapter). It has by default one input and one output pipe. The class is supposed to be inherited from by a child class to use either the input, output or both interfaces. The default run() function does a processing step after each read before it writes the records to the output pipe. The functions are defined using the virtual keyword by the FlowForwarder. This allows for a dynamic redefinition by the class inheriting from the FF to change the behavior. The functions of the runtime object are then chosen according to the dynamic binding principle. For example if the process() function is redefined by the child class, that function is used even when the parent class calls it in its run() function.

```
class FlowForwarder {
public:
  // Constructor _in: name of the input pipe, _out: name of the output pipe
  FlowForwarder(char* _in, char* _out);
  virtual ~FlowForwarder();

  // Start the processing loop of the flows
  virtual void run();

  // Things done after the loop
  virtual void afterRun();

  // Stop processing
  virtual void stop();

  // Helper function to print some record infos
  void print(NetflowRecord &rec, NetflowHeader &header);

protected:
  // Functions to read or write the next flow packet from the pipes
  virtual void readPacket();
  virtual void writePacket();

  // Function where the processing of the current packet is done
  virtual void process();
};
```

Listing 5.1: FlowForwarder Class Definition

5.2 NetflowReader

The NetflowReader reads Netflow traces from files. It outputs netflow packets in the internal format. The reader can read generic traces as well as v5 and v9 traces. The NetflowReader builds on the FlowForwarder and therefore has the same basic functions. The readRecord function is redefined since the NR no longer reads from a pipe, but from a file. In addition the NR uses a NetflowInput class from the NetflowVxPlusPlus package.

Important: All values are read in network byte-order and converted to host byte-order. Internally all components work with values in host byte-order.

Usage: ./NetflowReader inFile outPipe

```
// Inherits from FlowForwarder
class NetflowReader: public FlowForwarder {

public:
  // Constructor _in: name of the input file, _out: name of the output pipe
  NetflowReader(char* _in, char* _out);

private:
  // Redefines readPacket to read from file
  void readPacket();

  // use NetflowInput to read from a file
  NetflowVxPlusPlus::NetflowInput* reader_;
};
```

Listing 5.2: NetflowReader Class Definition

5.3 NetflowWriter

The NetflowWriter takes as input NFPs from a named pipe and writes them to disk. Currently the only format supported for writing by the NetflowVxPlusPlus packet is v5. Like the NR the NW builds on the FlowForwarder and has the same basic functions. It redefines the writeRecord function to write to the specified file instead of the pipe. The NetflowOutput2 class of the NetflowVxPlusPlus package is used for the actual writing to the file. If the output filename has a .bz2 oder .gz extension the output is automatically compressed in that format.

Important: All values are written in network byte-order.

Usage: ./NetflowWriter inPipe outFile

```
// Inherits from FlowForwarder
class NetflowWriter: public FlowForwarder {

public:
  // Constructor _in: name of the input pipe, _out: name of the output file
  NetflowWriter(char* _in, char* _out);
  ~NetflowWriter();

private:
  // Redefines writeRecord to write to file
  void writePacket();

  // use NetflowOutput2 to write to a file
  NetflowVxPlusPlus::NetflowOutput2* writer_;
};
```

Listing 5.3: NetflowWriter Class Definition

5.4 NetflowMerger

The NetflowMerger takes as input two NFP streams as sources and outputs the records to one new NFP stream. The inputs can be from any component implementing the flow forward interface. The NME checks if both streams are compatible by comparing the first header versions. To be able to read from 2 different pipes, the NME extends the FlowForwarder and implements an additional read and write function as well as a new run function which calls the correct writer and reader for the next record. Internally there is alway one record cached from each input stream. The record with the smaller time stamp gets written to the output pipe and a new record will be fetched from the according input stream. The comparison of the time stamps first compares the seconds stamp and when they match compares the nanoseconds stamp to decide which record has to be written first. The sequence number is also reset for both streams to an internal counter starting at the value 1 to create a correctly numbered flow packet stream.

Call: ./NetflowMerger inPipe1 inPipe2 outPipe

```
// Inherits from FlowForwarder
class NetflowMerger: public FlowForwarder {

public:
  // Constructor _in1: name of the first input pipe,
  // _in2: name of the second input pipe _out: name of the output pipe
  NetflowMerger(char* _in1, char* _in2, char* _out);

  // Redefines the run function to call the correct reader/writer
  void run();

private:
  // implements an additional read function to read from the 2nd pipe
  void readPacket2();

  // Implements an additional write function to write data from the 2nd pipe
  void writePacket2();
};
```

Listing 5.4: NetflowMerger Class Definition

5.5 NetflowDeleter

```
// Inherits from FlowForwarder
class NetflowDeleter: public FlowForwarder {

public:
  // Constructor:   _script: name of the python script
  NetflowDeleter(char* _in, char* _out, char* _script);
  ~NetflowDeleter();

  // Redefined FF functions
  void writePacket();
  void afterRun();
  void process();

  // function with the delete decision
  virtual void del();

  // Set current record/packet to be deleted
  void delRec();
  void delPacket();

  // Functions to access current record
  uint32_t get_addr();
  uint32_t get_dstAddr();
  uint32_t get_nextHop();
  uint16_t get_input();
  uint16_t get_output();
  uint32_t get_dPkts();
  uint32_t get_dOctets();
  uint32_t get_first();
  uint32_t get_last();
  uint16_t get_port();
  uint16_t get_dstPort();
  uint8_t  get_tcpFlags();
  uint8_t  get_prot();
  uint8_t  get_tos();
  uint16_t get_as();
  uint16_t get_dstAs();
  uint8_t  get_mask();
  uint8_t  get_dstMask();

  // Function to access header time in milliseconds
  uint64_t get_headerTime();

  // Helper function for current record time (calculated using header)
  // Return miliseconds since 1970 (unixtime * 1000)
  uint32_t get_start();
  uint32_t get_end();
  uint32_t get_dur();

  // Helper functions to access IPs from records
  uint8_t get_addr1();
  uint8_t get_addr2();
  uint8_t get_addr3();
  uint8_t get_addr4();
  uint8_t get_dstAddr1();
  uint8_t get_dstAddr2();
  uint8_t get_dstAddr3();
  uint8_t get_dstAddr4();
};
```

Listing 5.5: NetflowDeleter Class Definition

The NetflowDeleter gets a NFPs and deletes records from it using a python script. The script extends the NetflowDeleter with a new function to decide if the current record will be deleted. It outputs the modified NFPs. To still have a correct sequence number all packets get new sequence numbers counted by an internal counter which starts at 1. If all records in a packet are deleted, the deleter doesn't write anything and continues with the next packet.

The loaded python script has to define a class called **Deleter** which inherits from the NetflowDeleter class and has to define a function called **delete**. The currently active record in the NetflowDeleter can be accessed through getter methods and the record can be deleted by calling the method **self.delRec()**. The following 2 python script code samples show examples on how to access the fields as well as how to delete the according record.

```
class Deleter(NetflowDeleter):
    def delete(self):
        if self.get_addr1() == 129:
            self.delRec()
        return
```

Listing 5.6: Sample python script to delete records having source address starting with 129

```
class Deleter(NetflowDeleter):
    def delete(self):
        if (self.get_start() > 1189385993872) and
           (self.get_end() < 1189385994128):
            self.delRec()
        return
```

Listing 5.7: Sample python script to delete records in a specific time intervall

```
class Deleter(NetflowDeleter):
    def delete(self):
        if (self.get_headerTime() > 1188196352000) and
           (self.get_headerTime() < 1188196805000):
            self.delPacket()
        return
```

Listing 5.8: Sample python script to delete whole flow packets using header timestamp

The function **delete** is called by the derived python class which inherits from the NetflowDeleter class. For a more detailed description of the approach see 5.10.2.

