ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Gabriel Müller

Timing, ... in Firewall Testing

Semester Thesis
April 2, 2007

Winter Semester 2006/07
Supervisor: Diana von Bidder
Professor: David Basin

Abstract

Firewalls play an important role in today's networks. Therefore it is crucial, that firewalls work as expected. To ensure this, firewalls are tested and the observed behavior is compared to the expected one.
One firewall testing tool is fwtest. The user of fwtest specifies testcases. According to the content of the testcases, fwtest creates and sends a set of network packets. A timer is started and fwtest receives network packets which passed the firewall. When the timer expires, fwtest analyzes which network packets could pass the firewall. Then the next set of network packets is send.
In this semester thesis, the processing time of fwtest was decreased in two different ways. First, the timer value is now estimated and set dynamically, adapted to the current situation. Second, fwtest is now able to interleave testcases automatically and process them in parallel.
It could be shown that both modifications substantially shorten the processing time of fwtest.

Contents

List of Figures

Chapter 1

Introduction

Firewalls are used to restrict access between different networks. Therefore, a firewall is placed at the interconnection of the networks, acting as a gatekeeper. Every access attempt, originating from one network and directed to another one, is controlled by the firewall. The firewall decides whether it permits this access attempt or not. An access attempt consists of one or more network packets, sent to the other network. The firewall permits a certain access attempt by allowing the network packets to pass to the destinated network. If the firewall does not permit an access attempt the corresponding network packets are blocked by the firewall. Decisions made by a firewall are based on configurations, set up by a human before.
These configurations may contain errors: On the one hand, not wanted access attempts may not be blocked by the firewall. On the other hand, the firewall may block access attempts which are intended to pass the firewall.
One way to test a firewall to verify its configuration is to create access attempts and see whether they are successful or not. By comparing the test results against the predefined, expected behavior of the firewall, errors in the configuration can be detected. One tool, performing such tests, is *fwtest* [Zau05, Str06, Sch06].

1.1 fwtest

In this thesis, the firewall testing tool fwtest will be extended. We begin by describing fwtest v1.0 before explaining our goals.

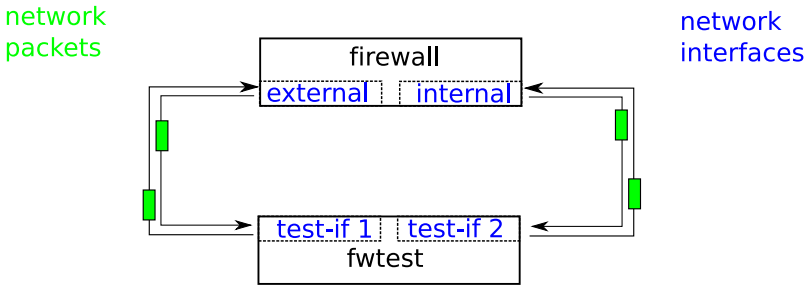

Figure 1.1: fwtest Test Setup

For a test run, the firewall is directly connected with the computer which runs the fwtest software (see Figure 1.1). Tests are performed according to predefined test specifications, containing network packet descriptions. The fwtest software creates and sends network packets to the firewall and observes if these network packets can pass the firewall or not. Behavior deviating from the test specifications is recorded. The firewall is seen as a blackbox. The blackbox approach is motivated by the goal to develop a firewall testing tool which can be used for various types / implementations of firewalls. This means, that nothing is known about the

```
define(ipa, 172.16.70.3)
define(ipb, 192.168.72.3)
testcase 1 {
             packet 1  { TCP
                       send { ipa ipb 4000 25 S 60 - }
                       receive { ipa ipb 4000 25 S 60 - }
                       }
             packet 9  { TCP send { ipa ipb 2002 25 S 60 - } receive ok }
}
testcase 2 {
             packet 1  { UDP send { ipb ipa 5000 22 } receive {} }
}
```

Figure 1.2: tp-file Example

firewall (hardware, operating system, software, configuration).

How a firewall is tested by fwtest has to be specified by the user. Therefore the user defines so called *testcases* (in a text file, which we call *tp-file* from now on). Each testcase contains one or more entries, specifying packets which should be sent at different times during the test. An example is shown in Figure 1.2.

The most important aspects of a tp-file are explained next. More information about tp-file specifications can be found in the README (Appendix A). More extensive information can be found in the semester thesis of Adrian Schüpbach [Sch06].

packet X { **protocol** { **send** { **...** } **receive** { **...** } }}: Describes details of a network packet. The packet id (`X`) defines the relative time (also called *timestep*) when the packet should be sent. Packets with higher ids will be sent later, those with lower ids will be sent sooner. `protocol` defines the protocol type of the packet. Possible values are `TCP` [DAR81b], `UDP` [DAR80] or different values for the different ICMP message types [DAR81a]. Values defined in the `send` statement are included in a network packet which is sent by fwtest. Values specified in the `receive` statement are expected to be found in the network packet when it is received by fwtest. In the `receive` statement two shortcuts can be used: `ok` indicates that the received network packet is expected to look like exactly as it was specified in the `send` statement. {} means that the network packet is not expected to be received at all.

testcase K { **...** }: A testcase contains one or more packet descriptions. Testcase ids must be unique. In a testcase every packet id can be used only once. Packet ids in different testcases may be equal. The corresponding packets will be sent in the same timestep. Testcases are seen and processed independently from each other by fwtest.

define(variable, value): The statement is used to assign a value (an IP address or a port) to a variable. From then on the variable can be used in packet descriptions. A preprocessor will replace variables with their corresponding values before the tp-file is parsed.

At start-up, a parser will transfer the content of the tp-file into an *internal structure* of fwtest. Packet descriptions are sorted according to their timestep numbers. The information of each packet description is stored in a structure of type `event`. Structures, which contain packet descriptions of the same timestep, are grouped together in a linked list. Each such list is referenced by a structure of type `tnode`. For the tp-file of Figure 1.2 this results in the structure given in Figure 1.3.

After the tp-file has been parsed, the actual firewall test is started. Packet descriptions referenced by the first `tnode` structure are built, sent, received and analyzed. Then the packet descriptions of the second `tnode` structure are processed and so on. What follows, is a more detailed description of this procedure: According to the packet descriptions of the first `tnode` structure, network packets are built and sent by fwtest. Network packets, which can pass the firewall, are received by fwtest (if a packet was injected by the interface *test-if 1* it will be received by the interface *test-if 2* and vice versa, see also Figure 1.1). Because the network packets are delayed by the firewall, fwtest will wait some time before analyzing the received packets. This is done by a timer which is started after the last packet has been sent. After the timer expired, the received network packets are analyzed:

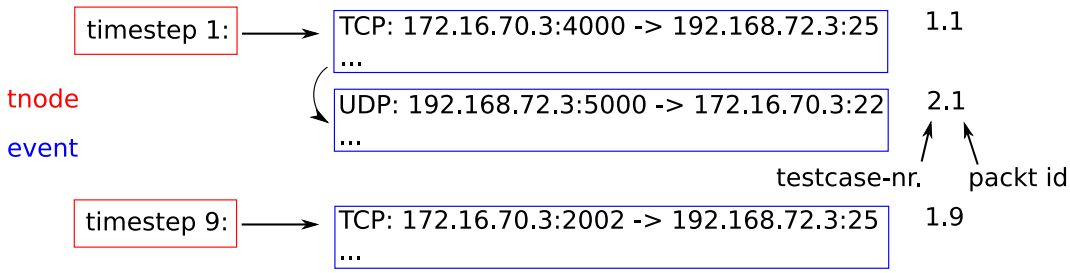

Figure 1.3: Internal Structure

- Were packets received which should have been blocked by the firewall?
- Were all packets received which were assumed to pass the firewall?

Irregularities are written to a log file.

1.2 Goal

The current implementation of fwtest has two drawbacks. First, the timer value used to wait for network packets is fixed. Therefore, a test run of fwtest takes a lot of time. Second, an efficient, parallel processing of testcases must be defined explicitly by the user and is not done automatically by fwtest. Parallel processing of testcases is desireable because it reduces the runtime of fwtest.
The goal of this semester thesis is to eliminate these drawbacks: The timer value should be estimated with respect to the current situation at run time. The testcase definitions of the user should be interleaved automatically by fwtest.
The extended version of fwtest (*fwtest v2.0*) must produce the same test results as fwtest v1.0.

Timer Value

So far, the timer value is set to a predefined value. Because of this, the timer value does not take the current situation of the test into account: How many packets must be sent in this timestep? How many packets are expected to be received by fwtest? The timer value also does not take into account other aspects such as the delay which is introduced by the firewall. Therefore it would be desireable that the timer value is set dynamically to fulfil the following requirements:

- The timer value is chosen with respect to the number of packets of the current timestep.
- The timer value is chosen with respect to the test environment

Interleaving

So far, only (parts of) testcases are processed in parallel which contain packet descriptions with equal packet ids. It is the burden of the user to enforce a parallel processing by setting the right packet ids in the tp-file. This task becomes even more complex if the user has different testcases with similar packet descriptions. Here the user has to take care that those testcases are not processed in parallel: If similar testcases are processed in parallel it is possible that their network packets influence each other. This influence can falsify the test results of fwtest.
Therefore it would be desireable, that testcases are interleaved automatically by fwtest to take away the burden from the user. The goal is to interleave / rearrange different testcases as close together as possible. This would allow a great speed up of the test duration because those testcases can then be processed in parallel.

1.3 Organization of this Thesis

We start by disscussing different timing issues in networks and firewalls in Chapter 2. This background knowledge is then used in Chapter 3 to design solutions to the problems, presented in Section 1.2. These are then implemented and tested in Chapter 4 and 5. Finally we conclude and report on future work in Chapter 6.

Chapter 2

Background Knowledge

The knowledge presented here will be necessary to understand the set of problems, presented and attacked in the next chapter.
Firewalls introduce some delay to network packets when they examine the network packets. The paper, presented in section 2.1, shows that such a delay can strongly vary and explains why this is the case.
Network packets of different testcases can influence each other (regarding their success of passing the firewall). This is definitely not the intention of the user which specified these test-cases. Section 2.2 explains in which situations such an influence takes place and how it can be prevented.

2.1 Packet Delay Aspects

The paper ”Packet Delay and Loss at the Auckland Internet Access Path” [KM] describes the study of packet delay and loss of IP data, traversing the University of Auckland Internet access path. The study concentrates on measuring delay and loss of network packets, caused by fire-wall and router. We were not able to find more papers attacking these issues.
First, test setup and the measured values of the study are described. Then an explanation for the observations is given and it is mentioned why this result of the paper is of importance for this thesis.

