Specification-based Firewall Testing

DISS. ETH NO. 17172

Specification-based Firewall Testing

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
DIANA VON BIDDER-SENN
Dipl. Informatik-Ing. ETH

born 14.11.1978
citizen of Zürich, Obersiggenthal AG, Basel, Genève

accepted on the recommendation of

Prof. D. Basin, Ph.D, examiner
Dr. G. Caronni, co-examiner
Prof. Dr. B. Plattner, co-examiner

2007

To my late grandfather
Werner Senn

Acknowledgements

First of all I want to thank all the people that nurtured my interest in mathematics: my late grandfather, my parents, Mrs. Fritschi, and Mr. Gamper. Without them I wouldn't be where I am. That I am still here and now finishing my dissertation is due to Germano and Paul S. — thank you for your support in hard times.

Then I would like to thank armasuisse for the interesting and challenging project I could work on, and also for the freedom they gave me. And I want to thank David and Germano for their supervision. Further, I want to thank all the students that helped me building a prototype tool to validate my approach: Gerry Zaugg, Markus Frauenfelder, Stefan Hildenbrand, Beat Strasser, Adrian Schüpbach, Lukas Brügger, and Gabriel Müller.

I want to thank all the people proofreading (parts of) my thesis: Paul Sevinç, Adrian von Bidder, Boris Köpf, Felix Klaedtke, Cas Cremers, Ivo Blöchliger, and Alexander Pretschner. And I want to thank my office-mate Jürgen Doser for all his help on mathematics and writing better understandable texts.

I quite liked being the only female researcher in our group. For this I want to thank all "my men" for all the interesting discussions during lunch or at the BQM. I also thank them for prodding me to bake more and better cakes.

And last but not least, I want to thank Adrian for his love, for listening to my problems and for staying longer in Zurich than he intended.

Contents

Part IV Appendix

List of Figures

Zusammenfassung

Firewalls sind heutzutage omnipräsent. Jedermann kennt das Wort, doch die Funktionsweise einer Firewall bleibt vielen verborgen. Viele sehen in Firewalls einen umfassenden Schutz, den diese aber leider nicht bieten können. So stehen zwar vielerorts Firewalls, die irgendwie konfiguriert sind, aber es kümmert sich niemand darum, ob diese auch tun was man von ihnen erwartet.

Der springende Punkt hier sind die Erwartungen, die man an eine Firewall stellt. Diese sind für verschiedene Benutzer und Umgebungen unterschiedlich: Eine Firewall in einer Bank muss einen anderen Zweck erfüllen als eine Firewall an einer Universität. Und auch innerhalb einer Bank gibt es mehrere Firewalls die unterschiedliche Zwecke erfüllen.

Um zu überprüfen, ob eine Firewall die an sie gestellten Erwartungen erfüllt, müssen die Erwartungen klar sein, zum Beispiel durch schriftliches Notieren in einer sogenannten security policy. Leider gibt es aktuell zwei Probleme, die dies verhindern: 1) security policies existieren oft nicht oder sind sehr informal gehalten. 2) es gibt keine Methoden um so eine Überprüfung durchzuführen.

Das Ziel dieser Dissertation ist es die obengenannten Probleme zu beheben: Das Ermöglichen eines automatisierten Firewall-Tests basierend auf einer formalen policy, soll die Leute motivieren policies zu schreiben. In dieser Dissertation wird also eine Methode zum spezifikationsbasierten Firewall-Testing entworfen.

Da wir die zu testenden Firewalls als Black-Boxes betrachten und auf Netzwerkebene testen, werden sowohl Firewall-Konfiguration als auch Firewall-Implementation getestet. Die Firewall-Konfiguration sagt aus welche Verbindungen erlaubt sind und wird vom Firewall-Benutzer geschrieben. Die Firewall-Implementation sagt aus was zu einer Verbindung gehört (Protokoll-Automaten) und wird vom Firewall-Hersteller geliefert. Unsere Methode gliedert sich in zwei Stufen, je eine für den Test von Konfiguration und Implementation.

Im ersten Teil der Dissertation wird ein Überblick über die gesamte Testmethode gegeben, mit Schwerpunkt auf dem Test der Firewall-Konfiguration. Zum Testen der Firewall-Konfiguration wird eine Sprache vorgestellt, die es ermöglicht security policies formal zu spezifizieren. Eine solche policy ist der Startpunkt unserer Methode, die daraus Testfälle generiert. Mit diesen Testfällen kann geprüft werden, ob eine gewisse Firewall die vorgegebene policy erfüllt. Da unsere Methode die zu testende Firewall als black-box betrachtet, funktioniert unser Ansatz mit jeder Firewall. Dies haben wir mit einer Fallstudie validiert.

Der zweite Teil der Dissertation befasst sich dann mit dem Problem, wie eine Firewall ein gewisses Protokoll behandeln soll. Dieses Wissen brauchen wir, um die Firewall-Implementation testen zu können. Wir zeigen auf wieso und inwiefern Firewalls sich von Endpunkten unterscheiden und was die Folge von fehlenden Protokollspezifikationen für Firewalls ist. Basierend auf dieser Analyse, schlagen wir einen Algorithmus vor, der Firewall-Protokollspezifikation aus Protokollspezifikationen für Endpunkte generiert. Wir

beweisen, dass eine Firewall, welche eine solche Spezifikation umsetzt, nur Pakete akzeptiert, die von protokoll-konformen Endpunkten stammen. Trotz der immensen Wichtigkeit dieses Themas, auch für Firewall-Hersteller, stellt die vorliegende Arbeit unseres Wissens nach die erste (systematische) Arbeit zu diesem Thema dar.

Die Kombination beider Teile dieser Dissertation ermöglicht also ein automatisiertes Prüfen der Erfüllung einer security policy durch eine Firewall (bestehend aus Regeln und Implementation). Damit haben wir das gesteckte Ziel erreicht, wie auch in der Fallstudie gezeigt wird.

Abstract

Firewalls are a central component in network security. They are widely deployed, unfortunately without having good means for determining whether they are accomplishing their job correctly.

To be able to determine if a firewall accomplishes its job correctly, we must know the expectations the firewall under test has to fulfil. These expectations need to be specified in a security policy, which is rarely done in practice at the moment.

The aim of this thesis is to develop a methodology for the specification-based firewall testing. We develop a method for testing given firewalls for conformance to a security policy. To achieve this goal, our method needs to be able to generate separate tests for every environment, based on the security policy of the respective environment.

A firewall consists of two main parts: implementation and configuration. Implementation is what is done by the firewall vendor. Configuration is what is done by the owner of the firewall. Simplified, the configuration states which connections are allowed, whereas the implementation states (as protocol automata) how such connections look like. As we do black-box testing at the network level, we have to both test the configuration and the implementation of a firewall. We do this by a two-stage approach.

The first part of this thesis gives an overview of the whole testing process, but concentrates on testing the configuration of a firewall. For this we design a language to formally specify network security policies. Further we propose a method for checking whether a given firewall, which we treat as a black box, correctly enforces such a policy. Using prototype tool support, we validate this approach.

The second part of this thesis then addresses the question of how a firewall, or more generally a midpoint, should handle a protocol. This question is central to the testing of a firewall's implementation. To our knowledge the problem has not been identified before. We show why midpoints are different from endpoints, illustrate the consequences of the current lack of protocol specifications for midpoints, and give an algorithm to generate midpoint specifications from endpoint specifications systematically. Roughly speaking, the algorithm tracks all possible endpoint states at each point in time, taking into account messages in transit and possible network behaviour. We prove that the midpoint automata constructed forward only those messages that could have resulted from protocol-conform endpoints.

By combining both parts of this thesis, it is possible to automatically check whether the security one relies on is justified, i.e. if firewall specifications and implementations conform to their specification.

Chapter 1

Introduction

The Need for Firewalls
In today's world, more and more companies (and individuals) want to benefit from the Internet. But connecting a private network to the Internet endangers proprietary data and the network itself. Security (protection) measures are needed to protect data from unauthorised access. Firewalls are an integral part of such security measures.

A firewall can be compared to a door-lock or a gatekeeper. Both can prevent unauthorised access to a building. In fact, we need a door-lock or gatekeeper for every critical door in the building. We explain this with the help of an example. Consider a bank with 4 doors. The bank has a main door (door 1) leading to an anteroom containing an ATM. From there, another door (door 2) leads to the counter area, which is separated by another door (door 3) from the offices. Finally, we have a door (door 4) separating the vault from the offices. In this example we see that the doors are placed between areas with different functionality.

An *access policy* defines who is authorised to enter which room and when. This access policy should be implemented by the door-locks, together with the corresponding keys and cards. Note that doors alone are not sufficient – we for example also need walls – but they are the policy enforcement points. In our example, the policy could be that door 1 can be opened 24 hours by customers of the bank using their account card, door 2 should only be open during the opening hours of the bank, door 3 can only be opened by employees of the bank and door 4 can only be opened by 2 or more members of the board together.

The Need for Security Policies
Similar to the access policy in our above example, a *security policy* is needed in a networked environment. A security policy specifies the jobs of the firewalls in a network as well as the expected behaviour of users like "do not write down your password". A security policy is used:

- To establish consensus about the security measures in place,
- as a guideline for the (firewall) administrator, and

- as foundation in testing.

Let us elaborate on the first item. There are mainly two reasons why consensus, about the security measures in place, is important. The first reason is that we have different people involved in the development of a security policy. We have a security officer having the knowledge to propose reasonable security measures, the management having to approve these, and a (firewall) administrator having to implement them. If they do not talk about the same thing, will we have security at the end? The second reason is that security cannot be enforced by technical means only. The people in a company need to understand, and adhere to, the security measures of their company. If they do not, security incidents will be the result.

Experience shows that consensus can only be reached if the security policy exists in a concise, written form and if everybody knows about it. Unfortunately policies often only exist in the heads of the administrator, or are written so informally, that they can be interpreted in many ways.

The Need for Firewall Testing
The bank from our above example becomes worthless if all doors are left wide-open. Similarly, we can only be satisfied if a firewall is working as expected. Firewall testing is a means to determine if a firewall correctly implements a given security policy.

What we should never forget when doing firewall testing is that an attack can come from either inside or outside, and it need not have a criminal motivation.

1.1 State of the Art in Firewall Testing

This is just a short overview on the state of the art in firewall testing. More details can be found in chapter 6.

Currently, firewall testing is mostly penetration testing [Sch96, WTS03]. Penetration testing consists of scanning all the ports to find the open ones. It also includes running known attacks against the firewall to find vulnerabilities. The problems with penetration testing are manyfold:

1. Though there are tools for port scanning, their results as such are worthless. To be of help, very *time-intensive* analyses of these results by *experts* are needed.

2. As new attacks occur every day, resistance against known attacks is only part of the story. The more important part – whether a firewall would resist new attacks – cannot be tested by penetration testing.

3. Penetration testing cannot determine if a firewall does what it is expected to.

Let us visualise these problems with the analogon of the door-lock. Consider a room with a locked door where nobody has got a key to that door. Penetration testing consists in many unauthorised persons trying to enter the room. They will not succeed. Therefore

the door-lock will be assumed secure. The problem in this setting, namely that authorised persons cannot enter the room either, is not found by penetration testing.

In the above example, the difference between *vulnerability testing* and *conformance testing* can clearly be seen. Vulnerability testing searches for vulnerabilities (possible attacks), whereas conformance testing analyses the conformance of a firewall to a policy. Both methods are needed to achieve adequate security. Unfortunately, the only means for conformance testing is rule analysis by experts.

1.2 Goal

The goal of this thesis is to develop a method for the specification-based testing of firewalls. Such a method shall consist of the following ingredients:

- a formal language for the specification of security policies,
- a network model,
- algorithmic methods for generating the test cases, and
- tool support for carrying out tests.

Our thesis is that it is possible to construct such a method whose resulting test-cases can be used to test whether a specific installation satisfies the stated policy.

To be of practical use, the method should work for multiple firewalls. The 'multiple' can be understood in two ways: 1.) The method should work for products of different vendors, and 2.) the method should work for more than one firewall in a network. Note that the goal of this thesis is not to implement a tool, but to design a method. Therefore only prototypical tool-support will be provided.

Note further that it is not our goal to find vulnerabilities but to test for conformance. If the policy states that everything should be allowed, we are satisfied if the firewalls allow everything. Thus we do not answer the question if a certain policy is secure. This must be answered by expert analysis or vulnerability testing.

1.3 Possible Approaches

In this section we will elaborate on different, possible approaches to ensure the conformance of a firewall to a given policy. Let us first introduce the terms used (see Figure 1.1). We use the term *informal policy* to denote policies written in natural language. These cannot be used for any kind of formal analysis. Therefore we need a policy with a clear syntax and semantics, which we call a *formal policy*. The term *firewall rules* is used to denote an actual firewall configuration in a vendor specific format, whereas the term *abstract firewall rules* is used to denote firewall configuration in a vendor-independent

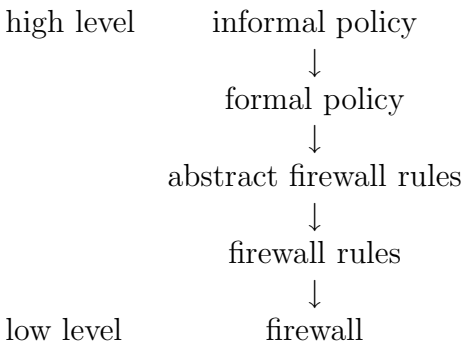


Figure 1.1: Policy versus rules

format. Finally, we use the term *firewall* to denote the behaviour of the firewall software or hardware implementing given firewall rules. A firewall therefore consists of two parts: configuration and *implementation*. Only if the firewall implementation is correct, the conformance of a firewall to a policy can be deduced from the conformance of its firewall rules to the policy.
The different approaches to ensure the conformance of a firewall to a given policy are to:

1. generate the firewall rules from the security policy,
2. prove the equivalence between a formal policy and an abstract firewall ruleset,
3. prove the equivalence between the abstract and the actual firewall rules,
4. generate the abstract firewall rules from the actual ones and then prove (theorem proving) the equivalence to the formal policy, and
5. generate test cases from the formal policy and run them against the firewall.

Note that approaches 2 and 3 can only be used together. As actual firewall rules are vendor- and release-dependent, all approaches using the firewall rules would need considerable (re-)engineering. This re-engineering must be repeated every time a new firewall software (version) is released. This is not very practical and also highly error-prone. Furthermore, all these approaches assume the firewall implementation to be correct.

The 5th approach shown is the most general. It can be used independently of the kind of firewall installed, as it treats the firewall as a black box. Furthermore, it is the only approach that not only checks the *firewall configuration* (ruleset) but also the firewall implementation[1]. Therefore we chose this approach. Thus we assume a testing environment where the firewall rules are already present (i.e. written by hand). More detail about this approach can be found in the next section.

[1]What does it help to have a ruleset conforming to a given policy if the firewall does not do what is written in the ruleset (i.e. the firewall implementation is buggy)?

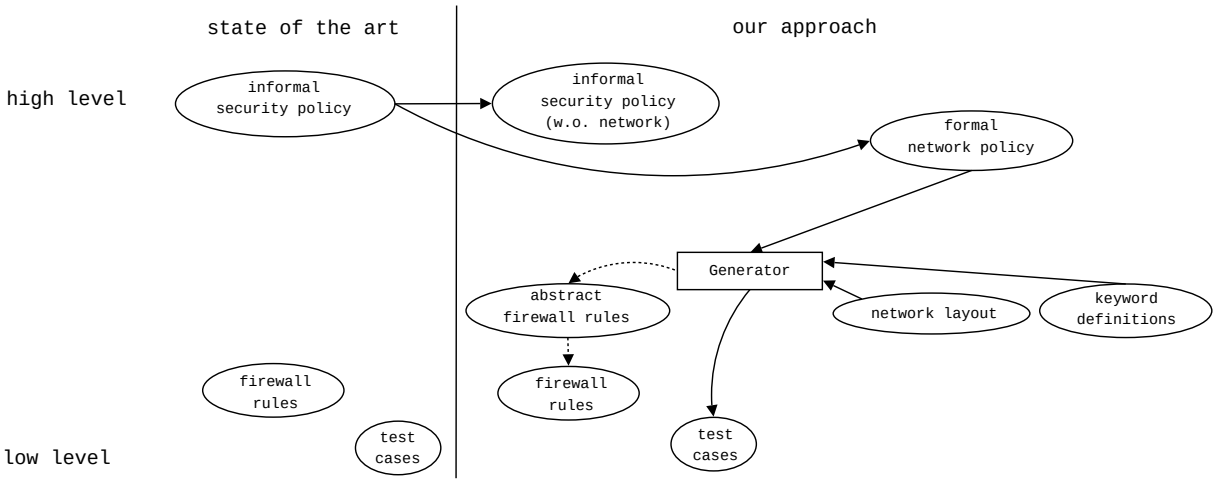


Figure 1.2: Overview

1.4 Overview of our Approach

In this section we give a short overview of our approach. All elements of our approach and their relation can be found in Figure 1.2.

Policy Current security policies are normally informal and can therefore not be used to reason about them in a formal manner. There exist even companies that have no written policy at all. But to be able to determine if a firewall implements a given security policy, at least the *network policy* (which is part of the security policy) needs to be formally stated. Such a formal network policy (*formal policy*) must specify what shall be allowed (but not how this should be done). Our formal policy is expressed at a high-level; this way it is both manageable and understandable by managers as well as security specialists. See section 3.2 for more on this topic.

Low-level details To test a firewall against a given formal policy, low-level details are needed. Consider the following policy statement: "only secure connections are allowed from the Internet to our corporate network." To be able to test this statement, we need to know what secure connections are and what our network looks like. This low-level information, which may be subject to frequent change, is stored separate from the policy in the so-called *keyword definitions* and the *network layout* respectively. See Sections 2.3, 3.1, and 3.2 for more on this topic.

Test case generation The formal network policy, the network layout and the keyword definitions provide all the information we need for automatically generating *test tuples*. These test tuples can then be used to instantiate the *abstract test cases*, representing the firewall implementation, to get *concrete test cases*. A concrete test case consists of several

network packets, and an expected result for each. These network packets are inserted to the network on one side of the firewall, and the "answer" of the firewall is compared to the expectation. Note that such a test must not be done in a production network, but in a copy of it. See section 4.3 for more on this topic.

Test evaluation An unexpected test outcome can result from:

- An error in the firewall configuration (the firewall rules do not correspond to the formal network policy),
- a bug in the firewall implementation (the firewall is not doing what it is told by the firewall rules),
- network problems (packets were changed or lost due to heavy network traffic), or
- a bug in our test environment.

To find the source of an unexpected test outcome, knowledge of the firewall rules is needed. As we are treating firewalls as black boxes, we do not have this knowledge. Therefore, the test evaluation cannot be fully automated. This represents no problem, as an expert can easily find the problem when given the unsuccessful test case, together with the policy-rule it was generated from.

1.5 Contributions

Our results can be divided into two parts.

First part In the first part of this thesis, a method, as described in section 1.4, is designed. It consists of a language for the formal specification of security policies and a method for the automatic generation of test cases from such a policy. Using prototype tool support, we validated our thesis – *it is possible to construct a method (for the specification-based testing of firewalls) whose resulting test-cases can be used to test whether a specific installation satisfies the stated policy* – by applying our method to a real-world scenario.

Our contribution for Part I is the first method for firewall conformance testing. It advances the state-of-the-art by eliminating time-intensive, highly error-prone rule analyses of experts.

Second part Apart from the policy, we needed another specification. To test if a firewall handles a certain connection correctly, it does not suffice to know if the connection should be allowed or not. Additionally we need to know how the connection (the protocol) should be handled by the firewall. To our knowledge, this important problem has not been tackled before.

In the *second part* we therefore analyse why a protocol specification for a firewall, or more general a midpoint, needs to be different from a protocol specification for endpoints. Based on this analysis we present an algorithm to transform (endpoint) protocol specifications into midpoint specifications. We prove the correctness of this algorithm and show its applicability to the TCP protocol.

Our contributions for Part II are an analysis of why different protocol specifications are needed for midpoints than for endpoints, what the implications of a lack of such specifications are, and an algorithm to generate midpoint specifications from endpoint specifications. We prove that the midpoint automata constructed by our algorithm forward only those messages that could have resulted from protocol-conform endpoints.

Our solution should be of interest to at least two groups: those building midpoints and those analysing (e.g. testing) them. It advances the state-of-the-art by being the first general method to systematically construct midpoint specifications from those for endpoints. Only by having such a method, the specification-based testing of firewalls is possible.

1.6 Organisation of this Thesis

As already stated, this thesis consists of two main parts. Both of them build on basic knowledge given in chapter 2.

The first part consists of defining Network Layout, Keyword Definitions and Formal Policy, introduced above, in chapter 3. This is followed by our test methodology in chapter 4, which is then validated in chapter 5. In chapter 6 we provide a comparison to related work. We summarise the first part in chapter 7.

The second part is organised as follows. In chapter 8, we explain why midpoints (firewalls) are different from endpoints and therefore need their own protocol specifications. In chapter 9, we present an algorithm to generate a midpoint automaton from endpoint automata and prove it correct. We discuss and summarise the second part in chapter 10.

Finally, in chapter 11, overall conclusions are drawn and the most important future work is discussed.

Chapter 2

Background

In this chapter we elaborate on the basic concepts on which we build our work, namely Mealy Machines, Network Protocols, Security Policies, Firewalls, and Testing.

2.1 Mealy Machines

Network protocols can be understood as *Mealy machines (automata)* [Mea55]. Firewalls filter protocol messages and their implementations consist of one or more Mealy machines. In short, a Mealy machine is an automaton taking *inputs* and returning *outputs* based on its current *state*. Formally, a Mealy machine is defined as a six-tuple

$$M = (Q, \Sigma, \Gamma, \delta, \lambda, q_1)$$

where

- $Q = \{q_1, q_2, ..., q_{|Q|}\}$ is a finite set of *states*;
- $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_{|\Sigma|}\}$ is a finite *input alphabet*;
- $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_{|\Gamma|}\}$ is a finite *output alphabet*;
- $\delta : Q \times \Sigma \to Q$ is the *transition function*;
- $\lambda : Q \times \Sigma \to \Gamma$ is the *output function*;
- $q_1 \in Q$ is the *initial state*.

Automata are often specified graphically. Figure 2.1 is an example of a graphical representation of a Mealy machine. In this example, q_1, q_2 and q_3 represent states, where q_1 is the start state and q_3 is an *accepting state* (or final state). If an accepting state is reached, using some input, the input is accepted. In a ticket machine, for example, the accepting state is the state where enough money for the chosen ticket has been inserted, which will result in issuing the ticket. The arrow from q_1 to q_2 represents a transition with input x and output y. Note that in this example, the accepting state q_3 is not reachable. Throughout this thesis we use '$-$' to denote empty input or output.

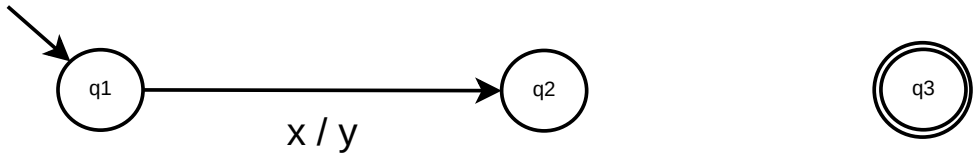


Figure 2.1: A graphical Mealy machine

2.2 Network Protocols

Today, nearly every computer is connected to a network through which it can *communicate* with other computers. In this section, we explain the points about such communications that are relevant for our work.

Internet layer
Communication in the digital world happens between unique addresses, so-called *IP addresses*, using *network packets*. The forwarding of these packets is done by devices that are called *Routers*. All this forms the *internet layer*. Its job is to deliver packets to their *destination IP address*.

There are two different types of IP addresses: *IPv4* (IP version 4) [ISI81a] and *IPv6* [CN98]. An IPv4 address is composed of four octets. Thus there are 2^{4*8} different IPv4 addresses between 0.0.0.0 and 255.255.255.255. Some of them are special. 10.x.x.x, 172.16.x.x, 192.168.x.x for example represent *private address spaces* and are not routed in the Internet.

The use of the Internet in the developing world and from mobile devices led to a need for more than the available 2^{32} IPv4-addresses. One solution for this problem is to map several private IP addresses to one public address (*Network Address Translation – NAT* [SCR+96, ST99])[1]. The real solution would be to switch to IPv6, which is intended as replacement of IPv4. IPv6 offers 2^{128} different addresses. An address consists of 8 blocks of 16 bit.

On top of the internet layer there are further layers: the *transport layer* and the *application layer*. The idea is that every layer uses the services of the next-lower layer and just add additional services. Note that, contrary to the ISO/OSI model [Org96], the *internet protocol suite* (TCP/IP reference model) has no further layers between the internet layer (OSI layer 3), the transport layer (OSI layer 4) and the application layer (OSI layers 5 – 7).

Transport layer
The transport layer is the next-higher layer after the internet layer. The most important protocol in this layer is the *Transmission Control Protocol* [ISI81b]. TCP is used for flow control, reliability, and multiplexing purposes.

[1] As the private IP addresses are not routed in the Internet, every institution can use the same ones.

Alice		message sent		Bob
CLOSED				LISTEN
SYN-SENT	—	SYN	→	
	←	SYN & ACK	—	SYN-RECEIVED
ESTABLISHED	—	ACK	→	ESTABLISHED

Figure 2.2: TCP three-way-handshake

How can it be ensured that all packets reach their destination? This is done by the use of *sequence numbers*. We will explain their use with the help of an example. Consider Alice sends the three packets 11 – 14 to Bob. This helps Bob to know in which order he has to read the packets and to see if some intermediate packet is missing: if he receives 11, 12 and 14, then he knows that 13 is missing. But what if Bob only receives packets 11 and 12? He cannot know if there need to be more packets. But Alice knows! Thus when Bob sends an answer to Alice (numbered with his own sequence number), he also tells her which packet number (= *acknowledgement number*) he expects next from her (13 in this case). In this way Alice knows which packets he received and can resend (= *retransmission*) the missing packets (here 13 and 14).

The sequence and acknowledgement number, together with so-called *flags* – named SYN, ACK, FIN, URG, PSH, and RST – are part of the TCP *header*. The flags are used for different purposes like the connection initiation (SYN: synchronising on the first sequence number), marking the presence of an acknowledgement number (ACK), and the connection closing (FIN). This means that the individual packets are independent at the IP level, whereas the communication at the TCP level is *connection*-based.

The TCP *connection initiation* is shown in Figure 2.2. CLOSED, LISTEN, and so on therein represent the *states* of the endpoints: TCP is implemented as Mealy machine by the endpoints. In the given setting, Alice is called the *initiator* of the connection, whereas Bob is called the *responder* of the connection.

Application layer

In the internet protocol suite, the application layer is the next-higher layer after the transport layer. It comprises protocols like http, ftp, and so on. Like with TCP and IP, the application level protocols use the next-lower protocol (TCP in this case) and just add some additional information. Like in the TCP case, some of this information is added in the form of a header. But here also the data part is important. We will explain this with the help of two examples.

Voice over IP (VoIP) [RSC+02, ITU03] for example consists of several protocols in series, where the *connection information* for one protocol can be found in the data part of the previous protocol. This means that both endpoints and intermediate firewalls need to be able to extract this connection information. If for example an intermediate firewall fails to extract this information, and allow the corresponding connections, then VoIP will not be possible.

The content of a website represents the data part of the http protocol. A company that does not want their employees to visit certain websites (racism, pornography, ...), therefore needs to be able to block http-traffic based on its data part. This can be done with the help of an application layer firewall that parses the data part of http-traffic and blocks all the packets that contain words from a *blacklist*.

2.3 Security Policy

The American National Institute of Standards and Technology (NIST) defines the term *computer security policy* as "documentation of computer security decisions" [oSN95]. Another definition is "A security policy is a formal statement of the rules by which people who are given access to an organization's technology and information assets must abide." [Gro97].

The policy statements can be grouped under the following headings:

1. Corporate Policy
2. Information Security Policy
3. Personnel Security Policy
4. Physical and Environmental Security Policy
5. Computer and Networks Security Policy
   - System Administration
   - Network Policy
   - Application Development Policy
6. Business Continuity Planning

A policy should describe a certain security topic, why it is needed (important), and explain what is allowed and what is not allowed. It should contain general directives that are not architecture or system dependent. A policy should also tackle enforcement, i.e. it should be clear what disciplinary measures are to be expected if the policy is breached. Policies should be concise, a good balance of productivity and security, be backed up by appropriate security tools, and be easy to understand.

Policy versus Model
As explained above, a policy is normally an informal statement about everything, including physical or administrative issues, that has to do with security at a specific site (or for a specific system). In contrast, a model is generally expressed in a formal language (including proofs for correctness, consistency, ...) where characteristics of policies are explicitly stated. Therefore models are used to write policies.

An example is the Bell-LaPadula model [BL73]. Its purpose is to prevent that confidential information (object) is given to non-trustworthy persons (subject). To achieve this purpose, objects are assigned security labels, e.g. *confidential*, and subjects are given clearance for a security label. A subject is then only allowed to read objects that are at most his security label (*simple security property*: no read-up) and to write documents with at least his security label (* *property*: no write-down).

Examples
The following are some representative excerpts of security policies:

[Uni03] "The RMITCS data stored on RMITCS computer systems [...] should be protected from unauthorised access, removal or destruction."

[oCS03] "All departmental computers which are accessible on the public Internet should have all non-essential services disabled, to minimise the possibility of security compromises."

[UoI03] "However, the owner of a privately owned machine is responsible for the behaviour of the processes running on that machine and all the network traffic to and from the machine."

All these examples visualise one problem with informal policies: they are not precise enough. What is *unauthorised access*? What is a *non-essential* service? What falls under a *security compromise*? When, and what for, is an owner of a private machine made responsible and how?

2.4 Firewalls

The computing world uses the term firewall for hardware or software that is put between two (or more) networks to prevent communication forbidden by the network policy. A firewall normally runs on a dedicated network device or computer. Such a firewall filters all the traffic entering or leaving the connected networks. Filtering means deciding whether or not each network packet is allowed to cross the boundary between the two networks. More sophisticated filtering also modifies packets. NAT (translating the private IP addresses of hosts behind a firewall to routable addresses), for example, can be done by firewalls.

Two major categories One can distinguish between two major categories of firewalls:

- *packet filters*, which work at the transport layer, and
- *application layer firewalls*, which work at the application layer.

These two types of firewalls may overlap, indeed some systems implement both. A proxy device may act as a firewall by responding to input packets in the manner of an application, whilst blocking other packets.

```
1 # delete everything
2 /sbin/iptables -F
3 /sbin/iptables -X
4
5 # allow all outgoing
6 /sbin/iptables -P OUTPUT ACCEPT
7
8 # allow only incoming ssh
9 # (and related or established connections)
10 /sbin/iptables -P INPUT DROP
11 /sbin/iptables -A INPUT -p tcp --syn --dport ssh -j ACCEPT
12 /sbin/iptables -A INPUT -m state --state RELATED,ESTABLISHED\
13                -j ACCEPT
14
15 # allow connections from localhost to localhost
16 /sbin/iptables -A INPUT -d 127.0.0.1 -j ACCEPT
```

Figure 2.3: An example of an iptables ruleset (stateful)

Firewall configuration versus implementation What is important to understand is the difference between the *firewall configuration*, by a so-called *ruleset*, and the *firewall implementation*. The firewall implementation is the software or hardware delivered by the firewall vendor. The firewall configuration consists of rules that tell the firewall what to do. This means that there exist many firewalls with the same implementation (including firmware, patchlevel, ...), but most probably there are no two firewalls with the same configuration.