Important: All time values returned are in miliseconds since 1970 (Unixtime * 1000). One needs to convert time values to that format before putting them into the script.
All IP addresses given as 32 bit integers are in host byte-order.

Usage: ./NetflowDeleter inPipe outPipe script.py

5.6 PacketGenerator

The PacketGenerator is responsible for the simulation of actual network traffic. It uses settings and distributions passed to it to create that traffic. The PacketGenerator outputs packet headers as defined below. The packet headers are collected by the NetflowGenerator and converted to NFPs for further processing.

```
struct GenericPacket {
  // Source Address
  uint32_t srcAddr;
  // Destination Address
  uint32_t dstAddr;
  // Protocoll
  uint8_t prot;
  // Total header length in bytes
  uint16_t len;
  // Source port
  uint16_t srcPort;
  // Destination port
  uint16_t dstPort;
  // TCP flags
  uint8_t flags;
  // ICMP Type
  uint8_t icmpType;
  // ICMP Code
  uint8_t icmpCode;

  // Load in bytes
  uint32_t load;
  // Timestamp in micro seconds since 1970
  uint64_t time;
};
```

Listing 5.9: GenericPacket Struct

There are 2 implemented PacketGenerator classes, one is for unidirectional traffic and the other for bidirectional traffic. The correct class gets automatically chosen by parsing a configuration file which which has to be passed by the main function of the PacketGenerator. To create a bidirectional class object, some additional parameters need to be set: a round trip time and a packet size in the opposite direction. In the bidirectional case for every created packet in the main direction a packet in the other direction is also created (switched source/destination IP and port plus another packet size). To create dynamic values for any field one can enter python scripts instead of values in the config file. The PacketGenerator then creates objects for all classes passed in the scripts. The objects all get individual names and are all in the same python namespace. This namespace is passed to the PacketGenerator objects to allow the extraction of the values in the according functions. The PGs internally use function pointers which either point to the function returning static values or calling the python objects in the passed python namespace. The following attributes are needed to create a PacketGenerator (the values in brackets are optional if you want a bidirectional PacketGenerator):

- Source Address	- Start Time	- Packet Size
- Destination Address	- End Time	- (Round Trip Time)
- Source Port	- Protocol	- (Packet Size Back)
- Destination Port	- Inter Arrival Time	

All values are set in a config file. Instead of fixed values one can pass python scripts for all the values except the start time and the end time.

```
class PacketGenerator {
public:
  // Constructor & Destructor
  PacketGenerator(char* _out, uint32_t _prot, uint32_t _sIP, uint32_t _dIP,
    uint32_t _sPort, uint32_t _dPort, uint64_t _start, uint64_t _end,
    uint32_t _pSize, float _IAT);
  virtual ~PacketGenerator();

  // Start creating packets
  virtual void run();

  // Pass a python namespace to get values from
  void setPyNamespace(object* _ns);

  // Set to use python function instead of static value
  void setPySourceIP();
  void setPyDestinationIP();
  void setPySourcePort();
  void setPyDestinationPort();
  void setPyProtocol();
  void setPyInterArrivalTime();
  void setPyPacketSize();

  // Print statistics
  void printStats();

protected:
  // Functions to access the static values
  uint32_t getSourceIP();
  uint32_t getDestinationIP();
  uint32_t getSourcePort();
  uint32_t getDestinationPort();
  uint32_t getProtocol();
  uint32_t getInterArrivalTime();
  uint32_t getPacketSize();

  // Functions to access the values from python
  uint32_t getPySourceIP();
  uint32_t getPyDestinationIP();
  uint32_t getPySourcePort();
  uint32_t getPyDestinationPort();
  uint32_t getPyProtocol();
  uint32_t getPyInterArrivalTime();
  uint32_t getPyPacketSize();
};
```

Listing 5.10: Unidirectional PacketGenerator Class

```
class BiPacketGenerator: public PacketGenerator {
public:
  // Constructor
  BiPacketGenerator(char* _out, uint32_t _prot, uint32_t _sIP, uint32_t _dIP,
    uint32_t _sPort, uint32_t _dPort, uint64_t _start, uint64_t _end,
    uint32_t _pSize, float _IAT, float _rtt, uint32_t _pSize2);

  void run();

  // Set to use python function instead of static value
  void setPyRoundTripTime();
  void setPyPacketSizeBack();

private:
  // Helper functions
  void deQueue();
  void writePacket(GenericPacket* _p);
};
```

Listing 5.11: Bidirectional PacketGenerator Class

Important:

- It is important that created packets always increase time to their previous packet (using the inter arrival time) if this is not the case there may be unexpected output from the flow generator.
- The PacketGenerator (as well as the NetFlowGenerator) internally works using micro seconds time values. But all values passed via the config or the python functions are in miliseconds!

Limitations:

- Currently the PacketGenerator does not support the following fields: icmpType, icmpCode.
- It is currently not possible to create answer packets only for some packets. For this reason one can use the PythonPacketGenerator or use the workaround to delete the according flows with a deleter.

Usage: ./PacketGenerator outPipe config

5.6.1 Configuration

As mentioned before the PacketGenerator is configured via a configuration script. This section describes the fields of the configuration file and gives some samples for python scripts.

```
SourceIP = 244.12.1.12
DestinationIP = 172.34.12.165
SourcePort = 423
DestinationPort = ../scripts/packetgenerator/portscan.py
Protocol = tcp
StartTime = 100
EndTime = 5000000
InterArrivalTime = 10.0
PacketSize = 4540
RoundTripTime = 7.8
PacketSizeBack = 10
```

Listing 5.12: Sample Config File

The sample configuration file contains almost all possible fields for the packet values. There is only one additional field which is called **StartScript** which sets the name of a python script that executes some needed includes or other settings. For example global variables could be set which can then be accessed by all other python classes set in the other fields of the configuration file. The StartScript is executed prior to all other scripts.

A python script to replace a value in the config has to fulfill some definitions to work:

1. The script has to define a class called PacketGenerator
2. The class has to have at least a function called getXxx, where Xxx is the name of the configuration field it represents. The function takes as argument a variable self which is used to access other members.
3. The scripts for SourcePort, DestinationPort, Protocol and PacketSize have to return integer values.
4. The scripts for SourceIP and DestinationIP have to return 32bit integer values (IPs have to be converted prior to using them since a conversion for every packet would be too computationally expensive).
5. All time values are in miliseconds and in case of the InterArrivalTime and RoundTripTime have to be floating point values.
6. The InterArrivalTime script or value is read first for each step. If a global counter or so it to be used the scripts are called in the following order: InterArrivalTime, SourceIP, DestinationIP, SourcePort, DestinationPort, Protocol, PacketSize, (RoundTripTime), (PacketSizeBack).

```
class PacketGenerator:
    port = 1023
    def getDestinationPort(self):
        self.port = self.port + 1
        if self.port <= 65535:
            return self.port
        else:
            global stop
            stop = True
            return 0
```

Listing 5.13: Sample python script for DestinationPort: portscan.py

The python script may abort the generation of packets by setting the global variable **stop** = True. This global variable gets tested after the creation of every packet in the PacketGenerator classes.
It is also possible to pass values between different classes by using global variables or other means, since the objects all reside in the same python namespace.

5.7 PythonPacketGenerator

The PythonPacketGenerator is a more flexibel implementation of a PacketGenerator, which can be configured much more freely but is therefore more complicated to configure. It basically is just a skeleton to send correct packet headers to the NetflowGenerator, most of the logic has to be implemented in a Python class by the user. The PythonPacketGenerator only ensures that packets in both directions are sent in the correct order and that they have the correct format.

The PythonPacketGenerator is extended by a Python class which has to implement the **request** and **requestBack** methods... those methods are called in the main loop in run. The request and requestBack methods have to call the **send** and **senBack** methods to actually send packets... If no packet is sent anymore when calling request the main loop stops and the PythonPacketGenerator halts. A sample implementation of a Python class extending the PythongPacketGenerator is given on the next page.