Test Setup and Measurement

Router and firewall connect the university network with the internet (see Figure 2.1). During 96

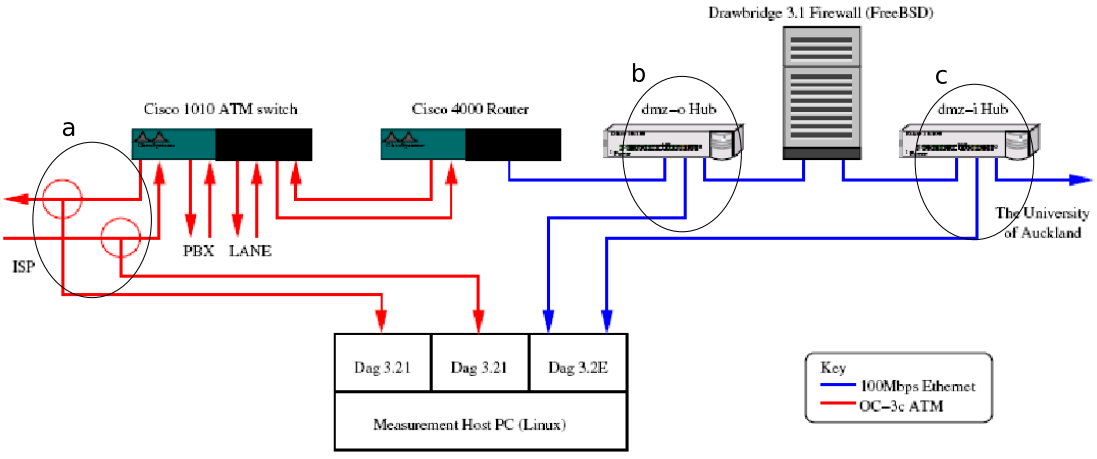

Figure 2.1: Network Architecture of Auckland University (picture taken from paper)

hours the first 40 bytes of each network packet were captured at three points in the network:

- at the ISP[1] entry point (circle a in Figure 2.1)
- between router and firewall (circle b in Figure 2.1).
- at the entry point of the university network (circle c in Figure 2.1).

A packet was identified at the different points (a,b and c) by comparing the captured 40 bytes. Specialized network adapters were used to measure the arrival time of network packets. The achieved measurement accuracy was about 10^{-6} seconds.

Results

The measured delay of network packets, introduced by the router, was as follows:

- For incoming traffic (to the firewall/university network): 99% were below 0.53ms
- For outgoing traffic (to the internet): 99% were below 82ms

Explanation

The difference of two orders of magnitude can be explained with the uplink speed of the ISP connection which is limited to 4.048Mbit/s. In contrast to that, the university network is operated at 100Mbit/s. As a consequence, when the router received too much traffic directed to the internet, the router had to queue packets which lead to the high delays.

Consequences for this Thesis

Although fwtest is not used to test routers but firewalls, the paper shows the consequences of queuing packets. This could also happen at a firewall when it is not able to process the network packets fast enough. So the range of possible delay values can be large which has to be kept in mind, when timeout values are estimated.

2.2 Connection Tracking

Stateful firewalls keep track of the state of current connections (called *connection tracking*). Ignoring this fact could lead to wrong test results and must be taken into account when reordering testcases. This section explains the basic idea of connection tracking.
Connection tracking associates incoming network packets with different connections / data streams[2] and keeps track of the current state of these connections / data streams. Depending on the network protocol, connection tracking differentiates between several states of a certain connection. Whenever connection tracking recognizes a further network packet of that connection, the state of that connection might change. This is important, since the timeout value of a connection depends on the current state of that connection.

Connection tracking introduces timeouts. This means, that after a certain time of inactivity of a certain connection, the information about this connection is removed. Then a new arriving network packet of a connection is treated as the first seen network packet of that connection. The following example demonstrates the described behavior:
Assume that the firewall is configured to only let pass an ICMP echo reply, if it belongs to a previously sent ICMP echo request. Furthermore, assume a timeout value of 30 seconds for ICMP traffic. This can result in the following unwanted scenario: An ICMP echo reply that reaches the firewall 40 seconds after its corresponding ICMP echo request, is dropped by the firewall.

[1] Internet Service Provider

[2] The term *connection* is usually used for traffic with some kind of state (like TCP). The term *data stream* describes one or more packets without a state but where the packets are somehow related/depend with/on each other. Note that a firewall can also maintain a state for a stateless *data stream*. To simplify things, from now on we will use the term *connection* for connections and data streams.

One the other side, imagine that the time gap between the two ICMP packets is smaller than 30 seconds. Both ICMP packets will pass the firewall. This is problematic if the two ICMP packets belong to different testcases.

Consequences

With the help of the ICMP example it was shown, why and how previously sent network packets can influence the success of network packets passing the firewall. The firewall keeps track of connections by storing the current state of them for a certain amount of time. But the time amount is finite and was introduced above as timeout. Because of the finite timeout values, it is possible to solve the problem: Testcases, containing conflicting packet descriptions, must be processed with a sufficient large time gap in between. This time gap must be larger than the timeout value used by the firewall's connection tracking. Because of the blackbox approach, it will be difficult to find out these timeout values. For iptables, the states and the corresponding timeout values were derived by looking at the timeout values used at the endpoints of the connection. These states and timeout values are described in the protocol specifications (e.g. for TCP in the TCP state diagram). The values can be used to test iptables based firewalls. Timeout values of other firewalls may be different.
Note, that this section only explained the basic ideas of connection tracking. At least for iptables, there are much more sophisticated extensions. These extentions enable iptables to keep track of more complicated application layer connections. The most famous two might be FTP [JP85] and SIP [JR02] related connections. These connections use several network connections as control and data channels.

Chapter 3

Design

The goal of this thesis is to speed up the runtime of fwtest. This is done by choosing suitable timer values and by reordering testcases. This chapter describes the design decisions taken to reach this goal.
Section 3.1 first explains the consequences of too short timer values. Then it describes, how timer values are estimated and based on which information these estimations are done. Finally, a method for detecting too short timer values is presented.
Section 3.2 discusses the parallelization of test cases.

3.1 Timing

The packet ids, contained in the packet descriptions of the tp-file, define the sending order of the network packets. All network packets with the same id are expected to be sent in the same timestep. Furthermore, it is expected that all network packets of a certain timestep have been processed by the firewall before network packets of a following timestep reach the firewall.
If the timer of fwtest expires too early, it is possible that these requirements are no longer fulfilled. Then it might be the case that a network packet of the next timestep is processed by the firewall before all network packets of the current timestep were processed. An example is given in Figure 3.1: The network packets `a` and `b` fulfil the requirements. This is not the case for network packet `c` and `d`: Packet `d` is processed by the firewall before packet `c`.

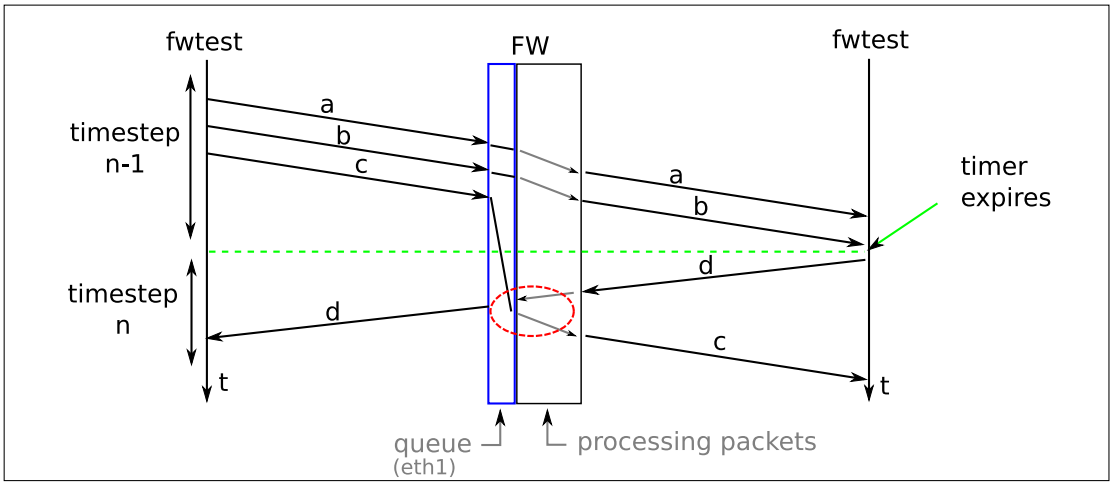

Figure 3.1: Processing Order of Network Packets at the Firewall

The impact of a wrong processing order can be severe, as the following example shows: Assume that packet `c` of Figure 3.1 is a TCP SYN packet and packet `d` is its corresponding ACK. A stateful firewall might drop packet `d` because it has not seen a SYN packet before. The same

configured stateful firewall might not drop packet `d` if it processes packet `c` first. Hence, a wrong processing order can influence the success of network packets passing the firewall. This can lead to wrong test results.
Because of the blackbox approach, network packets can only be observed at fwtest but not at the firewall. Therefore it is not possible to directly detect whether a network packet was already processed by the firewall or not. This can only be done indirectly, by receiving a network packet after it has been processed by the firewall. To ensure that all network packets are processed by the firewall in the correct order, the timer value must be chosen sufficiently large. Sufficiently large means, that the timer does not expire before all network packets of the current timestep have been received. For the situation, depicted in Figure 3.1 this means that the timer should expire after packet `c` has been received. Otherwise fwtest may be sends network packets of the next timestep too early. It is then impossible to detect if all network packet of the previous timestep have already been processed by the firewall.
The usage of the timer is necessary, since one cannot wait infinite long, until all packets of a timestep have been received. It is not possible to detect, whether a packet was not processed yet by the firewall or it was blocked by the firewall. Therefore fwtest has to continue its processing at some time. The timer value defines when fwtest continues. Note, that this design assumes that network packets might only be dropped by the firewall, but do not vanish during the transport inside the network environment.
It is likely, that it will not be possible to predict an appropriate timer value every time. If the timer value was chosen too short, the above mentioned requirements may no longer be fulfilled.
Two challenges were identified which will be attacked next. The first challenge is to estimate a suitable timer value. The second challenge is to detect too short chosen timer values and react appropriate to them.

3.1.1 Timer Value Estimation

This section describes how the timer value can be adapted to the current test situation. Simplified, the timer value estimation is based on the measured timing values of the previous timesteps and the number of network packets to be sent in the current timestep. Only network packets which were sent and received in the same timestep, before the timer expired, are used for evaluation purposes.