Proper configuration of firewalls demands skill. It requires considerable understanding of network protocols and of computer security. Small mistakes can render a firewall worthless as a security tool. Trust in misconfigured firewalls is misplaced indeed.

Types of packet filters When speaking of packet filters, two types have to be distinguished: *stateless* and *stateful packet filters*. Stateless and stateful refer to the kind of protocol inspection made by the packet filter. Let us explain the difference between the two types with the help of the stateful TCP protocol. If one endpoint, say Alice, wants a TCP connection to another endpoint, say Bob, then she has to initiate it with the three-way-handshake shown in Figure 2.2. Having a stateless packet filter, each of these three messages will be examined (i.e. compared to the firewall rules) separately, whereas with stateful inspection, the firewall remembers the messages seen so far (i.e. the state of the connection)[2].

Thus with stateless packet filters we cannot constrain incoming packets to answers (we

[2]It does so by having an internal automaton for the TCP protocol.

```
1  # delete everything
2  /sbin/ipchains -F
3  /sbin/ipchains -X
4
5  # allow all outgoing (and incoming: answers) traffic
6  /sbin/ipchains -P output ACCEPT
7
8  # allow incoming ssh and answers
9  /sbin/ipchains -P input DROP
10 /sbin/ipchains -A input -p tcp --dport ssh -j ACCEPT
11 /sbin/ipchains -A input -p tcp -y --syn -j DROP
12 /sbin/ipchains -A input -p tcp -j ACCEPT
13
14 # allow connections from localhost to localhost
15 /sbin/ipchains -A input -d 127.0.0.1 -j ACCEPT
```

Figure 2.4: An example of an ipchains ruleset (stateless)

can only constrain it to non-SYN packets, which can be more than answers), whereas with stateful packet filters we can. Let us explain this with an example. Figure 2.4 shows a ruleset for the stateless ipchains firewall, whereas Figure 2.3 shows a ruleset for the stateful iptables firewall (the successor of ipchains)[3]. Both should implement the following policy: everybody in our company network can connect to any machine on the Internet, using any protocol they want, but external people can only connect to our machines using ssh.

In iptables we can perfectly implement this policy by stating that everything is allowed if the initiator of a connection is inside (line 6). But from the outside we only allow ssh to be initiated (line 11). Other traffic from the outside is only allowed if it is an answer (line 12), i.e. belonging to an existing connection.

In ipchains we cannot base the rules on the state of a connection. It is therefore not possible to determine if a packet at hand represents an answer. We can only determine if the connection it belongs to is allowed. The only measure we can take is to block external SYN-packets (line 11). This will make the task of connecting to our machines from the outside more difficult, but unfortunately will not render it impossible.

Stateless packet filters are not anymore used in practice. This is due to the fact that they are less powerful than stateful packet filters. We therefore do not consider them in our work.

Correct firewall configuration is difficult For this simple example, the configuration of iptables was simple. But for more complex networks we need thousands of rules making the whole task very error-prone. Other problems are that every brand of firewall has its

[3]For details on the *iptables* firewall, which is the Linux default, refer to Appendix D .

own rule language. Even general concepts like the order of rule-matching (first vs. last rule that matches is used) differ between the different rule languages. Sometimes there are even hidden default rules one might not want: Checkpoint for example has a lot of these rules. They call them *global properties*. By default, properties like "Accept outgoing packets originating from Gateway" and "Accept VPN-1 & FireWall-1 control connections" are allowed. To see the impact of these global properties, the user has to choose to see *implied rules* in his ruleset. All this makes the task of proper firewall configuration even more difficult and is a prominent source of errors.

2.5 Testing in General

Testing is used for *validation* (Am I building the right system?), not to be confused with *verification* (Am I building the system right?) done by model checking and theorem proving.

There are more-or-less two different kinds of testing: white box and black box testing [Mye04]. *White box testing* is code-based. It consists of testing whether all the code is executed or whether copies of the code with inserted faults can be distinguished from the original. *Black-box testing* is specification-based. It tests whether the code implements the specification. As the name already suggests, the code cannot be used to determine the answer. The testing consists of feeding the *test data (input)* to the system under test (SUT) and comparing the output of the system with the *expected test output (results)*. As we want a methodology that is applicable to any kind of firewall, we do black box testing.

To determine if a test was successful, the expected test outcome has to be known in advance. Therefore every *test case* must consist of test data and expected test output.

Criteria for test cases The difficult part in (automated) testing is to find good test cases. Some of the first definitions in this direction can be found in a paper by J. Goodenough and S. Gerhart [GG75]. In this paper, the authors show how to define criteria which test data must satisfy such that a successful execution of the test data implies that there are no errors in a tested program. In general, such criteria C must be *reliable* and *valid*. In short, reliability refers to the consistency with which results are produced, regardless of whether the results are meaningful. Validity, in contrast to reliability, customarily refers to the ability to produce meaningful results, regardless of how consistently such results are produced. A test set is called *complete*, according to a given test data selection criterion C, if it satisfies C. A test set is called *successful* if the result of every test is as expected. This leads to the *fundamental theorem of testing*: Given a test set T and a test data selection criterion C for a program P; if T is complete according to C, C is reliable and valid, and the execution of T is successful, then P is correct.

Other researchers have also defined criteria for test cases. The following two definitions state the same as Goodenough and Gerhart, but in a more formal way.

Definition (test case reliability) [How76]: If P is a program to implement function

F on domain D, then a test set T $\subset$ D is reliable for P and F if $\forall$ t $\in$ T, P(t) = F(t) $\Rightarrow \forall$ d $\in$ D, P(d) = F(d).

Definition (test case adequacy) [BA82]: If P is a program to implement function F on domain D, then a test set T $\subset$ D is *adequate* for P and F if for all programs Q, if Q(D) $\neq$ F(D) $\Rightarrow \exists$ t $\in$ T such that Q(t) $\neq$ F(t).

Unfortunately there is no effective procedure for either generating adequate test sets or for detecting that a given test set is adequate. Thus the crux of the testing problem is to find an adequate test set, one large enough to span the domain and yet small enough that the testing process can be performed for each element in the set [ABC82].

2.6 Test Case Generation for Mealy Machines

As stated in section 2.4, firewalls are implemented by the means of Mealy machines. To black box test firewalls, we therefore need to generate test cases from Mealy machines. In this section we give a survey on the most common methods for this purpose. We will later use one of these methods for our test case generation in subsection 4.3.1.

2.6.1 General Introduction

There exist many methods to generate test cases for automata. In this thesis we are interested in the problem of *Machine Verification (Conformance Testing)*. Conformance Testing consists of testing whether an implementation machine IMP conforms (is equivalent) to the specification machine SPEC. Two finite state machines (FSMs) are *equivalent* if in their minimised version they have the same number of states, and if there exists a one-to-one correspondence between equivalent states. Two states are equivalent if for every input sequence the machine produces the same output sequence[4].

Assumptions The following assumptions are usually made in conformance testing:

- SPEC is *strongly connected*: A machine M is strongly connected if for any two states s and s' of M, s' is reachable from s.
- SPEC is *reduced*: all states are pair-wise distinguishable.
- IMP *does not change* during the experiment and has the *same input alphabet* as SPEC.
- IMP does *not* have *more states* than SPEC.

[4]Said in other words: If we have two completely specified, minimal automata with the same number of states, all of which equal (i.e. having the same outgoing transitions), they must be the same.

Some methods also make one or more of the following assumptions about SPEC or IMP:

- *minimal*: there is no equivalent FSM with fewer states.
- *completely specified*: each state has outgoing transitions for every input symbol. Normally it is also okay if we have an incompletely specified FSM where *non-core input* (input for which there is no transition) is ignored.
- *fixed initial state*: the FSM always starts in the same state (start state).
- every state is *reachable*: starting from the start state, every other state can be reached by traversing a finite number of transitions.
- *deterministic*: at each state there is only one possible transition per input symbol (as opposed to multiple possibilities in non-deterministic automata).
- *reliable reset*: there is a reset input which brings the FSM back into its start state (from any other state).

Fault detection The faults that can be found are the following:

- Operation error: an error in the output function δ.
- Transfer error: an error in the next-state function λ.
- Extra or missing states.

A method is said to have *full fault coverage* if it can find all these faults.

Basic test structure The basic structure of all test methods for solving this problem is similar: we want to make sure that every transition of the specification FSM SPEC is correctly implemented in FSM IMP. This is achieved as follows: for every transition of SPEC, say from state s_i to state s_j, do the following:

1) Bring machine IMP into the initial state s_1.
2) Transfer machine IMP into state s_i.
3) Test the transition (apply its input and see if the output is correct).
4) Verify that the automaton now is in state s_j.

Step one is easy if there is a reliable reset: Just apply the reset input to go back to the initial state. If there is no reset input, a *homing sequence* can be constructed to fulfil this task. We do not further elaborate on how this is done. Refer to [Gil62] for more information on this topic.
Steps two and three can be solved by building a *test tree* T (see Figure 2.5 for an example) according to the following rules [Cho78] and then walking along all the paths:

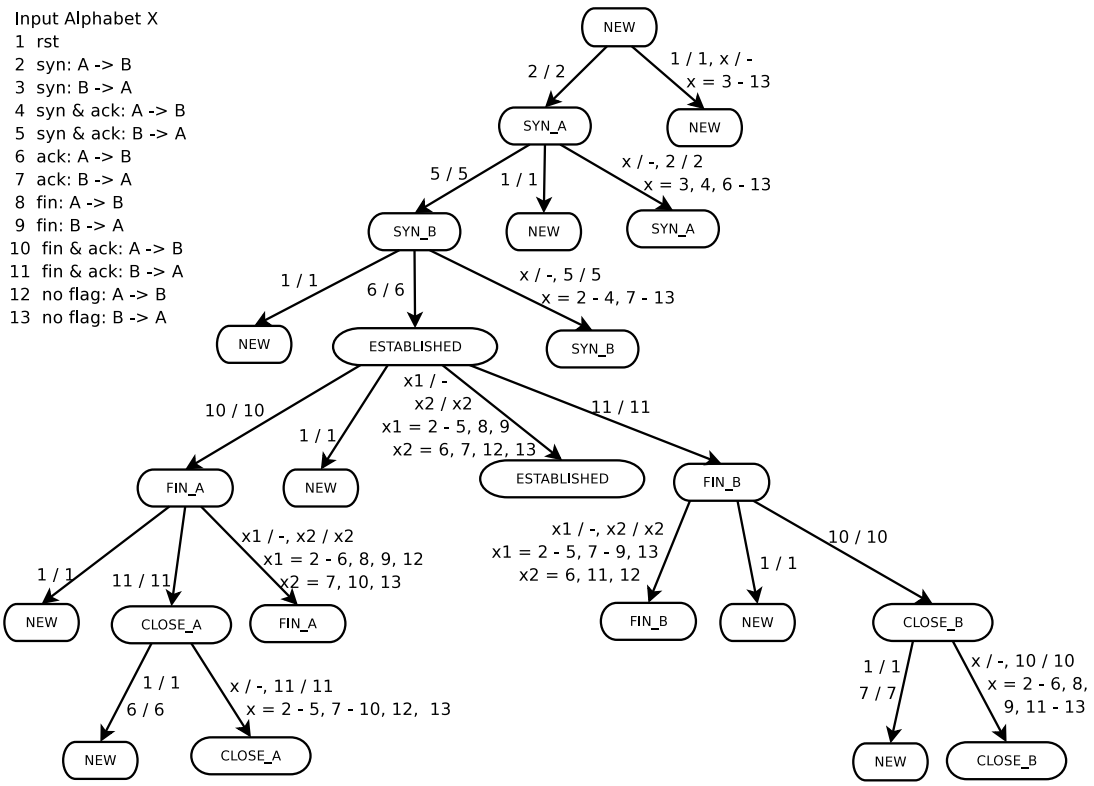


Figure 2.5: Test Tree for tcp

a) Label the root of T with the initial state of SPEC. This is level 1 of T.

b) Suppose we have already built T to a level k. The (k+1)-th level is built by examining nodes in the k-th level from left to right. A node in the k-th level is terminated if its label is the same as a nonterminal at some level j, j $\leq$ k. Otherwise, let $SPEC_i$ denote its label. If on input x, machine SPEC goes from state $SPEC_i$ to state $SPEC_j$, we attach a branch and a successor node to the node labelled $SPEC_i$ in T. The branch and the successor nodes are labelled with x and $SPEC_j$, respectively.

Step four is the one where the test methods differ. In the following we present the most common methods, namely *distinguishing sequences* [Gil61, Gil62], the *W-method* [Cho78], *UIO sequences* [SD88], the *partial W-method* [FvBK+91], and the *UIOv method* [CVI89]. For more information about these methods please refer to the original papers or to the overview paper by Lee and Yannakakis [LY96].

2.6.2 Verifying the State of a FSM

W-Method [Cho78]
Assumptions on the FSMs: completely specified, minimal, start with a fixed initial state, every state is reachable.

The W-method uses the *characterisation set* W (W-set) for accomplishing step four. A characterisation set is a set of input sequences that can distinguish between the behaviour of every pair of states. This means that no two states have the same output for all of these inputs:

$$\forall s_i, s_j \; \exists x \in W \; . \; i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$$

Contrary to other methods, the W-method does not assume the implementation FSM to have the same number of states as the specification FSM. In the W-method, the maximum number of states in the implementation – denoted by m – is estimated by the tester. Using the set

$$\begin{aligned} Z &= W \cup X \cdot W \cup \cdots \cup X^{m-n} \cdot W, \qquad \text{where} \\ X &= \text{input alphabet} \\ n &= \text{number of states of the specification} \\ X \cdot W &= \text{concatenation of the strings X and W} \end{aligned}$$

instead of W, the W-method will find all faults in the implementation if it has at most m states.
The complete W-method (steps 1 – 4) consists of

$$P.Z = \{p \cdot z | p \in P, z \in Z\},$$

where P consists of all partial paths (including $\{\}$) in the testing tree (steps two and three). P is called the *P set* or the *transition cover set*.

partial W-Method (Wp-Method) [FvBK+91]
Assumptions on the FSMs: as for the W-method.
This method is a variation of the W-method which has the same fault coverage but provides shorter test sequences. Additionally to Z and P from the W-method, we need the *state cover set* Q:

$$\forall s_i \in SPEC \; \exists pi \in Q \; . \; \lambda(s_1, pi) = s_i$$

The Wp-Method consists of two phases:

$$\begin{aligned} &\text{Phase 1:} && Q.Z \\ &\text{Phase 2:} && R \otimes W = \bigcup_{p \in R} \{p\}.Wj, \quad \text{where} \\ & && R = P \setminus Q \end{aligned}$$

The test sequences of the Wp-method are shorter than those of the W-method because in the Wp-method, Z is used only once per state (the shorter W is used for the other transitions) whereas in the W-method Z is used for every transition.

Unique Input-Output (UIO) sequences method [SD88]
Assumptions on the FSMs: deterministic, initial state, every state reachable, reliable reset, minimal, either completely specified or non-core input is ignored (completeness assumption).
The UIO method uses UIO sequences for accomplishing step four. A UIO sequence is a sequence x for a state s that distinguishes state s from all other states:

$$\forall s_j \exists x \forall s_i \ . \ i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$$

Contrary to the initial claim, this method does not have full fault coverage. The problem occurs when a UIO sequence is not unique in a faulty implementation. This problem was resolved in the UIOv method.

UIOv method [CVI89]
Assumptions on the FSMs: the same as for UIO, plus completely specified.
The problem of the UIO method is solved by the UIOv method by adding a *verification step ˜Uv* before the testing. The idea of ˜Uv is to verify that the UIO sequences indeed are unique or, if not, to detect the faulty state. This is done by applying all different input sequences from all UIO sequences to all states and verifying that every UIO sequence is encountered only once.
The UIOv method has full fault coverage if the number of states in the implementation is the same as in the specification.

Distinguishing Sequences (DS) method [Gil61, Gil62]
The Distinguishing Sequences method uses *Distinguishing Sequences* for step four. A distinguishing sequence (or generally a test) can be *preset* — if an input sequence is fixed ahead of time — or can be *adaptive* — if at each step of the test, the next input symbol depends on the previously observed outputs. Preset DSs are a special case of a W-set (see W-method): A W-set with only one sequence. That is, an input sequence that produces different output for each initial state:

$$\exists x \forall s_i, s_j \ . \ i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$$

Therefore DSs can also be used for state identification (determining the current state of an automaton) and a reliable reset is not required. The problem is that not every FSM has a DS.

Summary
We have seen three methods (plus two variations) for the testing of Mealy Machines. The difference lies in the number of input sequences that are used. By looking at the formulas from above, we can compute the number of input sequences used (n = number of states):

method	definition of input sequences x	number of sequences
W	$\forall s_i, s_j \exists x \ . \ i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$	$\frac{n*(n-1)}{2}$
UIO	$\forall s_j \exists x \forall s_i \ . \ i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$	n
DS	$\exists x \forall s_i, s_j \ . \ i \neq j \rightarrow \lambda(s_i, x) \neq \lambda(s_j, x)$	1

By looking at the formulas, we see that the W-method uses an input sequence x per pair of states, whereas the UIO sequences method uses an input sequence per state, and the DS method uses only one sequence at all.

2.6.3 Time of Generation: adaptive vs. preset

As mentioned in the description of the DS-method, there are basically two possibilities of test case generation (for all methods): adaptive and preset. A *preset test case* is generated before use, whereas with an *adaptive test case* the next input is determined in reaction to the previously observed output. In this section we will discuss the advantages and disadvantages of both approaches. Later, in section 4.4, we will combine both approaches for our tests.

Preset experiment The biggest advantage of a preset experiment is that the generated test cases can be reused. Additionally, the generation is easy and not time critical. The disadvantages are that not all information is available at that time – In the case of NAT, for example, the source port of a packet after the firewall might not be known in advance. – which results in incorrect test cases. Furthermore, as no correction or reaction is possible during the tests, we need many test cases: one for every possible reaction of the system.

Adaptive experiment The main advantage of an adaptive experiment is that the test can be based on the reaction of the system under test. This means fewer and correct test cases. The disadvantages are time and storage overhead: Time is needed for the building of test cases on the fly. Storage is needed to carry along all possible scenarios.

2.6.4 Comparison of the Methods

The UIO method has shorter test cases than the W- and the DS methods. Also UIO sequences nearly always exist (and if they do not, a special signature can be used instead). As long as only the DS, the W and the UIO method existed – and one believed they all have the same fault coverage – the UIO method was the best choice.

Taking the newer methods (UIOv and Wp) into account, the UIOv method is a special case of the Wp-method, and the DS method is a special case of the UIOv method. All have full fault coverage. This means that the Wp-method is the most general and most widely applicable method from these three. As the Wp-method achieves the same fault coverage as the W-method but with shorter test sequences, and as the UIO method has

no full fault coverage, it is in general the best to choose the Wp-method. Therefore we will use the Wp-method. Information on how we incorporate it into our setting can be found in subsection 4.3.1.

Part I

Specification-based Firewall Testing

Chapter 3

Specification

The first part of this thesis gives an overview of our methodology for specification-based firewall testing. It concentrates on testing the configuration of a firewall. In this chapter, we start by introducing the specifications we need.

3.1 Network Layout

As stated in section 2.4, firewalls are machines that filter traffic between two or more networks. All the traffic between these networks has to pass the firewall. To test a firewall for conformance to a policy, we need low-level information about the firewall and its adjacent networks. To determine the information we need, we first analyse the ingredients of a network.

Andrew Tanenbaum defines a *computer network* as an interconnected collection of autonomous computers [Tan96]. We distinguish between three types of such computers: *clients*, *servers*, and *routers*. A router is as defined in section 2.2. A server is a computer offering a *service* to others. All the remaining computers are clients. Note that firewalls normally are routers (or sit just next to a router). Therefore we will make no distinction between routers and firewalls. Note that it is considered good security practice not to mix the different types of computers. We therefore assume clients, servers and routers to be distinct.

Graphical network layout Network layouts are often visualised to help people understand them. One example of such a drawing can be found in Figure 3.1. It consists of three networks (clouds), and two firewalls (FW1, FW2) separating these networks. Furthermore, there are two servers in the *demilitarised zone (DMZ)*. As servers need to be accessed, we need to know where they are (and then talk about from where and how they should be accessed). Note that we do not differentiate the clients in the Intranet. We assume that all clients in a zone are equivalent, in that differences in their IP addresses have no effect on the firewalls' behaviour. This represents our *uniformity hypothesis*. The justification for this uniformity hypothesis is that it is considered good network layout

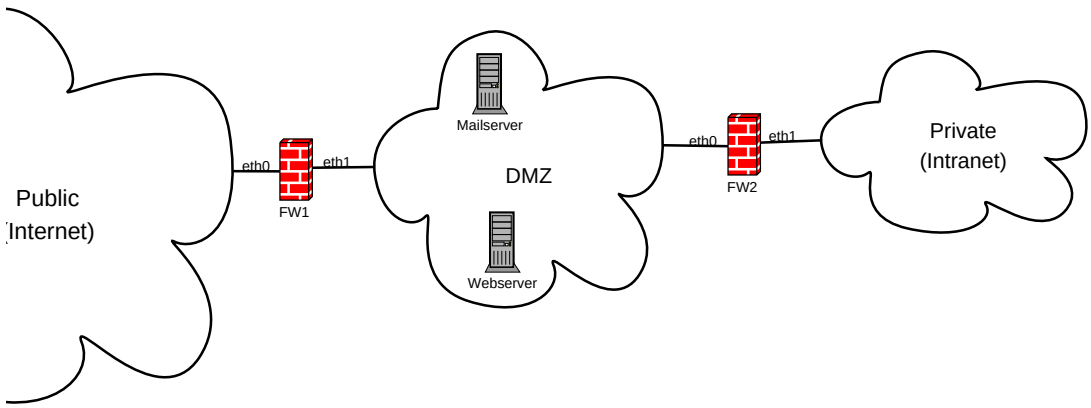


Figure 3.1: A typical network layout

practice to put clients with different rights into different networks. Customer information, for example, should not be endangered by havoc running tests.

The network layout in Figure 3.1 is typical for small companies. The DMZ is used for servers which need to be accessible from the outside world. The firewall FW1 regulates the accesses to these servers whereas FW2 protects the internal computers (which may store sensitive data) and network infrastructure from unauthorised outside access.

Textual network layout Whereas a graphical representation of a network is useful for people, a textual representation is needed for the use in an automated test case generation. The information needed can be divided into three categories: networks, firewalls and servers.

For the networks we need the mapping between name and IP address-range because policy specification is done using network names, but test packets need IP addresses.

For the firewalls we need to know the IP addresses of their interfaces and their capabilities. The IP addresses of the firewall interfaces tell us which networks are adjacent to which firewalls. To be able to test a connection between two networks, we need to know which firewall separates these networks. The capabilities — packet filter or application level firewall — tell us at which level we have to test.

For the servers we need to know the IP addresses and the services they offer. The need for IP addresses is the same as with the networks. With the services it is different: They are not needed for firewall testing — whether a firewall forwards some protocol to a server does not depend on the server accepting that protocol — but intended as a help to the policy writer.

Specification language To specify this low-level information, we designed the language given in Figure 3.2. We will call specifications written in this language *network layout*. Note that the language at hand is very simple on purpose. Our aim was to make it very easy to specify a network layout. The language contains all the needed elements, as

```
NETLAYOUT  =  NETWORKS '***' FIREWALLS '***' SERVERS
NETWORKS   =  {NET | COMMENT}
NET        =  NAME':' RANGE {',' RANGE}
NAME       =  letter {letter | digit} .
RANGE      =  IP'/'DD | '!'NAME
IP         =  DDD'.'DDD'.'DDD'.'DDD .
D          =  [digit] .
COMMENT    =  '@' TEXT '\n'
TEXT       =  {letter | digit | ' '}
FIREWALLS  =  {FIREWALL | COMMENT}
FIREWALL   =  FW IF PROP
FW         =  NAME
IF         =  ['eth'digit] '('IP')'
PROP       =  [TEXT]
SERVERS    =  {SERVER | COMMENT}
SERVER     =  NAME IP PROTO
```

Figure 3.2: Grammar for textual network layout

```
DMZ: 172.16.72.0/24
Private (Intranet): 192.168.70.0/24
Internet: !DMZ, !Private
                    ***
@ Name of the Firewall  Interface              Type
FW1                     eth0 (129.132.178.193)
FW1                     eth1 (172.16.72.1)
FW2                     eth0 (172.16.72.3)     Packet filter
FW2                     eth1 (192.168.70.3)    Packet filter
                    ***
          @ Name (fac.) IP           Service
          Mailserver    172.16.72.4  smtp
          Mailserver    172.16.72.4  imap
          Webserver     172.16.72.5  http
```

Figure 3.3: A sample textual network layout

identified above. An example of its use can be found in Figure 3.3, which is the textual representation of Figure 3.1.

There are different ways to gather the information needed for the network layout: discovery[1] and declaration. We do not have preferences in how to acquire this data. Our testing methodology is independent of how this information is gathered. It only has to be valid and (preferably) complete.

As humans prefer graphical representations, the specification of the network layout can also be done graphically. For the use in our test case generation, such a graphical specification can be converted to a textual one. For this purpose, a tool called *NetMap* has been written by Markus Frauenfelder [Fra05]. More information about this tool can be found in section 5.1.

3.2 Formal Policy and Keyword Definitions

As stated in section 2.3, a **security policy** should express **what** should be achieved and **not how** it should be achieved. Consider the following example: An individual having the policy to only travel to secure countries. This individual will stick to his policy for his whole life. During this time, the interpretation of a secure country can change. An armed conflict or the bird flu will most probably render a country insecure (for this individual). Thus, before travelling somewhere, he might consider the recommendations of the "Bundesamt für Gesundheit" on secure destinations.

By this example we can see very well that the *abstract* description (secure country) is *stable*, but the *concrete* description changes over time. The abstract description defines criteria that can be evaluated by a third-party (Bundesamt für Gesundheit) to get the *current* concrete description. This shows very nicely why low-level (implementation) details do not belong in a policy, which also holds for security policies. Thus when designing a formal language for the specification of security policies, we incorporate this separation of concerns.

Reasons for a formal policy Why do we have to formalise security policies? The aim of our work is to test a given set of firewalls for conformance to a policy. For this, the policy must be clearly defined. Thus we need a formal language to state at least the network policy, which is part of the security policy.

In practice, network policies are either expressed informally (just some text) or as access models. Most often a policy even does not exist in a written form, but only in the head of the firewall administrator. Unfortunately, no formal policy language exists. Therefore we have to design one. In the remainder of this thesis we will refer to a network policy written in our policy language as a *formal policy*.

[1] The UNIX-commands `/sbin/ifconfig` and `/sbin/route`, for example, can be used on each firewall to discover the information about the firewalls and the networks.

Demands on security policies There are only few companies using security policies. Unfortunately, most of them do not want to disclose their policies because they fear that this would have a negative influence on their security. Even if *security by obscurity* is not something one should build upon, we have to live with this fact.

Due to the situation, we based our requirements analysis on a few policies from universities[2] and on guides to developing security policies [Gro97]. We decided to start with a simple policy language.

Our demands for a policy language are that it is *high-level* and *stable*. The intuition for this demands is given by the above example, but there are also arguments. A policy has to be *stable* mainly to minimise errors: Every change is inserting a possible error into the policy. If changes happen too often, it is not feasible to check the policy every time for correctness (= stating what we want). An additional issue is that changing a policy too often, renders it nearly impossible for people to know what they have to adhere to.

A policy has to be *high-level* because Managers need to understand it (they are the ones with the decision power), to get the big picture (firewall rulesets often have thousands of rules making it impossible to determine what they are all about), and to make it stable.

Example (high-level and stable policy) We illustrate this at another example. Consider the following two statements

1. "Only company employees are allowed to possess a key to the company's main entrance."

2. "Only Fred, Bob, ... are allowed to possess a key to the company's main entrance.", where Fred, Bob, ... are all employees of the company.

The first statement is preferable because of the following points:

- It is easy to understand (one does not have to think about the connection between Fred, Bob, ...).

- It is always correct (we do not have to alter it every time somebody joins or leaves the company).

- It is on the right level (being allowed to possess a key depends on a persons status and not on his name. The mapping status to names can then be made by the HR department which does not have to know anything about this policy).

Specification language After having discussed the nature of a formal policy, let us discuss its ingredients. Recall our setting, where we have networks with machines in them and communication taking place between the individual machines. We are interested in all communications which cross a firewall. And we are interested in the same kind of abstraction as in the network layout: networks and servers, but not clients.

[2]Interestingly, some universities have their security policies on their webpages.

POLICY	=	{RULE | COMMENT}
RULE	=	SOURCE '→' DEST : ACTION KEYWORDS
SOURCE	=	NETWORK
DEST	=	NETWORK
NETWORK	=	NAME
NAME	=	letter {letter | digit} .
ACTION	=	'ACCEPT' | 'DENY' .
KEYWORDS	=	('∗′ | NAME) {',' KEYWORDS}
COMMENT	=	'@' TEXT '\n'
TEXT	=	{letter | digit ...}

Figure 3.4: Grammar for network policies

@ Connections to Private	
DMZ → Private:	ACCEPT securetraffic
Internet → Private:	DENY ∗
@ Connections to the DMZ	
∗ → Webserver:	ACCEPT webtraffic
∗ → Mailserver:	ACCEPT mailtraffic
@ Connections to the Internet	
Private → Internet:	ACCEPT ∗
DMZ → Internet:	DENY ∗

Figure 3.5: A sample formal network policy

KEYWORD-DEFINITIONS	=	{DEFINITION | COMMENT}
DEFINITION	=	NAME '=' PROTO {',' PROTO}
NAME	=	letter {letter | digit} .
PROTO	=	letter {letter | '-' | '+' | digit | '.' | '_'} | NUM .
NUM	=	{digit} .
COMMENT	=	'@' TEXT '\n'
TEXT	=	{letter | digit ...}

Figure 3.6: Grammar for keyword definitions

securetraffic	=	ssh, scp, https, imaps
webtraffic	=	http, https
mailtraffic	=	smtp, imap, imaps

Figure 3.7: Sample keyword definitions

We need to be able to specify how a network or server is allowed to communicate with another network or server. We do this by rules (RULE) that say who (SOURCE) is allowed or denied (ACTION) what (KEYWORDS). Here the KEYWORDS stand for a high-level category which can then be defined separately. This helps us achieve a high-level, stable policy. The low-level details can be specified in the *keyword definitions*, which may be changed frequently. These changes need not be made by the person writing the policy (division of concerns).

The grammars for formal policy and keyword definitions are given in figures 3.4 and 3.6, respectively. Examples of their use (for the network in Figure 3.1) are given in figures 3.5 and 3.7, respectively.

Note that it depends on the testing level (packet or application level) how a protocol in the keyword definitions is interpreted. If we test on the application level, 'ssh' really means the ssh-protocol, whereas on the packet level we can only interpret 'ssh' as destination port 22.

3.3 Summary

In this chapter we have presented formal languages for the specification of security policies, network layouts, and keyword definitions. All the languages are simple on purpose. An easy understandable language was more important to us than covering every special case.

Security policies, network layouts and keyword definitions are used as input to our testing methodology, which is presented in chapter 4. We will see in chapter 5, that our simple languages serve their intended purpose.