Important: It is important that there is some kind of abort condition that will be met in the request method implemented in python, since the C++ part will keep running until there is no more packet after the request method was called. In the example on the next page this condition is simply the reaching of a certain time value.

For the NetflowGenerator not to abort due to packets arriving in the wrong order, the user has to make sure that consecutive packets always increase their time values. This doesn't have to be true for the answer packets, those get sorted automatically in the right order using a priority queue in the C++ part of the class. The only condition for answer packets is of course that it has to arrive after its request packet.

All time values used in the PythonPacketGenerator are in micro seconds! This is due to the fact that internally the NetflowGenerator as well as the normal PacketGenerator work using micro seconds.

All IP addresses given as 32 bit integers are in host byte-order.

Usage: ./PythonPackGenerator outPipe script.py

```
class PythonPacketGenerator {
public:
  // Constructor & Destructor
  PythonPacketGenerator(char* _out);
  virtual ~PythonPacketGenerator();

  // Start the main processing loop
  void run();

  // Send a packet
  void send(  uint32_t _srcAddr, uint32_t _dstAddr, uint32_t _prot,
              uint32_t _srcPort, uint32_t _dstPort, uint32_t _load,
              uint64_t time, uint8_t flags );

  // Send an answer packet
  void sendBack(  uint32_t _srcAddr, uint32_t _dstAddr, uint32_t _prot,
                  uint32_t _srcPort, uint32_t _dstPort, uint32_t _load,
                  uint64_t time, uint8_t flags);

  // Request a packet (Implemented in Python)
  virtual void request();

  // Request an answer packet (Implemented in Python)
  virtual void requestBack();


private:
  void deQueue();
  void writePacket(GenericPacket* _p);
};
```

Listing 5.14: PythonPacketGenerator Class definition

```
import random

class PacketGenerator(PythonPacketGenerator):
    # Some Variables (time values are in micro seconds)

    # Daniela's IP as int
    srcIP = xxx #(censored)
    srcPort = 423
    dstPort = 563
    start = 1188196352 * 1000000
    end = 1188196385 * 1000000
    iat = int(0.01 * 1000)
    size = 20
    rtt = int(7.8 * 1000)
    sizeBack = 10
    flags = 2
    flagsBack = 18

    time = start
    timeBack = start + rtt

    # Read addresses to be scanned from a file
    f = open('unique_addresses', 'r')
    dstIP = int(f.readline())

    # Send a scan packet when requested by the c++ part
    def request (self):
        if self.time >= self.start and self.time < self.end:
            self.send(self.srcIP, self.dstIP, 6, self.srcPort, self.dstPort,
                      self.size, self.time, self.flags)

    # Choose an answer packet randomly, either SYN/ACK, RST or ICMP
    def requestBack (self):
        ran = random.randint(1,3)
        if ran == 1:
            self.sendBack(self.dstIP, self.srcIP, 6, self.dstPort,
                          self.srcPort, self.sizeBack, self.timeBack, 18)
        elif ran == 2:
            self.sendBack(self.dstIP, self.srcIP, 6, self.dstPort,
                          self.srcPort, self.sizeBack, self.timeBack, 4)
        elif ran == 3:
            self.sendBack(self.dstIP, self.srcIP, 1, 0, 0, self.sizeBack,
                          self.timeBack, 0)
        self.next()

    # Prepare the values for the next packet
    def next (self):
        self.time += self.iat
        self.timeBack = self.time + self.rtt
        self.dstIP = int(self.f.readline())
```

Listing 5.15: Implementation of a host scan using the PythonPacketGenerator

5.8 NetflowGenerator

The NetflowGenerator is the class that simulates the router. It receive packets from a list of PacketGenerators and assigns them to currently running flows or creates new ones. It keeps track of the actives flows in a hashtable. Flows can either timeout due to an active timeout which states the maximum length a flow can be active before it gets split in another flow (the old one is passed to the outpu) or a flow can timeout after some inactivity. Therefore the NetflowGenerator periodically checks all active flows and compares their time of the last update to the current router time and forwards flows to the ouput if the flow was inactive for a certain time. When there are no more packets arriving (all PacketGenerators have stopped sending packets) the NetflowGenerator flushes the hash table and outputs all remaining flows. The NetflowGenerator takes a configuration file as second argument which sets some fields in the flow packet header.

Usage: ./NetflowGenerator outPipe config [PG inPipes]+

```
class NetflowGenerator: public FlowForwarder {
public:
  // Constructor: takes outPipe name and a vector of input pipe names
  // to receive packets from PacketGenerators
  NetflowGenerator(char* _outPipe, vector<string> _pipes);
  ~NetflowGenerator();

  // Run the main logic
  void run();

  // Change router or flow settings
  void routerSettings(RouterSettings _s);
  void flowSettings(FlowSettings _f);

private:
  // Process a packet from a PacketGenerator
  void processPacket(PacketContainer* _pc);

  // Output an entry from the hash table
  void outputEntry(FlowtableEntry* _entry);

  // Output the NFP packet to the outPipe
  void outputPacket();

  // Check for inactive timeouts
  void inactiveCheck();

  // Flush the hash table (clean up at the end)
  void flushTable();
};
```

Listing 5.16: NetflowGenerator Class Definition

5.8.1 Configuration

The configuration file is mandatory and the 4 fields can be defined and should match the field of the router where you're background traffic traces come from (when merged). The fields are: Version, EngineId, EngineType and SourceId (If settings get left away they are set to 0 except Version which is set to 5 by default.

```
Version = 5
EngineId = 6
EngineType = 0
SourceId = 0
```

Listing 5.17: NetflowGenerator Sample Config File

5.8.2 Remarks

It has to be noted that the NetflowGenerator is a very basic router implementation. It misses for example the ability to dynamically set certain fields of the flow like "as", "mask", "tos", "nextHop". Furthermore does the NG not act upon any received TCP flags (like FIN, RST), but handles everything with the 2 timeouts (which are the same for all protocolls).

We didn't use already implemented functionality from other tools for the reasons described in the following. The main reasons are that the performance of the outcoming solution would have been kind of bad and the implementation cost wouldn't have been smaller than when the implementation was done from scratch. The reason for this is that in order to achieve the following data chain: [PacketGenerators] –> [NetflowGenerator] –> NFP; we would have had to build a chain that looks something like this: [PacketGenerators] –> [PacketCollector –> Converter to used format] –> [Existing Router Simulator] –> [NetflowReader] –> NFP; which is clearly less effective due to various data format conversions.

5.9 Other Components

Additionally to the 5 main components there are some helper components which can be used to analyze or test existing or new components. First there is the **NetflowSplitter** which is the opposite of the NetflowMerger and was basically created to test the latter. It takes one input pipe and two output pipes and randomly writes the current NFP to one of those output pipes. The other component, the **NetflowHelper** has no fixed purpose and can very easily be changed and recompiled as it contains almost no code. It is used to do things like printing every NFP on screen and waiting for a key press by the user before it continues with the next packet. The NetflowHelper is also a good starting point if a new component needs to be written.