Measurement of Values

After a network packet is sent by fwtest it may traverse the firewall and in this case is captured by fwtest. This is depicted in Figure 3.2: The network packets `a`, `b` and `c` were sent at times `t1`, `t2` and `t3`. They successfully passed the firewall and were captured at points in time `t4`, `t5` and `t6`. For the following, the time between sending and capturing a network packet will be

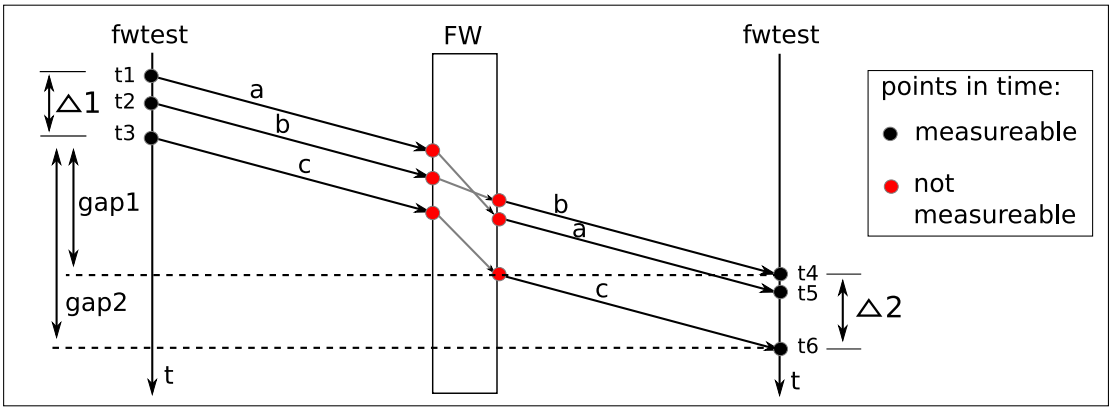

Figure 3.2: Points in Time

called *traveling time*. The traveling time can be calculated with the sending and receiving time of a network packet. Additionally the following values can be calculated (see also Figure 3.2):

- The time needed to send all packets of a timestep ($\Delta 1$).
- The time needed to receive all packets of a timestep ($\Delta 2$).
- The delay introduced by the firewall (`delay_fw`): $\Delta 2$ minus $\Delta 1$.
- The minimal traveling time of the current timestep (`min`).
- The maximal traveling time of the current timestep (`max`).
- The average traveling time of the current timestep (`ave`).
- The variance regarding the traveling time of the current timestep.
- The time between sending the last packet and receiving the first packet (`gap1`).
- The time between sending the last packet and receiving the last packet (`gap2`).

Note, that `gap2` does not always describe the traveling time of the last network packet. As shown in Figure 3.2, the order of the network packets my be changed by the firewall. Also remember, that only network packets which were received before the timer expired, can be used for the calculations.

Additionally it is possible to calculate values over several timesteps, such as:

- The average of `delay_fw` of several previous timesteps (e.g. over the last 10 timesteps) (`ave_delay_fw`).
- The average of `min`, `max` or `ave` of several previous timesteps (`ave_min`, `ave_max`, `ave_ave`).

Finally, global values can be determined. Global values consider data of all previous timesteps. Examples are the global maximum of `gap2` (`glo_gap2`), the global maximum traveling time (`glo_max`) and the global maximum delay of the firewall (`glo_delay_fw`).

Next, the meaning of the presented values is described.
The different measurements of the travelling time (`min, max, ave, variance`) indicate how strong the travelling time of the current timestep varies.
$\Delta 1$ describes how fast fwtest sent the network packets. This information can be used as an indicator for the performance of the network and/or the network interfaces. $\Delta 2$ shows how long it took fwtest to receive all network packets of the current timestep. This value might be strongly influenced by the performance of the firewall. Therefore, the difference of $\Delta 2$ and $\Delta 1$ (`delay_fw`) can be very interesting. It describes the amount of time the firewall needed to process the network packets of the current timestep. This information can be a good indicator for the performance and / or current load situation of the firewall.
`gap1` is the time, fwtest is idle after it has sent the last network packet and until it receives the first network packet. `gap2` would have been the perfect timer value for the current timestep. But because it depends on the number of network packets and the current situation, it cannot be used directly for further timer settings.
The observation of certain values over several timesteps can be used to recognize tendencies. E.g. an increasing `ave_delay` value indicates that the firewall might become more and more congested and it may be necessary to increase further timer values.
Global values such as `glo_delay_fw` can be useful, whenever worst case assumptions must be made.

Different Estimations

There exist several possibilities to estimate a timer value based on the measureable values. We will discuss the following:

- Estimation 1
  The timer is set to `glo_gap2`.

- Estimation 2,3 and 4
  The timer is set to the sum of (`glo_max` or `ave_max`) and (`glo_delay_fw` or `ave_delay_fw`).

Estimations 2 to 4 take into account the number of packets of the current timestep as follows: When values such as `delay_fw` are calculated, their are divided by the number of received network packets of that timestep. This is done to get a 'per-packet'-value. For the estimation of a timer value these per-packet-values are multiplied by the number of network packets of the next timestep.
The firewall delay values (`glo_delay_fw` and `ave_delay_fw`) used in Estimation 2, 3 and 4 try to take into account the current load situation of the firewall.

Estimation 1 describes a very simple timer value estimation: As the timer is started, after the last network packet of a timestep has been sent, in most cases, it should be sufficient to set the timer value to `glo_gap2`.
Estimation 2 uses the maximum traveling time detected so far (`glo_max`). Additionally it takes into account the delay introduced by the firewall, but only uses the global maximum value of `delay_fw` (`glo_delay_fw`). Note, that these two values can be measured in different timesteps.
Estimation 3 tries to detect whether the global maximum traveling time (`glo_max`) has increased too much or not. This is done by comparing this value to the average maximum traveling time of several previous timesteps (`ave_max`). If `glo_max` has reached a certain threshold value, it is no longer used for the timer. Instead `ave_max` is used from now on. `glo_max` is primarily used at the beginning, since then not sufficient average values are available. As in Estimation 2, `glo_delay_fw` is used here.
Estimation 4 The first part of estimation 4 is equal to the first part of estimation 3 (sum of `glo_max` or `ave_max`). For the second part, the maximum global delay (`glo_delay_fw`) is only used as long as it stays below a certain threshold value. As soon, as the threshold value is reached, estimation 4 uses the averages value `ave_delay_fw`. The usage of the global value is primarily intended for the beginning, as long as not sufficient average values are available.

The complexity of the estimations increases more and more from Estimation 1 to Estimation 4. Whether such a complex estimation as *Estimation 4* is necessary or not for proper timer settings, will be seen in the evaluation of this thesis. Next, some theoretically possible disadvantages of the estimations are described.

Estimation 1 has several disadvantages: First, especially at the beginning, the value of `glo_gap` can be too small. Second, `gap2` is only increased but never decreased, which will slow down the processing time. Third, the estimation does only partly take into account the delay introduced by the firewall (indirectly, since the delay might be contained in `glo_gap`).
Estimation 2 suffers from similar disadvantages as estimation 1, since only global values are used here, too.
Estimation 3 still takes into account a global value (`glo_delay_fw`) for its timer value estimation. But because the size of this global value may often be very small (compared to the other values), the negative impact seems to be lower.
Estimation 4 seems to have no disadvantages, but is very complicated and may take more computation time.

3.1.2 Delayed Packets

After having seen, how timer values can be estimated, the following part describes how delayed network packets can be detected. This is motivated by the fact, that estimations are never perfect. Remember that we must detect if the timer expired too early due to a too short set timer value. Furthermore, appropriate actions must be taken.

Delayed Packets and Test Results

In every timestep fwtest sends network packets and then starts the timer (see Figure 3.3). In the background fwtest continuously captures network packets which passed the firewall. These

network packets are stored in the receiving queue. When the timer of the current timestep expires, fwtest empties the receiving queue and delivers its content to the analysis function. If the timer value was too short, the timer expired too early. Not all network packets of the current timestep (which passed the firewall) were already captured. The analysis function can not take them into account in the current timestep.

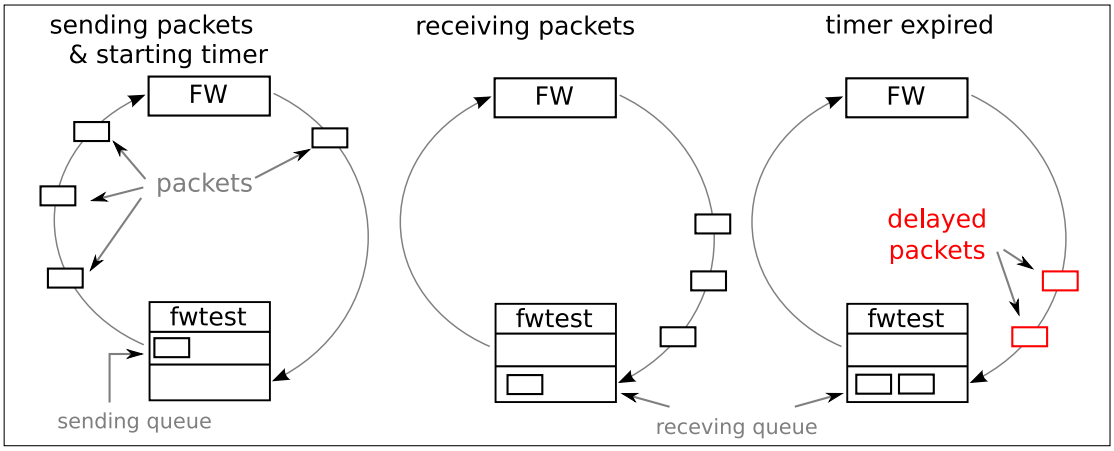

Figure 3.3: Consequences of a too short Timer Value

These delayed network packets are not lost. They will be received and stored in the receiving queue and delivered to the analysis function when it is called again. But the analysis function of fwtest v1.0 compares network packets in the receiving queue only against packet descriptions of the current timestep. For an example, see Figure 3.4: The analysis function is currently processing timestep 4. The network packets, contained in the receiving queue, are `e8`, `e9`, `e10`, `e7` and `e4`. These network packets are compared against the packet descriptions of timestep 4. The analysis function will be able to assign the network packets `e8`, `e9` and `e10` to their corresponding packet descriptions. But it will fail to assign the network packets `e4` and `e7` to their corresponding packet descriptions of timestep 2 and 3, because only packet descriptions of timestep 4 are considered by the analysis function.

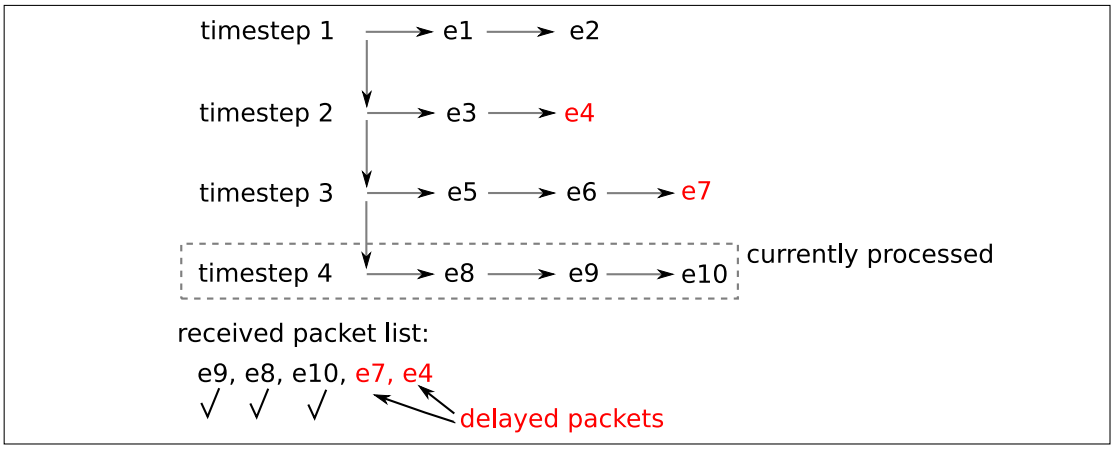

Figure 3.4: Delayed Packets

Therefore, a too short timer value can be responsible for wrong entries in the log file. For an example assume that the analysis function is currently processing timestep 3 (again see Figure 3.4). Packet `e7` is delayed and was not received yet. Because the function cannot find network packet `e7` in the receiving queue, the following happens (fwtest v1.0):
If the packet is assumed to be dropped by the firewall according to the packet description, the analysis function assumes `true positive`[1]. No log entry is created .