Chapter 4

Test Methodology

Recalling Figure 1.2, we want to generate test cases – for testing the conformance of firewalls to a policy – from the following ingredients: a formal network policy, a network layout and a keyword definition. After having introduced these in chapter 3, we now focus on the main part, the test case generation.

4.1 Test Objectives

Before talking about how and what to test, we first discuss the reasons for conducting tests and about the expected outcomes of such tests.

Roughly we can distinguish between the *first test* of a system before it is deployed and *consecutive retests* after a changing of rules or policy or the substitution of a firewall (software). What a test aims to show can be divided into three *categories*:

1. correct firewall implementation,
2. correct firewall specification (ruleset conforms to the policy),
3. resistance against attacks.

Resistance against attacks For the third category many tools [Aud06, GFi06, Sec06] already exist that test aspects like vulnerability to known attacks, origin validation (no spoofing possible), ways around the firewall (does traffic really has to pass the firewall) and so on. As already stated in section 1.2, we will not cover this category. Instead we recommend to use the existing methods for this kind of test complementary to our method.

Correct firewall implementation The first category is normally tested by the firewall vendors or by independent companies (see subsection 6.2.1). These tests consider timing, behaviour under stress, logging and so on. As not every firewall undergoes such a test, and as this is a really important point – the conformance of the ruleset to the policy is

worthless if the firewall implementation has a serious bug – we have decided to test the firewall implementation as well.

Correct firewall specification And finally the second category: As this depends on a policy, it cannot be tested once and for all. Rather it has to be tested individually for every policy. Furthermore, it has to be retested after every change of a policy. This is much work, but it is crucial to test this category as well. Companies rely on their firewalls. A firewalls that is not doing its job correctly, represents a major security issue. Unfortunately, to our knowledge, the only method for conformance-testing is handwork by experts. Thus, due to the importance and the missing methods, our biggest interest lies in this category.

Outcome of a test For each of these three categories, the the outcome of a test is different: implementation, design or configuration errors. The effect of an attack is also important. If the attack causes that no other traffic can pass the firewall (for some time) this is a different problem than if all traffic is let through the firewall. In the first case, we have denial of service (DoS). Not being able to do business, costs large amounts of money. In the case of banks or international parcel services (like DHL), some hours or few days will result in bankruptcy. In the second case, the problem is that everybody can enter a network unhindered and compromise machines.

4.2 The System under Test

In this section we will elaborate on how a system under test looks like, before giving our test methodology in the next section.

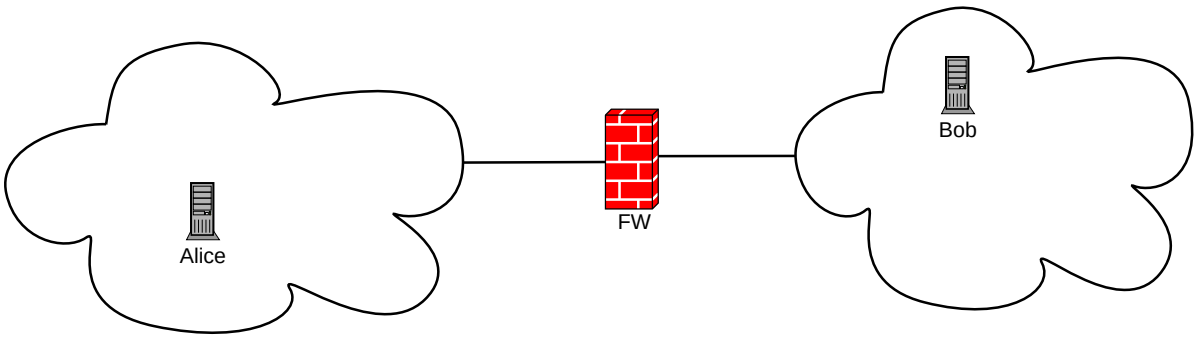


Figure 4.1: System under test

Figure 4.1 shows a system under test. It consists of two *networks*, with one machine each (Alice and Bob), and a *firewall*. All the traffic between Alice and Bob has to pass

the firewall. The firewall's job is to *filter* this traffic according to its *rules* (which should implement the *security policy*).

High-level versus low-level view When testing such a system, there is a high-level and a low-level view. Seen from a high-level, we simulate connections between Alice and Bob. We want to determine if the firewall only allows policy conform connections. To simulate a connection, we have to, at the low-level, simulate the endpoint automata. This is reflected by taking two ingredients for our tests: *test tuples* telling us which connections we have to test and how we expect the firewall to react, and *abstract test cases* which simulate the endpoint automata for us — Actually we simulate the expected firewall behaviour in a protocol run between two endpoints.

Questions answered When proceeding as described above, we answer the following questions with our tests:

1. Are the protocol automata of the firewall correct[1]?
2. Are only connections allowed by the policy accepted by the firewall?
3. Are all connections allowed by the policy accepted by the firewall?

Question 1 solely *depends on the firewall soft- / hardware used* and needs to be answered every time we encounter a new (version of a) firewall. To answer this question, we need models of the relevant protocol automata. How to generate *abstract test cases* from these is explained in subsection 4.3.1.

Questions 2 and 3 solely *depend on the policy* and need to be answered every time a policy is created or changed. To achieve this, we generate test tuples from the policy and use them to instantiate the abstract test cases. How test tuples are generated is shown in subsection 4.3.2.

4.3 Test Case Generation (Technical Details)

An overview on the relation between the different concepts introduced in this section can be found in Figure 4.2.

4.3.1 Abstract Test Cases

As described above, we use abstract test cases to determine if the protocol automata of a firewall under test are correct. Recall that firewalls monitor the state of a connection by internal automata (see section 2.4). We will call these automata *midpoint protocol*

[1]The protocol automata of a firewall are correct if they are equivalent to their specification. How these specifications look like, is discussed in part II of this thesis.

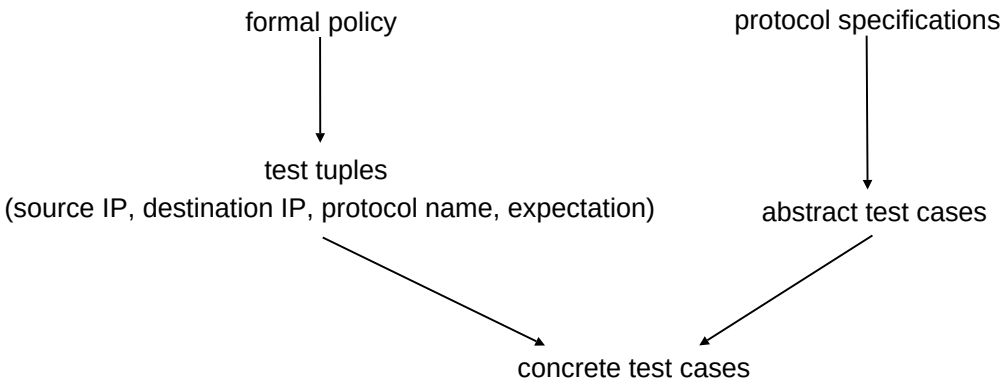


Figure 4.2: Test ingredients

automata. In part II we will elaborate on why midpoint automata are different from endpoint automata and how they look like. For the moment we just assume their existence.

To generate abstract test cases for a protocol p, a specification of a midpoint protocol automaton for protocol p is needed. As the firewalls behave as Mealy machines — a packet reaches the firewall (input) and is either forwarded (output) or dropped (no output) — such a specification must be in the form of a Mealy machine. For the sake of simplicity we will not consider modification — as with NAT — at the moment.

Starting from a specification, test cases can be generated using well-known methods. These methods were introduced and evaluated in section 2.6. As already explained, we chose the Wp-method, as it is the most widely applicable method with full fault coverage.

4.3.2 Test Tuples

A test tuple is a four-tuple (sIP, dIP, *proto*, *exp*), where sIP and dIP represent IP addresses, *proto* is the name of a protocol, and $exp \in \{ACCEPT, DROP\}$ represents an expectation. Note that we do not consider other firewall actions, like sending ICMP error codes. A test tuple describes whether a connection from the source sIP to the destination dIP (direction matters) using protocol *proto* is allowed by the formal policy. If the policy allows a connection, we expect the firewalls to let this data through, and therefore *exp* in this case would be $ACCEPT$. If a connection is not allowed (or explicitly forbidden) by the policy, *exp* will be $DROP$. Test tuples are policy-specific and thus must be generated for every policy. Note that the statefulness of a connection is not modelled by these test tuples, but rather by the abstract test cases.

Conversion to low-level rules We generate test tuples in two steps: first we convert them to low-level rules, then we select test tuples. The first step works as follows. We combine the formal policy with the low-level details contained in the keyword definitions and the textual network layout. This means that we transform every rule

`source → destination:  action keyword`

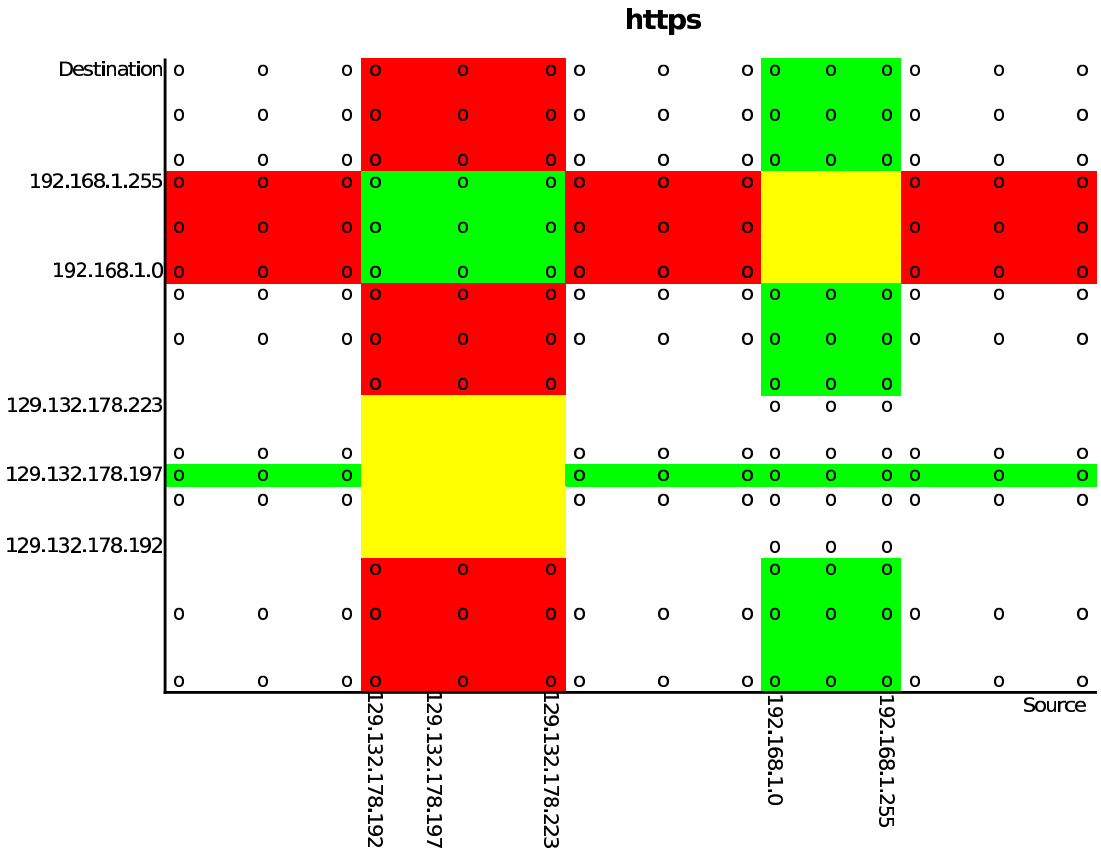


Figure 4.3: Policy for https with test points

from the formal policy into n low-level rules, where n is the number of protocols contained in *keyword*. In these low-level rules, the names of `source` and `destination` are replaced with the corresponding IP ranges (`sIPr` and `dIPr`).

These low-level rules can be represented graphically using one two-dimensional graph per protocol, where the x-axis represents the source IP addresses and the y-axis represents the destination IP addresses. For each low-level rule

`sIPr` $\rightarrow$ `dIPr: action protocol`

the cross-product $sIPr \times dIPr$ defines a rectangular region in the graph. We colour this region according to the given action: green for ACCEPT, red for DROP.

Implicit policy statements Until now, we just considered what the policy explicitly states. But we should also test implicit statements, i.e. what is not explicitly allowed is forbidden. This is best explained on the graphical representation. In the graph, we coloured all the areas where we have an explicit policy statement (either in green or red). This means that for all the yet uncoloured areas there exists no explicit policy statement. Note that a part of these areas is not testable since, as we stated earlier, policies for traffic within zones cannot be enforced by firewalls (this part is marked in yellow). The

remaining uncoloured areas can be partitioned into rectangles and then test tuples can be chosen, analogous to the procedure given above, where the expectation is set to DROP.

Ambiguous policies Overlapping rectangular regions (from explicit policy statements) mean that the *policy is ambiguous*. In such a case the testing process is stopped and a warning is issued. This warning tells the policy engineer which rules interfere and how they interfere. Based on this warning, a policy engineer can then remove the ambiguity in the policy.

Test tuple selection The second step of our test tuple generation is the selection of test tuples from the low-level rules. This is necessary because it is generally infeasible to test every possible combination of IP addresses. However, as we assume uniformity within zones, it is sufficient to choose for each low-level rule an arbitrary IP from the source IP range and an arbitrary IP from the destination IP range. As boundary points are a source of errors in practice, we also select addresses to test these. That is, we choose the lowest IP address, an arbitrary (intermediate) IP address, and the highest IP address per range. This results in nine (three times three) test tuples per low-level rule. An example (for the formal policy in Figure 3.5, the network layout in Figure 3.3, the keyword definitions in Figure 3.7 and the https protocol) is given in Figure 4.3. Note that taking special addresses (e.g. private networks and broadcast) into account is left for future work.

4.3.3 Concrete Test Cases

In the last two sections, we have explained the generation of test tuples and abstract test cases. Recall that abstract test cases test the correct stateful handling of a protocol, and they contain variables for source and destination addresses (A and B respectively). Recall further that test tuples are of the form *(sIP, dIP, proto, exp)*, formalising whether a connection from the IP address *sIP* to the IP address *dIP* using protocol *proto* is allowed by the policy or not. We now explain how to instantiate the abstract test cases with the test tuples. The resulting concrete test cases then can be used to test whether the policy is correctly implemented in a stateful manner.

Instantiation Given a test tuple *(sIP, dIP, proto, exp)* and abstract test cases a_i for the protocol *proto*, the instantiation proceeds as follows:

- replace every occurrence of A in every a_i with *sIP*,
- replace every occurrence of B in every a_i with *dIP*, and
- if *exp* equals *DENY* then replace the expected output in every a_i with "–".

The resulting test data represent network packets. These packets can then be built and injected into the actual network and the results (the answer of the firewall to the packets) can be compared to the expectations of the given test cases. Refer to section 5.1 for a description of *fwtest* which we implemented for this purpose.

Two-stage test As already stated in section 2.6, we must assume that the midpoint protocol automata do not change during the test. If the midpoint automata would change, we could make no statement at all about their correctness (at a certain point in time). Using the above assumption, we do not need to test the protocol automaton for every test tuple. Instead we divide our tests in two stages:

1. test the midpoint protocol automata, and
2. test the conformance to the policy.

For the first stage, we only take a *few accepting test tuples* per protocol to instantiate the abstract test cases. If the midpoint automaton does not change during the test, the test outcome regarding the midpoint automaton should be the same for all test tuples. Therefore one test tuple is enough. Taking some more, helps minimising environmental influences and the possibility of taking a test tuple with a wrong expectation.

For the second stage, we instantiate *a run-through* (shortest path from the start state to the end state of the midpoint automaton) of the protocol with the test tuples. This is enough as we only have to distinguish between accept and drop in this stage. If the midpoint automaton is correct – which is determined by stage one – it will either accept all possible run-throughs or none. Therefore one run-through is enough to decide.

Using this two-stage testing yields less test cases without loosing accuracy: As stated above, the midpoint automaton is either always correct or never. Further, the two-stage procedure enables us to only run one stage of the test at a time: if we have a new firewall (version) we only run the first stage of the test, whereas if we changed the policy or the firewall ruleset we only run the second stage of the test. Running both stages will only be required the first time a firewall setup is tested.

If both testing stages are successful, it can be assumed that the firewall (configuration and implementation) under test is working correctly. For the practitioner this is satisfying. But we should never forget about Dijkstras famous saying "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij70].

4.3.4 Remarks

Until now we were abstracting from the differences between packet level and application level. In this section we want to give some level-specific remarks to extend the general explanations given above.

For the rest of this thesis we will then restrict ourselves to the packet level. More precisely, we will just look at TCP from now on. But as UDP (stateless) is much simpler than TCP (stateful), all the results can be applied to UDP as well. This has the following reason: The application level builds on the packet level. Thus we first have to understand the packet level, before being able to work at the application level.

Packet Level (TCP)
On the packet level we only need abstract test cases for TCP and UDP. Thus, instead

of instantiating the abstract test cases generated for *proto* with test tuples of the form *(sIP, dIP, proto, exp)*, we instantiate the abstract test cases for TCP with these tuples. To model *proto* at the TCP-level, we use the TCP port-number *pnum* of *proto* as the destination port. Thus, B in the abstract test cases is replaced with *dIP:pnum* (instead of *dIP*) in this case, to produce the concrete test cases. Note that we do not take sequence numbers, timing, and fragments into account. An example looks as follows:

an abstract test case:	(rst: A → B / rst: A → B) (fin: A → B / -) (syn & ack: B → A / -) (syn: A → B / syn: A → B)
a test tuple:	(1.1.1.2, 3.3.3.0, https, ACCEPT)
the resulting concrete test case:	(rst: 1.1.1.2 → 3.3.3.0:443 / rst 1.1.1.2 → 3.3.3.0:443) (fin: 1.1.1.2 → 3.3.3.0:443 / –) (syn & ack: 3.3.3.0:443 → 1.1.1.2 / –) (syn: 1.1.1.2 → 3.3.3.0:443 / syn: 1.1.1.2 → 3.3.3.0:443)

In this example, the abstract test case simulates one part of the TCP protocol. Four packets need to be sent: a packet with the RST-flag set from A to B, a packet with the FIN-flag set from A to B, a packet with the SYN- and ACK-flag set from B to A and a packet with the SYN-flag set from A to B. To correctly handle the TCP protocol, the firewall has to block the second and third packet in any case. The test tuple states that a https-connection between the IPs 1.1.1.2 and 3.3.3.0 must be accepted by the firewall. Instantiating the abstract test case with these values, as described above, results in the network packets we can use to test this statement.

Application Level

With application level firewalls, the problem becomes far more complex. To answer the questions given in the introduction, application data has to be taken into account additionally. I.e. instead of answering the question *Are http connections from A to B able to pass the firewall?* we now have to answer the question *Are connections from A to B* ***not containing the words x,y,z*** *able to pass the firewall?*. Additionally, http really means the http-protocol, not port 80. I.e. we are interested if http is able to pass, not if anything on port 80 is able to pass.

To master the complexity, we need additional specifications for at least the following: 1) contents, their possible values, and their ordering (payload), and 2) the interaction between different protocols.

Another fact we have to keep in mind is that we only can test the correct handling of an application level protocol if it is unencrypted. If for example IPSEC is used, we can

only check the correct initialisation of IPSEC. And if the policy states that all connections between A and B need to be encrypted, we can test this additionally by sending unencrypted data.

Thus, to be able to test application level firewalls, we need to adapt our policy such that statements about contents of and interaction between protocols are possible. Also we need a way to describe the packet structure of a protocol, in order to correctly generate test packets. As stated above, the solution of these questions is beyond the scope of this thesis.

4.4 Practical Considerations

Preset test cases are easier to construct than adaptive ones (for a definition please refer to subsection 2.6.3). However, in our setting, certain values cannot be determined in advance. The correct acknowledgement number of a packet, for example, is known only at the time the packet is sent, because it depends on which previous packets reached their destination.

We decided to take the best of both approaches: We pregenerate the test cases as far as possible (preset). The missing information – sequence and acknowledgement numbers for all test cases, and ports and IPs in the case of NAT – is then completed during the test (adaptive).

4.5 Summary

When generating test cases we basically need to take two things into account: the policy they have to represent and the protocol specifications of the used protocols. As protocol specifications do not change, but policies do, we use a two-phase generation.

In the first phase, abstract test cases are generated for every protocol (if not already present) on the one hand and test tuples are generated for the policy on the other hand. The abstract test cases for a protocol represent correct (and incorrect) runs of that protocol. They only need to be generated once per protocol. The test tuples represent the policy in the form of connections to be tested and thus need to be (partially) regenerated after every change of the policy.

Instantiating the abstract test cases with the test tuples in the second phase yields concrete test cases. These concrete test cases specify test packets that can directly be fed into the network to test if the policy is implemented correctly by the firewalls.
To minimise testing efforts, we can restrict ourselves to testing only the stage needed:

- Testing the midpoint protocol automaton of a firewall – by instantiating the abstract test cases with only a few accepting test tuples – after a change of the firewall (software).

- Testing the conformance to the policy – by instantiating only a run-through of the protocol with the test tuples – after a change of the policy, keyword definitions, network layout or firewall rules.

Note that these two stages of test execution are not the same as the two phases of the test generation.

If the outcome of a test is not as expected, this can have one or several of the following causes: incorrect firewall rules, an incorrect firewall implementation, a bug in our tool, or different network problems. Searching the cause of an unexpected test outcome has to be done by hand. This is due to the fact that we treat firewalls as black boxes, thus cannot search for errors in their rules.

Chapter 5

Validation

In section 1.2 we stated our thesis as follows: It is possible to construct a method for the specification-based testing of firewalls, whose resulting test-cases can be used to test whether a specific installation satisfies the stated policy. In this chapter we validate this thesis.

This chapter is organised as follows: First, we explain our prototype tool suite which we implemented for the purpose of validation. Then we give a small example presenting how to conduct a whole test run, before we end with a real-world case study.

5.1 Tools

With the help of students we have implemented an essential part of our research in a prototype test harness. Figure 5.1 gives an overview of the various tools and their relation to each other. Note that we do not have an implementation for the test tuple generation. For the example test run in the next section and the case study in section 5.3, the test tuple generation is done by hand.

NetMap As explained in section 3.1, we need to know the network under test in order to generate test cases. This network layout can be graphically specified in NetMap, written by Markus Frauenfelder [Fra05], and then be converted to a textual representation needed for the generation. Additionally the conversion works in the other direction (textual to graphical) as well. Unfortunately the handling of the tool is not very intuitive. While this is not a problem for a prototype tool, this is definitely not desired in an industrial-strength tool.

End-Mid.lhs This is the implementation (see Appendix B for the code) of Part II of this thesis in Haskell. By entering the specification of an endpoint automaton, the corresponding midpoint automaton can be derived on a step-by-step basis. The resulting

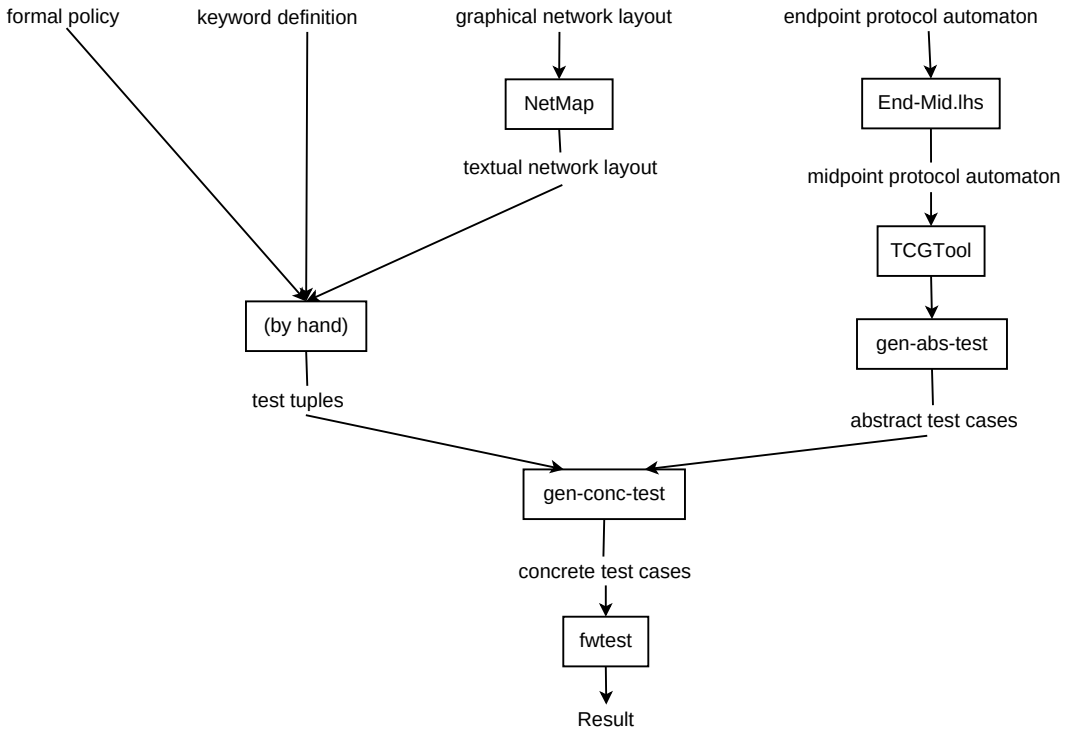


Figure 5.1: Our tools – the big picture

midpoint automaton then needs to be minimised[1] before it can be entered (graphically) into the TCGTool.

TCGTool The TCGTool (TCG standing for Test Case Generation) is an extension of JFLAP [RF] written by Stefan Hildenbrand [Hil05]. The TCGTool can generate abstract test cases for protocols: protocols can be specified graphically in the form of Mealy machines. The alphabet being used can be defined by the user. Using the Wp-Method described in section 2.6, abstract test cases are generated within seconds.

Even if there are some shortcomings (e.g. the tool only works with minimised automata), this tool represents an extraordinary work. Also, it is the first such tool which is freely available[2]

[1]Neglecting the triples of states (see section 9.2), i.e. only considering their incoming and outgoing transitions, we get a lot of equivalent states that can be merged.

[2]With the agreement of the JFLAP authors, we put the TCGTool under the GPL. Also with the hope that others in the field will invest their time in extending it instead of inventing the wheel once more (there already existed such a tool – see `http://www.site.uottawa.ca/~ural/tsg/` – but unfortunately it was not available for use). Somebody who wanted to invent the wheel once more, and is now using and extending our tool, is Dr. Mark Utting from the University of Waikato, New Zealand.

gen-abs-test `gen-abs-test` is a small compiler that converts the output of the TCG-Tool (test cases and alphabet) to *abstract test cases* in fwtest's input format. These test cases contain variables for IPs and ports.

gen-conc-test `gen-conc-test` is a small compiler that uses given test tuples to instantiate the abstract test cases generated by gen-abs-test. The resulting concrete test cases can then directly be used with fwtest.

fwtest The first version of *fwtest* was written by Gerry Zaugg in his diploma thesis [Zau04]. This initial version (0.5) of fwtest was able to craft, inject, capture and analyse predefined test packets as shown[3] in Figure 5.2. The tool was written with TCP and UDP in mind, but at that time only TCP was supported.

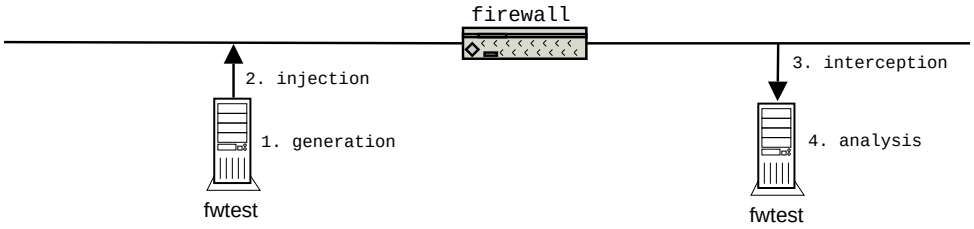


Figure 5.2: fwtest v0.5

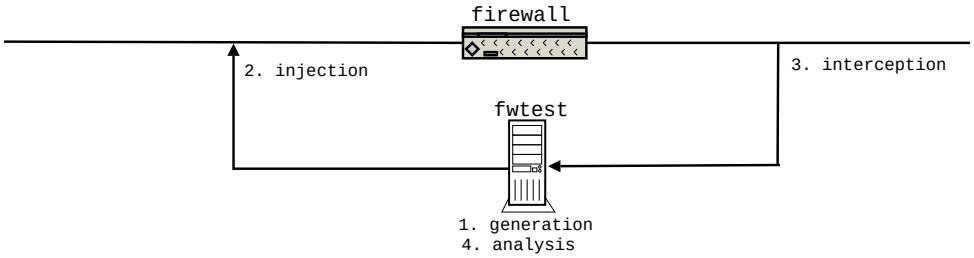


Figure 5.3: fwtest v1.0

Fwtest was extended (v1.0) by Beat Strasser [Str06] and Adrian Schüpbach [Sch06] to support UDP, ICMP and NATting. The test is now conducted by a single test instance (see Figure 5.3) instead of two.

Analysis of the Result As we are treating firewalls as black boxes, the cause of an unexpected test outcome has to be searched by hand with the help of the fwtest log-files.

[3]While there is only an unidirectional flow shown, fwtest works bidirectional.

5.2 An Example Test Run

In this section we demonstrate, step by step, a complete test run with the help of a simple example.

5.2.1 Policy

Our formal policy, network layout and keyword definitions can be found in Figures 5.4, 5.5, and 5.6.

```
@ Connections to the Intranet
DMZ → Intranet:              DENY *

@ Connections to the DMZ
* → Webserver:               ACCEPT webtraffic
* → Mailserver:              ACCEPT mailtraffic
```

Figure 5.4: Demo – formal policy

```
                    DMZ: 172.16.0.0/16
                    Intranet: 192.168.0.0/16
                              * * *
@ Name of the Firewall  Interface          Network behind  Properties
charly                  eth0 (172.16.70.3) DMZ             Packet filter
charly                  eth1 (192.168.72.3) Intranet       Packet filter
                              * * *
                @ Name (fac.)  IP           Service
                Mailserver     172.16.70.5  smtp
                Mailserver     172.16.70.5  imap
                Mailserver     172.16.70.5  imaps
                Webserver      172.16.70.4  http
                Webserver      172.16.70.4  https
```

Figure 5.5: Demo – network layout

```
webtraffic   =   http, https
mailtraffic  =   smtp, imap, imaps
```

Figure 5.6: Keyword definitions

5.2.2 Setup

To do firewall testing we need at least two machines: a firewall and a tester. These are connected as given in Figure 5.7.