5.10 Encountered Problems

5.10.1 Inter Process Communication - Internal Packet Format

The NetflowVxPlusPlus package provides the structs as described in 4.3. What is not mentioned is that a NetflowRecord also has place for a destructor. This lead to some problems when passing whole structs instances between components, since the destructor addresses were passed along and consequently the objects containing the NetflowRecords couldn't be deleted properly. To solve the problem, the records sent between the components skip the destructor when writing and reading and start at the address of the first field of the struct (because the destructor reference is at the beginning of the struct).

```
// Calculate how much bytes have to be skipped to exclude the destructor
// Workaround in order to work on both 32 and 64 bit systems
uint64_t skip = ((uint64_t)&(records_[0].addr) - (uint64_t)&(records_[0]));
recordSize_ = sizeof(records_[0]) - skip;

// Extract from the readPacket function
if (read(inPipe_, &(records_[i].addr), recordSize_) <= 0) {
```

Listing 5.18: Code from FlowForwarder.cpp

5.10.2 NetflowDeleter

The first version of the NetflowDeleter used to execute the python script internally and recreated a python object each time the delete decision had to be made. Therefore the whole Deleter became slower by a factor of about 10. The new and correct approach was to create a derived python class which inherits from the NetflowDeleter class and is then instantiated in python and afterwards started as a casted NetflowDeleter object in C++. The trick to achieve a correct polymorphism between C++ and python was to write a wrapper class which enables one to replace the C++ del() function with a python function. This is necessary because the del() function is called from within the C++ code. This approach was then also used for the implementation of the PythonPacketGenerator and a very similar one for the implementation of the PacketGenerator.

```
struct NetflowDeleterWrap : NetflowDeleter
{
  NetflowDeleterWrap(PyObject* self_, char* in_, char* out_)
          : NetflowDeleter(in_, out_), self(self_) {}
      void del() { call_method<void>(self, "delete"); }
      PyObject* self;
};
```

Listing 5.19: Code from NetflowDeleter.cpp: Wrapper Class

Chapter 6

Evaluation

The main purpose of the evaluation was to test the functionality of the framework as e whole and of all the individual components. Therefore three different kind of anomalies were modeled using the flame implementation and injected into real flow traces. The modeling of the anomalies is done as described in the flame paper [3].
At the end of the chapter performance, limitations and possible artifacts are discussed.

6.1 Subtractive Anomaly: Loss event

I tried to simulate a loss event using a real event as template. The original loss event occurred due to an outage of the NetFlow packet collector. The simulated loss event was injected 24 hours before the original event to get conditions as similar as possible.
As a result of the packet collector outage, the 1 hour trace packet running at that time got cut off. After its re-initialization the packet collector skipped the next sequence number (for the next 1 hour trace file). Therefore we observe an apparently missing file in our sequence of one hour trace files. The time of the outage was calculated to be approximately 453 seconds (time stamp of the first packet after the outage minus time stamp of the last packet before the outage). The figures on the following pages show the UDP flows count plots of the original event as well as the injected loss. The plotted interval length is different due to the shorter first data packet in the original event. Other differences can be explained by the different time of the original and the injected event (24 hours). On the plot of the simulated loss the solid line represents the injection and the dotted line the original data.

6.1.1 Configuration: Loss Event

The following two boxes show the main script which is run for the loss injection as well as the deleter script which is used to delete the records in the desired time frame. The configuration is very simple and all it does is delete all flows in the specified intervall.

```
#!/bin/sh

DIR="./Release"
DATA="xxx"  #(censored)
OUT="loss91_1.dat.bz2"

# Create pipes
mkfifo pipe1
mkfifo pipe2

$DIR/NetflowReader $DATA pipe1 &
$DIR/NetflowDeleter pipe1 pipe2 loss.py &
$DIR/NetflowWriter pipe2 $OUT
```

Listing 6.1: Script for running the Loss 'Injection'

```
class Deleter(NetflowDeleter):
    def delete(self):
        if (self.get_headerTime() > 1188196352000) and
           (self.get_headerTime() < 1188196805000):
            self.delPacket()
        return
```

Listing 6.2: Deleter Script for the Loss Event

6.1.2 Plots: Loss Even

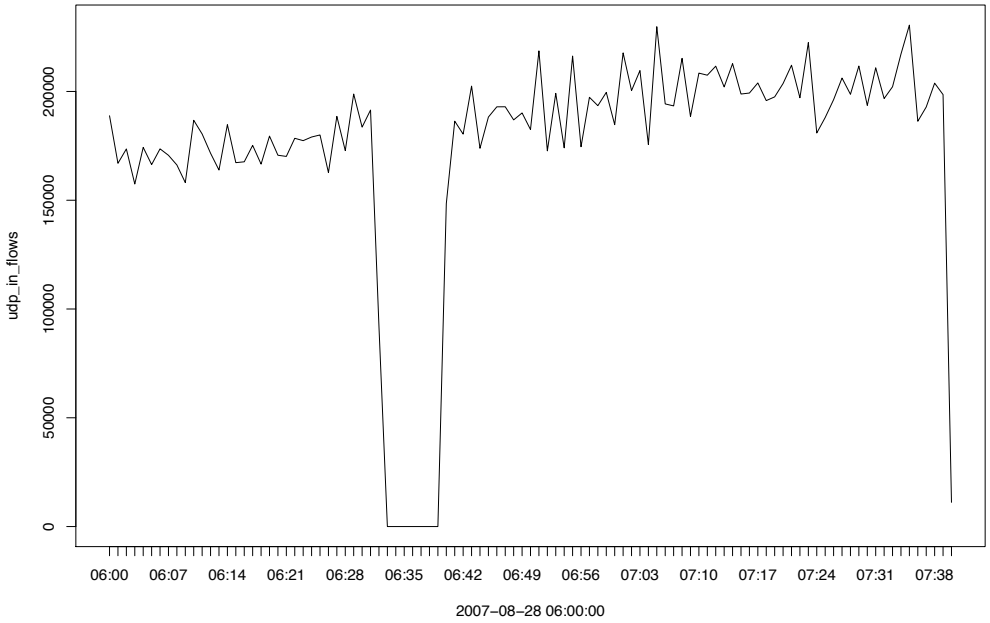

Figure 6.1: UDP Flows count of the original loss event

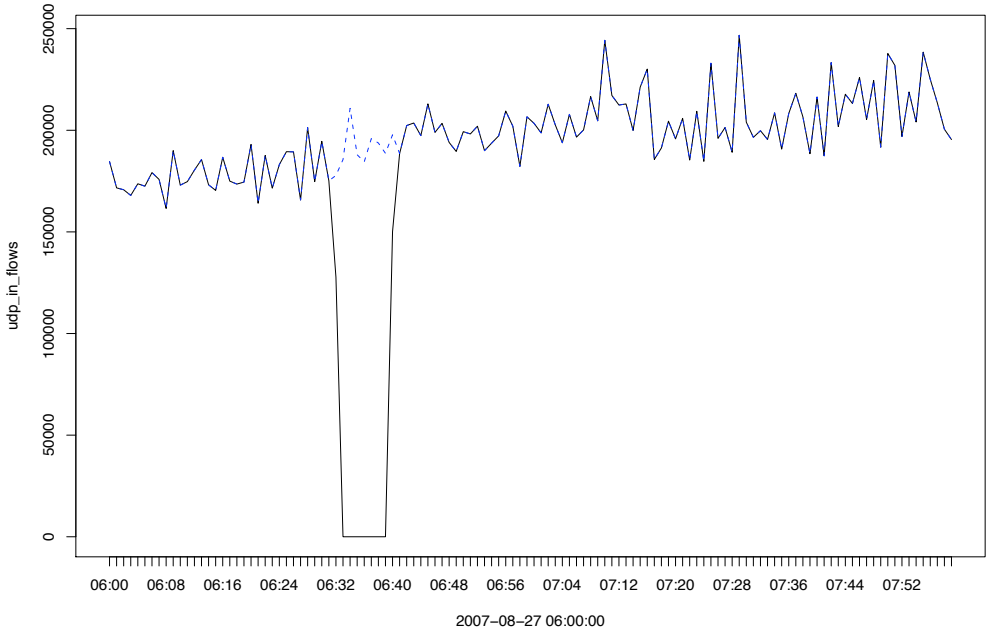

Figure 6.2: UDP Flows count with (solid line) and without Injection

6.2 Additive Anomaly: TCP SYN Host Scan

The host scan is a simulated anomaly where one "attacking" host scans a range of IP addresses to determine if the hosts are alive and how they respond. The scan packets are TCP SYN packets. In our simulation there is always an answer seen on our router which will either be a TCP SYN/ACK, RST packet or an ICMP destination unreachable packet.