[1] Network packet is assumed to be dropped, and was not received

If the packet is assumed to be passed by the firewall according to the packet description, the analysis function assumes `false positive`[2]. A log entry is created altough no entry should be created.
An example for a wrong log entry (second case mentioned above, created by fwtest v1.0) is depicted below: The analysis function could not find the packet and assumed `false positive` (the network packet was assumed to pass the firewall but was not found in the receiving queue). The network packet was found in the receiving queue later. It is mentioned in the log file as an unknown received packet and timestamped with its receiving time.

```
LOG:
# [time]        [type]      [id][packet]
#
17:16:37.501869 false_pos   1.1       --
17:16:36.992900 false_neg   --  (ICMP 10.0.0.5 > 192.168.100.6, type: 8, code: 0, ...)

TP FILE:
define(ipa, 10.0.0.5)
define(ipb, 192.168.100.6)
testcase 1 {
            packet  1 { icmpecho send { ipa ipb ECHO_REQUEST 10 910 } receive ok }
           }
```

The problem can easily be solved by comparing network packets in the receiving queue against packet descriptions of previous timesteps. Log entries, containing information about detected delayed network packets must be created.

Limited Countermeasures

Remember that it is impossible to detect whether delayed packets where processed too late by the firewall or not. Therefore it must be assumed that delayed network packets were processed too late by the firewall. It is possible that those network packets influenced the test outcome of the corresponding testcase. As a consequence, the testcases of such network packets must be re-run manually (by the user) with sufficient large timer values.

3.2 Parallel Processing of Testcases

After having discussed how to reduce the time of one timestep we will now elaborate on how (to save more time by) parallelizing the testcases.
So far, testcases are only processed in parallel if the user specified a parallel processing by using equal packet ids in several testcase definitions. For the user this is a cumbersome and difficult task, mainly because not all testcases are allowed to be processed in parallel, since this can influence their test outcome. This section explains why time can be saved with parallel processing of testcases, why not all testcases can be processed in parallel and how testcases are automatically interleaved for parallel processing.

3.2.1 Sequential Versus Parallel Processing

Why time can be saved when processing several testcases in parallel (which are intended to be processed sequentially)? This can be seen best with the help of an example: Assume the user specified the tp-file depicted in Figure 3.5. Sequentially processed, the network packets of these testcases are sent in three different timesteps. This means, the timer is started 3 times and one must wait 3 times for a single packet to be captured by fwtest. In contrast, if all testcases are processed in parallel, the timer is only started once, and the network packets of the three testcases are all captured in the same timestep.
Depending on the test setup and environment, typical timer values are in the range of 5ms and 500ms. The more testcases can be processed in parallel, the more time can be saved, since the timer is used less often. Especially for large tp-files the overall processing time of fwtest is strongly reduced.

[2]Network packet is expected to pass the firewall, but was not received

```
testcase 1 {
             packet 1 { udp send { ipa ipb 100 200 } receive ok }
           }
testcase 2 {
             packet 2 { udp send { ipc ipd 10 20 } receive ok }
           }
testcase 3 {
             packet 3 { udp send { ipc ipb 100 200 } receive ok }
           }
```

Figure 3.5: tp-file

3.2.2 Parallel Processing

A maximum time saving would be achieved by processing all test cases of a tp-file in parallel. Unfortunately this is not possible due to several reasons which are described and explained next.
Different testcases must be tested independently by fwtest. This means, the test results of a modified and processed tp-file must be the same, as the results of the same tp-file, not modified before processing. This is possible as long as the firewall does not classify network packets of different testcases to the same connection (Section 2.2).

Dependent Testcases

As seen in Section 2.2, dependent testcases cannot be processed at the same time. Two testcases, containing network packets which would be classified to a single connection by the firewall, are called *dependent testcases* from now on. Whether two testcases are dependent or not is determined by analyzing their packet descriptions. If any two packet descriptions of two testcases are equal in all the following points, the corresponding testcases will be classified as dependent:

For TCP / UDP:

- Protocol
- Source IP
- Destination IP
- Source port
- Destination port

For ICMP:

- Protocol
- Source IP
- Destination IP
- Sequence number and ID (for other then ICMP echo: content of ICMP header)

This also holds if source and destination IPs or ports of the two packet descriptions are exchanged.

Efficient Classification of ICMP Network Packets

In the case of the ICMP protocol there are two types of network packets which can conflict with each other: A ICMP echo request can conflict with an equal ICMP echo request. Furthermore the ICMP echo request can conflict with a matching ICMP echo reply. Equal requests are problematic, since they re-new the timeout value at the firewall for connection tracking. Matching replies are problematic, since they can answer a request of another testcase. All possible combinations are depicted below:

- request A→B conflicts with: request A→B and reply B→A
- request B→A conflicts with: request B→A and reply A→B
- reply A→B conflicts with: reply A→B and request B→A

- reply B→A conflicts with: reply B→A and request A→B

Conflicting combinations must be grouped together in the same dependency. Again, this must be done to prevent the parallel processing of dependent testcases and to assign and enforce time constraints.
For an efficient classification of successive ICMP packet descriptions of further testcases, the following is proposed. At the first occurrence of a certain ICMP echo request or reply two dependencies are created at the same time. Depending on the direction of the ICMP packet, one of the following dependency pairs is created:

Dependency Pair 1

- Dependency 1: Request A→B
- Dependency 2: Reply A→B

Dependency Pair 2

- Dependency 1: Request B→A
- Dependency 2: Reply B→A

It is possible to create two dependencies, since an ICMP request and an ICMP reply in the same direction are independent. Therefore they can be processed in parallel. Both dependencies of a pair have to be created, since it is now much more easier to find conflicting ICMP packets and group them together in appropriate dependencies.

Variables for IPs and Ports

fwtest allows the user to define variables inside the tp-file for IPs and ports. The motivation for this are scenarios where IPs or ports can change (e.g. NAT, refer to the fwtest README for more details). When variables are used, the values of these variables are only known and assigned at run time. Any value (of the set of possible values) can be assigned to the variable at run time. Therefore, one must assume that any pair of a given IP and a variable (of two different packet descriptions) are equal. The same holds for ports. Two examples:

Example 1
192.168.1.10:80 → 192.168.2.10:**2555**
and
192.168.1.10:80 → 192.168.2.10:**VarA**

Example 2
192.168.1.10:80 → 192.168.2.10:**VarB**
and
192.168.1.10:80 → 192.168.2.10:**VarA**

Because the port number `2555` can be assigned to `VarA` at run time, the two packet descriptions can become equal (Example1). Therefore they must be grouped together in the same dependency. It is also possible that the same port number is assigned to `VarA` and `VarB` (Example2). Therefore, the two packet descriptions on the right side must be grouped together in the same dependency.

Time Constraints

Recall that a firewall remembers a connection for a certain time amount (Section 2.2). When processing two dependent testcases sequentially, we have to wait at least this time amount between the last packet of the first test case and the first packet of the second test case. The problem is that this time amount is different for every firewall and protocol. Therefore the user can specify these values. Whenever needed, a *time constraint* (consisting of such a value) is added to a test case (see Figure 3.6). In this example, at least 10 second must be in between the receiving time of packet A→B and the sending time of packet C→D. This time constraint needs to be fulfilled during testing.

Handling Time Constrains

To prevent that two dependent testcase interfere with each other, the time constraint value was introduced. A further dependent testcase of a certain dependency will not be processed until

- dependency 1: A→B(time constraint: 10s), C→D and E→F
- dependency 2: X→J, V→X, J→Z
- dependency 3: L→K, O→Q

Figure 3.6: Dependency Example with Time Constraint

the time constraint of the previous testcase is fulfilled. What should be done when fwtest has to wait for a specific time constraint to become true. There are two possible procedures:

A Do nothing until the time constraint is fulfilled.

B Try to process packets of other dependencies (which have currently no time constraints). This is possible since different dependencies do not conflict.

Assume that packet descriptions of several testcases were sorted to three different dependencies (see Figure 3.6). The first packet description of dependency 1 contains a time constraint. The time constraint states that fwtest has to wait 10 seconds, before it can send the second network packet of that dependency.
Depending on the chosen procedure the testcases will be processed as depicted below:

A
- Send *A*→*B*, *X*→*J* and *L*→*K*
- Wait 10 seconds, do nothing
- waiting ...
- waiting ...
- Send *C*→*D*, *V*→*X* and *O*→*Q*
- Send *E*→*F* and *J*→*Z*

B
- Send *A*→*B*, *X*→*J* and *L*→*K*
- Send *V*→*X* and *O*→*Q* (t-c n.f.)
- Send *J*→*Z* (t-c n.f.)
- Wait until t-c is fulfilled
- Send *C*→*D*
- Send *E*→*F*

(t-c n.f. = time constraint not fulfilled jet)

Procedure A just waits until the time constraint is fulfilled, before continuing. In contrast to A, B processes independent testcases of dependency 2 and 3 (during dependency 1 is blocked by the time constraint). In the mean time, the time constraint of dependency 1 decreases more and more. After having processed all testcases/packets of dependency 2 and 3, procedure B waits until the remaining time constraint is fulfilled (if necessary) and processes the remaining packets of the first dependency.
B processes independent testcases while the time constraint of a certain testcase is not fulfilled. Therefore the overall processing time will be smaller (or equal, in a worst case scenario) than the one of A, because A does not process other dependencies while the time constraint is not fulfilled.

What follows is a more general comparison of the two procedures, naming their advantages and disadvantages:
Procedure A: In a worst case scenario, which containts a time constraint in every processing step, the overall processing time becomes huge. The advantage of this procedure is that all testcases are executed in a deterministic order. This means that the test run can be repeated easily.
Procedure B: The higher the number of dependencies, the higher the propability that there are some dependencies without time constraints. While processing these dependencies, the time constraints of not fulfilled dependencies decrease. The propability increases that some time constraints are fulfilled, after having processed the other dependencies without time constraints. The disadvantage is that the testcases are not processed in a deterministic fashion. It is possible but very complicated to repeat the test run so that all packets are sent in the same order as it was done in the first test run.
Because of huge possible time savings, Procedure B is chosen and will be implemented.

3.2.3 Conclusion

Section 3.2 showed what has to be taken into account when several testcases should be processed in parallel. Only independent testcases can be processed at the same time. Furthermore, when processing two dependent testcases one after the other, time constrains have to be respected. Preventing concurrent processing of dependent testcases and getting deterministic results can be achieved by

- grouping together dependent testcases,
- introduce time constraints between two dependent testcases, and
- enforcing these time constraints at run time.