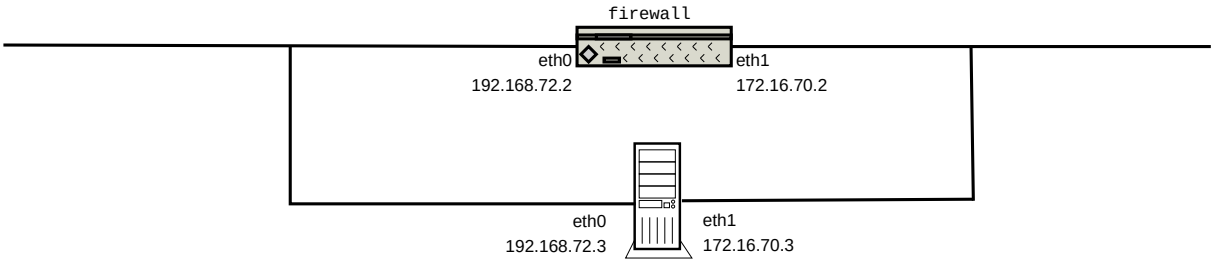


Figure 5.7: Network setup

Firewall-Setup

For the firewall we choose a Checkpoint product, which we configure as follows:

- Install Checkpoint R55W
- configure the interfaces
  - eth0 192.168.72.2
  - eth1 172.16.70.2
- define hosts and networks
  - Intranet 192.168.0.0 255.255.0.0
  - DMZ 172.16.0.0 255.255.0.0
  - Webserver 172.16.70.4
  - Mailserver 172.16.70.5
  - alice 192.168.72.3
  - bob 172.16.70.3
- write the rules (Figure 5.8)
- install the rules

Tester-Setup

On the tester we need at least fwtest installed. If we want to generate our test cases on the tester, we need to install all the other tools – TCGTool, gen-abs-test, and gen-conc-test – as well.

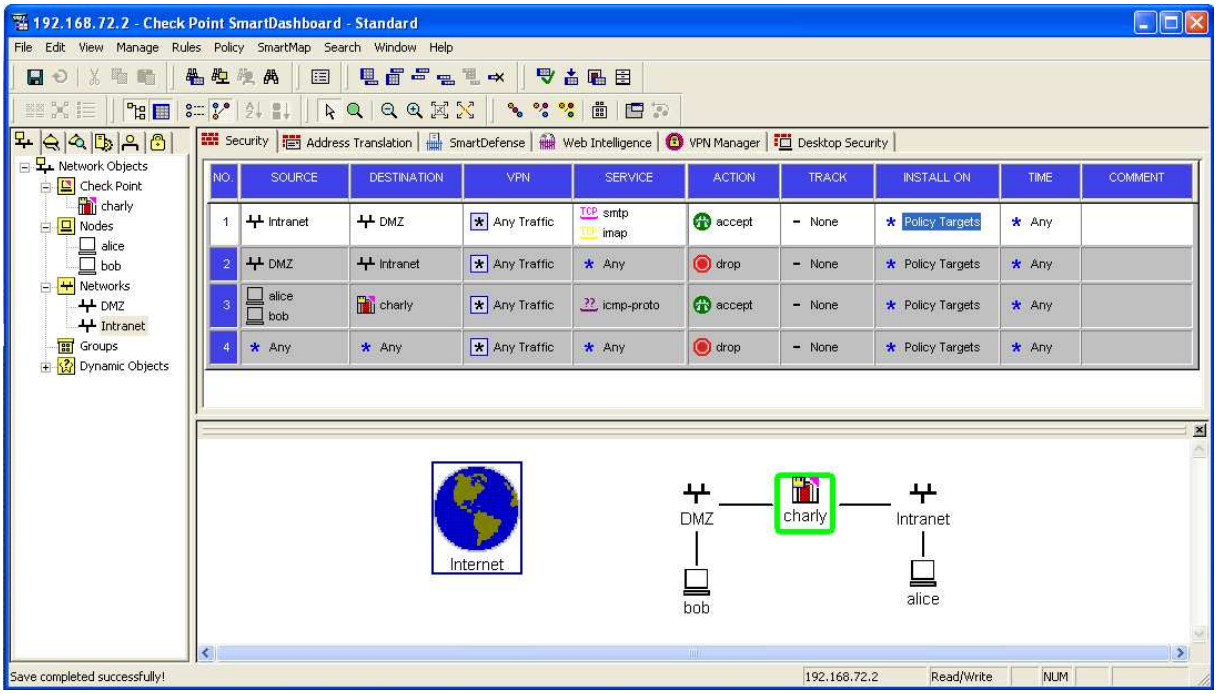


Figure 5.8: Firewall rules

Configure the interfaces:

```
ifconfig eth0 down
ifconfig eth1 down
ifconfig eth0 192.168.72.3
ifconfig eth1 172.16.70.3
route add -net 192.168.0.0/16 gw 192.168.72.2
route add -net 172.16.0.0/16 gw 172.16.70.2
```

5.2.3 Test Case Generation

Generation of test tuples

The test tuple generation, as of subsection 4.3.2, was done by hand. It resulted in 75 test tuples (`Small_Example.diana`):

```
(172.16.0.0, 192.168.0.0, http, DROP)
(172.16.70.2, 192.168.0.0, http, DROP)
(172.16.255.255, 192.168.0.0, http, DROP)
[..]
(192.168.0.0, 172.16.70.4, https, ACCEPT)
[..]
```

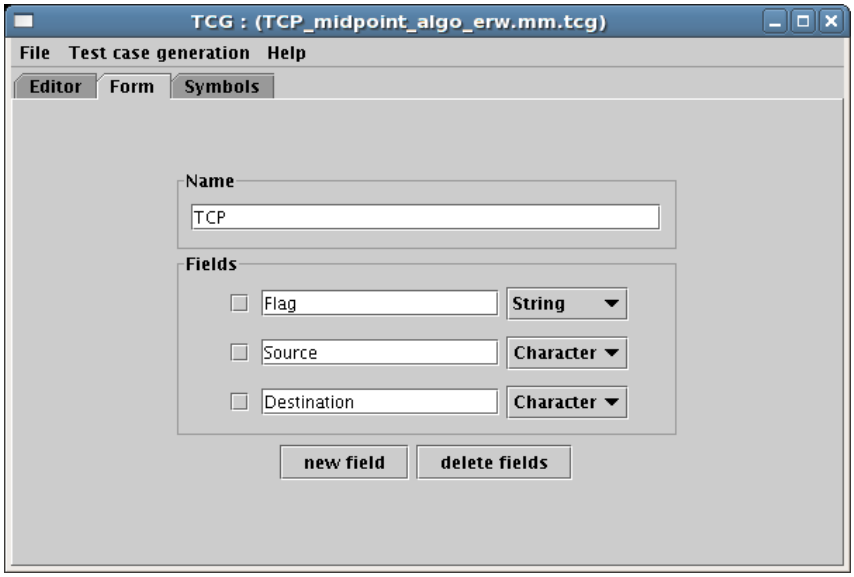


Figure 5.9: TCGTool – The form of a TCP packet

Generation of abstract test cases

We begin with drawing the TCP automaton of Checkpoint R55W in the TCGTool:

```
< dsenn@gin > java-1.4.2 -jar TCGTool.jar
```

We first have to define the fields of a TCP packet[4] (Figure 5.9 – `TCP.alpha.tcg`), before we can specify the individual packets (Figure 5.10). Then we can draw the TCP automaton of Checkpoint R55W (Figure 8.5 – Figure 5.11 – `TCP_CheckpointR55W.mm.tcg`). To generate the abstract test cases, we choose `test case generation` → `Wp-method` from the menu. The resulting abstract test cases look as follows, where every line represents a test case, and each (input/expected output)-tuple represents a test packet.

```
(SYN_A_B/SYN_A_B)(FINACK_A_B/FINACK_A_B)(SYN_A_B/SYN_A_B)
(SYN_A_B/SYN_A_B)(SYNACK_B_A/SYNACK_B_A)(ACK_A_B/ACK_A_B)[..]
(SYN_A_B/SYN_A_B)(ACK_A_B/ACK_A_B)(ACK_B_A/-)
[..]
```

These abstract test cases then need to be converted to the fwtest-format:

```
< dsenn@gin > ./gen-abs-test TCP_CheckpointR55W.tests.tcg TCP.alpha.tcg \
TCP_CheckpointR55W.tp
```

```
testcase 1 {
    packet 1 { TCP send {ipB ipA portB portA R 222 -} receive ?}
    packet 6 { TCP send {ipA ipB portA portB S 1 -} receive ok}
    packet 7 { TCP send {ipA ipB portA portB FA 1 1} receive ok}
```

[4] At the moment we are only interested in header fields.

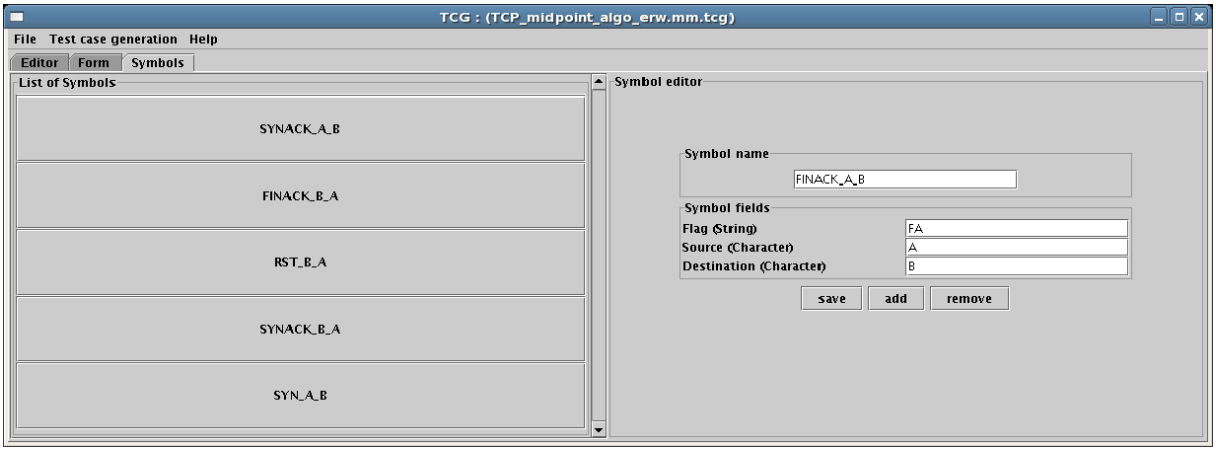


Figure 5.10: TCGTool – The different TCP packets

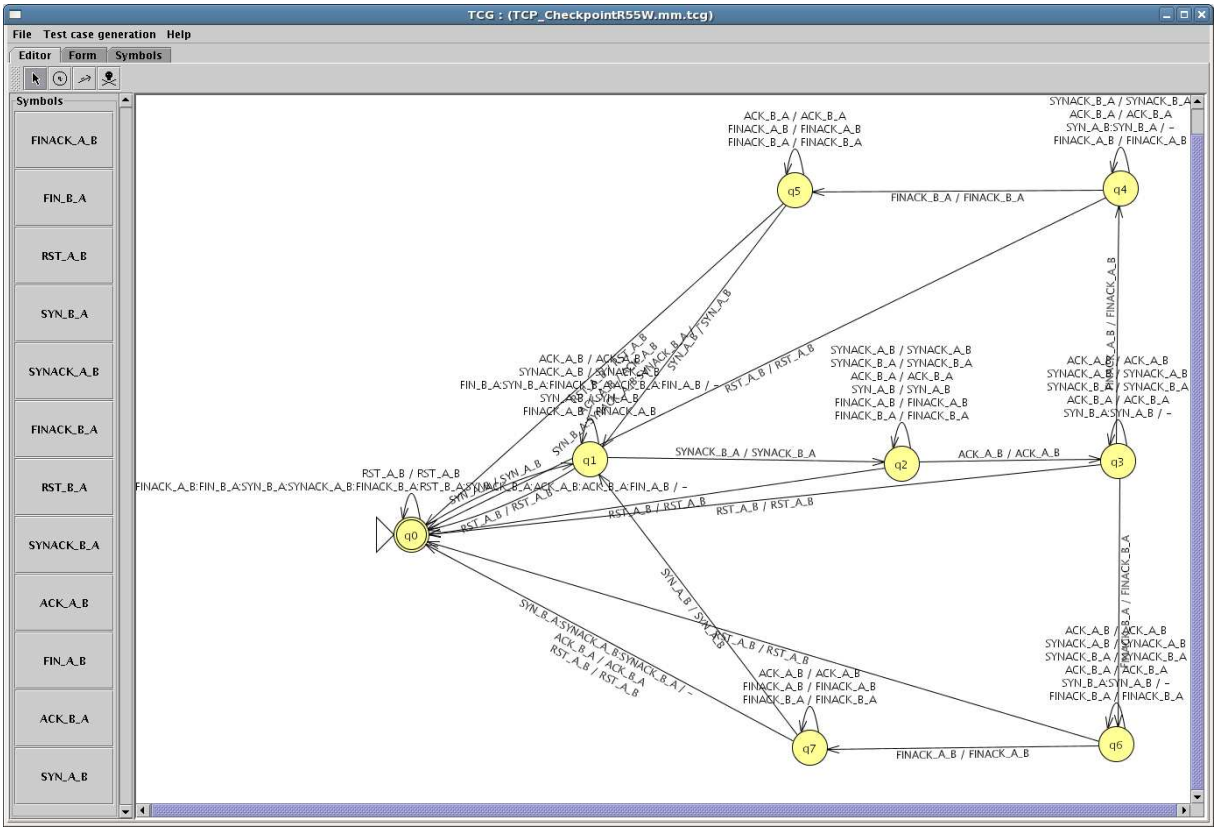


Figure 5.11: TCGTool – TCP automaton in Checkpoint R55W

```
    packet 8 { TCP send {ipA ipB portA portB S 1 -} receive ok}
}
testcase 2 {
    packet 9 { TCP send {ipB ipA portB portA R 222 -} receive ?}
    packet 14 { TCP send {ipA ipB portA portB S 2 -} receive ok}
    packet 15 { TCP send {ipB ipA portB portA SA 1 3} receive ok}
    packet 16 { TCP send {ipA ipB portA portB A 3 2} receive ok}
    packet 17 { TCP send {ipA ipB portA portB FA 4 2} receive ok}
    packet 18 { TCP send {ipA ipB portA portB R 5 -} receive ok}
    packet 19 { TCP send {ipA ipB portA portB SA 6 2} receive {}}
}
[..]
```

Note that every `testcase` represents one line and every `packet` represents one (send/ receive)-tuple from above.

Generation of concrete test cases

For this example we only run the second testing-stage: conformance to the policy.

```
< dsenn@gin > ./gen-conc-test TCP_Simple.tp Small_Example.diana \
Small_Example_simple_diana.tp
```

The concrete test cases for the second testing-stage (`Small_Example_simple_diana.tp`) look as follows:

```
testcase  1  {
    packet 1 { TCP send {192.168.0.0 172.16.0.0 http 1025 R 222 -}
                   receive ?}
    packet 6 { TCP send {172.16.0.0 192.168.0.0 1025 http S 2 -}
                   receive {}}
    packet 7 { TCP send {192.168.0.0 172.16.0.0 http 1025 SA 1 3}
                   receive {}}
    packet 8 { TCP send {172.16.0.0 192.168.0.0 1025 http A 3 2}
                   receive {}}
    packet 9 { TCP send {172.16.0.0 192.168.0.0 1025 http FA 4 2}
                   receive {}}
    packet 10 { TCP send {192.168.0.0 172.16.0.0 http 1025 FA 5 -}
                    receive {}}
}
testcase  2  {
    packet 1 { TCP send {192.168.0.0 172.16.70.2 http 1025 R 222 -}
                   receive ?}
    packet 6 { TCP send {172.16.70.2 192.168.0.0 1025 http S 2 -}
                   receive {}}
    packet 7 { TCP send {192.168.0.0 172.16.70.2 http 1025 SA 1 3}
```

```
                    receive {}}
    packet 8 { TCP send {172.16.70.2 192.168.0.0 1025 http A 3 2}
                    receive {}}
    packet 9 { TCP send {172.16.70.2 192.168.0.0 1025 http FA 4 2}
                    receive {}}
    packet 10 { TCP send {192.168.0.0 172.16.70.2 http 1025 FA 5 -}
                     receive {}}
}
```

Here we can clearly see the instantiation of the abstract test cases with a test tuple. For example `portB` from above has been replaced by `http`.

5.2.4 Execution

The concrete test cases, generated in the last section, can directly be fed into fwtest. Following the instructions in the fwtest-README the following steps are needed:

1. Ping the firewall from the tester:

   ```
   < dsenn@gin > ping -c 1 192.168.72.2
   < dsenn@gin > ping -c 1 172.16.70.2
   ```

   Note that this should either be allowed by the rules, as in our case, or be done before the policy is in place. Alternatively, the MAC-Addresses corresponding to the pinged IPs could be set by hand.

2. Install the policy on the firewall (if not already done).

3. Run fwtest on the tester:

   ```
   < dsenn@gin > ./run_fwtest.sh Small_Example_simple_diana.tp \
   Small_Example_simple_diana_20060912.log 192.168.0.0/16 eth0 \
   172.16.0.0/16 eth1
   ```

If we now examine the log-file, we see a number of false positives and false negatives, which can be divided into the following categories:

- 46 - 48: 192.168.x.x → 172.16.70.4 http ok
- 49 - 51: 192.168.x.x → 172.16.70.4 https ok
- 55 - 57: 192.168.x.x → 172.16.70.4 imap nok
- 67 - 69: 192.168.x.x → 172.16.70.5 imaps ok

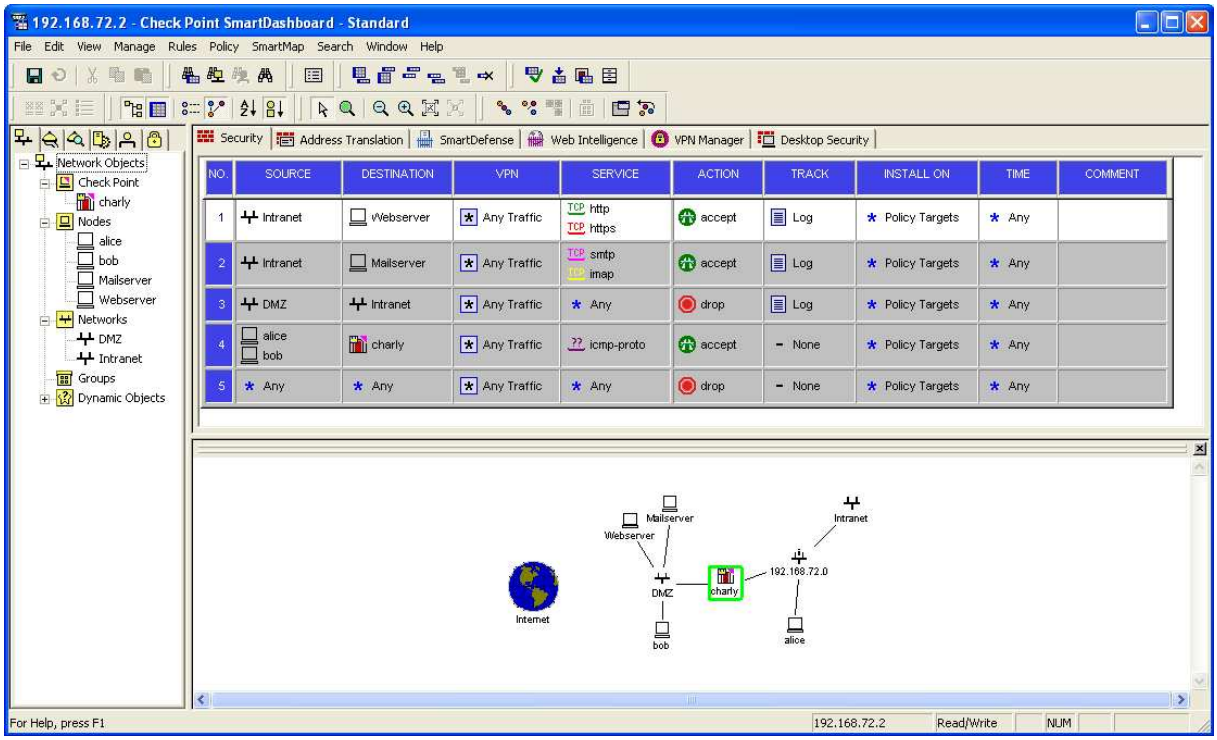


Figure 5.12: Firewall rules – improved version

For 46 - 48, 49 - 51, and 67 - 69 all packets have been blocked instead of passed. For 55 - 57 all packets were passed instead of blocked.

Comparing the ruleset (Figure 5.8) to the policy (Figure 5.4) with the above findings in mind, we see that rule 1 is wrong. We therefore change its destination to "Mailserver" and add a rule for the "Webserver". This yields the ruleset in figure 5.12. We now run our tests again and obtain the following log-file (`Small_Example_simple_diana_20060912_3.log`):

```
# Fwtest v1.0 [Firewall Testing Tool]
# 10/05/2006 11:40:14.841064
# Timeout between time slots: 1 second(s)
# [time]        [type]          [id]    [packet]
#
11:41:18.601734 false_pos       69.46   --
11:41:18.601891 false_pos       68.46   --
11:41:18.602039 false_pos       67.46   --
11:41:19.614018 false_pos       69.47   --
11:41:19.614168 false_pos       68.47   --
11:41:19.614316 false_pos       67.47   --
11:41:20.621963 false_pos       69.48   --
11:41:20.622114 false_pos       68.48   --
```

```
11:41:20.622262 false_pos       67.48   --
11:41:21.634112 false_pos       69.49   --
11:41:21.634263 false_pos       68.49   --
11:41:21.634411 false_pos       67.49   --
11:41:22.654111 false_pos       69.50   --
11:41:22.654259 false_pos       68.50   --
11:41:22.654407 false_pos       67.50   --
```

The only entries left are:

- 67 - 69: 192.168.x.x → 172.16.70.5 imaps

These reveal a problem of the firewall used: Checkpoint R55W does not seem to know `imaps`. The elimination of this problem is a task for the security officer and the firewall administrator.

5.2.5 Summary

In this section, the use of our tools for specification-based firewall testing has been shown at a small example. The specification of policy, network layout and keyword definitions was straightforward. The test case generation worked well and the resulting concrete test cases were able to reveal errors in the firewall configuration. This means that our approach works in principle.

In the next section, we will now apply our approach to a real-world setting: the firewall which separates the armasuisse "Wissenschaft und Technologie" department from the rest of the world.

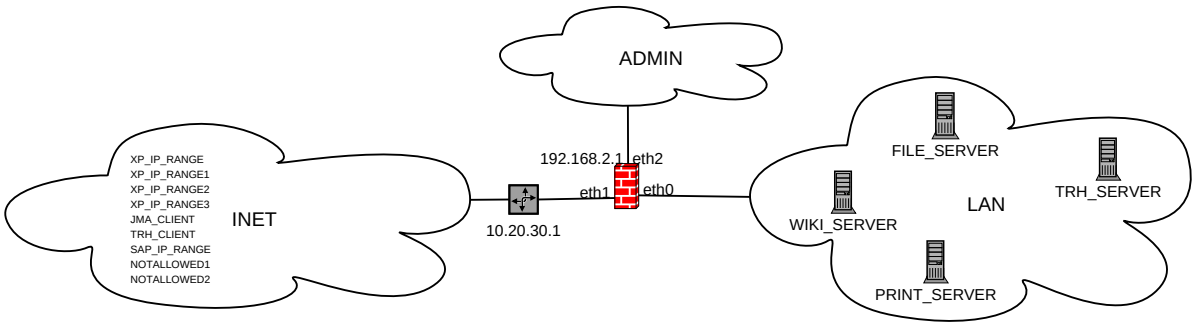
5.3 Armasuisse Case Study

5.3.1 Specification

The policy was derived in a two-step process. First, we reverse-engineered it from the comments in the firewall rules. Then Jürg Gertsch from armasuisse made some small corrections.

Network Layout

Note that the real IP addresses have been replaced by private ones for confidentiality reasons.



LAN:	10.20.30.0/24
XP_IP_RANGE:	10.20.24.0/23
XP_IP_RANGE1:	10.20.51.0/24
XP_IP_RANGE2:	10.20.48.0/23
XP_IP_RANGE3:	10.20.60.0/23
TRH_CLIENT:	10.20.99.0/24
SAP_IP_RANGE:	192.168.66.0/24
NOTALLOWED1:	192.168.33.0/24
NOTALLOWED2:	172.20.13.0/24
INET:	XP_IP_RANGE, XP_IP_RANGE1, XP_IP_RANGE2, XP_IP_RANGE3, TRH_CLIENT, SAP_IP_RANGE, NOTALLOWED1, NOTALLOWED2

**

@ Name of the Firewall	Interface	Network behind	Properties
FW	eth0 (-)	LAN	Bridge
FW	eth1 (-)	INET	Bridge
FW	eth2 (192.168.2.1)	ADMIN	

**

@ Name	IP	Service
FILE_SERVER	10.20.30.11	
PRINT_SERVER	10.20.30.16	
TRH_SERVER	10.20.30.26	
WIKI_SERVER	10.20.30.45	
LAN_BCAST	10.20.30.255	
JMA_CLIENT	10.20.71.60	
SCANNER0	172.100.200.19	
SCANNER1	172.100.100.97	
SCANNER2	10.33.0.99	

Formal Policy

@ Connections to ADMIN

@ Connections to LAN

*	→	LAN-BROADCAST	DROP *
XP_IP_RANGE1-3	→	FILE_SERVER	ACCEPT xxx, web
JMA_CLIENT	→	FILE_SERVER	ACCEPT data, xxx
*	→	PRINT_SERVER	ACCEPT control
SAP_IP_RANGE	→	PRINT_SERVER	ACCEPT print
TRH_CLIENT	→	TRH_SERVER	ACCEPT xxx
*	→	WIKI_SERVER	ACCEPT web, sec-web
XP_IP_RANGE	→	LAN	ACCEPT *
SCANNER0-2	→	LAN	DROP *

@ Connections to INET

LAN	→	!NOTALLOWED	ACCEPT *

@ OTHER

ADMIN	→	FW	ACCEPT *

Keyword Definitions

```
data = ftp
xxx = NetBios
web = http
sec-web = https
print = printer
control = icmp
```

5.3.2 Test Case Generation

We have generated the test tuples by hand. As boundary points for a \24-network, we used .2 and .254. The resulting test tuples can be found in Appendix A. These test tuples were used to instantiate the abstract test cases for a single TCP connection initiation followed by a connection tear-down (*TCP_simple.tp*), to get the concrete test cases (*armasuisse_simple.tp*).

We only did a check of the policy, but no check of the firewall. The justification for this is that we could not run our tests on the real hardware but only on a copy. Therefore we would have found bugs only in the implementation of our firewall, but not in the real one. As we already know how the automaton of our firewall looks like – see Figure 8.4 in section 8.2 – checking it again made no sense.

5.3.3 Test Execution

As we could not run our tests in armasuisse's productive environment, we duplicated the environment using VMware, which we used for our tests.

As fwtest, at the moment, is not able to simulate more than one network per side, we had to run one test per network contained in the Internet. Overall, all test cases run through without false positives (a packet we expected did not arrive) or false negatives (a packet that should have been dropped by the firewall, arrived at its destination).

5.3.4 Conclusion and Future Work

To summarise, there is good and bad news. The good news is that our approach passed the test and did work in a real-life scenario. The bad news is that the tool-support needs many improvements before it can be used by a normal user. As we were only interested in a proof of concept, rather than an industrial-strength tool, we leave them as future work. Apart from GUIs, the following problems and questions need to be fixed and answered, respectively.

[specification] How to specify connections to the firewall? When designing our policy language we decided that a policy should talk about communication between networks. Unfortunately we forgot one special case: if the firewall configuration does not take place at the machine itself (physically), communication between the administration machine and the firewall needs to be allowed (remote). Thus the formal policy and the test tuple generation need to be adapted.

[TCGTool] Where is the error? The Wp-method, used for test generation, expects completely specified, minimal automata with a fixed start state, where every state is reachable. At the moment these requirements are not checked by the TCGTool. This results in stack overflows if an automaton does not fulfil the requirements. Two improvements are needed here: First, we need error handling which consists of finding the source of the error. Second, we need a method for the automatic minimisation of a given Mealy machine.

[test tuples] Automation We have shown that our method works and does help in finding errors. To reduce the amount of time needed, and errors introduced by humans, the test tuple generation should be automated.

[test tuples] Traceability For the analysis of test logs it is important to know from which policy rule a test tuple was generated. This knowledge should finally end as a comment in the executed test packets file.

[**test tuples**] **Which are the boundaries of the network?** Recall that we choose 9 test cases per graph-region. For this choice, specialised network knowledge need to be taken into account. Choosing .0 as IP-address is of no help, as it represents the network itself. Care has to be taken with .255 and .1: .255 represents broadcast, and .1 often is a router. Finally, private IP addresses can be used multiple times and are not routed in the Internet.

[**abs_to_conc2**] **Support for UDP and ICMP are needed.** At the moment the instantiation of abstract test cases with test tuples only works for TCP. Support for UDP and ICMP is needed. But to distinguish xxx over TCP from xxx over UDP, the keyword definitions have to be adapted.

[**fwtest**] **What if the firewall is a bridge?** Fwtest assumes the firewall interfaces to have an IP. It uses this IP to lookup the MAC addresses of the firewall interfaces, where it sends the packets to. If the firewall acts as a bridge – the routing may be done by a separate router on one side of the firewall – it does not have IPs. Although this is a special case, we need a solution for it.

[**fwtest**] **Support for more than one net on one side.** When running fwtest, we need to tell it which networks it should simulate on either side. Unfortunately, at the moment this can only be one network per side. But, for example, the Internet consists of many different networks. It would be nice if this could be tested in one run, instead of many runs (one per network) as it has to be done now.

[**fwtest**] **It also needs to work with 200 packets at one timeslot.** We tried to run approximately 200 of our test cases at the same time. Unfortunately it seems that 200 test cases per timeslot asks too much of fwtest. We saw output of the form

```
IP-ID (TCP/UDP): expected ID=436, received ID=69
IP-ID (TCP/UDP): expected ID=430, received ID=69
```

and all the test packets were logged as false positives despite their arrival. This issue should definitely be resolved. We need to be able to parallelise our tests.

Chapter 6

Related Work

The definitions of terms used in this chapter can be found in chapter 2.

6.1 Security Policy

[BCG+01] presents very similar ideas to ours. It is not clear if there is more than just ideas. Their approach relies on the help of device vendors. We doubt that device vendors will supply models for new devices and make existing devices confirm to standards.

A Language for Security Constraints on Objects (LaSCO) is defined in [HPL98]. The authors use it to nicely graph Access Control Constraints (Security Policies for Access Control) but claim that it could also be used for security policies. We doubt that LaSCO is suitable for network security policies. With firewalls we, for example, need to be able to express that an event A has to happen before a B can happen. This is not expressible in LaSCO.

There exist guidelines [Gro97, SAN] on how to write a security policy, on security services, security incident handling, et cetera. These discuss informal policies and are intended as a guideline for system administrators and as information for management. These guidelines helped us to identify some demands to a security policy, which we needed during the design of our formal policy language.

6.2 Firewalls

6.2.1 Product Testing

Most people, and all firewall testing methods working at rule-level, rely on the correctness of the firewall implementation. There are several companies conducting intensive tests to ensure this correctness. ICSA Labs[1] describe their tests in [Wal04]. All the bugs they find during their tests, show that it is not safe to rely on the correctness of a firewall.

[1] formerly known as the International Computer Security Association

This justifies our decision to conduct our tests at the network level. Further, the ICSA white paper confirms our findings with respect to different firewalls having different TCP automata: "What has been permitted through firewalls prior to session establishment has varied greatly depending on just how much attention the firewall is paying to the TCP header during connection establishment."

6.2.2 Penetration Testing

Penetration testing [Sch96, WTS03] is the state of the art in firewall testing. While probing a firewall for known attacks definitively should have a place in the whole testing process (see section 4.1), it should not be the only test conducted. The usage of penetration testing as a 'tool' in fully manual conformance testing[2] includes too much interpretation work (and work in general). Due to this, and the ever growing complexity of systems, it should definitely be replaced with a (semi-)automated method based on a clear specification. We see this field partially (probing for known attacks) as complement to our method and partially (manual conformance testing) as predecessor of our work.