6.2.1 Configuration: Host Scan

To keep the simulation as simple as possible one of the three possible answer packets is chosen randomly (SYN/ACK, RST, ICMP destination unreachable). The inter arrival time was set to a constant 0.01 ms and represents the scanning rate of the scanning host. The round trip time was also set to a constant value as we think for this test it wouldn't make much of a difference. The packet size in both directions was set to 40 bytes (only headers). The IP address range that was "scanned" was chosen arbitrarily as 100.0.0.0 - 100.255.255.255.
The following boxes show the bash script which is used to set up and start the simulation as well as the configuration/implementation of the PacketGenerator used.

```
#!/bin/sh

DIR="Release"
DATA="xxx"    #(censored)
OUT="injected_001ms.dat.bz2"

# Create pipes
mkfifo pipe1
mkfifo pipe2
mkfifo pipe3
mkfifo pipe4

$DIR/PythonPacketGenerator pipe1 pypg.py &
$DIR/NetflowGenerator pipe2 ng.conf pipe1 &
$DIR/NetflowReader $DATA pipe3 &
$DIR/NetflowMerger pipe2 pipe3 pipe4 &
$DIR/NetflowWriter pipe4 $OUT
```

Listing 6.3: Bash Script for the Host Scan

```
import random

class PacketGenerator(PythonPacketGenerator):
    # Some Variables (time values are in micro seconds)

    # Daniela's IP as int
    srcIP = xxx #(censored)
    srcPort = 4231
    dstPort = 145
    start = 1188196352 * 1000000
    end = 1188196385 * 1000000
    iat = int(0.01 * 1000)       #Inter Arrival Time => Scan Rate
    size = 40
    rtt = int(7.8 * 1000)
    sizeBack = 40
    flags = 2

    time = start
    timeBack = start + rtt

    # Start with 100.0.0.0
    dstIP = 1677721600

    # Send a scan packet when requested by the c++ part
    def request (self):
        # Check if IP in interval and smaller than 100.255.255.255
        if self.time >= self.start and self.time < self.end
        and self.dstIP <= 1694498815:
            self.send(self.srcIP, self.dstIP, 6, self.srcPort, self.dstPort,
                      self.size, self.time, self.flags)

    # Choose an answer packet randomly, either SYN/ACK, RST or ICMP
    def requestBack (self):
        ran = random.randint(1,3)
        if ran == 1:
            self.sendBack(self.dstIP, self.srcIP, 6, self.dstPort,
                          self.srcPort, self.sizeBack, self.timeBack, 18)
        elif ran == 2:
            self.sendBack(self.dstIP, self.srcIP, 6, self.dstPort,
                          self.srcPort, self.sizeBack, self.timeBack, 4)
        elif ran == 3:
            self.sendBack(self.dstIP, self.srcIP, 1, 0, 0, self.sizeBack,
                          self.timeBack, 0)
        self.next()

    # Prepare the values for the next packet
    def next (self):
        self.time += self.iat
        self.timeBack = self.time + self.rtt
        self.dstIP += 1
```

Listing 6.4: Host Scan Python Script for the PythonPacketGenerator

6.2.2 Plots: Host Scan

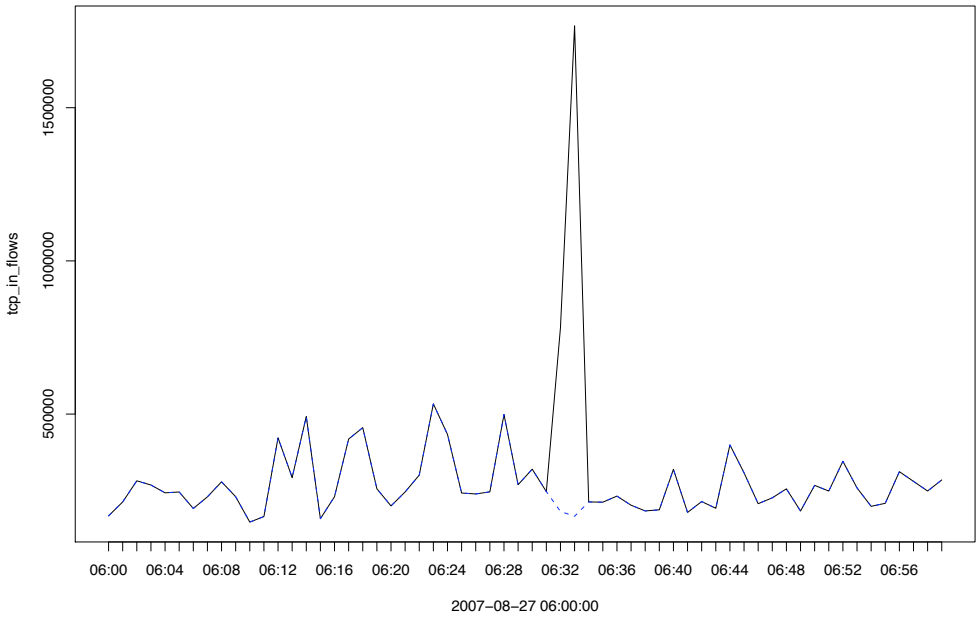

Figure 6.3: Outgoing Flows Count: with (solid line) and without Scan Injection

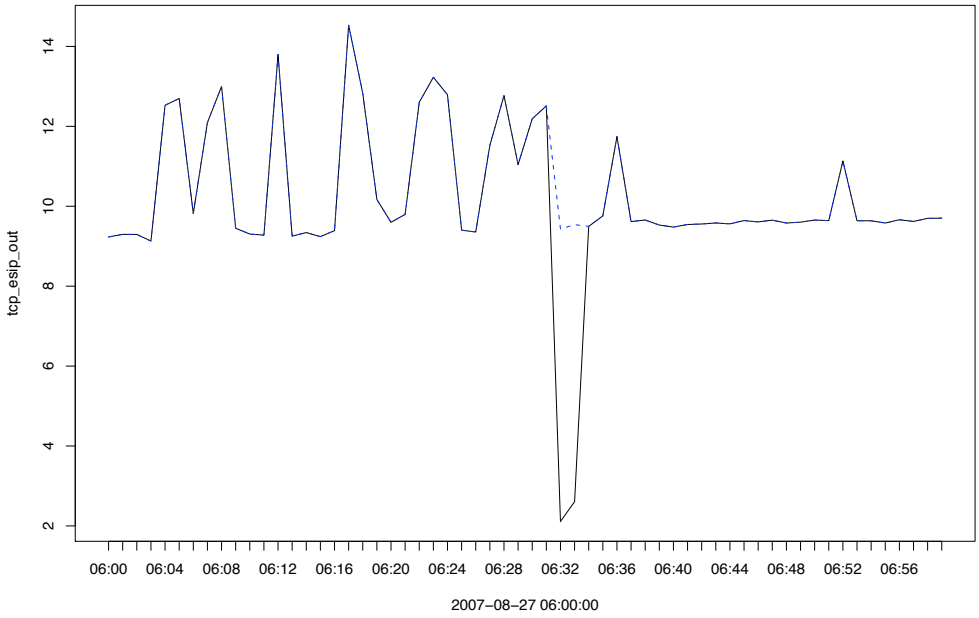

Figure 6.4: Destination IP Entropy: with (solid line) and without Scan Injection

6.3 Interactive Anomaly: TCP SYN flooding DoS Attack

The DOS attack is an anomaly simulated which uses all the components of the framework as illustrated in chapter 4.1. Flows are not only created for the scan event, but a loss of existing flows originating from the victim host is also simulated using the deleter. The overall configuration is very similar to the previous event (host scan). The attacker address is now the address of the victim which is attacked by an unknown host that spoofs the source address. As expected when we plot the event it looks practically identical to the host scan, since there's also answers created in the other direction. The answers correspond to the scan in the last event. And the TCP/SYN packets represent the responses from the last event. The only differences in the plot are noticed in the ICMP section, since in this event there are no ICMP packets created. If we however start to set the source address to a fixed address (no spoofing) the event already becomes less visible as more and more packets fall into the same flows. If we then also set the source port to a fixed value the event almost disappears from the plots (we only notice a slight increase in the packet TCP in and out packet count which is even smaller than other normal peaks). An event like this will be hard to detect.
The plots show the injection for spoofed source IP addresses and random source ports. We noticed that the plot for the TCP out flows count looks almost identical to the previous host scan injection. This is because this time the victim was inside the observed AS and all answer TCP packets went outside. Whereas before the attacker was inside and the victims were outside the observed AS and the attack. So previously the peak was due to the attack packets and this time it's due to the answer packets.