Chapter 4

Implementation

This chapter describes the implementation of the design decisions taken in Chapter 3. In more detail, the estimation of the timer value (Section 4.1), the handling of delayed network packets (Section 4.2) and the parallelization of testcases (Section 4.3) is described.

4.1 Timer Value Estimation

This section describes data structures and functions added to fwtest and those which have been modified to provide the necessary functionality for the timer value estimation.

4.1.1 Measuring

For nearly all calculations two points in time are needed:

- the sending time of a network packet
- the receiving time of a network packet (if not blocked by the firewall)

These points in time are stored in the corresponding `event` item. To be able to store these points in time, the `event` structure was expanded as follows:

```
struct timeval t_send, t_recv;
```

Moreover, during the sending process, three points in time are stored in the `progvars` structure (pv):

- The time, before fwtest starts to create the network packets.
- The time, after fwtest finshed creation of the network packets.
- The time, when all network packets have been sent.

These timestamps are used to determine the processing time of different functions of a single timestep, during a test run. The this, the `progvars` structure was extended by three structs of type `timeval`.

Points in time are measured with the help of the function `gettimeofday`. The function has a resolution of 10^{-6} seconds (on a linux system). This is sufficient for the intended purpose of measuring the traveling time of network packets (which is usually somewhere between 10^{-5} and 10^{-2} seconds).

Just before a network packet is sent, `gettimeofday` is used to store the current time in the corresponding `event` item (sending time of the packet). This is done inside the function `process_send_events`. An incoming network packet is timestamped by the receiving routine. Because received network packets are deleted after fwtest has analyzed them, it is necessary to copy the timestamp information to the corresponding `event` item. The `event` item which corresponds to a received network packet is searched as follows: The ID of a network packet is

compared against the IDs contained in the `event` items of the current timestep. If they match, network packet and `event` item correspond to each other. The receiving time of the network packet is then copied to the corresponding `event` item.
Furthermore, two flags are used to indicate whether the timestamp information is valid or not. These flags are set when a network packet of the corresponding `event` item is sent and received respectively by fwtest: The flag `t_marked_send` is set inside `process_send_events`, the flag `t_marked_recv` is set inside `cmp_tcp_udp_pkts` or `cmp_icmp_pkts`.This ensures that only `event` items with proper time information are analyzed.

4.1.2 Evaluation

The measured data is analyzed at the end of a timestep. When `process_recv_events` has finished, the function `evaluate` is invoked. `evaluate` creates a new instance of type `t_statistics`, inserts it in the linked list of already existing instances, and stores the calculated time values (described in Section 3.1) in this structure. The start of the linked list can be reached by `pv.t_info.t_stat_start`, its end by `pv.t_info.t_stat_end`. Every instance contains information about one timestep in which at least one network packet was sent and also received by fwtest.
Most of the information derived by `evaluate` is based on the two structs `t_send` and `t_recv`, mentioned above. `evaluate` will only analyze those `event` items in which the flags `t_marked_send` and `t_marked_recv` are set. An example is given in Figure 4.1: The packet was sent at 101.01532 seconds and received at 103.923555 seconds. Both, send and receiving flag are set and therefore *evaluate* can assume that `t_send` and `t_recv` contain proper values.

Figure 4.1: Information Contained in the Event Structure

It can be distinguished between three types of values, which are evaluated and stored by the function `evaluate`:

- local values,
- global values and
- memory values.

local values: Only data of the current timestep is used to calculate such values. Examples are the minimum traveling time of the network packets of the current timestep or the mean of the traveling time.
global values: Evaluated data of the current timestep is compared to data of previous timesteps. Examples are the maximum traveling time of a network packet valid for the whole test run or the maximum delay introduced by the firewall so far.
memory values: Data which describes the tendency of certain values. Those values are calculated by taking into account values of the previous timesteps. How many previous timesteps should be used for the calculations can be specified by the user (in `config.h`). Examples are the average of the maximum traveling time or the average of the delay introduced by the firewall.

4.1.3 Estimation

In fwtest v2.0, a timer value is estimated by calling `get_new_timer_value` at the beginning of a timestep. The function estimates a new value based on evaluated data of previous timesteps. Because all time statistics are standardized to one packet, the function first has to count the number of network packets to send.
The four timer estimations, described in Chapter 3, are implemented and can be chosen by the user (at the command line, when calling the shell script `run_fwtest.sh`). Furthermore, the user can tell fwtest to use `TIMING_DEF` or `TIMING_MIN` for every timestep of the testrun. These values can be preset by the user in `config.h`.
Note: Because no evaluated data is available in the first timestep, the timer is set to `TIMING_DEF` here (specified in `config.h`). The user can also specify a timer value for the last timestep (`TIMEOUT_LAST`). This value must be chosen sufficiently large. Otherwise fwtest is not able to capture all network packets. This is because there is no further timestep which can be used to receive delayed packets.

4.2 Finding Delayed Packets

The existing `analyze` function of fwtest v1.0 works as follows: It first compares the packet descriptions of the current timestep with the network packets in the receiving queue. Irregularities are logged (such as a network packet in the receiving queue which was assumed to be dropped according to its packet description). The `analyze` function will then log all remaining network packets of the receiving queue as unknown packets.
In fwtest v2.0, `check_for_delay` is placed between these two processing steps. It detects delayed network packets. Therefore the IDs of the network packets are compared with the the IDs of the `event` items of previous timesteps. Testcase and packet number of the delayed network packet are stored and written to the logfile (by `delayed_packets_to_log_file`). Also the timestep in which the packet was received (and detected) is stored by the same function. These network packets are removed from the receiving queue. This is done to prevent `analyze` to log them as unknown network packets.

4.3 Parallel Processing of Testcases

This section describes functions and data structures which were added to process testcases in parallel. During the parsing process, dependent testcases are grouped together and time constraints are assigned. At run time, the internal structure[1] of fwtest is built up step by step and time constraints are enforced.

4.3.1 Operation Mode

In fwtest v2.0, the user defines how testcases are processed. If fwtest should process testcases in parallel, the user has to choose the parallel operation mode. If the user wants fwtest to process the testcases as defined in the tp-file, the normal operation mode has to be chosen (in `config.h`).
Depending on the operation mode, chosen by the user, the parser will either create the internal structure (for the normal operation mode) or will call `build_dependency_list`. In the later case, `build_dependency_list` receives a pointer, pointing to the packet descriptions of a testcase and the corresponding testcase number. `tpparser.y` was modified accordingly.

4.3.2 During the Parsing Process

If the user decides to use the parallel operation mode, `build_dependency_list` will group together dependent testcases and assign time constraints. The function is called by the parser for every testcase of the tp-file. `build_dependency_list` receives a linked list of `event` items

[1] see 1.1 on page 1

and the testcase number of this testcase. The processing steps, `build_dependency_list` now performs, can be summarized as listed below.

- Reverse linked list of `event` items.
- Create linked list of `event_item` items.
- Find (or create) `dependency_node` item.
- Add linked list of `event_item` items to found (or created) `dependency_node` item.
- Do a protocol specific analysis of the current testcase to determine a possible time constraint for the current testcase.

The goal is to build a structure in which dependent testcases are grouped together. Further more the structure contains time constraints between two dependent testcases. At run time, the structure is used to determine independent testcases which can be processed in parallel. The time constraints are used to ensure that two dependent testcases do not influence each other.

Reverse linked list of `event` items

The linked list of `event` items, received from the parser, contains the packet descriptions of the testcase. It is necessary to reverse this list because it is received in the wrong order: The first `event` item in the linked list describes the last packet of the testcase, the last `event` item describes the first packet of the testcase. For easier further usage, the `event` item list is reversed (the first `event` item now describes the first packet of the testcase, the last `event` item now describes the last packet of the testcase). This is done by `re_order_event_list`.

Create linked list of `event_item` items

An `event_item` item is attached to every `event` item of the linked list. It is used to store a possible time constraint of the `event` item. Time constraints are assigned during the protocol specific analysis at the end of `build_dependency_list`. At run time, a time constraint of a `event` item will delay the sending of the network packet, described in the next `event` item, until the time constraint is fulfilled. To store a possible time constraint, the `event_item` structure contains a `timeval` struct. The `event_item` items are interconnected to a linked list. At run time, this linked list will be traversed instead of the linked list of `event` items. The linked list is created by `build_event_item_list`.

Find (or create) `dependency_node` item

Every `dependency_node` item describes one specific dependency. A `dependency_node` contains two `packetinfo` items. Every `packetinfo` item describes one direction of the dependency, e.g. for TCP:

- Direction 1: 192.168.1.10:80 → 192.168.2.10:2555
- Direction 2: 192.168.2.10:2555 → 192.168.1.10:80

A `dependency_node` item points to the beginning of the linked list of `event_item` items. This list contains all testcases which match the dependency described by the `dependency_node` item.
The `packetinfo` field of the first `event` item of a testcase is compared with the `packetinfo` fields of the dependency nodes (by `search_dep_node`). A match indicates that a dependent testcase was seen before. If there is no match among all `dependency_node` items, the currently processed testcase is independent to all previously seen testcases.
If a matching `dependency_node` item was found, the linked list of `event_item` items is attached to this `dependency_node` item (attaching the current testcase to the dependency). If no matching `dependency_node` item was found, one is created and initialized. Then the linked list of `event_item` items is linked to the dependency node.

Note, that the current implementation only compares the first `event` item of the linked list with the `packetinfo` fields of the `dependency_node` items. Therefore every testcase may only describe one connection. Furthermore this implementation requires that the network protocol of all packet descriptions is the same within on testcase.

Protocol Specific Analysis

Protocol specific functions (such as `tcp_evaluation`) are used to evaluate testcases and assign the necessary time constraints. Recall that the time constraint values can differ for different types of firewalls (Section 2.2). The user can configure these time constraint values (in `config.h`).

4.3.3 Example

The following example demonstrates how `build_dependency_list` transfers the information of a tp-file into its own internal dependency structure. This internal dependency structure is later used to determine, which network packets are sent next.
To the left side of Figure 4.2 the tp-file is depicted, on the right side the final internal dependency structure is shown. How this internal dependency structure was built, is explained next:
Testcase 1 is the first testcase which is processed by the parser and `build_dependency_list`. Therefore, a new `dependency_node` item is created and initialized. The content of the `packetinfo` struct of the packet description is copied to the one of the `dependency_node` item. Furthermore, during the protocol specific analysis, a time constraint is placed inside the `timeval` item of the `event_item` item.
Next testcase 2 is processed. The packet is compared against the dependency description of the previously created `dependency_node` item. It is an ICMP echo reply with the same ID and sequence number. It is directed in the opposite direction of the ICMP echo request. Therefore, testcase 1 and testcase 2 are dependent and testcase 2 is added to dependency 1. In this case, no time constraint is needed, since an ICMP echo reply cannot be answered.
The packet of testcase 3 is processed next. No matching `dependency_node` item is found, therefore a new item is created and initialized. The packet is added to the new created dependency. The protocol specific analysis function finds a time constraint which is added.
Testcase 4 is processed at the end. No matching `dependency_node` item is found and therefore a new item is created and initialized. The packet is added to the new created item. No time constraint is assigned here (since a RST packet immediately resets the connection).