6.2.3 Analysis and Simulation

[FKSF01, KFS+03] analyse existing firewall vulnerabilities. The authors first define different stages of packet traversal in firewalls. These stages are then analysed, which results in a classification of the types of occurring errors and vulnerabilities. The authors claim that the knowledge about known vulnerabilities can help to develop an intuition about which errors a firewall operation is most vulnerable to. This approach is orthogonal to ours as we consider firewalls to be black-boxes.

The most mature work in this category is a series of tools by Wool et al. The tools named Firmato [BMNW99, BMNW03], Fang [MWZ00] and FA (earlier called LFA: Lumeta Firewall Analyzer) [Woo01] represent the evolution from a prototype tool to one that is being sold [MWZ05]. The idea behind FA is that the user only has to say where his firewall configuration files lie. FA then automatically translates the rulebases into an intermediate format and automatically retrieves the network layout from the firewall routing tables. Then all possible connections are analysed and the output is given in HTML. For an expert user this might be interesting, as he can really see in detail which services are allowed between which endpoints (this is difficult with most of today's firewalls as much information is hidden in options and behind service names). What the user is also shown is whether he uses services known to be insecure. This approach is a big help for firewall administrators to give them an overview on their firewall rules and possible security implications. Unfortunately, the difficult part of determining if these rules conform to the policy is completely left to the firewall administrator.

[2]Manual conformance testing: Humans using tools to find out what is able to pass a firewall and then manually checking (very time intensive) if that is what should happen.

Another white-box approach at the rule-level is [ASH03]. There the connection between different firewall rules (completely disjoint, exactly matches, inclusively matches, partially matches, are correlated) is analysed. This helps to find errors caused by overlapping rules, and to insert new rules at the correct place. This approach has the same shortcomings as all on this level have: First, it only works for firewalls whose rule languages it knows, and second, implementation errors cannot be found. Also this approach does not take into account whether the rules express what they should. Therefore, this approach does not solve our problem, but could be used complementary to ours.

Two other, simple approaches are [EZ01] and [Haz00]. There the user can issue questions regarding his firewall rules, e.g. "which services does IP X offer (to whom)?", which the tool will then answer. The problems with these two approaches are on one hand that they only know Cisco Access Lists and on the other hand that they are only as good as the person using it (i.e. issuing the questions). Note that this is not true for our approach: We also test implicit policy statements (i.e. what is not explicitly stated).

6.2.4 Formal Testing

There are not many formal approaches to firewall testing yet.

A paper claiming to do formal firewall testing is [JW01]. Unfortunately the really important questions, for example how to formulate test case specifications, are not at all tackled.

Another work claiming to do specification-based firewall testing is [Ma04]. What the author sees as specification-based testing – to answer certain questions with the help of the firewall ruleset (which is the specification in her case) – can in our point of view not be called specification-based testing. Specification-based testing means comparing something to a specification, but here there is no comparison at all. The biggest challenge for her is to answer the test queries. The real challenge in our experience, namely how to find the right queries, is not addressed at all.

6.2.5 Other related work

What does the best firewall solution help if your CIO does not think it is needed? [Gro97] explains in an easy to understand way why protection is needed, why security policies are needed, how a security policy is written and so on.

A direction which is orthogonal to our goal, but also has its strengths, is the generation of firewall rules from the policy. One paper implementing this approach is [Gut97].

Chapter 7

Summary

7.1 Conclusion

In the first part of this thesis we solved the problem of automating the specification-based (conformance) testing of firewalls. As to our knowledge, this is the first such approach. The few other approaches to automate firewall testing, do not test for conformance but rather for vulnerabilities, which is important as well. Or more to the point: Only when using conformance-testing and vulnerability-testing together, the "security" of a firewall can be assured.

Conformance-testing needs at least two ingredients: an implementation to test and a specification to test against. The implementation in our case is the firewall, which we treat as black box. This makes our approach generic. One of the specifications is the security policy. As there was no formal language to specify network security policies in, which is needed to automatically test for conformance to a policy, we designed a language for this purpose.

The other specification that is needed is a midpoint protocol specification, specifying how a certain protocol should be handled by a firewall. We are the first to elaborate on how such a midpoint protocol specification must look like, which we do in the second part of this thesis.

Our contribution for Part I is a product-independent, automated approach for firewall conformance testing. Using prototype tool support, we validated this approach.

7.2 Future Work

The field of firewall conformance testing is a fascinating field, which unfortunately has not been given the necessary attention yet. But in today's world, where everything depends on digital data, firewall conformance testing is essential. We see many interesting problems to be solved, both in research and engineering. We elaborate here on the problems we feel to be most pressing.

Adaptation to the application layer A very important problem is to adapt our method to the application layer. A firewall test method will only be useful if it can handle application level firewalls.

Research-wise this includes adapting the formal policy language to being able to specify dependencies between and payload of protocols. An example for a dependency is if a client is only allowed to use a service after he has authenticated himself. Another example is that Voice over IP (VoIP) consists of several protocols where the connection information about one protocol might be negotiated using another protocol. This example additionally shows why payload is important.

The application layer also presents an engineering problem: How to build a tool that can generate messages for any protocol? I.e. how to let a tool craft a protocol message of an unknown protocol.

Industrial-strength tool Another very important, mostly engineering problem, is to implement an industrial-strength tool for our approach. Besides adding nicer user interfaces and the like to our prototype tool — see subsection 5.3.4 for details — this also includes adapting our formal specification language to the needs of the users, and to enhance the presentation of test results.

The problem with the formal specification language is that the requirements are unclear: Unfortunately the few companies having (informal) security policies do not want to disclose them, because they fear that this will decrease their security level. But companies can probably be won for a *case study* using our prototype tool support. The resulting feedback, on the suitability of our formal policy language, could then be used to improve the formal policy language.

With respect to the test evaluation, feedback from users could also be of help. Is it enough to tell the tester only which test case did not succeed? Or does he need to know from which policy rule this test case originated? Does he even need to know other things? For example about related test cases that did succeed?

Only when we manage to make our approach easily usable, and adapted to the needs of the users, we really solved the problem. That such a tool is absolutely necessary, is shown by the interest in this thesis by military and banks. To achieve this goal, many person-years would be necessary.

Part II

Endpoints versus Midpoints

Chapter 8

Motivation

Networks contain different kinds of principals. Some are communication endpoints, such as clients and servers, while others are midpoints that forward, filter, or, more generally, transform traffic. A midpoint that simply forwards traffic is straightforward to implement. But as soon as stateful filtering comes into play, the midpoint must know the communication protocols used. This is TCP for packet filters and diverse application-level protocols for application-level firewalls. If a midpoint does not know enough about the protocols it filters, there exist ways to bypass a security policy. A prominent example is sending file-sharing traffic over http when using packet filters.

Protocol specifications are normally written for endpoints. Starting from such specifications, it is not clear how a midpoint should enforce the protocol-conform execution by the endpoints, as it can neither observe nor correctly track the protocol states of endpoints (see section 8.1 for more details on this problem). Another problem is that filtering midpoints need to be as secure (i.e. as strict) as possible. However, they should also be user-friendly (and therefore not overly strict). This leads to different interpretations on how a midpoint should handle a given protocol.

The implications of the lack of protocol specifications for midpoints are that manufacturers of devices acting as midpoints have no guidelines on how they should implement a protocol. In practice, midpoint manufacturers implement the same protocol differently, based on their own interpretation of how the midpoint should handle the endpoint data. This implementation is then incrementally adapted based on practical experience. To show how this looks in practice, we present the TCP automata of three different firewalls in section 8.2 and analyse their differences.

As a solution to this problem, we show how to systematically generate midpoint specifications from endpoint specifications. We propose an algorithm that, given the protocol automata for the endpoints, generates a protocol automaton for the midpoint. Roughly speaking, the algorithm tracks all possible endpoint states at each point in time, taking into account messages in transit and possible network behaviour. We prove that the midpoint automata constructed forward only those messages that could have resulted from protocol-conform endpoints.

Despite the fact that midpoint automata are central for the construction of firewalls

Alice				Midpoint				Bob
CLOSED								LISTEN
SYN-SENT	—		SYN	→	—	SYN	→	
		| ←	SYN & ACK	—	←	SYN & ACK	—	SYN-RECEIVED

Figure 8.1: TCP scenario 1

Alice				Midpoint				Bob
CLOSED								LISTEN
SYN-SENT	—		SYN	→	—	SYN	→	
	←		SYN & ACK	—	←	SYN & ACK	—	SYN-RECEIVED
ESTABLISHED	—		ACK	→ |				

Figure 8.2: TCP scenario 2

and other security gateways, the problem we address has not, to our knowledge, been identified before. The closest related work is [BCMG01], which describes how to build a monitor to find out if a system correctly implements an endpoint specification. The authors report on the same problems as ours in determining the state of an endpoint. This problem arises as packets can be reordered or lost between the monitor and the endpoint. They propose several algorithms for a monitor. Unfortunately, their algorithm that takes arbitrary reordering and loss into account is very inefficient (the authors call it brute-force) and their refinements are too constrained to be useful in our setting. In principle, their solution could be used to solve our problem, by using one monitor per endpoint and attaching the output of one monitor to the input of the other. As their monitors are inefficient, this is not a practical solution since firewalls should execute very efficiently, i.e., make each decision with minimal overhead[1]. Note that we also solve a different problem than they do: We do not care if the endpoints correctly implement a protocol. Our goal is to only let messages arising from correct protocol runs pass the firewall, independent of how the endpoints create them.

8.1 The Source of the Problem

In this section, we explain why midpoints are different from endpoints and why they thus need a different protocol specification. The problem arises with the filtering midpoints. These base their decisions — basically drop or forward — on the protocol states the endpoints are in. Unfortunately the two endpoints of a connection can be in different states and not all of these states are observable by the midpoint.

Consider the following example: the TCP connection initiation (three-way-handshake) shown in Figure 2.2. Imagine the second packet gets lost after being forwarded by the midpoint (Figure 8.1). Alice is now in the state `SYN-SENT`, whereas Bob is in state `SYN-RECEIVED`. To the midpoint, this situation looks the same as the situation where the

[1]This is not a problem with our generated automata. The generation takes time but their execution is very fast.

second packet reaches Alice, but the third packet gets lost before being received by the midpoint (Figure 8.2).

Given that the midpoint cannot differentiate between the scenario in Figure 8.1 and the scenario in Figure 8.2, in what state should the midpoint be? And how should it react upon receiving a SYN packet from Alice? In scenario 1, the SYN is a retransmission and should be forwarded. Whereas in scenario 2, a SYN does not conform with a correct protocol execution (Alice should not send SYN packets in state `ESTABLISHED`) and should therefore be dropped.

The endpoints see the situation differently. Alice can clearly distinguish between scenario 1, where she would repeat her SYN, and scenario 2, where she would repeat her ACK. Bob cannot distinguish between the scenarios (or at least not until he has seen Alice's reaction), but he does not need to: he would repeat his SYN&ACK in any case.

Another scenario is possible: Alice may have crashed and the SYN at hand could represent a new connection initiation (with the same source port as before). This scenario can also happen later on in a connection. What should the midpoint then do? Should it forward the SYN and risk damage to Bob? Should it just block the packet and hinder Alice from communicating with Bob? Should it send a RST in Bob's name? Should it also send a RST in Alice's name to Bob? All these questions must be answered when giving a midpoint specification of TCP.

In this example, we see that one reason why midpoints cannot always track the protocol states of the endpoints lies in packet loss. But packet loss is only part of the problem. Another reason lies in the fact that certain endpoint constructs lead to ambiguity from the midpoint's perspective. These are:

1. Multiple transitions with the same output:
   Consider two transitions $q_i \xrightarrow{-/a} q_j$ and $q_i \xrightarrow{b/a} q_k$. Assume that the midpoint has previously forwarded b to an endpoint with these two transitions and afterwards receives an a. It cannot know from which transition this a originated.

2. Transitions without output:
   In this case, a midpoint cannot distinguish between the start state of the transition and the end state of the transition (at least until it has seen other, unambiguous output from the endpoint). A special case is hidden states, which are states where all incoming and outgoing transitions have no output. Such states can never be identified by a midpoint.

3. A packet can be sent in different states:
   This makes an unambiguous mapping between packets and states impossible. While this is not a source of tracking problems, it does make recovering from them difficult.

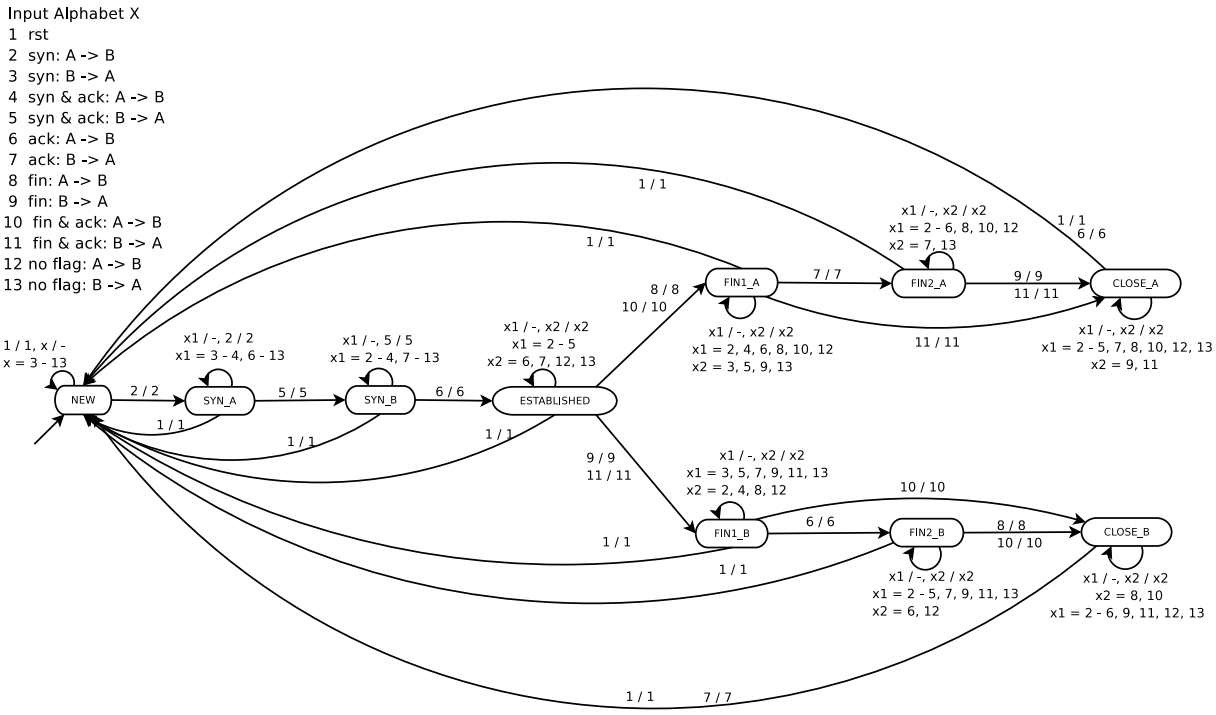


Figure 8.3: TCP specification for a midpoint

8.2 Case Study Differences in midpoints based on TCP

In the last section, we explained why it is nontrivial to build a midpoint from endpoint specifications. In this section, we now show the implications of a lack of midpoint specifications by documenting the current state of affairs. For this, we took three commonly used firewalls — Checkpoint [Ltd], netfilter (iptables) [ea], and ISA Server [Mic] — and reverse-engineered them, testing them against our (as there is no other) TCP midpoint specification given in Figure 8.3 and then analysing the results by hand.

Additionally we also tested the transition from state NEW to state SYN_A with all different flag combinations. Doing this we found that the following flag-combinations cause the transition from state NEW to state SYN_A:

netfilter	SYN; SYN & PSH; SYN & FIN; SYN & FIN & PSH
Checkpoint	SYN; SYN & PSH
ISA Server	SYN

As a result, we derived three distinct TCP automata (see Figures 8.4, 8.5 and 8.6). It may appear that there are few differences, but for security-critical devices, every difference represents a possible fault and hence is one too many. Also we have not taken into account sequence numbers and fragmentation, where we might have found other differences. Below

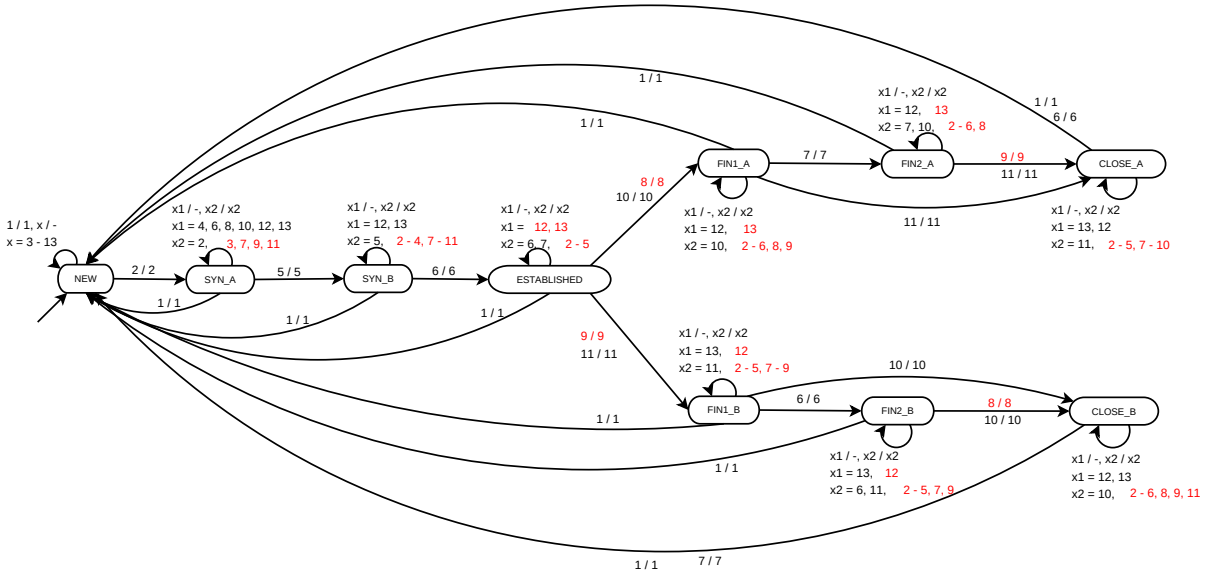


Figure 8.4: TCP automaton in iptables (ip_conntrack 2.1)

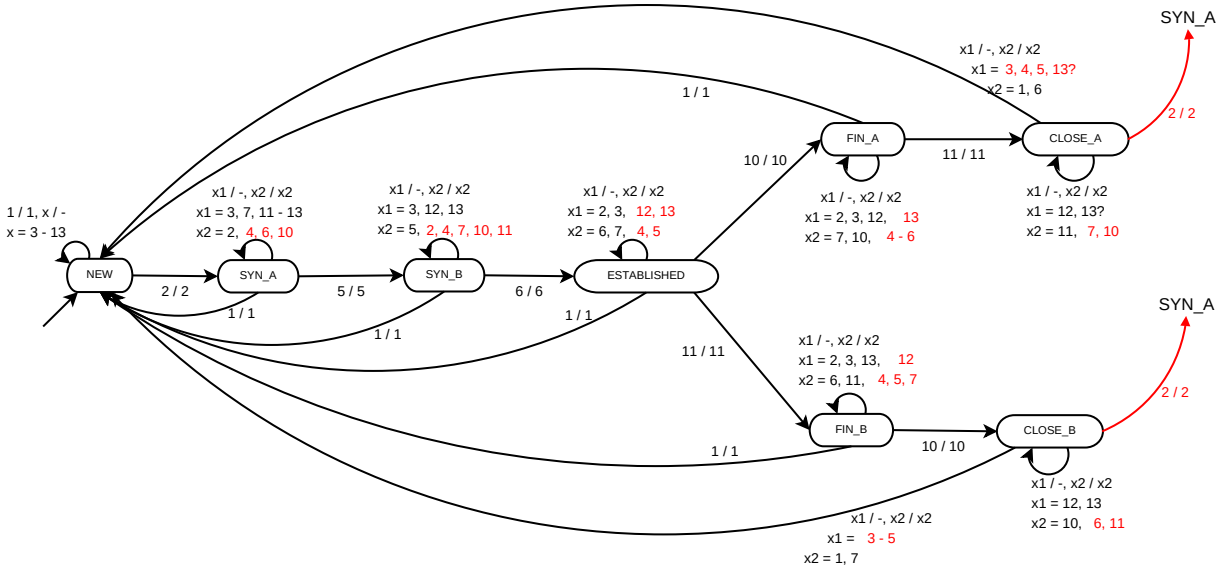


Figure 8.5: TCP automaton in Checkpoint R55W

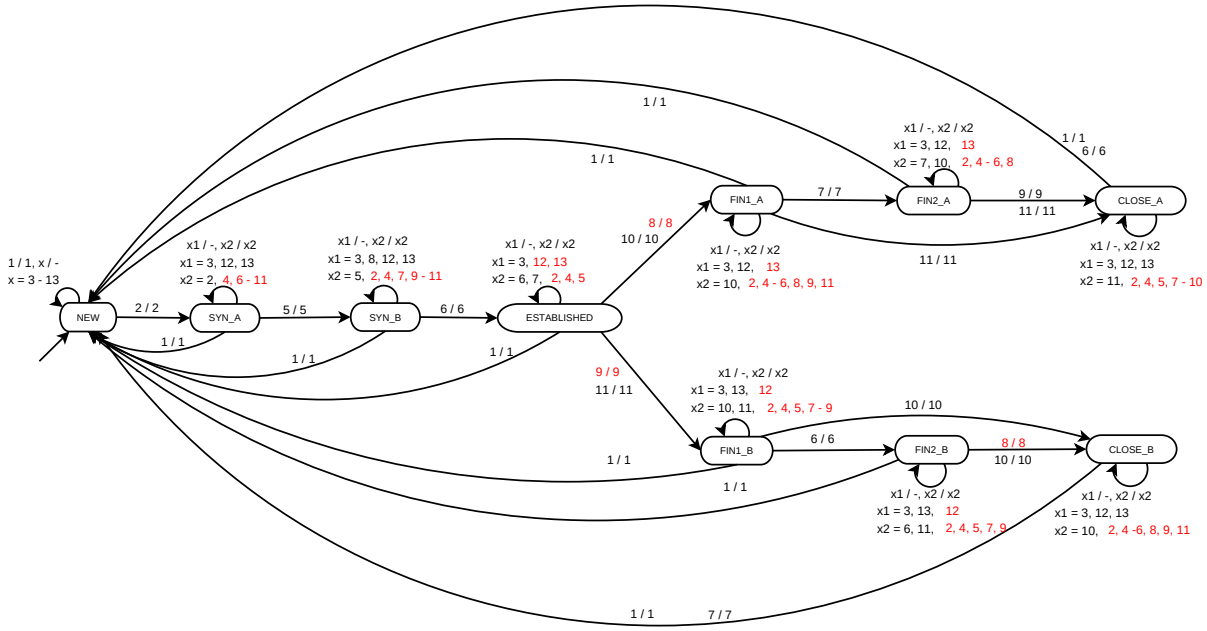


Figure 8.6: TCP automaton in ISA Server v4.0.2161.50

we describe three of the differences:

A 'clean' three-way-handshake is not enforced
To initiate a TCP connection, a so called three-way-handshake is used (see Figure 2.2). So let us assume the firewall has accepted a SYN from an endpoint E_0 (Alice) to another endpoint E_1 (Bob). If there is now a SYN&ACK from E_1 to E_0, then everything works as expected: the packet should be let through and the firewall should enter the next state. If there is another SYN from E_0 to E_1, then this will be a retransmission (it could be that the first SYN was lost between the firewall and E_1) and should be allowed as well. If there is a RST from E_1 to E_0, then E_1 does not want this connection, the packet should be let through, and the TCP automaton be initialised. All other packets make no sense at this time and therefore should be blocked. Unfortunately, in all of the tested firewalls, additional packets were let through. As one example, a FIN from E_1 to E_0 is allowed during connection initiation in netfilter. But there is no connection to be closed: If E_1 is not accepting the connection then it would send a RST.

After a FIN, data from both sides is still accepted
If E_0 sends a FIN to E_1, then this means that E_0 wants to close the connection. After this FIN, E_1 is still allowed to send data, but E_0 is not (it makes not sense to send data after requesting to close the connection), except for the ACK belonging to E_1's FIN. As packets may not arrive in their correct ordering at the firewall, the firewall cannot just drop all packets from E_0 after having seen a FIN from B. But the firewall should just let

through older packets (based on the sequence number), a retransmission of the FIN, and the ACK to E_1's FIN&ACK (after having received E_1's FIN&ACK). To accomplish this task, the firewall has to keep track of the sequence numbers. This appears not to be done, and therefore too many packets are let through.

SYNs are accepted during already established connections
SYNs are only used for connection initiation. That means that if a connection is fully established, there will be no more legitimate SYNs (based on the sequence numbers) belonging to that connection. But netfilter and ISA Server accept SYNs (from the initiator of the connection) all the time. Checkpoint does block the SYNs, but always allows SYN & ACK, which is not much better.

These findings show that there is a lack of consensus, at best, and a general lack of understanding, at worst, about how TCP should be handled by a firewall. The result is that every vendor does something different. We would like to contribute a solution to this problem by showing, systematically, how to construct midpoint specifications from endpoint specifications.

Chapter 9

Construction of a Midpoint Automaton from Endpoint Automata

In chapter 8, we saw why there is a need for midpoint specifications. Basically there are two ways to construct such a specification: write it directly or generate it from the endpoint specification. The first alternative has two major drawbacks: 1) the consistency with the endpoint specification must somehow be assured and 2) it requires additional work for protocol designers. The second alternative overcomes both problems.

9.1 Setting

We subsequently consider only two-party protocols, i.e. protocols for only two endpoints. This covers most network protocols[1]. Let E_0 and E_1 be the endpoints and M the midpoint through which communication passes. Communication takes place in the form of messages, where the endpoint specification of the communication protocol determines when an endpoint may send which kind of message. For every message arriving at the midpoint, the midpoint can either forward the message (Figure 9.1) or drop it (Figure 9.2).

We write $X \rightarrow Y : m$ to express that the message m is sent from the endpoint X to the other endpoint Y, where $X \in \{E_0, E_1\}$, $Y \in \{E_0, E_1\}$, and $Y \neq X$. As there is a midpoint M between E_0 and E_1, every $X \rightarrow Y : m$ can be divided into the two parts $X \rightarrow M : m$ and $M \rightarrow Y : m'$. This makes explicit on which side of the midpoint a message is and also simplifies the specification of the actions of the midpoint: $m' = m$ if the midpoint forwards the message unaltered, $m = -$ (where $-$ signifies no external output) if the midpoint drops the message, and $m' \neq m$ if the midpoint alters the message before forwarding it. The messages may be altered, for example, when using Network Address Translation (NAT) in the firewall. For the sake of simplicity we will not consider this case.

[1] It is straightforward to expand our approach to protocols with more endpoints.

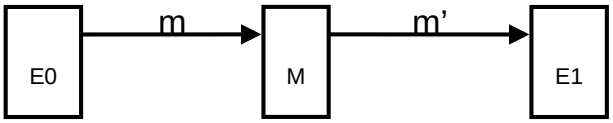


Figure 9.1: A message m from endpoint $E0$ to endpoint $E1$, forwarded by the midpoint M.

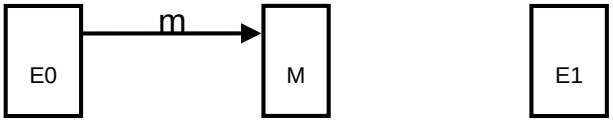


Figure 9.2: A message m from endpoint $E0$ to endpoint $E1$, dropped by the midpoint M.

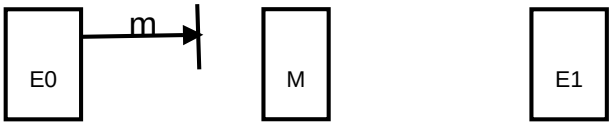


Figure 9.3: A message m from endpoint $E0$ to endpoint $E1$, lost by the network.

We will construct our midpoints to be *permissive* rather than *restrictive*. This means that our midpoint forwards messages if they could have resulted from protocol-conform endpoints. Thus our midpoint can possibly accept an incorrect message, but only in the cases where there is a correct scenario where this message could occur.

For the transport of the messages between the endpoints (via the midpoint) we assume a network that either (1) delivers messages, although not necessarily preserving the order, or (2) looses them (Figure 9.3).

9.2 Idea

Before giving the construction of a midpoint automaton from endpoint automata in section 9.3, we first sketch the ideas behind our construction.

We model the global state of a system (the endpoints, midpoint, and network) at some time t as a state $st^t = (q_0^t, q_M^t, q_1^t, net^t)$, where q_i^t is the state of endpoint E_i at time t, q_M^t is the state of the midpoint M at time t, and net^t consists of all messages travelling between the endpoints at time t.

Base midpoint-actions on environment-state The midpoint M has to base its actions on the state of its environment. If M could observe all actions in the system, $q_M^t = (q_0^t, q_1^t, net^t)$ would hold at any time t, meaning that M always knows the exact states of the endpoints and the contents of the network. But midpoints generally cannot always determine the correct values of these components and hence we will let the state of our midpoint be a set of such triples, where each of these triples represents a possibly correct view of the system. Thus the triples of one state q_M^t are equivalent in the sense

that they are not distinguishable by the midpoint with its current knowledge, i.e. based on the traffic it has previously observed.

Example 1 To provide further intuition about the functioning of a midpoint, let us consider an example. Suppose the system starts in the global state $st^1 = (q_1, \{(q_1, q_1, net^1)\}, q_1, net^1)$. This means that E_0 and E_1 both are in their start states, the network content is net^1 and the midpoint knows about all this. Note that q_1 of E_0 and q_1 of E_1 are not the same as they do not belong to the same automaton. Assume the following steps are taken:

1. M sends (forwards) a message x to E_0.
   To track the fact that the network now contains x, M must change its state to $q_M^2 = \{(q_1, q_1, net^2)\}$, where $net^2 = net^1 \cup \{M \to E_0 : x\}$. The global state is then $st^2 = (q_1, \{(q_1, q_1, net^2)\}, q_1, net^2)$.

2. x is received by E_0 and used as input to its automaton. Suppose, for E_0 in state q_1, there are only two transitions: $q_1 \xrightarrow{x/y} q_2$ and $q_1 \xrightarrow{-/z} q_3$. As M does not know if and when E_0 makes a transition, it cannot directly act on this step and thus its state is wrong. The global state is $st^3 = (q_2, \{(q_1, q_1, net^2)\}, q_1, net^3)$, where $net^3 = (net^2 \setminus \{M \to E_0 : x\}) \cup \{E_0 \to M : y\}$.

3. A message y reaches M.
   To take a correct decision, M must compute what could have happened (all possible successor steps) since its last step. This is either:

   - nothing, i.e., (q_1, q_1, net^2),
   - E_0 consumes x, makes a transition to q_2, and outputs y: (q_2, q_1, net^3), or
   - E_0 makes a transition to q_3, and outputs z: $(q_3, q_1, net^2 \cup \{E_0 \to M : z\})$.