6.3.1 Configuration: DoS flooding Attack

```
!/bin/sh

DIR="./Release"
DATA="xxx" #(censored)
OUT="injected_dos_001ms.dat"

# Create pipes
mkfifo pipe1
mkfifo pipe2
mkfifo pipe3
mkfifo pipe4
mkfifo pipe5

$DIR/PythonPacketGenerator pipe1 pypg.py &
$DIR/NetflowGenerator pipe2 ng.conf pipe1 &
$DIR/NetflowReader $DATA pipe3 &
$DIR/NetflowDeleter pipe3 pipe4 delete.py &
$DIR/NetflowMerger pipe2 pipe4 pipe5 &
$DIR/NetflowWriter pipe5 $OUT
```

Listing 6.5: Bash Script for the TCP/SYN Flood

```
import random

class PacketGenerator(PythonPacketGenerator):
    # Daniela's IP as int
    dstIP = xxx #(censored)
    srcPort = 1025
    dstPort = 80
    start = 1188196352 * 1000000
    end = 1188196385 * 1000000
    iat = int(0.01 * 1000)      #Inter Arrival Time => Scan Rate
    size = 20
    rtt = int(7.8 * 1000)
    sizeBack = 10
    flags = 2

    time = start
    timeBack = start + rtt

    # Probability answer gets lost
    prob = 0.2

    # Read addresses to be scanned from a file
    f = open('unique_addresses', 'r')
    srcIP = int(f.readline())

    # Send a scan packet when requested by the c++ part
    def request (self):
        if self.time >= self.start and self.time < self.end:
            self.send(self.srcIP, self.dstIP, 6, self.srcPort, self.dstPort,
                      self.size, self.time, self.flags)

    # Send TCP SYN/ACK with prob 1-prob
    def requestBack (self):
        ran = random.random()
        if ran >= self.prob:
            self.sendBack(self.dstIP, self.srcIP, 6, self.dstPort,
                          self.srcPort, self.sizeBack, self.timeBack, 16)
        self.next()

    # Prepare the values for the next packet
    def next (self):
        self.time += self.iat
        self.timeBack = self.time + self.rtt
        self.timeBack += int(random.randint(0,40)*100)
        self.srcIP = int(self.f.readline())
        self.srcPort = random.randint(1025,47000)
```

Listing 6.6: DoS Python Script for the PythonPacketGenerator

```
import random

class Deleter(NetflowDeleter):
    matches = 0
    deleted = 0
    # Daniela's IP as int
    srcIP = xxx #(censored)
    start = 1188196352000
    end = 1188196385000
    prob = 0.2

    def delete(self):
        # If IP matches and in correct time interval...
        if (self.get_addr() == self.srcIP and self.get_start() >= self.start
            and self.get_end() <= self.end):
            self.matches += 1
            ran = random.random()
            if ran <= self.prob:
                self.deleted += 1
                self.delRec()
        return
```

Listing 6.7: DoS Python Script for the NetflowDeleter

6.3.2 Plots: TCP SYN DOS Attack

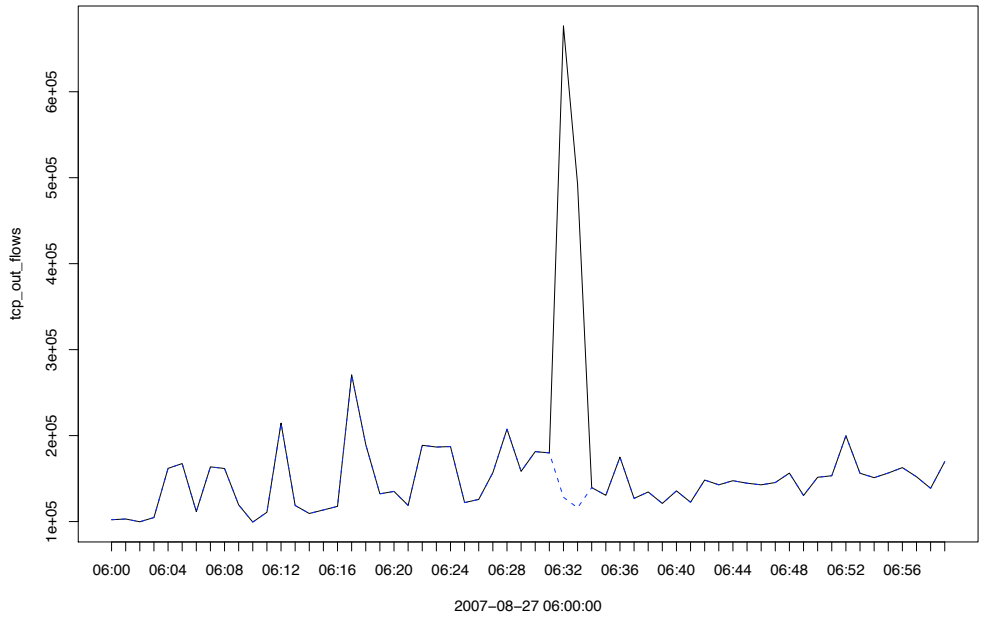

Figure 6.5: Outgoing Flows Count: with (solid line) and without DoS Injection

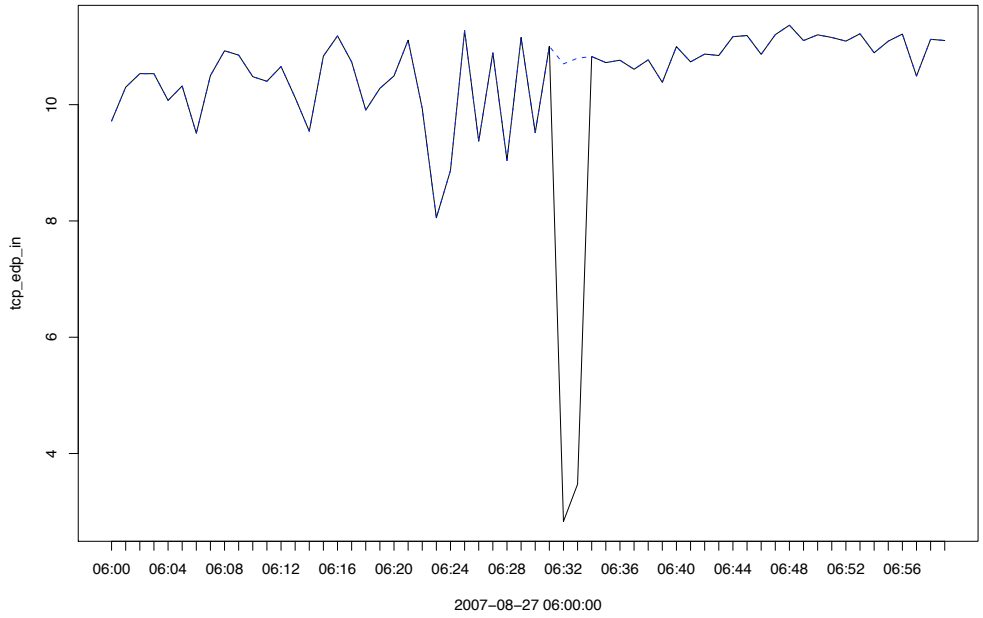

Figure 6.6: Destination Port Entropy: with (solid line) and without DoS Injection

6.4 Performance

The performance of the framework depends greatly on the number of components used and the complexity of the embedded Python Scripts.