Figure 4.2: Building the Internal Dependeny Structure

4.3.4 During the Test Run

Due to the time constraints, it is not possible to schedule the network packets of the testcases at the beginning. This has to be done at run time. Furthermore, time constraints have to be enforced at run time to guarantee proper test results.
Running in the parallel operation mode, network packets of the next timestep must be determined on the fly. `prepare_timestep` will determine the `event` items which should be sent in the next timestep. These `event` items are added to the internal structure (Section 1.1). In this way, fwtest's internal structure is built step by step, only adding the `event` items of the next timestep each time. Creating the internal structure means that for the sending, receiving and analysis processes the already existing functionality of fwtest v1.0 can be used.
A time constraint of a certain `event` item is enforced by `prepare_timestep`. As long as the time constraint of that `event` item is not fulfilled, `prepare_timestep` does not add that `event` item to the list of `event` items for the next timestep. If `prepare_timestep` cannot find a least one `event` item without a time constraint, it will interrupt, until the smallest time constraint is fulfilled.

Time Constraint Fulfilled ?

`update_delta` checks if the previous sent network packet had a time constraint (stored in the corresponding `event_item` item). If there is a time constraint, the time constraint is added to the receiving time of the previously sent packet. The sum of this is stored inside the corresponding `dependency_node` item. For example:
The last network packet was received at 15:31:05 and has a time constraint of 30 seconds. Therefore, the time constraint stored now inside the `dependency_node` item will be 15:31:35. If the previously sent packet does not have a receiving time (e.g. it was dropped), the current time is used as reference instead.
Next, the `timeval` item of the `dependency_node` item is compared against the current time. If the current time is larger than the time stored in the `timeval` item, the next packet of the `dependency_node` item can be added to the linked list of `event` item, which will be sent next.

Maximum Number of Concurrent Packets

The user can specify the maximum number of packets to be sent in one timestep (in `config.h`). Assume, for example, that there are 20 `dependency_node` items and the user set `MAX_NUMBER_PACKETS` to 10. Assuming that no `dependency_node` item has a time constraint, `prepare_timestep` will create a linked list with the first 10 `event` items in the first run. In the following run, `prepare_timestep` will create a list with the second 10 `event` items. Hence, the 20 network packets are sent in two successive timesteps.

No `event` Items Without a Time Constraint

If all `dependency_node` items contain time constraints, no `event` items can be added to the sending list. If this happens, `prepare_timestep` determines the smallest time constraint. The smallest time constraint is the one which will be fulfilled first (regarding time). Now a timer is set, which timeouts right after the smallest time constraint has been fulfilled. Then at least one `event` item can be added to the sending list. At least one network packet can be sent then in the next timestep.

Chapter 5

Evaluation

In this chapter we evaluate whether the different implementations meet the requirements of the design. The two most important questions are: Does the implementation do what was derived and proposed in the design? How well does the designed and implemented functionality improve the processing time of fwtest?
This chapter describes what kind of tests were performed and how these tests were performed. Furthermore, the results of these tests are presented.

5.1 Test Setup

Tests during the implementation phase were done inside a VMware[1] environment. Very soon problems regarding timing were detected. Time measurements inside the VMware environment did not correspond to the real time clock. Because parts of the implementation heavily rely on correct time measurements, the final tests were done on two computers, one running fwtest, and the other one running an iptables based firewall.

5.2 Testing

Tests were performed during the implementation process and after it was finished. Tests during the implementation concentrated on the correct content of data structures, calculated results and proper exception handling. Tests at the end evaluated the correctness and performance of the implementations.
In the following, tests performed at the end and their results are described in more detail. This is done separately for timer estimation, detection of delayed packets and parallel processing of testcases.

5.2.1 Timer Value Estimation

The timer value estimation is based on data which is measured, evaluated and saved before. It is therefore crucial that this data collection works as intended. This was successfully verified during the implementation phase by looking at the content of the data items.
The performance of the 4 timer value estimation functions (Section 3.1) was tested with a large tp-file. It contained 1344 network packet descriptions inside 225 testcases, the highest packet id was 2244. The network packets were sent in 1340 different timesteps (not every packet id between 1 and 2244 was used for the packet descriptions). All network packets, specified in the tp-file, were TCP packets. Here, all tests were performed using the normal operation mode.
All 4 estimation functions needed about 30 seconds to process the tp-file. At maximum, 3 network packets were delayed, no network packets got lost. This is a huge performance increase compared to fwtest v1.0, which needed about 2300 seconds (about 38 minutes) to process the same tp-file.

[1] VMware Workstation, see `http://www.vmware.com`

Testing the estimation functions more detailed was very difficult, since the traveling time of network packets did not vary very much in the test setup. At a first glance, one may think of stressing the firewall, e.g. by a cpu consuming application, to delay network packets. Such an application is usually run as an user space process. In contrast to that, netfilter[2], responsible for processing and forwarding the network packets, runs in the kernel space. Since priorities of kernel space processes are higher than the ones of user space processes, stressing with cpu consuming applications does not help. One way to delay network packets is described in Section 6.2. Due to a lack of time, this method could not be used in this evaluation.
Because of the low variance, the assumed disadvantage of estimation functions 1 and 2 (more delayed packets, if the runtime suddenly increases) could not be verified. Estimation function 3 never decreased or increased its estimated timer value, since the traveling time of the network packets did not vary enough to trigger a change. Estimation function 4 is more sensitive, therefore an increase and decrease of the estimated timer value could be observed. This showed the ability of estimation function 4 to adapt to changes during runtime.

5.2.2 Detecting Delayed Network Packets

To detect delayed network packets, one first has to produce delayed network packets. They were produced by using a timer value of 10^{-6} seconds and by increasing the priority of fwtest (using the `nice` command):

```
# nice --19 ./run_fwtest ...
```

Delayed network packets were successfully detected. A packet is delayed if it sent in one timestep and received in a successive timestep. By comparing the timestep of the packet description of a network packet and the timestep which is currently processed, a delayed network packet can easily be detected. It was verified that for every delayed network packet an entry in the log file was created:

```
17:21:31.026935: packet [3.2/2]  delayed, found it in timestep 3
```

The network packet of testcase 3 (`[3.2/2]`) was sent in timestep 2 (`[3.2/2]`), but found in timestep 3. The last number (`[3.2/2]`) is the packet ID (as described in the tp-file).

5.2.3 Parallel Processing of Testcases

When testcases are processed in parallel, it is crucial that dependent testcases are detected, proper time constraints are assigned and that these time constraints are enforced during runtime.
Different tp-files were used to test, if the dependency analyze function is capable of detecting dependent testcases. The function was able to detect dependent testcases and successfully grouped them together in the same dependency. It was also tested if the function detects time constraints and if the appropriate time constraint values were assigned. This was successfully verified.
While fwtest was processing the testcases in parallel, it was checked if all network packets are sent in the right order and if time constraints are really enforced. This was done with the help of the network sniffing program `tcpdump`[3], which was used to capture the network packets at the firewall. Network packets were sent in the correct order and all time constraints were enforced by fwtest.
All mentioned tests were performed for TCP, UDP and ICMP echo.

Example

Since the two packets of the tp-file depicted below are dependent, they must not be processed at the same time. Furthermore, after the first packet, a sufficient large time constraint has to be assigned and enforced, so that the firewall forgets about the first network packet.

[2] the kernel space part of iptables
[3] `http://www.tcpdump.org`

```
testcase 1 {
        packet  1 { icmpecho send { ipa ipb ECHO_REQUEST 10 910 } receive {} }
}
testcase 2 {
        packet  1 { icmpecho send { ipb ipa ECHO_REPLY 10 910 } receive {} }
}
```

The network packets of the two testcases are grouped together under dependency 1 and a time constraint (`delta`) of 2 seconds is assigned to the first network packet (this is shown in the `Dependency Information`):

```
# Dependency Information
  Dependency 1:
  TC: 1.1 10.0.0.5 -> 192.168.100.6  type: 8  ID:10  Seq:910  delta: 2.000000
  TC: 2.1 192.168.100.6 -> 10.0.0.5  type: 0  ID:10  Seq:910  delta: 0.000000
```

(**1.**1 is the testcase number, 1**.1** is the packet ID).
After the first network packet has been sent, fwtest pauses to enforce the time constraint. This can be seen by looking at the output of `tcpdump` (recorded at the firewall):

```
11:42:28.751365 IP 10.0.0.5 > 192.168.100.6: ICMP echo request, id 10, seq 910, length 8
11:42:30.759198 IP 192.168.100.6 > 10.0.0.5: ICMP echo reply, id 10, seq 910, length 8
```

The second network packet was captured more than 2 seconds later, which indicates that the time constraint was successfully enforced. Approximately , this information can also be found in the timing statistics of the log file:

```
# Timing Statistics
# [start time]          [ts]      [timeout]      [min]          [max]

11:42:28.750850          0        1.000000       0.000194       0.000194
11:42:30.758814          1        1.000000       0.000157       0.000157
```

The two network packets were logged approximately two seconds one after the other.

5.3 Conclusion

The evaluation of the extended version of fwtest (fwtest v2.0) showed, that a lot of time can be saved with the new timer value estimation functionality. In case of a too short chosen timer value, delayed packets are successfully detected.
Altough not demonstrated here, it is obvious that even more time can be saved by interleaving testcases. The tests showed, that the interleaving functionality successfully detects dependencies and take the appropriate actions to guaranty correct test results (by reordering packets and by enforcing time constraints).

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Working at this semester thesis was very interesting. First of all, it was a good exercise how to incorporate into existing source code of fwtest. I got familiar with the design of fwtest and figured out, which parts of the source code are relevant for me.
I recognized again, how well the VMware Workstation software can be used for development tasks like the one of this thesis but also saw one weakness of this software (the deviation of clocks inside VMware environments).
But most of all, I liked the process of thinking about a problem, finding ab (hopefully) appropriate design, see whether it is possible to implement the design, and see if the implementation works as intended.

In the first part of this semester thesis, I designed and implemented estimation functions for the timer value. Here, the challenge was to determine which data can be used for this estimation. I designed 4 different estimation functions and implemented them. All functions use data, which is in some way based on the traveling time of the network packets, sent and received in previous timesteps. Using the traveling time as information source for the current situation has been a good design decision. As I could show during the tests, at least estimation function 4 is able to detect changes in the current situation and react appropriately. More tests would have been necessary to really verify the assumed advantages of estimation function 4 in comparison to estimation function 1 to 3. Nevertheless, the tests demonstrated that my implementation dramatically reduces the time, needed for testing a firewall. Since it is in the nature of estimations, sometimes the estimation of the timer value is wrong, the value is too short and network packets become delayed. But these network packets are detected and documented in the log file. This information gives the user the possibility to manually re-run the corresponding testcases.