   Now, M can determine its reaction on y. If there is one or more triple having y in its net (a possibly correct scenario where y occurred), y is forwarded and M's next state will consist of all these matching triples with their nets updated: $q_M^4 = \{(q_2, q_1, net^4)\}$, where $net^4 = (net^3 \setminus \{E_0 \to M : y\}) \cup \{M \to E_1 : y\}$. The global state is $st^4 = (q_2, \{(q_2, q_1, net^4)\}, q_1, net^4)$.

Example 2 In the example above, there was only one endpoint transition between two consecutive midpoint transitions. In such a case, tracking (computing all possible successor states) is not difficult. The situation is more complex if more than one endpoint transition can happen between two consecutive midpoint transitions. Figure 9.4 provides an example. Since communication need not be order preserving, after two consecutive endpoint transitions, the second message ($E_0 \to M : z2$) may reach the midpoint first. Thus it would not be enough if the midpoint computed only the next possible state (reachable in one step), but all possible successor states are needed, as only this

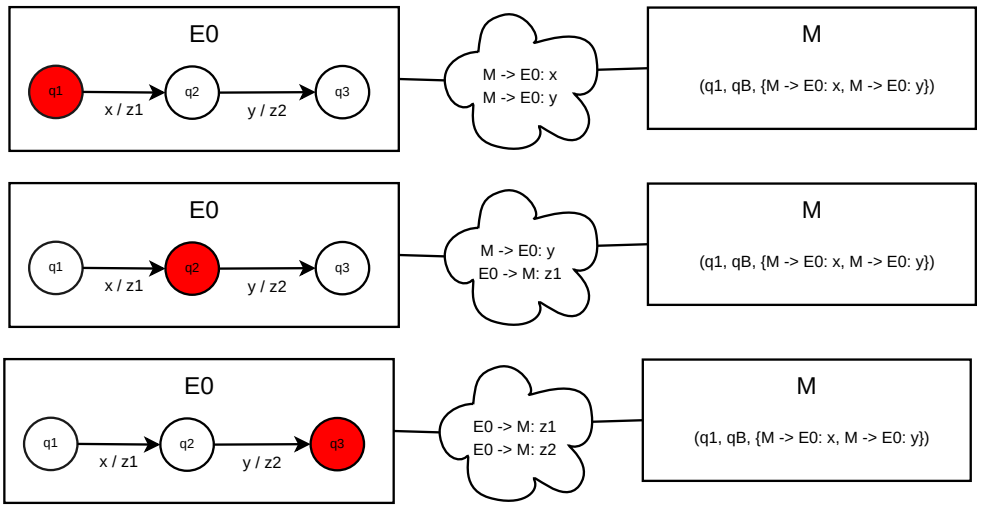


Figure 9.4: Two consecutive endpoint transitions

would lead to $\{(q_1, q_B, \{M \to E_0 : x, M \to E_0 : y\}), (q_2, q_B, \{M \to E_0 : y, E_0 \to M : z1\}), (q_3, q_B, \{E_0 \to M : z1, E_0 \to M : z2\})\}$ and thus lead to the correct action — forwarding $z2$ — and the correct next state $\{(q_3, q_B, \{E_0 \to M : z1, M \to E_1 : z2\})\}$.

All possibly correct messages must be forwarded Let us return to our first example to illustrate why all possibly correct messages must be forwarded. Assume that we have the following sequence of actions:

1. M forwards a message x to E_0.
2. x is lost by the network.
3. E_0 takes the transition to state q_3.
4. An intruder sends message y (which is incorrect at q_3).
5. y reaches M.

For the midpoint, this scenario looks exactly the same as the one before. But here y is an incorrect message. As we never want to block correct messages (our decision for a permissive rather than restrictive midpoint), we have to accept y here (it could be the correct one from above).

9.3 Construction

We will now give the technical details of our construction of a midpoint automaton from endpoint automata. A two-party protocol p can be specified by two Mealy machines (one

for each endpoint):

$$A_0 = (Q_0, \Sigma_0, \Gamma_0, \delta_0, \lambda_0, q_{0,1})$$
$$A_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, \lambda_1, q_{1,1})$$

Note that $\Sigma_0 = \Gamma_1$ and $\Gamma_0 = \Sigma_1$ since these automata must be able to communicate with each other. Often even A_0 and A_1 are the same.

The network can be modelled as a multiset (also called bag), which stores all messages in transit between the midpoint and the endpoints. Specifically

$$net \subseteq \mathcal{M}(S) = \{x' \mid x \subseteq S, x' =_s x\},$$

where S is the set of messages allowed by the protocol and $=_s$ denotes set equality (the sets contain the same elements, ignoring repetition).

In our construction, *net* will be part of a state of a deterministic automaton. Since we cannot handle a network of infinite size we must forbid actions of the endpoints and the network that can put infinitely many packets into the network. Hence, for the endpoints, we forbid loops without input in their protocol automata. For the network, we do not model message duplication. These restrictions are not problematic as the former should not be present and the latter can easily be detected and handled on another layer. Thus, it suffices to consider

$$net \subseteq \mathcal{P}(S), \qquad \text{where } S = \{X \to Y : m | X, Y \in \{E_0, E_1, M\}, Y \neq X, m \in (\Sigma_0 \cup \Gamma_0)\}.$$

Before starting with the construction of the midpoint automaton, we need to define the actions that are possible in our system. These are either transitions by the endpoints or the midpoint, based on their automata, network loss, or a message inserted by an intruder.

Definition 1 *$st^{t+1} = (q_0^{t+1}, q_M^{t+1}, q_1^{t+1}, net^{t+1})$ is a* successor state *of st^t =(q_0^t, q_M^t, q_1^t, net^t), denoted $st^t \vdash st^{t+1}$, if one of the following conditions holds.*

Midpoint transition: For any $msg \in net$ with $msg = (E_i \to M : m)$,

$$\begin{aligned} q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= \delta_M(q_M^t, msg), \\ net^{t+1} &= (net^t \setminus \{msg\}) \cup \{\lambda_M(q_M^t, msg)\} \end{aligned} \tag{9.1a}$$

Correct endpoint transition: For any $msg \in net^t$ with $msg = (M \to E_i : m)$ (the endpoint taking an input from the network) or for $m = -$ (no input)

$$\begin{aligned} msg' &= \begin{cases} (E_i \to M : \lambda_i(q_i^t, m)) & if \lambda_i(q_i^t, m)) \neq -, \\ - & otherwise, \end{cases} \\ net^{t+1} &= (net^t \setminus \{msg\}) \cup \{msg'\}, \\ q_i^{t+1} &= \delta_i(q_i^t, m), \\ q_{1-i}^{t+1} &= q_{1-i}^t, \\ q_M^{t+1} &= q_M^t \end{aligned} \tag{9.1b}$$

Incorrect transition:

$$\begin{aligned} msg' &\in (\Gamma_i \setminus \{\lambda_i(q_i^t, x) | (M \to E_i : x) \in net^t \text{ or } x = -\}, \\ net^{t+1} &= net^t \cup \{msg'\}, \\ q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= q_M^t \end{aligned} \tag{9.1c}$$

Network loss: for any $msg \in net^t$,

$$\begin{aligned} q_0^{t+1} &= q_0^t, \\ q_1^{t+1} &= q_1^t, \\ q_M^{t+1} &= q_M^t, \\ net^{t+1} &= net^t \setminus \{msg\} \end{aligned} \tag{9.1d}$$

Note that as the network is modelled by a multiset, the permutation of messages is handled implicitly. Furthermore, note that an endpoint transition must not produce output. We denote empty output as '−'.

Definition 2 *Transitions* (9.1a), (9.1b) *and* (9.1d) *represent* correct transitions. *We denote a message resulting from a correct transition as a* correct message *and all other messages as* incorrect messages.

Definition 3 *A* correct trace *is a trace $st^1 \vdash st^2 \vdash \cdots \vdash st^n$, where every transition $st^i \vdash st^{i+1}$, with $1 \leq i < n$, is a correct transition.*

Definition 4 *The* message history *of a trace $tr = st^1 \vdash st^2 \vdash \cdots \vdash st^n$ is a sequence of messages $m_1, m_2, \cdots, m_t$, where t is the number of non-midpoint transitions in tr that produce output, and m_i is the output from the ith of these transitions.*

Definition 5 *The* midpoint message history *of a trace $tr = st^1 \vdash st^2 \vdash \cdots \vdash st^n$ is a sequence of messages $m_1, m_2, \cdots, m_s$, where s is the number of midpoint transitions in tr, and m_i is the input of M at its ith transition in tr.*

Definition 6 *Two* traces *are* midpoint equivalent *if they have the same midpoint message history.*

Definition 7 *Two* triples (a, c, d) *and* (e, g, h) *are* midpoint equivalent *if there exist two midpoint equivalent traces* tr_1 *and* tr_2 *with* $tr_1 = s^1 \vdash s^2 \vdash \cdots \vdash (a, b, c, d)$ *and* $tr_2 = st^1 \vdash st^2 \vdash \cdots \vdash (e, f, g, h)$.

To have a correctly functioning midpoint, two properties about a midpoint state q_M must be satisfied: 1) one of the triples in q_M is the correct one (Definitions 8 and 9); and 2) only possibly correct messages are forwarded (Definition 10).

Definition 8 q_M^t *is a* correct tracking at time t *if* q_M^t *is neither too small nor too large.* Not too small *means that* $(q_0^t, q_1^t, net^t) \in q_M^t$. Not too large *means that all* $q \in q_M^t$ *are midpoint equivalent to* (q_0^t, q_1^t, net^t).

Definition 9 M tracks endpoints correctly *if for every midpoint transition* $(q_0^{n-1}, q_M^{n-1}, q_1^{n-1}, net^{n-1}) \vdash (q_0^n, q_M^n, q_1^n, net^n)$ *in a trace,* q_M^n *is a correct tracking at time* n.

Definition 10 M computes outputs correctly *if for every trace* $tr = st^1 \vdash \ldots \vdash st^n$ *and every* t, $1 \leq t \leq n$ *we have:*

$$\lambda_M(q_M^t, E_i \to M : m) = \begin{cases} M \to E_j : m & \text{if } E_i \to M : m \text{ occurs in the message history} \\ & \text{of any trace } tr' \text{ which is midpoint equivalent} \\ & \text{to } tr, \\ - & \text{otherwise.} \end{cases}$$

where $j = 1 - i$.
As M *cannot distinguish between* tr *and* tr'*, it must forward all messages that occur in any of these traces, in order to avoid ever dropping a correct message.*

Based on A_0 and A_1, we will now construct a Mealy machine A_M for the handling of the protocol p by the midpoint:

$$\begin{aligned} A_M &= (Q_M, \Sigma_M, \Gamma_M, \delta_M, \lambda_M, s_M) \\ Q_M &= \mathcal{P}(Q_0 \times Q_1 \times net) \\ \Sigma_M &= \{E_0 \to M : a \mid a \in (\Gamma_0 \setminus \{-\}\} \cup \{E_1 \to M : a \mid a \in (\Gamma_1 \setminus \{-\}\} \\ \Gamma_M &= \{M \to E_0 : a \mid a \in (\Sigma_0 \setminus \{-\}\} \cup \{M \to E_1 : a \mid a \in (\Sigma_1 \setminus \{-\}\} \cup \{-\} \\ q_{M,1} &= \{(q_{0,1}, q_{1,1}, \{\})\} \end{aligned}$$

Before we define the functions δ_M and λ_M, we first analyse the different possible scenarios. We do this with the help of Figure 9.5. There we examine the relationship between the actions of an endpoint E_0 (the situation for E_1 is analogous), the network, and the midpoint M. In particular, we consider how the four different types of transitions an

Endpoint E_0	Network	Midpoint	correct midpoint transition	Eq.
-			$\delta_M(q_M, -) = \{(q_2, q_{E_1}, net_M \setminus \{M \to E_0 : x\})\}$ $\lambda_M(q_M, -) = -$	(9.2b)
			$\delta_M(q_M, -) = \{(q_2, q_{E1}, net_M \setminus \{M \to E0 : x\})\}$ $\lambda_M(q_M, -) = -$	(9.2c)
			$\delta_M(q_M, -) = \{(q_3, q_{E_1}, net_M \setminus \{M \to E_0 : x\})\}$ $\lambda_M(q_M, -) = -$	(9.2c) (9.2b)
			$\delta_M(q_M, E_0 \to E_1 : y)$ $= \{(q_3, q_{E1}, (net_M \setminus \{M \to E_0 : x\}) \cup \{M \to E_1 : y\})\}$ $\lambda_M(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(9.2c)
			$\delta_M(q_M, E_0 \to E_1 : y)$ $= \{(q_4, q_{E1}, (net_M \setminus \{M \to E_0 : x\}) \cup \{M \to E_1 : y\})\}$ $\lambda(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(9.2b) (9.2d)
			$\delta_M(q_M, E_0 \to E_1 : y)$ $= \{(q_4, q_{E1}, net_M \cup \{M \to E_1 : y\})\}$ $\lambda(q_M, E_0 \to E_1 : y) = E_0 \to E_1 : y$	(9.2d)
			$\delta_M(q_M, -) = \{(q_4, q_{E1}, net_M)\}$ $\lambda_M(q_M, -) = -$	(9.2d) (9.2b)
			$\delta_M(q_M, -) = \{(q_5, q_{E1}, net_M)\}$ $\lambda_M(q_M, -) = -$	(9.2d)

Figure 9.5: A transition in an endpoint, from a midpoint's view

endpoint can take ($x/-$, x/y, $-/y$, and $-/-$, for $x \in \Sigma_0$, $y \in \Gamma_0$) look from the endpoints', the network's, and the midpoint's respective point of view. These are shown in columns 1 – 3, where one row represents one case. Note that one view (row) of one principal can belong to several views of another principal.

In the fourth column, the correct midpoint transition is shown. That is the transition the midpoint must take if it wants to track correctly the endpoint's state and the messages in the network[2]. For this we assume $q_M = \{(q_1, q_{E_1}, net_M)\}$ to be the state of the midpoint after its last transition (where applicable, this is forwarding x). Note that net_M contains all the messages in the network. Therefore a message has to be removed from net_M if it is no longer in the network, either because it was consumed by an endpoint or midpoint, or lost by the network.

As an example, let us explain the contents of the third row (in the first column, the third and the forth row coincide). Here a message x is forwarded by M to E_0, i.e. $M \to E_0 : x$ (3rd column). This message then reaches E_0 (2nd column), which uses it as input to its x/y-transition (1st+column). After this transition, E_0 is in state q_3 (1st column) and a message y ($E_0 \to M : y$) has been inserted to the network (2nd column). This message is then lost by the network (2nd column). To correctly represent these actions, M has to change its state as given: E_0 is now in state q_3 and the *net* no longer contains x (4th column). Note that net_M does not contain y as its insertion is compensated by its removal.

With the help of Figure 9.5, we will now define the successor function. This function computes all direct successor states of a triple of the midpoint state (the transition function will then later choose some of these triples, based on its input). The figure illustrates why we sometimes have more than one possible successor state: the midpoint (3rd column) cannot distinguish all the scenarios (rows). Note that Figure 9.5 only considers one endpoint, whereas *succ* considers both endpoints (the equations (9.2c) and (9.2d) for E_0 correspond to the equations (9.2e) and (9.2f) for E_1).

$$
\begin{aligned}
succ(q_M) = \bigcup_{q \in q_M} \; \bigcup_{(M \to E_0 : m_1) \in net_M} \; \bigcup_{(M \to E_1 : m_4) \in net_M} \; \bigcup_{msg \in net_M} & \\
& \{(q_0, q_1, net_M), && \text{(9.2a)} \\
& \quad (q_0, q_1, net_M \setminus \{msg\}), && \text{(9.2b)} \\
& \quad (\delta_0(q_0, m_1), q_1, (net_M \setminus \{M \to E_0 : m_1\}) \cup m_2) && \text{(9.2c)} \\
& \quad (\delta_0(q_0, -), q_1, net_M \cup m_3) && \text{(9.2d)} \\
& \quad (q_0, \delta_1(q_1, m_4), (net_M \setminus \{M \to E_1 : m_4\}) \cup m_5), && \text{(9.2e)} \\
& \quad (q_0, \delta_1(q_1, -), net_M \cup m_6)\} && \text{(9.2f)}
\end{aligned}
$$

[2]We will later give definitions of δ_M and λ_M that incorporate all these scenarios. The fifth column gives the number of the corresponding equations.

where

$$m_2 = \begin{cases} \{E_0 \to M\} : \lambda_0(q_0, m_1) & \lambda_0(q_0, m_1) \neq -, \\ \emptyset & \text{otherwise,} \end{cases} \tag{9.2g}$$

$$m_3 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, -)\} & \lambda_0(q_0, -) \neq -, \\ \emptyset & \text{otherwise,} \end{cases} \tag{9.2h}$$

$$m_5 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, m_4)\} & \lambda_1(q_1, m_4) \neq -, \\ \emptyset & \text{otherwise,} \end{cases} \tag{9.2i}$$

$$m_6 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, -)\} & \lambda_1(q_1, -) \neq -, \\ \emptyset & \text{otherwise.} \end{cases} \tag{9.2j}$$

The function *succ* computes all the states that are reachable in one step by an endpoint or the network. Since we are interested in all possible successor states, we must compute the closure of *succ*, defined as

$$cl(succ(x)) = \bigcup_{i=0}^{\infty} succ^i(x). \tag{9.3}$$

Observe that the closure is monotonic. It also has an upper bound, namely

$$cl(succ(q_M^t)) \subseteq \mathcal{P}(\{(q_0, q_1, n) | q_0 \in Q_0, q_1 \in Q_1, n \in net\}).$$

Hence, as Q_0, Q_1, and net are finite, $cl(succ(q_M^t))$ is also finite.

We now can define δ_M. The idea is to let our midpoint "track" all possible actions. We do this by first calculating the closure of all possible next states before actually executing a transition based on them.

$$\delta_M(q_M, m) = \bigcup_{(q_0, q_1, net_M) \in cl(succ(q_M^t))} \{(q_0, q_1, (net_M \setminus \{m\}) \cup \lambda_M(q_M, m))\} | m \in net_M\} \tag{9.4}$$

Note that $cl(succ(q_M^t))$ represents all possible successor states of q_M^t, whereas $\delta_M(q_M^t, m)$ only contains those successor states of q_M^t which can be "reached" with a message m. λ_M is now straightforward: If there is any triple where the input occurs, i.e. the message is correct in some midpoint-equivalent trace, the input is forwarded.

$$out((q_0, q_1, net_M), E_i \to M : y) = \begin{cases} \{M \to E_j : y\} & \text{if } \{E_i \to M : y\} \in net_M, \\ \emptyset & \text{otherwise.} \end{cases} \tag{9.5}$$

where $j = 1 - i$, and

$$\lambda_M(q_M, m) = \begin{cases} out(q, m) & \text{if there exists a } q \in cl(succ(q_M)) \text{ with } (out(q, m) \neq -), \\ - & \text{otherwise.} \end{cases} \tag{9.6}$$

Note that for each m, there is at most one non-empty (not '$-$') value for $out(q, m)$. Hence, λ_M is well-defined. This is due to the fact that our midpoint either drops or forwards a message. This would have to be revised for a midpoint that alters messages (e.g. a firewall performing Network Address Translation).

9.4 Correctness

As stated in section 9.2, a correctly functioning midpoint must satisfy two properties: 1) one of its state triples is correct; and 2) only possibly correct messages are forwarded. In this section, we prove that a midpoint, constructed as described in section 9.3, satisfies the above properties.

Property I: one state triple is correct.

We prove the first property with the help of the following lemmas. During the proofs, we will denote the correct triple (q_0^t, q_1^t, net^t) at time t as q_{corr}^t.

Lemma 1 *Given an M produced by our midpoint construction and a trace $st^1 \vdash st^2 \vdash \cdots \vdash st^n$, if there is a correct tracking at time $t1$, then the tracking after the next midpoint transition $st^{t2} \vdash st^{t2+1}$, $t2 \geq t1$, is also correct.*

Lemma 2 *Given an M produced by our midpoint construction and a trace $tr = st^1 \vdash st^2 \vdash \cdots \vdash st^n$, for any midpoint transition $st^{t2} \vdash st^{t2+1}$, there is a correct tracking at each time $t1$ with $1 \leq t1 \leq t2 < n$.*

Lemma 3 *Given an M produced by our midpoint construction, M tracks endpoints correctly (as defined in Definition 9).*

Lemma 3 follows from the other two. If for every midpoint transition there exists an earlier correct tracking (Lemma 2), then the tracking after the midpoint transition is correct (Lemma 1). Hence the tracking after every midpoint transition is correct.

Proof of Lemma 1

It suffices to show that $cl(succ(q_M^{t1}))$ is a correct tracking at time $t2$. The midpoint transition which then takes place — $net^{t2+1} = (net^{t2} \setminus \{msg\}) \cup \lambda_M(q_M^{t2}, msg)$ (Definition (9.1a)) is reflected by δ_M (Equation (9.4)).

To show that $cl(succ(q_M^{t1}))$ is a correct tracking at time $t2$, it suffices to show that $succ^{(t2-t1)+1}(q_M^{t1})$ is a correct tracking at time $t2$. We establish the following proposition by induction on $t = t2 - t1$.

$$succ^t(q_M^{t1}) \text{ is a correct tracking at time } t1 + t \ (= t2). \tag{9.7}$$

Basis $t = 1$
q_M^{t1} is a correct tracking at time t1 (as given). From Lemma 4, $succ(q_M^{t1})$ is a correct tracking at time $t2$.

Step We assume now that the induction hypothesis (9.7) holds for $t = n$, and we prove it for $t = n + 1$.
By the induction hypothesis, $succ^n(q_M^{t1})$ is a correct tracking at time $t1 + n$. By Lemma 4, established below, $succ(succ^n(q_M^{t1})) = succ^{n+1}(q_M^{t1})$ is a correct tracking at time $(t1 + n) + 1 = t1 + (n + 1)$. QED.

Proof of Lemma 4

Lemma 4 *$succ(q_M^t)$ is a correct tracking at time $t+1$, if q_M^t is a correct tracking at time t.*

To prove this lemma, we have to show the following:

1. For a correct transition $st^t \vdash st^{t+1}$, $q_{corr}^{t+1} \in \bigcup_{q \in q_M^t} succ(q)$.
2. A message from an incorrect transition is not added to net_M^t by *succ*, except in the case where it could also occur in a correct trace.

We now consider each case.

1. *succ* for correct transitions. We consider the following cases:
   (a) correct endpoint transition
      i. $msg = (M \to E_i : m) \in net^t$
      By Definition (9.1b) we have:

$$\begin{aligned} q_i^{t+1} &= \delta_i(q_i^t, m) \\ q_{1-i}^{t+1} &= q_{1-i}^t \\ q_M^{t+1} &= q_M^t \\ net^{t+1} &= (net^t \setminus \{M \to E_i : m\}) \cup msg' \\ msg' &= \begin{cases} \{E_i \to M : \lambda_M(q_i^t, m)\} & \lambda_M(q_i^t, m)) \neq -, \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned}$$

      By Equation (9.2c) it holds that $\forall (M \to E_0 : m_1) \in net_M^t, (\delta_0(q_0^t, m_1), q_1^t, (net_M^t \setminus \{M \to E_0 : m_1\}) \cup m_2) \in succ(q_{corr}^t)$,
      where $m_2 = \begin{cases} \{E_0 \to M : \lambda_0(q_0^t, m_1)\} & \lambda_0(q_0^t, m_1) \neq -, \\ \emptyset & \text{otherwise.} \end{cases}$

and
$\forall(M \to E_1 : m_4) \in net^t_M, (q^t_0, \delta_1(q^t_1, m_4), (net^t_M \setminus \{M \to E_1 : m_4\}) \cup m_5) \in succ(q^t_{corr})$
where $m_5 = \begin{cases} \{E_1 \to M : \lambda_1(q^t_1, m_4)\} & \lambda_1(q^t_1, m_4) \neq -, \\ \emptyset & \text{otherwise.} \end{cases}$

ii. $msg = -$
By Definition (9.1b) we have:

$$\begin{aligned} q_0^{t+1} &= \delta_0(q_0^t, -) \\ q_1^{t+1} &= q_1^t \\ q_M^{t+1} &= q_M^t \\ net^{t+1} &= net^t \cup msg' \\ msg' &= \begin{cases} \{E_i \to M : \lambda_M(q_i^t, -)\} & \lambda_M(q_i^t, -)) \neq -, \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned}$$

By Equation (9.2d) it holds that $(\delta_0(q^t_0, -), q^t_1, net^t_M \cup m_3) succ(q^t_{corr})$,
where $m_3 = \begin{cases} \{E_0 \to M : \lambda_0(q^t_0, -)\} & \lambda_0(q^t_0, -) \neq -, \\ \emptyset & \text{otherwise.} \end{cases}$
and
$(q^t_0, \delta_1(q^t_1, -), net^t_M \cup m_6)\} \in succ(q^t_{corr})$
where $m_6 = \begin{cases} \{E_1 \to M : \lambda_1(q^t_1, -)\} & \lambda_1(q^t_1, -) \neq -, \\ \emptyset & \text{otherwise.} \end{cases}$

(b) network loss
By Definition (9.1d), the following holds:

$$\begin{aligned} q_0^{t+1} &= q_0^t \\ q_1^{t+1} &= q_1^t \\ q_m^{t+1} &= q_m^t \\ net^{t+1} &= net^t \setminus \{msg\} \end{aligned}$$

where msg is an arbitrary element of net^t. By Equation (9.2b) it holds that $\forall msg \in net^t_M, (q^t_0, q^t_1, net^t_M \setminus \{msg\}) \in succ(q^t_{corr})$

2. *succ* for incorrect transitions
By Definition (9.1c), a message from an incorrect transition has the form: $msg \in ((\Gamma_i \setminus \{\lambda_i(q^t_i, m) | m \in net^t\}) \setminus \lambda_i(q^t_i, -))$. msg could be added to the net of some $q \in q^t_M$, if the following holds (see Equations (9.4) and (9.2c) – (9.2f)): $\exists(q_0, q_1, net) \in q^t_M$ with $\lambda_i(q_i, -) = msg$ or $\lambda_i(q_i, m) = msg$, $m \in net$. By the induction hypothesis, q^t_M represents a correct tracking at time t. From this it follows that only those

incorrect messages are added to the net of some triple that could also occur in a correct scenario.

QED.

Proof of Lemma 2

Basis first midpoint transition in a trace.
The first state st^1 is a correct tracking at time 1:

$$st^1 = (q_0^1, q_M^1, q_1^1, net^1) = (q_{0,1}, (q_{0,1}, q_{1,1}, \{\}), q_{1,1}, \{\}).$$

The first midpoint transition cannot take place before $st^1 \vdash st^2$. By Lemma 1, this means that the tracking after the first midpoint transition is correct.

Step nth midpoint transition, $n > 1$.
By the induction hypothesis, the tracking is correct after the $(n-1)$th midpoint transition. By Lemma 1, this implies that the tracking is also correct after the nth midpoint transition. QED.

Property II: only possibly correct messages are forwarded

Lemma 5 *M computes outputs correctly (as defined in Definition 10).*

We show the correctness of $\lambda_M(q_M^t, msg)$ in two steps:

1. $msg = (E_i \to M : m)$ was inserted by a correct transition.
   In the proof of Lemma 9.4 we have shown that there is a triple $q_{corr}^t \in q_M^t$ with $q_{corr}^t = (q_0^t, q_1^t, net^t)$. The output of this triple, defined by Equation (9.5), is correct, namely $msg' = (M \to E_j : m)$. For every other triple q, $out(q, msg)$ is either msg' or $-$. Thus, by Equation (9.6), $\lambda_M(q_M^t, E_i \to M : m)$ is correct.

2. msg was inserted by an incorrect transition.
   As seen above, there can only be a $(q_0, q_1, net) \in q_M^t$ with $msg \in net$ if this represents a possibly correct scenario. But, in this case, forwarding msg is correct.[3] QED.

[3]Note that in this case, the endpoints might not be able to continue their run of the protocol; the incorrect endpoint is only able to continue to send messages if they belong to a possibly correct scenario. This is the price of having a permissive firewall.

Chapter 10

Summary

10.1 Discussion

In section 8.2 we analysed the TCP automata of several firewalls. We now compute the TCP midpoint automaton using the construction just presented. For endpoint automata, we use the automaton from the TCP specification for endpoints [ISI81b, page 23].

The endpoint automaton in the TCP specification combines the initiator and the responder role. These roles are handled differently by firewalls, which distinguish between the networks outside and behind the firewall. Normally only one side is allowed to initiate a connection. Therefore we made two copies of the TCP endpoint automaton from the specification, one for each of the roles, which we adapt as follows. We chose E_0 to play the role of the *initiator*. Therefore we denote the state `CLOSED` as the start state of automaton A_0 and delete the state `LISTEN` from A_0. Furthermore, as we are only interested in one run of the protocol, we name the end state of A_0 `CLOSED2` (instead of `CLOSED`). To let E_1 play the role of the *responder*, we denote the state `LISTEN` as the start state of A_1 and delete the state `SYN-SENT` and the transitions from state `CLOSED` to state `LISTEN` from A_1. The resulting, minimised midpoint automaton can be found in Figure 10.1. The abstract test cases generated from this automaton can be found in Appendix C.

As expected, in each state of the midpoint, there is considerable uncertainty about the exact state of the endpoints. This is reflected in the fact that some midpoint states consist of over 60 triples. Despite this, the automaton is of manageable complexity, in particular the number of outgoing transitions per state is small (1 – 3). The multiple transitions reflect the (limited) ways that messages can be sent independently by the endpoints and how they can be reordered by the network.

Note too that our midpoint automaton has 7 more states (a – g) than our reverse-engineered TCP automata. This reflects the additional complexity necessary to properly track possible network events. Let us illustrate this with an example. In our midpoint automaton, it can clearly be seen that, to get from state `SYN_B` to state `FIN1_B`, two messages — an ACK from E_0 and a FIN from E_1 — are needed. These messages are independent and thus can arrive in either order at the firewall. If we look how actual

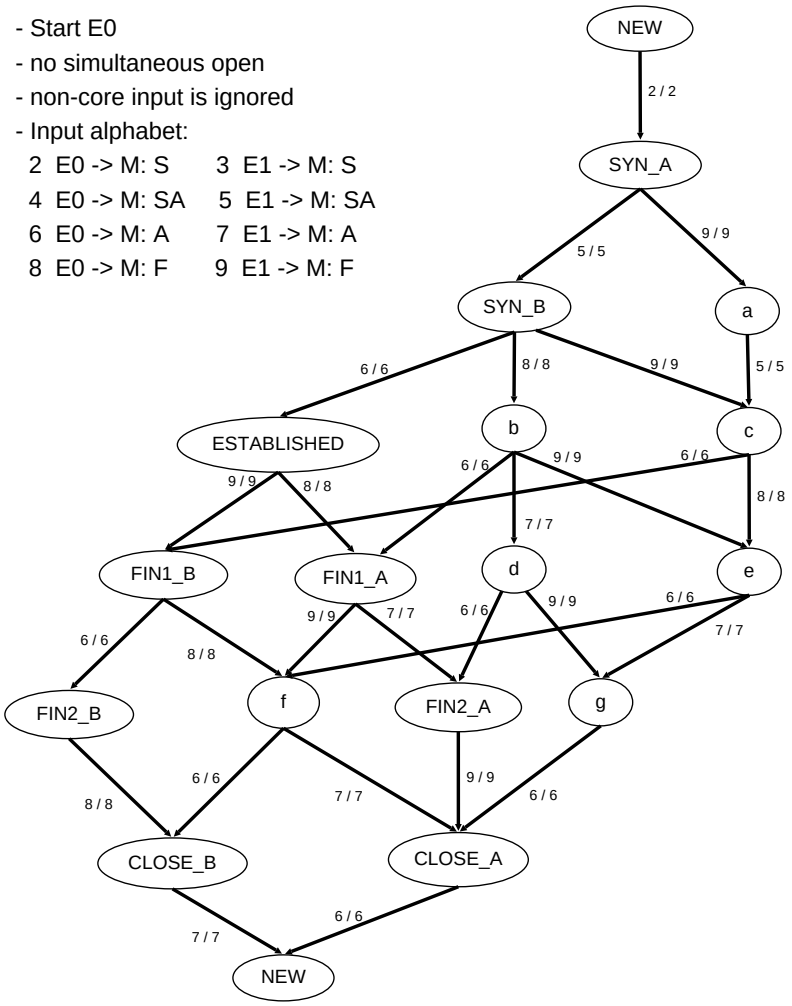


Figure 10.1: Midpoint automaton for TCP

firewalls handle this, we see that the intended order of sending the ACK before the FIN leads to the same result, but that the opposite order ends in state `ESTABLISHED`, leaving us without an explanation why the FIN needs to be allowed in state `SYN_B`.