When using a simple setup which only consists of pure C++ components we get a throughput of about 140'000 flow records per second. When we then add a NetflowDeleter component which uses a very simple Python script, the performance drops by about 50% already.
By then adding even more components like PacketGenerators, a NetflowGenerator and a NetflowMerger we lose even some more performance and as in the case of the Host Scan event are down to about 45'000 flow records per second.

The test system was running SMP Linux, had 2 dual-core AMD Opteron 275 CPUs running at 2200 MHz and had 8 gigabyte of main memory.

Setup:	**Flow Records**	**Description**
Basic Reader / Writer:	140'000 rec/s	Setup using only pure C++ component: The Reader and the Writer
Loss Event	70'000 rec/s	Setup using also the Deleter component with a very simple Python script
Host Scan	45'000 rec/s	Setup using Reader/Writer components and 1 PythonPacketGenerator as well as the NetflowGenerator and the Merger

6.5 Limitations

The main limitation of the framework is due to the fact that some components have no means of communicating with each other during their anomaly injection. For example the flow stream coming from the NetflowDeleter and the NetflowGenerator only meet for the first time after the injection at the NetflowMerger. But at this point in time the injection at the two points has already been made and the NetflowMerger only merges the two arriving streams. So it is up to the user to model the anomaly injection in a way considering these limitations.
Another limitation of the framework is coming from the traffic generation part. The NetflowGenerator only simulates very basic router behavior. It does for example not add certain fields like destination AS, source AS and others. Furthermore is the flow export only based on 2 different timeout settings, namely an inactive and an active timeout.

6.6 Artifacts

Artifacts in the traffic traces may occur due to several reasons. Either because of the limitations mentioned in the previous section or for other reasons which will be briefly discussed in this section.

Overlap of injected and existing flows: When injecting anomalies into existing flows it can happen that during that same time a flow with the same 5-tuple already exists. The new and existing flow should then be merged into one flow record which is something the NetflowMerger cannot do.
Missing fields from the NetflowGenerator: Some fields like source and destination AS are not entered at the flow generation in the NetflowGenerator component and therefore this introduces artifacts of that missing data.
NetflowGenerator only has basic timeouts: The NetflowGenerator outputs flows only based on active and inactive timeouts and not for example based on TCP flags.
Configuration of the NetflowGenerator not matching the settings in the flow trace: If the settings of the NetflowGenerator don't match with the ones from the original flow trace, artifacts will be introduced (The Netflow version for example will be checked by the Merger, but not everything).
Bad configuration of the PG: Sending illogical answers or packets with impossible settings)

Most artifacts can be avoided by thinking about their possibility and designing the anomaly in a according way to avoid these artifacts. So it is really essential that one thinks about the artifacts before modeling the anomaly.

Chapter 7

Conclusion

7.1 Outlook

So far the evaluation of the framework had the main purpose of testing the functionality. So the first and main task for future work would be to use the framework to actually test existing anomaly detection mechanism.

In the future more components may be added to further extend the framework for example with functionality for working with flow data in general. Like a modifier component which basically works like the deleter but has the ability to also modify all flow characteristics instead of just deleting it by its characteristics.
A component to extract statistical data from the flow records has been developed by Daniela Brauckhoff during the process of this work. She has also started the implementation of new components such as a data sampler to sample existing flow traces.
Another thinkable component would be a data anonymizer to obfuscate existing flows traces.

To further extend the usability of the framework in the future, the support for new formats such as NetFlow v.9 could be implemented.

7.2 Summary

The implementation and evaluation of the framework showed that it is fit for the task for which it was designed. I think that this work can greatly contribute to the evaluation and also development of anomaly detection systems in the future.
And the framework is also a great tool when working with flow data in general and can be easily extended to be able to handle new tasks.

Appendix A

Usage

This chapter is not explaining how the individual components are used since this is already covered in chapter 5. Three configuration examples are also already given in the evaluation chapter (6) and will not be covered here. Instead this chapter gives some examples and ideas on how to use the framework and its components in a more sophisticated or different way.

In the end of the chapter platform requirements to compile and use the framework are given.

A.1 Using the NetflowDeleter

The deleter can be used to do other things. For example it was used to extract a list of IP addresses n write them to a file which was then used in a PacketGenerator for source/destinaion addresses.
Therefore the deleter's python script had to be adjusted and the delete function just isn't called.

```
class Deleter(NetflowDeleter):
    f = open('addresses', 'wt')
    def delete(self):
        ip = self.get_dstAddr()
        ipstr = str(ip)
        self.f.write(ipstr)
        self.f.write('\n')
        return
```

Listing A.1: Deleter Python Script to extract IPs from a trace

A.2 Processing Multiple Files

The framework comes with several scripts, one of which is a script to process a bunch of items with the same components setup and write the output to a specific directory

```
!/bin/bash
# Settings
EXT="bz2"
OUTDIR=""

# Worker function
# Brief: creates the components and connects them to each other
worker() {
        if [ $# -ne 2 ]
        then
                echo "Missing arguments - usage: worker inFile outFile"
                exit 1
        fi

        # Create pipes
        mkfifo pipe1

        ./NetflowReader $1 pipe1 &
        ./NetflowWriter pipe1 $2

        # Delete pipes
        rm -f pipe1
}

if [ $# -le 1 ]
then
        echo "Missing arguments - usage: $0 outputdir/ files*"
        exit 1
fi

i=0
for file in "$@"
do
        i=`expr $i + 1`
        if [ $i -eq 1 ]
        then
                OUTDIR="$file"
        else
                echo "***PROCESSING FILE: " "$file" " ***"
                filename=`basename $file `"."$EXT
                outfile=$OUTDIR"/"$filename
                worker $file $outfile
        fi
done
exit 0
```

Listing A.2: Worker Bash Script to process multiple files

A.3 Other Tips and Trick

- **Writing to /dev/null:** If no file output is needed for some reason (for example if you only want to extract statistical data) instead of a pipe name one can just direct the output to /dev/null which will just destroy any output but still allow all components to run.
- **Using random pipenames:** If you want to run a script more than once in parallel in the same directory you need to make sure that they both use different pipes so they wont interfere with each other. Therefore it would be a good idea to enhance the bash script to use random names for the pipes and delete the pipes again at the end of the process.

A.4 Requirements

There are several requirements for the framework to compile and to run. The components should be able to compile on any unix-like system, but it was only tested under Linux.
To compile the components, depending on the paths and versions of the libraries and header files, the makefiles of the individual projects have to be adjusted.
The following things are necessary to compile the whole framework (might also work with older version):

- GCC 4.1+
- Python library and headers 2.4+
- Boost Python library and headers 1.35+

Appendix B

Time Table

The time period for the whole project was 4 months and the plan on how to spend the time was rather simple. The first month was used for the theoretical part. The 2 months in the middle were used for the detailed design and implementation of the components and the last month was reserved for evaulation and minor implementation tweak and changes.
The following figure shows a graph of how the time was actually spent.