The second part of this semester thesis was dedicated to the design and implementation of processing testcases in parallel. At the beginning, this task seemed to be easy. But it was not. During the design phase I realized, that one must be very careful, if fwtest is used to test stateful firewalls. As I encountered, stateful firewalls keep track of the state of connections. Firewalls use this information as a base for further decisions, made for following network packets of a connection. This can be problematic, since firewalls temporally store this information, and the information is eventually used for network packets of different testcases. Wrong test results are the consequence. But I figured out, that this information about connections is removed after a certain amount of time. This insight was the solution to the problem. I had to schedule different, problematic testcases with a sufficiently large time period in between. The implementation was very time consuming, since testcases must be analyzed in much detail. Furthermore, fwtest must ensure, that two problematic testcases are not processed in parallel or too close together in time. I also spent a lot of time on implementing this functionality. I was able to verify, that the implementation meets the design specifications and works as intended. Some restrictions remain (due to a lack of time): The current implementation is only able to handle testcases, describing a single connection. Also, the network protocol of all packet descriptions inside a

testcase must be the same. Besides TCP and UDP, the current implementation only partly supports ICMP messages.

In my opinion, taking into account stateful firewall behavior and recognizing its consequences, was the most important contribution of this semester thesis. The user does not have to pay attention to that aspect any longer when writting tp-files. fwtest v2.0 automatically detects critically testcases and introduces time constraints to guaranty an independent processing of testcases. If the stateful firewall behavior is not considered, wrong test results can occur, which will be very hard to find. Or the user might get a wrong picture of the tested firewall. Wrong test results still can occur in the case of delayed network packets. But these network packets are written to the log file. The user should carefully analyze it, before making statements about the tested firewall.

6.2 Future Work

As mentioned in the previous section, functions still need to be tested more exhaustive. Others must be implemented or at least the functionality must be extended.
To verify the assumed advantages and disadvantages, the estimation functions must be tested more deeply. It is necessary to perform tests with varying traveling times of the network packets. The traveling time can be influenced by intentionally delaying network packets. For linux, there exists a kernel module called `netem`[1] which can be used to delay packets. The kernel module offers different options to delay network packets, introduce randomness to the assigned delays and much more. It seems to be the perfect testing tool regarding timing aspects.
It would be desirable that fwtest re-runs testcases of delayed network packets automatically. This was already explained in detail in a previous semester thesis ([Sch06]).
When processing testcases in parallel, the current implementation does not support ICMP messages other than echo request/reply. Extending fwtest accordingly will not be difficult, since also these ICMP messages were considered in the design and partly in the implementation. Simplified, only a few functions must be extended, responsible for comparing the network packets and assigning time constraints.
The current implementation is also not able to process a testcase, containing several connections and/or protocols. This limitation can be resolved by adding additional data structures to the entity, describing a dependency.
Besides these smaller improvements, in my opinion, it is very important to attack the following issue: Todays firewalls are not only able to monitor single network connections with their connection tracking facilities. They are also able to track application layer connections. These connections often use several network connections. As we know, a network packet may change the state of the connection at the firewall. But network packets of application layer connections can change the state of other network connections, too! This means, that all these different possible network connections of an application layer connection must be considered by the dependency analysis. In my eyes, extending fwtest accordingly poses a big challenge. The first step will be to understand, how firewalls keep track of application layer connections. Analyzing the modules of iptables, which implement these functionality, might be a good place to start.

[1] `http://linux-net.osdl.org/index.php/Netem`

Appendix A

README

```
================================================================================

    FWTEST v2.0 (April 2007)
    Author:       Gerry Zaugg <zauggge@gmail.com>
    Contributors: Adrian Schuepbach <scadrian@student.ethz.ch>
                  Beat Strasser <b8@student.ethz.ch>
                  Gabriel Mueller <muellega@ee.ethz.ch>
    Maintainer:   Diana von Bidder <diana.bidder@inf.ethz.ch>

================================================================================

0.  Content

    1. Description
    2. Software Requirements
    3. How to Perform a Test Run
       3.1 Test Environment
       3.2 Mandatory Files
       3.3 run_fwtest.sh
       3.4 Test Packets File
           3.4.1 General layout
           3.4.2 Protocol specific fields (PROT_FIELDS)
           3.4.3 Preprocessor
           3.4.4 Variables
           3.4.5 Parse errors
       3.5 Timeout
       3.6 Operation mode
       3.7 Running Fwtest
       3.8 Example
    4. Source Files



1.  Description

Fwtest v2.0 provides a simple, portable interface to perform firewall testing.
It is executed on a so-called testing host that sends test packets through a
firewall. Fwtest crafts and injects the test packets and captures them if they
pass the firewall. When receiving packets, fwtest performs an exhaustive
analysis revealing irregularities (i.e. packets that are accepted by the
firewall although they were expected to be dropped or packets that are
discarded/changed although they were expected to be passed). The irregularities
are logged and serve as source of information to identify failures.

Fwtest v1.0 was a further development of fwtest v0.5 which evolved in the
context of a Diploma Thesis by Gerry Zaugg. Besides an extension to UDP and
ICMP, the new version has experienced some underlying restructuring in order to
fit for NAT. For more information, read the corresponding documentation(s):

Fwtest v2.0 adds further functionality to Fwtest v1.0: Version 2.0 is able to
addapt its processing speed to its current environment (regarding delays
introduced by the network and the tested firewall).
Further more, v2.0 can automatically reorder testcases and recognizes (and
resolves) dependencies between testcases. This is necessary to guaranty
correct test results.

[1] Gerry Zaugg. Firewall Testing. January 2005.
    http://www.infsec.ethz.ch/education/projects/archive/Bericht_Gerry.pdf
[2] Adrian Schuepbach. Testen von Firewalls mit NAT. February 2006.
    http://www.infsec.ethz.ch/education/projects/archive/Bericht_Adrian.pdf
[3] Beat Strasser. Eine UDP-/ICMP-Erweiterung fuer fwtest. February 2006.
    http://www.infsec.ethz.ch/education/projects/archive/Bericht_Beat.pdf
[4] Gabriel Mueller. Timing ... in Firewall Testing. April 2007
    http://www.infsec.ethz.ch/education/projects/archive/Bericht_Gabriel.pdf



2.  Software Requirements

Fwtest v2.0 is known to compile and run under Linux Debian 3.0 with the 2.4.24
kernel, Linux Red Hat 7.3 with the 2.4.18-3 kernel, Gentoo Linux 2005.1 with
the 2.6.14 kernel and Ubuntu 6.10 with the 2.6.17 kernel.  Probably, it also
runs under OpenBSD, FreeBSD and NetBSD.
```

You will need flex, bison, gcc, make, libc and libc-dev to build fwtest. We
make use of m4 and the administration tools iptables or ipchains.

We need the following libraries:
* libpcap-0.7.2
* libnet-1.1.2.1
* libdnet-1.8
* libc

Libpcap, libnet and libdnet have to be installed from source since we use the
header files (pcap.h, libnet.h, dnet.h).

NOTE: Run /sbin/ldconfig before using libdnet. ldconfig creates the necessary
links and cache (for use by the run-time linker, ld.so) to the most recent
shared libraries (for more information: man ldconfig, some times it was
necessary to invoke ldconfig as follows: # ldconfig /usr/local/lib).

3. How To Perform a Test Run

Disclaimer: Do not use fwtest in a productive environment.

3.1 Test Environment

This Section gives an overview on how a firewall test is performed with fwtest.

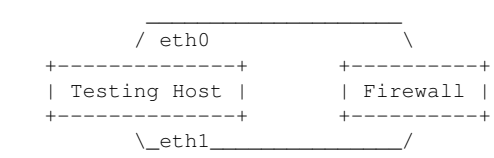

Figure 1: Sample test environment

We assume that you have set up a test environment similar to the one illustrated
in figure 1: A testing host is connected to a firewall via two network devices.
Fwtest will run on the testing host and craft, inject, capture and analyze
packets as well as log irregularities.
You have to provide a test packet file holding the specifications of the test
cases (consisting of several test packets) that fwtest will generate, inject and
capture. The layout of the this file is explained in Section 3.4.

3.2 Mandatory Files

We now discuss how to run fwtest on a testing host.

Copy the following files into the test directory:

(1) Source files: Listed in the last Section of this file.
(2) run_fwtest.sh: Fundamental shell script performing the test run.

3.3 run_fwtest.sh

run_fwtest.sh leads you through the process of firewall testing.
You have to be root when executing the script. Furthermore, you must provide
some arguments when starting the script. The test run will then be performed
automatically.

run_fwtest.sh requires at least seven arguments:
(1) Test packets file
    File containing the test packets
    Read more about the file format below.
(2) Log file
    File where the irregularities are reported
(3) Network 1: IP and mask in CIDR notation
    Specifies the first network that the testing host represents.
    (e.g. 192.168.1.0/29)

```
(4) Network 1: interface
    The network interface to capture the packets from network 1 (e.g. eth0)
(5) Network 2: IP and mask in CIDR notation
    Specifies the second network that the testing host represents.
    (e.g. 172.16.70.0/29)
(6) Network 2: interface
    The network interface to capture the packets from network 2 (e.g. eth1)
(7) timer value estimation function
    Values 0-5 (see Section 3.5)
(8) Option -p
    Transmit packet id in payload (TCP/UDP only*)
(9) Option -u
    Interpret TCPUDP packets as UDP instead of TCP

* ICMP packets usually don't allow to send a payload. The option -p covers only
TCP and UDP packets; the ICMP packet id is always sent in the IP header. Note
that the IP identification field is only 16 bit - if you deal with more than
2^16 packets including ICMP packets and together with the payload option, you
will certainly run into problems: statistics may be misinterpretable, so be
warned.

Before fwtest can be called, you must make sure that the gateway's MAC address
is contained in the local system's ARP cache. To achieve this, you may ping the
gateway on each device while the firewall/gateway is (still) accepting ICMP
packets.

Example how run_fwtest.sh may be called:
./run_fwtest test1 test1.log 192.168.72.0/29 eth0 172.16.70.0/29 eth1 2 -u

run_fwtest.sh executes the following steps:
(1) Check the arguments for validity.
    Exit if one of them is bad or not specified.
(2) Seek for the required programs, libraries and header files.
    Exit if one of them is missing.
(3) Compile fwtest if necessary.
(4) Set up local firewall rules to drop all incoming packets so that the testing
    host does not response to the packets it captures. This can only be done
    if either iptables or ipchains is installed on your system.
(5) Run fwtest.
    - Run a preprocessor on the packets definition file.
    - Depending on the operation mode (see Section 3.6) the sending order of
      packets is changed.
    - Testing is performed (i.e. packets are crafted, injected,
      captured, analyzed) and the irregularities are logged.
    - Fwtest will print some useful information, especially warnings and errors.

    To make this point clear: fwtest is the firewall testing tool that
    performs testing whereas run_fwtest.sh is only a shell script that
    prepares the testing host for the test run and compiles and runs fwtest
    for you. Read more about invoking fwtest in Section 3.5.
(6) Remove the local firewall rules.
(7) Exit successfully.

The irregularities are stored in the log file. Evaluate them to identify
problems and abnormalities.


3.4 Test Packets File

3.4.1 General layout

The general layout of a file containing test packet specifications looks like
this:

testcase I {
  packet K { PROTOCOL send { PROT_FIELDS } receive { PROT_FIELDS } }
  packet L { PROTOCOL send { PROT_FIELDS } receive ok }
  packet M { PROTOCOL send { PROT_FIELDS } receive {} }
  packet N { PROTOCOL send { PROT_FIELDS } receive ? }
}
testcase J { ... }

I,J,K,L,M and N are integers which define the testcase/packet id.
```

Using operation mode 1 (see Section 3.6), testcases are run in parallel.
The packet id stands for a global timeslot, so for example the packet 1.2
(testcase 1, packet 2) is sent at the same time as packet 3.2.
Using operation mode 2 (see Section 3.6), testcases and their packet
descriptions are reordered by fwtest for time efficient testing. The packet id
is no longer used for determining the sending timeslot.