Our construction builds permissive midpoint automata. This reflects our decision not to penalise protocol-conform endpoints for actions of the environment (here the network). But it is a simple matter to modify the approach to construct restrictive midpoint automata. These can be built by stopping — i.e. dropping everything from then on — at states that consist of more than one triple. But building a restrictive automaton makes little sense with current protocols: It requires dropping more or less everything, as there will be uncertainty already after a few packets.

10.2 Conclusion

In the second part of this thesis we attacked the problem we encountered in the first part: missing midpoint specifications. To be able to test a firewall, we need specifications telling us how a certain protocol should be handled by a firewall. As this problem was not being addressed before, we analysed the problem and developed an algorithm to convert endpoint protocol specifications to midpoint protocol specification.

For part two, our contributions are an analysis of why midpoints must behave, and hence be specified, differently from endpoints, what the implications of a lack of such specifications are, and a solution for this problem. Our solution should be of interest to at least two groups: those building midpoints and those analysing (e.g. testing) them. Both groups will benefit from having a general method to systematically construct midpoint specifications from those for endpoints.

10.3 Future Work

The construction presented has two minor limitations: it requires that the endpoint automata do not have loops without input and it does not take duplication in the network into account. The first point is unproblematic, as loops without input should not be present since these would enable one endpoint to loop infinitely without communicating (only "talking" not "listening") with the other endpoint.

The problem of duplicates (or retransmission), we believe, should be solved independent of protocol automata. The midpoint should remember the packets seen (unique id) and its decision, and then apply the same decision to duplicates received later.

An important step will be to go for other types of midpoints, for example firewalls doing NAT or protocol gateways. For this, our solution must be changed such that additional midpoint functions can be specified and are incorporated.

Part III

Conclusion

Chapter 11

Conclusion and Future Work

The aim of this thesis was to develop a specification-based firewall testing method. The problem to be solved is the one of not being able to determine – or only with great effort of experts – if a firewall accomplishes its job correctly. What "correct" means is different for each firewall, it depends on the corresponding security policy.

As already shown, we have achieved our goal. In this chapter we now want to look at our achievements from an overall point of view.

11.1 Conclusion

Specification-based firewall testing is a novel, exciting and important field. As we have shown in this thesis, it not only consists of testing the conformance of a firewall configuration to a given policy, but also of testing the firewall implementation for correctness. Only if both the firewall configuration conforms to the policy and the firewall implementation is correct, the firewall can be said to implement the policy.

One might say that the above statement is trivially true. Unfortunately it seems not to be so clear: None of the few earlier methods and tools for firewall testing did check both the firewall configuration and the firewall implementation.

Midpoint automata What came as a surprise to us was that it was unclear how a correct firewall implementation should look like. Although several companies are building or testing firewall implementations, there existed no specifications for midpoint (firewall) protocol automata.

As there cannot be a test without a specification, we developed an algorithm for the generation of midpoint automata from endpoint automata (see Part II). This algorithm builds on our analysis of the differences between endpoints and midpoints. We have proven that the midpoint automata, resulting from our algorithm, only accept those messages that could have resulted from protocol-conform endpoints.

With our algorithm, we contribute to a common understanding between midpoint vendors, midpoint testers and midpoint users. Security is only possible if the users are

aware of the capabilities of their firewalls, and the firewall implementation is correct. Only if the midpoint vendors implement what the midpoint users need, the midpoints are useful. The midpoint testers play an important role here: They have to test, on behalf of the users, that the midpoint vendors did their job correctly.

Our contributions for this part are an analysis of why midpoints must behave, and hence be specified, differently from endpoints, an analysis of the implications of a lack of such specifications, and a solution for this problem.

Specification-based firewall testing The main solution to our research question is presented in Part I. We start by defining a language for the formal specification of network security policies. To keep the policy high-level and stable, the low-level details must be specified separately from the policy. In this way, we achieved that the policy can be understood by the accountable managers, while the low-level details can be maintained by qualified technical personnel.

Starting from our specification, we show how to generate test tuples that test the conformance of a firewall configuration to the policy. These test tuples are then used to instantiate abstract test cases (corresponding to a test of the implementation) to get concrete test cases. By choosing this two-phase approach, we achieved a clear distinction between configuration and implementation testing. Using our method, a tester can decide whether he wants to test the firewall configuration (e.g. after a change of the rules or the policy), the firewall implementation (e.g. of a new firewall version) or both. Another advantage of this approach is that existing test cases can be reused. The work of generating abstract test cases for a certain protocol, for example, has only to be done once.

We showed that our method, implemented in a prototype tool, is able to find errors, both in the firewall configuration and the firewall implementation. Note that we cannot prove the absence of errors: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij70].

Our contribution for this part is a product-independent, automated approach for firewall conformance testing. It represents the first approach for the automated conformance testing of firewalls and the first firewall testing method that tests both the firewall configuration and the firewall implementation.

Success Combining both parts of this thesis, we attained our goal. The interest in our work, by banks and military, shows that we solved an important problem – the one of not being able to check (or only with a great effort of experts) if ones defense (by firewalls) is working as expected – and therefore our goal was right:w .

The validation of our approach shows that it is easily applicable, fast, and able to find errors. Therefore we can say that in principle our approach is useful. To use our approach in corporate environments, the usability of our prototype tool needs to be improved.

11.2 Future Work

Detailed future work was already discussed in Sections 5.3.4, 7.2, and 10.3. Here we want to summarise the most important issues.

Adaptation to the application layer During the last years, a shift from packet filters to application layer firewalls can be observed. This means that, to be of long-term use, a firewall testing method needs to be able to handle application layer firewalls.

To adapt our method to the application layer, two main issues need to be solved. On one hand, our formal policy language needs to be adapted such that the interaction between, and the content of, protocols can be specified. On the other hand, fwtest must be adapted such that it can craft messages for any protocol.

Industrial-strength tool The intention behind our prototype tool was validation. This means that usability and nice GUIs were not as important as for an industrial-strength tool. Therefore, to use the tool in corporate environments, its usability and appearance have to be improved. What this means in detail, has already been discussed in subsection 5.3.4.

Case studies In section 5.3 we evaluated and validated our approach. To obtain a more detailed view on the pros and cons of our approach, more case studies are necessary. Every environment has its own demands and its own special cases. Case studies in different environments – different businesses (banking, universities, and so on) as well as different company sizes – will help in improving our method and make it widely applicable.