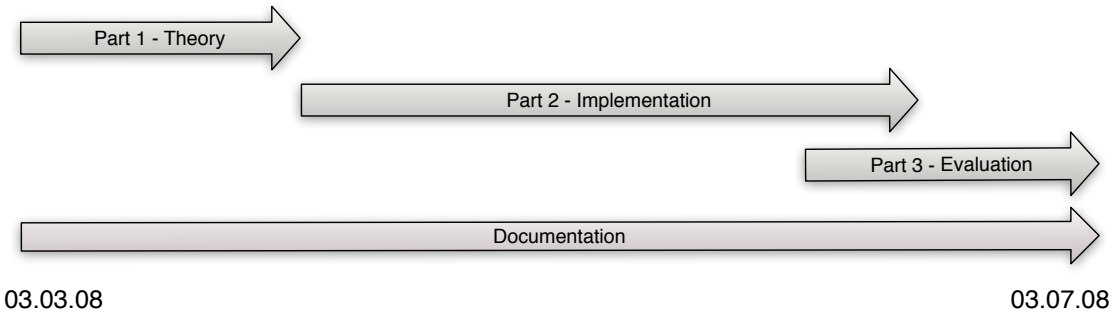

Figure B.1: Visualization of the Time Table

Appendix C

Diploma Thesis Task

C.1 Introduction

Anomaly detection, also known as behavior-based intrusion detection, is a promising approach for detecting interesting network events such as malicious attacks or failures. Anomaly detection systems can operate on different input data streams such as packets, system calls, or flows. A flow is defined as a unidirectional sequence of packets sharing the same characteristic values, e.g., source and destination IPs. Flow-based anomaly detection systems, which operate on aggregated packet header streams, represent a considerable alternative to packet-based detection systems especially on fast network links and for encrypted traffic. However, albeit the advantages flow-based anomaly detection systems face many challenges such as high false alarm rates or insufficient attack classification. Moreover, today's rudimentary evaluation methods, foremost the missing of suitable evaluation and benchmarking data, make a systematic assessment of the advantages and drawbacks of anomaly detection systems practically impossible. We believe that our approach, which aims at combining authentic background traffic with synthetic anomalies provides an optimal trade-off between controllability of the environment and authenticity of the generated traces since it guarantees for realistic and versatile background traffic as well as adjustable and controllable anomalies.
The goal of this thesis is to design and implement a framework for injecting anomalies into existing background traffic. First, the student needs to define a catalog of all possible insertion and deletion actions that can be performed on a trace. These simple actions will then be used to design more complex anomalies (e.g., scans, failures, or denial of service attacks). The student will develop such a model for at least two anomalies. In a second step, the injection framework needs to be developed. First priority is given here to implement the core functionality: the flow generation and injection part. A user interface that allows for parameterizing existing anomaly models and to design new anomaly models from the catalog actions completes the framework. All developed code should be well documented. Finally, the framework is to be evaluated with data from the SWITCH network [1].

C.2 The Task

This thesis is conducted at ETH Zurich. The task of this Diploma thesis is three-fold: First, the student studies literature on existing anomalies and develops an action catalog, as well as two models for more complex anomalies. Second, the core functionality and the user interface of the injection framework will be implemented. In the last step, the implemented anomaly models and the framework will be evaluated.

C.2.1 Action Catalog and Two Anomaly Models

Study literature [6, 7, 13] on frequent types of anomalies (e.g., scans, denial of service attacks, failures, heavy-hitters), and the implications of these anomalies for network traffic. Develop a catalog of possible insertion and deletion actions (e.g., insert constant-rate UDP traffic) that

model specific parts of an anomaly. Show that actions from this catalog can be used to model more complex anomalies by developing a model for at least two complex anomalies.

C.2.2 Injection Framework

Implement the core functionality of the injection framework in C++. This includes a *traffic generation module*, which takes the anomaly model as input and generates a list of trace manipulation rules (e.g., remove all flows between time 1 and time 2 from source 1, insert constant-rate flow from source 1 to destination 2 with n packets), as well as the *trace modification module*, which takes the original trace as input and outputs a trace that is modified according to the manipulation rules. Furthermore, the basic user functionality for parameterizing anomalies, and if time permits, also for constructing new anomalies, is to be implemented. [11, 12] have done something similar on the packet layer.

C.2.3 Evaluation

Evaluate the implementation with Netflow data from the Switch network. In particular, you need to show three things: first, that the modified trace created by the injection framework resembles real anomalies in the traces; second, that the injection does not cause any artifacts; and third, that the modified trace can be used to evaluate different detection systems.

C.3 Deliverables

The following results are expected:

- Catalog of possible insertion and deletion actions.
- Al least two anomaly models for, e.g., a scan, a loss event, or a denial of service attack.
- C++ implementation of the core functionality of the injection framework, i.e., the flow generation and the Netflow trace modification.
- A basic user interface to the injection framework that allows for parameterizing anomalies, and if time permits to design new anomalies from the action catalog.
- A detailed evaluation of the approach.
- A concise description of the work conducted in this thesis (motivation, related work, own approach, implementation, results and outlook). The survey as well as the description of the prototype and the testing results is part of this main documentation. The abstract of the documentation has to be written in both English and German. The original task description is to be put in the appendix of the documentation. One sample of the documentation needs to be delivered at TIK. The whole documentation, as well as the source code, slides of the talk etc., needs to be archived in a printable, respectively executable version on a CDROM, which is to be attached to the printed documentation.

C.4 Organizational Aspects

C.4.1 Documentation and presentation

A documentation that states the steps conducted, lessons learned, major results and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another developer within reasonable time. At the end of the thesis, a presentation will have to be given at TIK that states the core tasks and results of this thesis. If important new research results are found, a paper might be written as an extract of the thesis and submitted to a computer network and security conference.

C.4.2 Dates

This Diploma thesis starts on March 3rd 2008 and is finished on July 3rd 2008. It lasts 4 months in total. At the end of the second week the student has to provide a schedule for the thesis. It will be discussed with the supervisors.
After a month the student should provide a draft of the table of contents (ToC) of the thesis. The ToC suggests that the documentation is written in parallel to the progress of the work.
One intermediate informal presentation for Prof. Plattner and all supervisors will be scheduled 2 months into this thesis.
A final presentation at TIK will be scheduled close to the completion date of the thesis. The presentation consists of a 15 minute talk plus 5 minutes for questions. Informal meetings with the supervisors will be announced an organized on demand.

C.4.3 Supervisors

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

Arno Wagner, wagner@tik.ee.ethz.ch

Bibliography

[1] SWITCH - The Swiss Education and Reserach Network. http://www.switch.ch/.

[2] Paul Barford and David Plonka. Characteristics of network traffic flow anomalies. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2001.

[3] D. Brauckhoff, A. Wagner, and M. May. Flame: A flow-level anomaly modeling engine. In *CSET Workshop, USENIX Security, San Jose*, 2008.

[4] Myung-Sup Kim, Hun-Jeong Kong, Seong-Cheol Hong, Seung-Hwa Chung, and J.W. Hong. A flow-based method for abnormal network traffic detection. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, 2004.

[5] Anukool Lakhina, Mark Crovella, and Christophe Diot. Characterization of Network-Wide Anomalies in Traffic Flows. In *IMC'04, October 25-27, 2004, Taormina, Sicily, Italia*, 2004.

[6] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 219–230, New York, NY, USA, 2004. ACM Press.

[7] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining Anomalies Using Traffic Feature Distributions. In *ACM SIGCOMM 2005*, 2005.

[8] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.

[9] Gerhard Munz and Georg Carle. Real-time analysis of flow data for network attack detection. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium*, 2007.

[10] Andy Rupp, Holger Dreger, Anja Feldmann, and Robin Sommer. Packet trace manipulation framework for test labs. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 251–256, New York, NY, USA, 2004. ACM Press.

[11] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Internet Measurement Conference*, 2004.

[12] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *Internet Measurement Conference*, pages 82–87, 2004.

[13] Augustin Soule, Kavé Salamatian, and Nina Taft. Combining filtering and statistical methods for anomaly detection. In *IMC '05*, 2005.

[14] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *ACM SIGCOMM Computer Communication Review, 2006*, 2006.

[15] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *Workshop on Rapid Malcode, Proceedings of the 2003 ACM workshop on Rapid malcode, Washington, DC, USA*, 2003.