For each packet you have to declare the protocol type (see below) as well as a
send and a receive clause where protocol specific fields are specified. In the
receive part you specify which values you expect the packet to have when the
packet arrives at the destination host. "receive ok" is a shortcut for a receive
block with exactly the same values as in the send clause. You may also have an
empty receive clause {} if you expect the packet to be dropped by the firewall.
A question mark means you don't care about the firewalls reaction to this
packet.

3.4.2 Protocol specific fields (PROT_FIELDS)

Please read documentation [1] for further details on TCP and [3] for more
information on UDP and ICMP.

```
TCP (Transmission Control Protocol):
{ srcip dstip srcprt dstprt flags seqnr acknr }
    source ip, destination ip, source port, destination port, control flags (a
    combination of S/A/F/R/U/P), sequence number, acknowledgment number

UDP (User Datagram Protocol):
{ srcip dstip srcprt dstprt }
    source ip, destination ip, source port, destination port

ICMPecho (ICMP Echo request/reply):
{ srcip dstip type idnr seqnr }
    source ip, destination ip, type (0/8), identification number, sequence
    number

ICMPunreach (ICMP Destination unreachable):
{ srcip dstip code origid }
    source ip, destination ip, code (0-15), original packet id

ICMPredir (ICMP Redirect):
{ srcip dstip code gwip origid }
    source ip, destination ip, code (0-4), gateway ip, original packet id

ICMPtexc (ICMP Time exceeded):
{ srcip dstip code origid }
    source ip, destination ip, code (0-1), original packet id

ICMPtstamp (ICMP Timestamp request/reply):
{ srcip dstip idnr seqnr otime rtime ttime }
    source ip, destination ip, identification number, sequence number,
    originate timestamp, receive timestamp, transmit timestamp
```

3.4.3 Preprocessor

Fwtest v2.0 filters the given test packet file with a preprocessor (m4). So you
may define constants for often used IPs or ports. Fwtest includes by default the
file defined by the environment variable FWTEST_MACROS. run_fwtest.sh sets this
to macros.m4 which contains constants for possible ICMP types and codes.
It's possible to disable the preprocessor by setting the environment variable
FWTEST_USEM4 to 'no'.

3.4.4 Variables

You may use variables for IP and port numbers (not known at the beginning of
the test) in the packet specifications. This is of help for example in the case
where NAT is done at the firewall: we do not care about the rewriting of a
certain IP and port, but we want it to always be the same.
A variable will be set, to the observed value, by fwtest on its first
occurrence in a receive clause.

```
example:
testcase 1 {
  packet 1 {TCP send {1.1.1.1 2.2.2.2 2 22 S 60 -}
               receive {6.6.6.6 2.2.2.2 VarA 22 S 60 -}}
  packet 2 {TCP send {2.2.2.2 6.6.6.6 22 VarA SA 70 61}
               receive {2.2.2.2 1.1.1.1 22 2 SA 70 61}}
  packet 3 {TCP send {1.1.1.1 2.2.2.2 2 22 A 61 71}
receive {6.6.6.6 2.2.2.2 VarA 22 A 61 71}}
}
```

Here, VarA will be set to the value of the source port of packet 1 at its arrival. Then all further occurrences of VarA in this testcase will be set to this value.

Variable names have to start with an uppercase letter and may not be longer than 15 characters. Numbers can be used in variable names except for the first character. Variables are only allowed for IP numbers and for port numbers. A variable is valid in the scope of a testcase. Within a testcase, a variable is assigned only once and all packets that use the same variable, have to have the same value for the field the variable is used. In different testcases the same variable name is NOT the same variable, because it is another scope.

3.4.5 Parse errors

Whenever you get parse errors on a test packet file, the line number will be displayed. Whereas the number is correct for syntax errors, Fwtest v2.0 errs mostly on a semantical error e.g. the discovery of a duplicate packet id. Because packet specifications are internalized only at the end of a testcase, such errors are not detected until the end of the actual testcase.

Since the order of the packets is not relevant (only the packet id specifies the time slot), Fwtest reverses the packets to simplify matters because of the internal data structure; so, errors may be detected in the opposite order than they appear in the specification.

3.5 Timeout

A timer is startet by fwtest after the packets for a given timeslot are sent. When the timer expires after an estimated timeout value, fwtest steps to the next timeslot. Four different estimation functions (1 to 4) can be used. A new timer value is estimated for each timeslot. For further details, please refer to [4], Chapter 3.
Additionally, a default timer value (5) can be specified (in config.h) and used for every timeslot. It is also possible to use a minimal possible timer setting (0), most often used to provoke delayed network packets.
You must invoke the shell script (or fwtest directly) with one of the values, mentioned in the brackets. Recompile fwtest after modifications on config.h

3.6 Operation Mode

Two operation modes are available:
(1) Sends the network packets of the different testcases
    as it was specified in tp-file (->packet id).
(2) Before the test is started, fwtest re-orders the testcases
    and their network packets. fwtest will then process as many
    testcases as possible in parallel.
You must specify the operation mode in the configuration file config.h. Recompile fwtest after modifications on config.h

3.7 Running Fwtest

We briefly explain how to invoke fwtest without making use of run_fwtest.sh. fwtest expects the test packets file name as argument, and takes the following options:

```
        -l <log file>  Log file wherein the irregularities are stored
        -n <network>   Network 1: IP and mask in CIDR notation
        -i <interface> Network 1: interface to capture the packets.
        -m <network>   Network 2: IP and mask in CIDR notation
        -j <interface> Network 2: interface to capture the packets.
        -p             Transmit packet id in payload (neglecting ICMP)
```

```
        -u              Interpret TCPUDP packets as UDP instead of TCP
        -t              Specifies timer value estimation function
```

You have to specify the networks, the interfaces and the estimation function (-n, -i, -m, -j and -t).

Fwtest may be called like this (make sure you are root):

```
./main -l test1.log -n 192.168.72.0/29 -i eth0 -m 172.16.70.0/29 -j eth1 -t 1 test.tp
```

The test packets file (e.g. test.tp) has to be defined. You have no chance to perform a test without a test packets file. If the syntax of the file is invalid, fwtest will display an error message and quit instantly.

3.8 Example

Let's just show at a simple test run in a possible environment:

Test host setup:

```
ifconfig eth0 down
ifconfig eth1 down
ifconfig eth0 172.16.70.2 netmask 255.255.255.0 up
ifconfig eth1 192.168.72.2 netmask 255.255.255.0 up
route add -net 172.16.70.0/24 gw 172.16.70.1
route add -net 192.168.72.0/24 gw 192.168.72.1
```

Firewall setup:

```
ifconfig eth0 down
ifconfig eth1 down
ifconfig eth0 172.16.70.1 netmask 255.255.255.0 up
ifconfig eth1 192.168.72.1 netmask 255.255.255.0 up
echo 1>  /proc/sys/net/ipv4/ip_forward
echo 1>  /proc/sys/net/ipv4/conf/default/proxy_arp
```

Ping the firewall from the test host (the firewall is still accepting all incoming packets):

```
ping -c 1 172.16.70.1
ping -c 1 192.168.72.1
```

Install the firewall rules on the firewall (e.g. drop all packets, but forward tcp packets to 192.168.72.3:25):

```
iptables -F
iptables -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP
iptables -A FORWARD -p tcp -d 192.168.72.3 --dport 25 -j ACCEPT
```

Create a test file named 'test.tp' like this:

```
define(ipa, 172.16.70.3)
define(ipb, 192.168.72.3)
testcase 1 {
     packet 1 { TCP send { ipa ipb 4000 25 S 60 - } receive ok }
     packet 2 { ICMPunreach send { ipb ipa PORT_UNREACHABLE 1.1 }
                         receive {} }
}
testcase 2 {
     packet 1 { UDP send { ipa ipb 5005 domain } receive {} }
}
```

Now fire off fwtest by calling run_fwtest.sh:

```
./run_fwtest.sh test.tp test.log 172.16.70.0/24 eth0 192.168.72.0/24 eth1 1
```

After the test run is complete, the statistics will be displayed which report one packet as successfully forwarded (true_negative) and packets 1.2/2 and 2.1/1 as successfully rejected (true_positive). The notation is as follows:

```
<Testcase number>.<Timeslot number>/<Packet ID>
The timeslot number indicates the time when the packet was sent.



4.  Source Files

Makefile          - Makefile
main.c            - main program
tpparser.l        - lexer grammar
tpparser.y        - parser grammar
lpcap.c           - packet capturing routines
lnet.c            - packet crafting and injection routines
log.c             - log routines
util.c            - helper routines
symboltable.c     - a symbol table for the test packets file variables
semanticchecker.c - a semantic-checker for the test packets file
main.h            - main definitions & TIMEOUT definition
tpparser.h        - parse definitions
lpcap.h           - packet capture definitions
lnet.h            - packet crafting definitions
log.h             - log definitions
symboltable.h     - symbol table definitions
semanticchecker.h - semantic-checker definitions

timing.c          - measure timing aspects and estimate timer value
tdetection.c      - detect and log delayed network packets
dependency_list.c - re-order and process testcases in parallel
config.h          - configuration file
```

Appendix B

Task and Timetable

a

a

a

a

Bibliography

[DAR80] DARPA. User datagram protocol. RFC http://www.ietf.org/rfc/rfc768.txt, 1980.

[DAR81a] DARPA. Internet control message protocol. RFC http://www.ietf.org/rfc/rfc792.txt, 1981.

[DAR81b] DARPA. Transmission control protocol. RFC http://www.ietf.org/rfc/rfc793.txt, 1981.

[JP85] J. Reynolds J. Postel. File transfer protocol. RFC http://www.ietf.org/rfc/rfc959.txt, 1985.

[JR02] et al. J. Rosenberg, H. Schulzrinne. Sip: Session initiation protocol. RFC http://www.ietf.org/rfc/rfc3261.txt, 2002.

[KM] Stephen Donelly Klaus Maochalski, Jörg Micheel. Packet delay and loss at the auckland internet access path. http://www.wand.cs.waikato.ac.nz/old/wand/publications/pam2002-delay.pdf.

[Sch06] Adrian Schüpbach. Firewall testing with nat. http://www.infsec.ethz.ch/education/projects/archive/Bericht_Adrian.pdf, 2006.

[Str06] Beat Strasser. Extending fwtest to handle udp and icmp. http://www.infsec.ethz.ch/education/projects/archive/Bericht_Beat.pdf, 2006.

[Zau05] Gerry Zaugg. Firewall testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf, 2005.