Part IV

Appendix

Appendix A

Validation – Test Tuples

```
1  /**
2   * $Id: armasuisse_test_tuples.txt 32816 2006-11-30 12:24:57Z dsenn $
3   *
4   * armasuisse-Fallstudie -- test tuples
5  */
6  /* (a, b, c, d) = explizite test tuples
7          (e, f, g, h) = implizite test tuples
8  */
9
10
11
12 /*
13 *                 ->        LAN-BROADCAST    DROP *
14 */
15
16
17
18 /*
19 XP_IP_RANGE1-3  ->        FILE_SERVER       ACCEPT xxx, web
20 */
21 (10.20.48.2, 10.20.30.11, NetBios, ACCEPT)
22 (10.20.48.231, 10.20.30.11, NetBios, ACCEPT)
23 (10.20.49.254, 10.20.30.11, NetBios, ACCEPT)
24          (10.20.47.254, 10.20.30.11, NetBios, DROP)
25          (10.20.50.2, 10.20.30.11, NetBios, DROP)
26
27          (10.20.48.2, 10.20.30.10, NetBios, DROP)
28          (10.20.48.231, 10.20.30.10, NetBios, DROP)
29          (10.20.49.254, 10.20.30.10, NetBios, DROP)
30
31          (10.20.48.2, 10.20.30.12, NetBios, DROP)
32          (10.20.48.231, 10.20.30.12, NetBios, DROP)
33          (10.20.49.254, 10.20.30.12, NetBios, DROP)
34
35
36 (10.20.51.2, 10.20.30.11, NetBios, ACCEPT)
37 (10.20.51.128, 10.20.30.11, NetBios, ACCEPT)
38 (10.20.51.254, 10.20.30.11, NetBios, ACCEPT)
39          (10.20.50.254, 10.20.30.11, NetBios, DROP)
40          (10.20.52.2, 10.20.30.11, NetBios, DROP)
41
42          (10.20.51.2, 10.20.30.10, NetBios, DROP)
43          (10.20.51.128, 10.20.30.10, NetBios, DROP)
44          (10.20.51.254, 10.20.30.10, NetBios, DROP)
45
```

```
46          (10.20.51.2, 10.20.30.12, NetBios, DROP)
47          (10.20.51.128, 10.20.30.12, NetBios, DROP)
48          (10.20.51.254, 10.20.30.12, NetBios, DROP)
49
50
51  (10.20.60.2, 10.20.30.11, NetBios, ACCEPT)
52  (10.20.60.231, 10.20.30.11, NetBios, ACCEPT)
53  (10.20.61.254, 10.20.30.11, NetBios, ACCEPT)
54          (10.20.59.254, 10.20.30.11, NetBios, DROP)
55          (10.20.62.2, 10.20.30.11, NetBios, DROP)
56
57          (10.20.60.2, 10.20.30.10, NetBios, DROP)
58          (10.20.60.231, 10.20.30.10, NetBios, DROP)
59          (10.20.61.254, 10.20.30.10, NetBios, DROP)
60
61          (10.20.60.2, 10.20.30.12, NetBios, DROP)
62          (10.20.60.231, 10.20.30.12, NetBios, DROP)
63          (10.20.61.254, 10.20.30.12, NetBios, DROP)
64
65
66  (10.20.48.2, 10.20.30.11, http, ACCEPT)
67  (10.20.48.231, 10.20.30.11, http, ACCEPT)
68  (10.20.49.254, 10.20.30.11, http, ACCEPT)
69          (10.20.47.254, 10.20.30.11, http, DROP)
70          (10.20.50.2, 10.20.30.11, http, DROP)
71
72          (10.20.48.2, 10.20.30.10, http, DROP)
73          (10.20.48.231, 10.20.30.10, http, DROP)
74          (10.20.49.254, 10.20.30.10, http, DROP)
75
76          (10.20.48.2, 10.20.30.12, http, DROP)
77          (10.20.48.231, 10.20.30.12, http, DROP)
78          (10.20.49.254, 10.20.30.12, http, DROP)
79
80
81  (10.20.51.2, 10.20.30.11, http, ACCEPT)
82  (10.20.51.128, 10.20.30.11, http, ACCEPT)
83  (10.20.51.254, 10.20.30.11, http, ACCEPT)
84          (10.20.50.254, 10.20.30.11, http, DROP)
85          (10.20.52.2, 10.20.30.11, http, DROP)
86
87          (10.20.51.2, 10.20.30.10, http, DROP)
88          (10.20.51.128, 10.20.30.10, http, DROP)
89          (10.20.51.254, 10.20.30.10, http, DROP)
90
91          (10.20.51.2, 10.20.30.12, http, DROP)
92          (10.20.51.128, 10.20.30.12, http, DROP)
93          (10.20.51.254, 10.20.30.12, http, DROP)
94
95
96  (10.20.60.2, 10.20.30.11, http, ACCEPT)
97  (10.20.60.231, 10.20.30.11, http, ACCEPT)
98  (10.20.61.254, 10.20.30.11, http, ACCEPT)
99          (10.20.59.254, 10.20.30.11, http, DROP)
100         (10.20.62.2, 10.20.30.11, http, DROP)
101
102         (10.20.60.2, 10.20.30.10, http, DROP)
103         (10.20.60.231, 10.20.30.10, http, DROP)
104         (10.20.61.254, 10.20.30.10, http, DROP)
105
106         (10.20.60.2, 10.20.30.12, http, DROP)
107         (10.20.60.231, 10.20.30.12, http, DROP)
108         (10.20.61.254, 10.20.30.12, http, DROP)
109
```

```
110
111 /*
112 JMA_CLIENT       ->       FILE_SERVER      ACCEPT data, xxx
113 */
114 (10.20.71.60, 10.20.30.11, ftp, ACCEPT)
115         (10.20.71.59, 10.20.30.11, ftp, DROP)
116         (10.20.71.61, 10.20.30.11, ftp, DROP)
117
118         (10.20.71.60, 10.20.30.10, ftp, DROP)
119         (10.20.71.60, 10.20.30.12, ftp, DROP)
120
121
122 (10.20.71.60, 10.20.30.11, NetBios, ACCEPT)
123         (10.20.71.59, 10.20.30.11, NetBios, DROP)
124         (10.20.71.61, 10.20.30.11, NetBios, DROP)
125
126         (10.20.71.60, 10.20.30.10, NetBios, DROP)
127         (10.20.71.60, 10.20.30.12, NetBios, DROP)
128
129
130
131 /*
132 *               ->       PRINT_SERVER     ACCEPT control
133 *
134 */
135 (192.168.66.128, 10.20.30.16, icmp, ACCEPT)
136         (192.168.66.128, 10.20.30.15, icmp, DROP)
137         (192.168.66.128, 10.20.30.17, icmp, DROP)
138 (192.168.33.128, 10.20.30.16, icmp, ACCEPT)
139         (192.168.33.128, 10.20.30.15, icmp, DROP)
140         (192.168.33.128, 10.20.30.17, icmp, DROP)
141 (172.100.200.19, 10.20.30.16, icmp, ACCEPT)
142         (172.100.200.19, 10.20.30.15, icmp, DROP)
143         (172.100.200.19, 10.20.30.17, icmp, DROP)
144 (172.100.100.97, 10.20.30.16, icmp, ACCEPT)
145         (172.100.100.97, 10.20.30.15, icmp, DROP)
146         (172.100.100.97, 10.20.30.17, icmp, DROP)
147 (172.20.13.128, 10.20.30.16, icmp, ACCEPT)
148         (172.20.13.128, 10.20.30.15, icmp, DROP)
149         (172.20.13.128, 10.20.30.17, icmp, DROP)
150 (10.33.0.99, 10.20.30.16, icmp, ACCEPT)
151         (10.33.0.99, 10.20.30.15, icmp, DROP)
152         (10.33.0.99, 10.20.30.17, icmp, DROP)
153 (10.20.99.128, 10.20.30.16, icmp, ACCEPT)
154         (10.20.99.128, 10.20.30.15, icmp, DROP)
155         (10.20.99.128, 10.20.30.17, icmp, DROP)
156 (10.20.71.60, 10.20.30.16, icmp, ACCEPT)
157         (10.20.71.60, 10.20.30.15, icmp, DROP)
158         (10.20.71.60, 10.20.30.17, icmp, DROP)
159 (10.20.60.231, 10.20.30.16, icmp, ACCEPT)
160         (10.20.60.231, 10.20.30.15, icmp, DROP)
161         (10.20.60.231, 10.20.30.17, icmp, DROP)
162 (10.20.51.128, 10.20.30.16, icmp, ACCEPT)
163         (10.20.51.128, 10.20.30.15, icmp, DROP)
164         (10.20.51.128, 10.20.30.17, icmp, DROP)
165 (10.20.48.231, 10.20.30.16, icmp, ACCEPT)
166         (10.20.48.231, 10.20.30.15, icmp, DROP)
167         (10.20.48.231, 10.20.30.17, icmp, DROP)
168 (10.20.30.254, 10.20.30.16, icmp, ACCEPT)
169         (10.20.30.254, 10.20.30.15, icmp, DROP)
170         (10.20.30.254, 10.20.30.17, icmp, DROP)
171 (10.20.30.45, 10.20.30.16, icmp, ACCEPT)
172         (10.20.30.45, 10.20.30.15, icmp, DROP)
173         (10.20.30.45, 10.20.30.17, icmp, DROP)
```

```
(10.20.30.26, 10.20.30.16, icmp, ACCEPT)
        (10.20.30.26, 10.20.30.15, icmp, DROP)
        (10.20.30.26, 10.20.30.17, icmp, DROP)
(10.20.30.16, 10.20.30.16, icmp, ACCEPT)
        (10.20.30.16, 10.20.30.15, icmp, DROP)
        (10.20.30.16, 10.20.30.17, icmp, DROP)
(10.20.30.11, 10.20.30.16, icmp, ACCEPT)
        (10.20.30.11, 10.20.30.15, icmp, DROP)
        (10.20.30.11, 10.20.30.17, icmp, DROP)
(10.20.30.2, 10.20.30.16, icmp, ACCEPT)
        (10.20.30.2, 10.20.30.15, icmp, DROP)
        (10.20.30.2, 10.20.30.17, icmp, DROP)
(10.20.24.231, 10.20.30.16, icmp, ACCEPT)
        (10.20.24.231, 10.20.30.15, icmp, DROP)
        (10.20.24.231, 10.20.30.17, icmp, DROP)



/*
SAP_IP_RANGE    ->      PRINT_SERVER    ACCEPT print
*/
(192.168.66.2, 10.20.30.16, printer, ACCEPT)
(192.168.66.128, 10.20.30.16, printer, ACCEPT)
(192.168.66.254, 10.20.30.16, printer, ACCEPT)
        (192.168.65.254, 10.20.30.16, printer, DROP)
        (192.168.67.2, 10.20.30.16, printer, DROP)

        (192.168.66.2, 10.20.30.15, printer, DROP)
        (192.168.66.128, 10.20.30.15, printer, DROP)
        (192.168.66.254, 10.20.30.15, printer, DROP)

        (192.168.66.2, 10.20.30.17, printer, DROP)
        (192.168.66.128, 10.20.30.17, printer, DROP)
        (192.168.66.254, 10.20.30.17, printer, DROP)



/*
TRH_CLIENT      ->      TRH_SERVER      ACCEPT xxx
*/
(10.20.99.2, 10.20.30.26, NetBios, ACCEPT)
(10.20.99.128, 10.20.30.26, NetBios, ACCEPT)
(10.20.99.254, 10.20.30.26, NetBios, ACCEPT)
        (10.20.98.254, 10.20.30.26, NetBios, DROP)
        (10.20.100.2, 10.20.30.26, NetBios, DROP)

        (10.20.99.2, 10.20.30.25, NetBios, DROP)
        (10.20.99.128, 10.20.30.25, NetBios, DROP)
        (10.20.99.254, 10.20.30.25, NetBios, DROP)

        (10.20.99.2, 10.20.30.27, NetBios, DROP)
        (10.20.99.128, 10.20.30.27, NetBios, DROP)
        (10.20.99.254, 10.20.30.27, NetBios, DROP)



/*
*               ->      WIKI_SERVER     ACCEPT web, sec-web
*/
(192.168.66.128, 10.20.30.45, http, ACCEPT)
        (192.168.66.128, 10.20.30.44, http, DROP)
        (192.168.66.128, 10.20.30.46, http, DROP)
(192.168.33.128, 10.20.30.45, http, ACCEPT)
        (192.168.33.128, 10.20.30.44, http, DROP)
```

```
238         (192.168.33.128, 10.20.30.46, http, DROP)
239 (172.100.200.19, 10.20.30.45, http, ACCEPT)
240         (172.100.200.19, 10.20.30.44, http, DROP)
241         (172.100.200.19, 10.20.30.46, http, DROP)
242 (172.100.100.97, 10.20.30.45, http, ACCEPT)
243         (172.100.100.97, 10.20.30.44, http, DROP)
244         (172.100.100.97, 10.20.30.46, http, DROP)
245 (172.20.13.128, 10.20.30.45, http, ACCEPT)
246         (172.20.13.128, 10.20.30.44, http, DROP)
247         (172.20.13.128, 10.20.30.46, http, DROP)
248 (10.33.0.99, 10.20.30.16, http, ACCEPT)
249         (10.33.0.99, 10.20.30.15, http, DROP)
250         (10.33.0.99, 10.20.30.17, http, DROP)
251 (10.20.99.128, 10.20.30.45, http, ACCEPT)
252         (10.20.99.128, 10.20.30.44, http, DROP)
253         (10.20.99.128, 10.20.30.46, http, DROP)
254 (10.20.71.60, 10.20.30.45, http, ACCEPT)
255         (10.20.71.60, 10.20.30.44, http, DROP)
256         (10.20.71.60, 10.20.30.46, http, DROP)
257 (10.20.60.231, 10.20.30.45, http, ACCEPT)
258         (10.20.60.231, 10.20.30.44, http, DROP)
259         (10.20.60.231, 10.20.30.46, http, DROP)
260 (10.20.51.128, 10.20.30.45, http, ACCEPT)
261         (10.20.51.128, 10.20.30.44, http, DROP)
262         (10.20.51.128, 10.20.30.46, http, DROP)
263 (10.20.48.231, 10.20.30.45, http, ACCEPT)
264         (10.20.48.231, 10.20.30.44, http, DROP)
265         (10.20.48.231, 10.20.30.46, http, DROP)
266 (10.20.30.254, 10.20.30.45, http, ACCEPT)
267         (10.20.30.254, 10.20.30.44, http, DROP)
268         (10.20.30.254, 10.20.30.46, http, DROP)
269 (10.20.30.45, 10.20.30.45, http, ACCEPT)
270         (10.20.30.45, 10.20.30.44, http, DROP)
271         (10.20.30.45, 10.20.30.46, http, DROP)
272 (10.20.30.26, 10.20.30.45, http, ACCEPT)
273         (10.20.30.26, 10.20.30.44, http, DROP)
274         (10.20.30.26, 10.20.30.46, http, DROP)
275 (10.20.30.16, 10.20.30.45, http, ACCEPT)
276         (10.20.30.16, 10.20.30.44, http, DROP)
277         (10.20.30.16, 10.20.30.46, http, DROP)
278 (10.20.30.11, 10.20.30.45, http, ACCEPT)
279         (10.20.30.11, 10.20.30.44, http, DROP)
280         (10.20.30.11, 10.20.30.46, http, DROP)
281 (10.20.30.2, 10.20.30.45, http, ACCEPT)
282         (10.20.30.2, 10.20.30.44, http, DROP)
283         (10.20.30.2, 10.20.30.46, http, DROP)
284 (10.20.24.231, 10.20.30.45, http, ACCEPT)
285         (10.20.24.231, 10.20.30.44, http, DROP)
286         (10.20.24.231, 10.20.30.46, http, DROP)
287
288
289 (192.168.66.128, 10.20.30.45, https, ACCEPT)
290         (192.168.66.128, 10.20.30.44, https, DROP)
291         (192.168.66.128, 10.20.30.46, https, DROP)
292 (192.168.33.128, 10.20.30.45, https, ACCEPT)
293         (192.168.33.128, 10.20.30.44, https, DROP)
294         (192.168.33.128, 10.20.30.46, https, DROP)
295 (172.100.200.19, 10.20.30.45, https, ACCEPT)
296         (172.100.200.19, 10.20.30.44, https, DROP)
297         (172.100.200.19, 10.20.30.46, https, DROP)
298 (172.100.100.97, 10.20.30.45, https, ACCEPT)
299         (172.100.100.97, 10.20.30.44, https, DROP)
300         (172.100.100.97, 10.20.30.46, https, DROP)
301 (172.20.13.128, 10.20.30.45, https, ACCEPT)
```

```
302          (172.20.13.128, 10.20.30.44, https, DROP)
303          (172.20.13.128, 10.20.30.46, https, DROP)
304  (10.33.0.99, 10.20.30.16, https, ACCEPT)
305          (10.33.0.99, 10.20.30.15, https, DROP)
306          (10.33.0.99, 10.20.30.17, https, DROP)
307  (10.20.99.128, 10.20.30.45, https, ACCEPT)
308          (10.20.99.128, 10.20.30.44, https, DROP)
309          (10.20.99.128, 10.20.30.46, https, DROP)
310  (10.20.71.60, 10.20.30.45, https, ACCEPT)
311          (10.20.71.60, 10.20.30.44, https, DROP)
312          (10.20.71.60, 10.20.30.46, https, DROP)
313  (10.20.60.231, 10.20.30.45, https, ACCEPT)
314          (10.20.60.231, 10.20.30.44, https, DROP)
315          (10.20.60.231, 10.20.30.46, https, DROP)
316  (10.20.51.128, 10.20.30.45, https, ACCEPT)
317          (10.20.51.128, 10.20.30.44, https, DROP)
318          (10.20.51.128, 10.20.30.46, https, DROP)
319  (10.20.48.231, 10.20.30.45, https, ACCEPT)
320          (10.20.48.231, 10.20.30.44, https, DROP)
321          (10.20.48.231, 10.20.30.46, https, DROP)
322  (10.20.30.254, 10.20.30.45, https, ACCEPT)
323          (10.20.30.254, 10.20.30.44, https, DROP)
324          (10.20.30.254, 10.20.30.46, https, DROP)
325  (10.20.30.45, 10.20.30.45, https, ACCEPT)
326          (10.20.30.45, 10.20.30.44, https, DROP)
327          (10.20.30.45, 10.20.30.46, https, DROP)
328  (10.20.30.26, 10.20.30.45, https, ACCEPT)
329          (10.20.30.26, 10.20.30.44, https, DROP)
330          (10.20.30.26, 10.20.30.46, https, DROP)
331  (10.20.30.16, 10.20.30.45, https, ACCEPT)
332          (10.20.30.16, 10.20.30.44, https, DROP)
333          (10.20.30.16, 10.20.30.46, https, DROP)
334  (10.20.30.11, 10.20.30.45, https, ACCEPT)
335          (10.20.30.11, 10.20.30.44, https, DROP)
336          (10.20.30.11, 10.20.30.46, https, DROP)
337  (10.20.30.2, 10.20.30.45, https, ACCEPT)
338          (10.20.30.2, 10.20.30.44, https, DROP)
339          (10.20.30.2, 10.20.30.46, https, DROP)
340  (10.20.24.231, 10.20.30.45, https, ACCEPT)
341          (10.20.24.231, 10.20.30.44, https, DROP)
342          (10.20.24.231, 10.20.30.46, https, DROP)
343
344
345
346  /*
347  XP_IP_RANGE      ->      LAN              ACCEPT *
348  */
349  (10.20.24.2, 10.20.30.2, ftp, ACCEPT)
350  (10.20.24.231, 10.20.30.2, ftp, ACCEPT)
351  (10.20.25.254, 10.20.30.2, ftp, ACCEPT)
352          (10.20.23.254, 10.20.30.2, ftp, DROP)
353          (10.20.26.2, 10.20.30.2, ftp, DROP)
354  (10.20.24.2, 10.20.30.128, ftp, ACCEPT)
355  (10.20.24.231, 10.20.30.128, ftp, ACCEPT)
356  (10.20.25.254, 10.20.30.128, ftp, ACCEPT)
357          (10.20.23.254, 10.20.30.128, ftp, DROP)
358          (10.20.26.2, 10.20.30.128, ftp, DROP)
359  (10.20.24.2, 10.20.30.254, ftp, ACCEPT)
360  (10.20.24.231, 10.20.30.254, ftp, ACCEPT)
361  (10.20.25.254, 10.20.30.254, ftp, ACCEPT)
362          (10.20.23.254, 10.20.30.254, ftp, DROP)
363          (10.20.26.2, 10.20.30.254, ftp, DROP)
364
365
```

```
366 (10.20.24.2, 10.20.30.2, NetBios, ACCEPT)
367 (10.20.24.231, 10.20.30.2, NetBios, ACCEPT)
368 (10.20.25.254, 10.20.30.2, NetBios, ACCEPT)
369         (10.20.23.254, 10.20.30.2, NetBios, DROP)
370         (10.20.26.2, 10.20.30.2, NetBios, DROP)
371 (10.20.24.2, 10.20.30.128, NetBios, ACCEPT)
372 (10.20.24.231, 10.20.30.128, NetBios, ACCEPT)
373 (10.20.25.254, 10.20.30.128, NetBios, ACCEPT)
374         (10.20.23.254, 10.20.30.128, NetBios, DROP)
375         (10.20.26.2, 10.20.30.128, NetBios, DROP)
376 (10.20.24.2, 10.20.30.254, NetBios, ACCEPT)
377 (10.20.24.231, 10.20.30.254, NetBios, ACCEPT)
378 (10.20.25.254, 10.20.30.254, NetBios, ACCEPT)
379         (10.20.23.254, 10.20.30.254, NetBios, DROP)
380         (10.20.26.2, 10.20.30.254, NetBios, DROP)
381
382
383 (10.20.24.2, 10.20.30.2, http, ACCEPT)
384 (10.20.24.231, 10.20.30.2, http, ACCEPT)
385 (10.20.25.254, 10.20.30.2, http, ACCEPT)
386         (10.20.23.254, 10.20.30.2, http, DROP)
387         (10.20.26.2, 10.20.30.2, http, DROP)
388 (10.20.24.2, 10.20.30.128, http, ACCEPT)
389 (10.20.24.231, 10.20.30.128, http, ACCEPT)
390 (10.20.25.254, 10.20.30.128, http, ACCEPT)
391         (10.20.23.254, 10.20.30.128, http, DROP)
392         (10.20.26.2, 10.20.30.128, http, DROP)
393 (10.20.24.2, 10.20.30.254, http, ACCEPT)
394 (10.20.24.231, 10.20.30.254, http, ACCEPT)
395 (10.20.25.254, 10.20.30.254, http, ACCEPT)
396         (10.20.23.254, 10.20.30.254, http, DROP)
397         (10.20.26.2, 10.20.30.254, http, DROP)
398
399
400 (10.20.24.2, 10.20.30.2, https, ACCEPT)
401 (10.20.24.231, 10.20.30.2, https, ACCEPT)
402 (10.20.25.254, 10.20.30.2, https, ACCEPT)
403         (10.20.23.254, 10.20.30.2, https, DROP)
404         (10.20.26.2, 10.20.30.2, https, DROP)
405 (10.20.24.2, 10.20.30.128, https, ACCEPT)
406 (10.20.24.231, 10.20.30.128, https, ACCEPT)
407 (10.20.25.254, 10.20.30.128, https, ACCEPT)
408         (10.20.23.254, 10.20.30.128, https, DROP)
409         (10.20.26.2, 10.20.30.128, https, DROP)
410 (10.20.24.2, 10.20.30.254, https, ACCEPT)
411 (10.20.24.231, 10.20.30.254, https, ACCEPT)
412 (10.20.25.254, 10.20.30.254, https, ACCEPT)
413         (10.20.23.254, 10.20.30.254, https, DROP)
414         (10.20.26.2, 10.20.30.254, https, DROP)
415
416
417 (10.20.24.2, 10.20.30.2, printer, ACCEPT)
418 (10.20.24.231, 10.20.30.2, printer, ACCEPT)
419 (10.20.25.254, 10.20.30.2, printer, ACCEPT)
420         (10.20.23.254, 10.20.30.2, printer, DROP)
421         (10.20.26.2, 10.20.30.2, printer, DROP)
422 (10.20.24.2, 10.20.30.128, printer, ACCEPT)
423 (10.20.24.231, 10.20.30.128, printer, ACCEPT)
424 (10.20.25.254, 10.20.30.128, printer, ACCEPT)
425         (10.20.23.254, 10.20.30.128, printer, DROP)
426         (10.20.26.2, 10.20.30.128, printer, DROP)
427 (10.20.24.2, 10.20.30.254, printer, ACCEPT)
428 (10.20.24.231, 10.20.30.254, printer, ACCEPT)
429 (10.20.25.254, 10.20.30.254, printer, ACCEPT)
```

```
430         (10.20.23.254, 10.20.30.254, printer, DROP)
431         (10.20.26.2, 10.20.30.254, printer, DROP)
432
433
434 (10.20.24.2, 10.20.30.2, icmp, ACCEPT)
435 (10.20.24.231, 10.20.30.2, icmp, ACCEPT)
436 (10.20.25.254, 10.20.30.2, icmp, ACCEPT)
437         (10.20.23.254, 10.20.30.2, icmp, DROP)
438         (10.20.26.2, 10.20.30.2, icmp, DROP)
439 (10.20.24.2, 10.20.30.128, icmp, ACCEPT)
440 (10.20.24.231, 10.20.30.128, icmp, ACCEPT)
441 (10.20.25.254, 10.20.30.128, icmp, ACCEPT)
442         (10.20.23.254, 10.20.30.128, icmp, DROP)
443         (10.20.26.2, 10.20.30.128, icmp, DROP)
444 (10.20.24.2, 10.20.30.254, icmp, ACCEPT)
445 (10.20.24.231, 10.20.30.254, icmp, ACCEPT)
446 (10.20.25.254, 10.20.30.254, icmp, ACCEPT)
447         (10.20.23.254, 10.20.30.254, icmp, DROP)
448         (10.20.26.2, 10.20.30.254, icmp, DROP)
449
450
451 (10.20.24.2, 10.20.30.2, telnet, ACCEPT)
452 (10.20.24.231, 10.20.30.2, telnet, ACCEPT)
453 (10.20.25.254, 10.20.30.2, telnet, ACCEPT)
454         (10.20.23.254, 10.20.30.2, telnet, DROP)
455         (10.20.26.2, 10.20.30.2, telnet, DROP)
456 (10.20.24.2, 10.20.30.128, telnet, ACCEPT)
457 (10.20.24.231, 10.20.30.128, telnet, ACCEPT)
458 (10.20.25.254, 10.20.30.128, telnet, ACCEPT)
459         (10.20.23.254, 10.20.30.128, telnet, DROP)
460         (10.20.26.2, 10.20.30.128, telnet, DROP)
461 (10.20.24.2, 10.20.30.254, telnet, ACCEPT)
462 (10.20.24.231, 10.20.30.254, telnet, ACCEPT)
463 (10.20.25.254, 10.20.30.254, telnet, ACCEPT)
464         (10.20.23.254, 10.20.30.254, telnet, DROP)
465         (10.20.26.2, 10.20.30.254, telnet, DROP)
466
467
468
469 /*
470 SCANNER0-2      ->      LAN             DROP *
471 */
472 (172.100.200.19, 10.20.30.2, ftp, DROP)
473 (172.100.100.97, 10.20.30.2, ftp, DROP)
474 (10.33.0.99, 10.20.30.2, ftp, DROP)
475 (172.100.200.19, 10.20.30.128, ftp, DROP)
476 (172.100.100.97, 10.20.30.128, ftp, DROP)
477 (10.33.0.99, 10.20.30.128, ftp, DROP)
478 (172.100.200.19, 10.20.30.254, ftp, DROP)
479 (172.100.100.97, 10.20.30.254, ftp, DROP)
480 (10.33.0.99, 10.20.30.254, ftp, DROP)
481
482
483 (172.100.200.19, 10.20.30.2, NetBios, DROP)
484 (172.100.100.97, 10.20.30.2, NetBios, DROP)
485 (10.33.0.99, 10.20.30.2, NetBios, DROP)
486 (172.100.200.19, 10.20.30.128, NetBios, DROP)
487 (172.100.100.97, 10.20.30.128, NetBios, DROP)
488 (10.33.0.99, 10.20.30.128, NetBios, DROP)
489 (172.100.200.19, 10.20.30.254, NetBios, DROP)
490 (172.100.100.97, 10.20.30.254, NetBios, DROP)
491 (10.33.0.99, 10.20.30.254, NetBios, DROP)
492
493
```

```
494 (172.100.200.19, 10.20.30.2, http, DROP)
495 (172.100.100.97, 10.20.30.2, http, DROP)
496 (10.33.0.99, 10.20.30.2, http, DROP)
497 (172.100.200.19, 10.20.30.128, http, DROP)
498 (172.100.100.97, 10.20.30.128, http, DROP)
499 (10.33.0.99, 10.20.30.128, http, DROP)
500 (172.100.200.19, 10.20.30.254, http, DROP)
501 (172.100.100.97, 10.20.30.254, http, DROP)
502 (10.33.0.99, 10.20.30.254, http, DROP)
503
504
505 (172.100.200.19, 10.20.30.2, https, DROP)
506 (172.100.100.97, 10.20.30.2, https, DROP)
507 (10.33.0.99, 10.20.30.2, https, DROP)
508 (172.100.200.19, 10.20.30.128, https, DROP)
509 (172.100.100.97, 10.20.30.128, https, DROP)
510 (10.33.0.99, 10.20.30.128, https, DROP)
511 (172.100.200.19, 10.20.30.254, https, DROP)
512 (172.100.100.97, 10.20.30.254, https, DROP)
513 (10.33.0.99, 10.20.30.254, https, DROP)
514
515
516 (172.100.200.19, 10.20.30.2, printer, DROP)
517 (172.100.100.97, 10.20.30.2, printer, DROP)
518 (10.33.0.99, 10.20.30.2, printer, DROP)
519 (172.100.200.19, 10.20.30.128, printer, DROP)
520 (172.100.100.97, 10.20.30.128, printer, DROP)
521 (10.33.0.99, 10.20.30.128, printer, DROP)
522 (172.100.200.19, 10.20.30.254, printer, DROP)
523 (172.100.100.97, 10.20.30.254, printer, DROP)
524 (10.33.0.99, 10.20.30.254, printer, DROP)
525
526
527 (172.100.200.19, 10.20.30.2, icmp, DROP)
528 (172.100.100.97, 10.20.30.2, icmp, DROP)
529 (10.33.0.99, 10.20.30.2, icmp, DROP)
530 (172.100.200.19, 10.20.30.128, icmp, DROP)
531 (172.100.100.97, 10.20.30.128, icmp, DROP)
532 (10.33.0.99, 10.20.30.128, icmp, DROP)
533 (172.100.200.19, 10.20.30.254, icmp, DROP)
534 (172.100.100.97, 10.20.30.254, icmp, DROP)
535 (10.33.0.99, 10.20.30.254, icmp, DROP)
536
537
538 (172.100.200.19, 10.20.30.2, telnet, DROP)
539 (172.100.100.97, 10.20.30.2, telnet, DROP)
540 (10.33.0.99, 10.20.30.2, telnet, DROP)
541 (172.100.200.19, 10.20.30.128, telnet, DROP)
542 (172.100.100.97, 10.20.30.128, telnet, DROP)
543 (10.33.0.99, 10.20.30.128, telnet, DROP)
544 (172.100.200.19, 10.20.30.254, telnet, DROP)
545 (172.100.100.97, 10.20.30.254, telnet, DROP)
546 (10.33.0.99, 10.20.30.254, telnet, DROP)
547
548
549
550 /*
551 LAN                -> !NOTALLOWED ACCEPT *
552 */
553         (10.20.30.2, 192.168.33.2, ftp, DROP)
554         (10.20.30.128, 192.168.33.2, ftp, DROP)
555         (10.20.30.254, 192.168.33.2, ftp, DROP)
556         (10.20.30.2, 192.168.33.128, ftp, DROP)
557         (10.20.30.128, 192.168.33.128, ftp, DROP)
```

```
558          (10.20.30.254, 192.168.33.128, ftp, DROP)
559          (10.20.30.2, 192.168.33.254, ftp, DROP)
560          (10.20.30.128, 192.168.33.254, ftp, DROP)
561          (10.20.30.254, 192.168.33.254, ftp, DROP)
562          (10.20.30.2, 172.20.13.2, ftp, DROP)
563          (10.20.30.128, 172.20.13.2, ftp, DROP)
564          (10.20.30.254, 172.20.13.2, ftp, DROP)
565          (10.20.30.2, 172.20.13.128, ftp, DROP)
566          (10.20.30.128, 172.20.13.128, ftp, DROP)
567          (10.20.30.254, 172.20.13.128, ftp, DROP)
568          (10.20.30.2, 172.20.13.254, ftp, DROP)
569          (10.20.30.128, 172.20.13.254, ftp, DROP)
570          (10.20.30.254, 172.20.13.254, ftp, DROP)
571  (10.20.29.254, 192.168.33.2, ftp, ACCEPT)
572  (10.20.31.2, 192.168.33.2, ftp, ACCEPT)
573  (10.20.29.254, 192.168.33.128, ftp, ACCEPT)
574  (10.20.31.2, 192.168.33.128, ftp, ACCEPT)
575  (10.20.29.254, 192.168.33.254, ftp, ACCEPT)
576  (10.20.31.2, 192.168.33.254, ftp, ACCEPT)
577
578  (10.20.30.2, 192.168.66.128, ftp, ACCEPT)
579  (10.20.30.128, 192.168.66.128, ftp, ACCEPT)
580  (10.20.30.254, 192.168.66.128, ftp, ACCEPT)
581          (10.20.29.254, 192.168.66.128, ftp, DROP)
582          (10.20.31.2, 192.168.66.128, ftp, DROP)
583  (10.20.30.2, 172.100.200.19, ftp, ACCEPT)
584  (10.20.30.128, 172.100.200.19, ftp, ACCEPT)
585  (10.20.30.254, 172.100.200.19, ftp, ACCEPT)
586          (10.20.29.254, 172.100.200.19, ftp, DROP)
587          (10.20.31.2, 172.100.200.19, ftp, DROP)
588  (10.20.30.2, 172.100.100.97, ftp, ACCEPT)
589  (10.20.30.128, 172.100.100.97, ftp, ACCEPT)
590  (10.20.30.254, 172.100.100.97, ftp, ACCEPT)
591          (10.20.29.254, 172.100.100.97, ftp, DROP)
592          (10.20.31.2, 172.100.100.97, ftp, DROP)
593  (10.20.30.2, 10.33.0.99, ftp, ACCEPT)
594  (10.20.30.128, 10.33.0.99, ftp, ACCEPT)
595  (10.20.30.254, 10.33.0.99, ftp, ACCEPT)
596          (10.20.29.254, 10.33.0.99, ftp, DROP)
597          (10.20.31.2, 10.33.0.99, ftp, DROP)
598  (10.20.30.2, 10.20.99.128, ftp, ACCEPT)
599  (10.20.30.128, 10.20.99.128, ftp, ACCEPT)
600  (10.20.30.254, 10.20.99.128, ftp, ACCEPT)
601          (10.20.29.254, 10.20.99.128, ftp, DROP)
602          (10.20.31.2, 10.20.99.128, ftp, DROP)
603  (10.20.30.2, 10.20.71.60, ftp, ACCEPT)
604  (10.20.30.128, 10.20.71.60, ftp, ACCEPT)
605  (10.20.30.254, 10.20.71.60, ftp, ACCEPT)
606          (10.20.29.254, 10.20.71.60, ftp, DROP)
607          (10.20.31.2, 10.20.71.60, ftp, DROP)
608  (10.20.30.2, 10.20.60.231, ftp, ACCEPT)
609  (10.20.30.128, 10.20.60.231, ftp, ACCEPT)
610  (10.20.30.254, 10.20.60.231, ftp, ACCEPT)
611          (10.20.29.254, 10.20.60.231, ftp, DROP)
612          (10.20.31.2, 10.20.60.231, ftp, DROP)
613  (10.20.30.2, 10.20.51.128, ftp, ACCEPT)
614  (10.20.30.128, 10.20.51.128, ftp, ACCEPT)
615  (10.20.30.254, 10.20.51.128, ftp, ACCEPT)
616          (10.20.29.254, 10.20.51.128, ftp, DROP)
617          (10.20.31.2, 10.20.51.128, ftp, DROP)
618  (10.20.30.2, 10.20.48.231, ftp, ACCEPT)
619  (10.20.30.128, 10.20.48.231, ftp, ACCEPT)
620  (10.20.30.254, 10.20.48.231, ftp, ACCEPT)
621          (10.20.29.254, 10.20.48.231, ftp, DROP)
```

```
622             (10.20.31.2, 10.20.48.231, ftp, DROP)
623     (10.20.30.2, 10.20.30.254, ftp, ACCEPT)
624     (10.20.30.128, 10.20.30.254, ftp, ACCEPT)
625     (10.20.30.254, 10.20.30.254, ftp, ACCEPT)
626             (10.20.29.254, 10.20.30.254, ftp, DROP)
627             (10.20.31.2, 10.20.30.254, ftp, DROP)
628     (10.20.30.2, 10.20.30.45, ftp, ACCEPT)
629     (10.20.30.128, 10.20.30.45, ftp, ACCEPT)
630     (10.20.30.254, 10.20.30.45, ftp, ACCEPT)
631             (10.20.29.254, 10.20.30.45, ftp, DROP)
632             (10.20.31.2, 10.20.30.45, ftp, DROP)
633     (10.20.30.2, 10.20.30.26, ftp, ACCEPT)
634     (10.20.30.128, 10.20.30.26, ftp, ACCEPT)
635     (10.20.30.254, 10.20.30.26, ftp, ACCEPT)
636             (10.20.29.254, 10.20.30.26, ftp, DROP)
637             (10.20.31.2, 10.20.30.26, ftp, DROP)
638     (10.20.30.2, 10.20.30.16, ftp, ACCEPT)
639     (10.20.30.128, 10.20.30.16, ftp, ACCEPT)
640     (10.20.30.254, 10.20.30.16, ftp, ACCEPT)
641             (10.20.29.254, 10.20.30.16, ftp, DROP)
642             (10.20.31.2, 10.20.30.16, ftp, DROP)
643     (10.20.30.2, 10.20.30.11, ftp, ACCEPT)
644     (10.20.30.128, 10.20.30.11, ftp, ACCEPT)
645     (10.20.30.254, 10.20.30.11, ftp, ACCEPT)
646             (10.20.29.254, 10.20.30.11, ftp, DROP)
647             (10.20.31.2, 10.20.30.11, ftp, DROP)
648     (10.20.30.2, 10.20.30.2, ftp, ACCEPT)
649     (10.20.30.128, 10.20.30.2, ftp, ACCEPT)
650     (10.20.30.254, 10.20.30.2, ftp, ACCEPT)
651             (10.20.29.254, 10.20.30.2, ftp, DROP)
652             (10.20.31.2, 10.20.30.2, ftp, DROP)
653     (10.20.30.2, 10.20.24.231, ftp, ACCEPT)
654     (10.20.30.128, 10.20.24.231, ftp, ACCEPT)
655     (10.20.30.254, 10.20.24.231, ftp, ACCEPT)
656             (10.20.29.254, 10.20.24.231, ftp, DROP)
657             (10.20.31.2, 10.20.24.231, ftp, DROP)
658
659
660     (10.20.30.2, 192.168.33.2, NetBios, DROP)
661     (10.20.30.128, 192.168.33.2, NetBios, DROP)
662     (10.20.30.254, 192.168.33.2, NetBios, DROP)
663     (10.20.30.2, 192.168.33.128, NetBios, DROP)
664     (10.20.30.128, 192.168.33.128, NetBios, DROP)
665     (10.20.30.254, 192.168.33.128, NetBios, DROP)
666     (10.20.30.2, 192.168.33.254, NetBios, DROP)
667     (10.20.30.128, 192.168.33.254, NetBios, DROP)
668     (10.20.30.254, 192.168.33.254, NetBios, DROP)
669     (10.20.30.2, 172.20.13.2, NetBios, DROP)
670     (10.20.30.128, 172.20.13.2, NetBios, DROP)
671     (10.20.30.254, 172.20.13.2, NetBios, DROP)
672     (10.20.30.2, 172.20.13.128, NetBios, DROP)
673     (10.20.30.128, 172.20.13.128, NetBios, DROP)
674     (10.20.30.254, 172.20.13.128, NetBios, DROP)
675     (10.20.30.2, 172.20.13.254, NetBios, DROP)
676     (10.20.30.128, 172.20.13.254, NetBios, DROP)
677     (10.20.30.254, 172.20.13.254, NetBios, DROP)
678             (10.20.29.254, 192.168.33.2, NetBios, ACCEPT)
679             (10.20.31.2, 192.168.33.2, NetBios, ACCEPT)
680             (10.20.29.254, 192.168.33.128, NetBios, ACCEPT)
681             (10.20.31.2, 192.168.33.128, NetBios, ACCEPT)
682             (10.20.29.254, 192.168.33.254, NetBios, ACCEPT)
683             (10.20.31.2, 192.168.33.254, NetBios, ACCEPT)
684
685     (10.20.30.2, 192.168.66.128, NetBios, ACCEPT)
```

```
686 (10.20.30.128, 192.168.66.128, NetBios, ACCEPT)
687 (10.20.30.254, 192.168.66.128, NetBios, ACCEPT)
688         (10.20.29.254, 192.168.66.128, NetBios, DROP)
689         (10.20.31.2, 192.168.66.128, NetBios, DROP)
690 (10.20.30.2, 172.100.200.19, NetBios, ACCEPT)
691 (10.20.30.128, 172.100.200.19, NetBios, ACCEPT)
692 (10.20.30.254, 172.100.200.19, NetBios, ACCEPT)
693         (10.20.29.254, 172.100.200.19, NetBios, DROP)
694         (10.20.31.2, 172.100.200.19, NetBios, DROP)
695 (10.20.30.2, 172.100.100.97, NetBios, ACCEPT)
696 (10.20.30.128, 172.100.100.97, NetBios, ACCEPT)
697 (10.20.30.254, 172.100.100.97, NetBios, ACCEPT)
698         (10.20.29.254, 172.100.100.97, NetBios, DROP)
699         (10.20.31.2, 172.100.100.97, NetBios, DROP)
700 (10.20.30.2, 10.33.0.99, NetBios, ACCEPT)
701 (10.20.30.128, 10.33.0.99, NetBios, ACCEPT)
702 (10.20.30.254, 10.33.0.99, NetBios, ACCEPT)
703         (10.20.29.254, 10.33.0.99, NetBios, DROP)
704         (10.20.31.2, 10.33.0.99, NetBios, DROP)
705 (10.20.30.2, 10.20.99.128, NetBios, ACCEPT)
706 (10.20.30.128, 10.20.99.128, NetBios, ACCEPT)
707 (10.20.30.254, 10.20.99.128, NetBios, ACCEPT)
708         (10.20.29.254, 10.20.99.128, NetBios, DROP)
709         (10.20.31.2, 10.20.99.128, NetBios, DROP)
710 (10.20.30.2, 10.20.71.60, NetBios, ACCEPT)
711 (10.20.30.128, 10.20.71.60, NetBios, ACCEPT)
712 (10.20.30.254, 10.20.71.60, NetBios, ACCEPT)
713         (10.20.29.254, 10.20.71.60, NetBios, DROP)
714         (10.20.31.2, 10.20.71.60, NetBios, DROP)
715 (10.20.30.2, 10.20.60.231, NetBios, ACCEPT)
716 (10.20.30.128, 10.20.60.231, NetBios, ACCEPT)
717 (10.20.30.254, 10.20.60.231, NetBios, ACCEPT)
718         (10.20.29.254, 10.20.60.231, NetBios, DROP)
719         (10.20.31.2, 10.20.60.231, NetBios, DROP)
720 (10.20.30.2, 10.20.51.128, NetBios, ACCEPT)
721 (10.20.30.128, 10.20.51.128, NetBios, ACCEPT)
722 (10.20.30.254, 10.20.51.128, NetBios, ACCEPT)
723         (10.20.29.254, 10.20.51.128, NetBios, DROP)
724         (10.20.31.2, 10.20.51.128, NetBios, DROP)
725 (10.20.30.2, 10.20.48.231, NetBios, ACCEPT)
726 (10.20.30.128, 10.20.48.231, NetBios, ACCEPT)
727 (10.20.30.254, 10.20.48.231, NetBios, ACCEPT)
728         (10.20.29.254, 10.20.48.231, NetBios, DROP)
729         (10.20.31.2, 10.20.48.231, NetBios, DROP)
730 (10.20.30.2, 10.20.30.254, NetBios, ACCEPT)
731 (10.20.30.128, 10.20.30.254, NetBios, ACCEPT)
732 (10.20.30.254, 10.20.30.254, NetBios, ACCEPT)
733         (10.20.29.254, 10.20.30.254, NetBios, DROP)
734         (10.20.31.2, 10.20.30.254, NetBios, DROP)
735 (10.20.30.2, 10.20.30.45, NetBios, ACCEPT)
736 (10.20.30.128, 10.20.30.45, NetBios, ACCEPT)
737 (10.20.30.254, 10.20.30.45, NetBios, ACCEPT)
738         (10.20.29.254, 10.20.30.45, NetBios, DROP)
739         (10.20.31.2, 10.20.30.45, NetBios, DROP)
740 (10.20.30.2, 10.20.30.26, NetBios, ACCEPT)
741 (10.20.30.128, 10.20.30.26, NetBios, ACCEPT)
742 (10.20.30.254, 10.20.30.26, NetBios, ACCEPT)
743         (10.20.29.254, 10.20.30.26, NetBios, DROP)
744         (10.20.31.2, 10.20.30.26, NetBios, DROP)
745 (10.20.30.2, 10.20.30.16, NetBios, ACCEPT)
746 (10.20.30.128, 10.20.30.16, NetBios, ACCEPT)
747 (10.20.30.254, 10.20.30.16, NetBios, ACCEPT)
748         (10.20.29.254, 10.20.30.16, NetBios, DROP)
749         (10.20.31.2, 10.20.30.16, NetBios, DROP)
```

```
750 (10.20.30.2, 10.20.30.11, NetBios, ACCEPT)
751 (10.20.30.128, 10.20.30.11, NetBios, ACCEPT)
752 (10.20.30.254, 10.20.30.11, NetBios, ACCEPT)
753         (10.20.29.254, 10.20.30.11, NetBios, DROP)
754         (10.20.31.2, 10.20.30.11, NetBios, DROP)
755 (10.20.30.2, 10.20.30.2, NetBios, ACCEPT)
756 (10.20.30.128, 10.20.30.2, NetBios, ACCEPT)
757 (10.20.30.254, 10.20.30.2, NetBios, ACCEPT)
758         (10.20.29.254, 10.20.30.2, NetBios, DROP)
759         (10.20.31.2, 10.20.30.2, NetBios, DROP)
760 (10.20.30.2, 10.20.24.231, NetBios, ACCEPT)
761 (10.20.30.128, 10.20.24.231, NetBios, ACCEPT)
762 (10.20.30.254, 10.20.24.231, NetBios, ACCEPT)
763         (10.20.29.254, 10.20.24.231, NetBios, DROP)
764         (10.20.31.2, 10.20.24.231, NetBios, DROP)
765
766
767 (10.20.30.2, 192.168.33.2, http, DROP)
768 (10.20.30.128, 192.168.33.2, http, DROP)
769 (10.20.30.254, 192.168.33.2, http, DROP)
770 (10.20.30.2, 192.168.33.128, http, DROP)
771 (10.20.30.128, 192.168.33.128, http, DROP)
772 (10.20.30.254, 192.168.33.128, http, DROP)
773 (10.20.30.2, 192.168.33.254, http, DROP)
774 (10.20.30.128, 192.168.33.254, http, DROP)
775 (10.20.30.254, 192.168.33.254, http, DROP)
776 (10.20.30.2, 172.20.13.2, http, DROP)
777 (10.20.30.128, 172.20.13.2, http, DROP)
778 (10.20.30.254, 172.20.13.2, http, DROP)
779 (10.20.30.2, 172.20.13.128, http, DROP)
780 (10.20.30.128, 172.20.13.128, http, DROP)
781 (10.20.30.254, 172.20.13.128, http, DROP)
782 (10.20.30.2, 172.20.13.254, http, DROP)
783 (10.20.30.128, 172.20.13.254, http, DROP)
784 (10.20.30.254, 172.20.13.254, http, DROP)
785         (10.20.29.254, 192.168.33.2, http, ACCEPT)
786         (10.20.31.2, 192.168.33.2, http, ACCEPT)
787         (10.20.29.254, 192.168.33.128, http, ACCEPT)
788         (10.20.31.2, 192.168.33.128, http, ACCEPT)
789         (10.20.29.254, 192.168.33.254, http, ACCEPT)
790         (10.20.31.2, 192.168.33.254, http, ACCEPT)
791
792 (10.20.30.2, 192.168.66.128, http, ACCEPT)
793 (10.20.30.128, 192.168.66.128, http, ACCEPT)
794 (10.20.30.254, 192.168.66.128, http, ACCEPT)
795         (10.20.29.254, 192.168.66.128, http, DROP)
796         (10.20.31.2, 192.168.66.128, http, DROP)
797 (10.20.30.2, 172.100.200.19, http, ACCEPT)
798 (10.20.30.128, 172.100.200.19, http, ACCEPT)
799 (10.20.30.254, 172.100.200.19, http, ACCEPT)
800         (10.20.29.254, 172.100.200.19, http, DROP)
801         (10.20.31.2, 172.100.200.19, http, DROP)
802 (10.20.30.2, 172.100.100.97, http, ACCEPT)
803 (10.20.30.128, 172.100.100.97, http, ACCEPT)
804 (10.20.30.254, 172.100.100.97, http, ACCEPT)
805         (10.20.29.254, 172.100.100.97, http, DROP)
806         (10.20.31.2, 172.100.100.97, http, DROP)
807 (10.20.30.2, 10.33.0.99, http, ACCEPT)
808 (10.20.30.128, 10.33.0.99, http, ACCEPT)
809 (10.20.30.254, 10.33.0.99, http, ACCEPT)
810         (10.20.29.254, 10.33.0.99, http, DROP)
811         (10.20.31.2, 10.33.0.99, http, DROP)
812 (10.20.30.2, 10.20.99.128, http, ACCEPT)
813 (10.20.30.128, 10.20.99.128, http, ACCEPT)
```

```
814 (10.20.30.254, 10.20.99.128, http, ACCEPT)
815         (10.20.29.254, 10.20.99.128, http, DROP)
816         (10.20.31.2, 10.20.99.128, http, DROP)
817 (10.20.30.2, 10.20.71.60, http, ACCEPT)
818 (10.20.30.128, 10.20.71.60, http, ACCEPT)
819 (10.20.30.254, 10.20.71.60, http, ACCEPT)
820         (10.20.29.254, 10.20.71.60, http, DROP)
821         (10.20.31.2, 10.20.71.60, http, DROP)
822 (10.20.30.2, 10.20.60.231, http, ACCEPT)
823 (10.20.30.128, 10.20.60.231, http, ACCEPT)
824 (10.20.30.254, 10.20.60.231, http, ACCEPT)
825         (10.20.29.254, 10.20.60.231, http, DROP)
826         (10.20.31.2, 10.20.60.231, http, DROP)
827 (10.20.30.2, 10.20.51.128, http, ACCEPT)
828 (10.20.30.128, 10.20.51.128, http, ACCEPT)
829 (10.20.30.254, 10.20.51.128, http, ACCEPT)
830         (10.20.29.254, 10.20.51.128, http, DROP)
831         (10.20.31.2, 10.20.51.128, http, DROP)
832 (10.20.30.2, 10.20.48.231, http, ACCEPT)
833 (10.20.30.128, 10.20.48.231, http, ACCEPT)
834 (10.20.30.254, 10.20.48.231, http, ACCEPT)
835         (10.20.29.254, 10.20.48.231, http, DROP)
836         (10.20.31.2, 10.20.48.231, http, DROP)
837 (10.20.30.2, 10.20.30.254, http, ACCEPT)
838 (10.20.30.128, 10.20.30.254, http, ACCEPT)
839 (10.20.30.254, 10.20.30.254, http, ACCEPT)
840         (10.20.29.254, 10.20.30.254, http, DROP)
841         (10.20.31.2, 10.20.30.254, http, DROP)
842 (10.20.30.2, 10.20.30.45, http, ACCEPT)
843 (10.20.30.128, 10.20.30.45, http, ACCEPT)
844 (10.20.30.254, 10.20.30.45, http, ACCEPT)
845         (10.20.29.254, 10.20.30.45, http, DROP)
846         (10.20.31.2, 10.20.30.45, http, DROP)
847 (10.20.30.2, 10.20.30.26, http, ACCEPT)
848 (10.20.30.128, 10.20.30.26, http, ACCEPT)
849 (10.20.30.254, 10.20.30.26, http, ACCEPT)
850         (10.20.29.254, 10.20.30.26, http, DROP)
851         (10.20.31.2, 10.20.30.26, http, DROP)
852 (10.20.30.2, 10.20.30.16, http, ACCEPT)
853 (10.20.30.128, 10.20.30.16, http, ACCEPT)
854 (10.20.30.254, 10.20.30.16, http, ACCEPT)
855         (10.20.29.254, 10.20.30.16, http, DROP)
856         (10.20.31.2, 10.20.30.16, http, DROP)
857 (10.20.30.2, 10.20.30.11, http, ACCEPT)
858 (10.20.30.128, 10.20.30.11, http, ACCEPT)
859 (10.20.30.254, 10.20.30.11, http, ACCEPT)
860         (10.20.29.254, 10.20.30.11, http, DROP)
861         (10.20.31.2, 10.20.30.11, http, DROP)
862 (10.20.30.2, 10.20.30.2, http, ACCEPT)
863 (10.20.30.128, 10.20.30.2, http, ACCEPT)
864 (10.20.30.254, 10.20.30.2, http, ACCEPT)
865         (10.20.29.254, 10.20.30.2, http, DROP)
866         (10.20.31.2, 10.20.30.2, http, DROP)
867 (10.20.30.2, 10.20.24.231, http, ACCEPT)
868 (10.20.30.128, 10.20.24.231, http, ACCEPT)
869 (10.20.30.254, 10.20.24.231, http, ACCEPT)
870         (10.20.29.254, 10.20.24.231, http, DROP)
871         (10.20.31.2, 10.20.24.231, http, DROP)
872
873
874 (10.20.30.2, 192.168.33.2, https, DROP)
875 (10.20.30.128, 192.168.33.2, https, DROP)
876 (10.20.30.254, 192.168.33.2, https, DROP)
877 (10.20.30.2, 192.168.33.128, https, DROP)
```

```
878 (10.20.30.128, 192.168.33.128, https, DROP)
879 (10.20.30.254, 192.168.33.128, https, DROP)
880 (10.20.30.2, 192.168.33.254, https, DROP)
881 (10.20.30.128, 192.168.33.254, https, DROP)
882 (10.20.30.254, 192.168.33.254, https, DROP)
883 (10.20.30.2, 172.20.13.2, https, DROP)
884 (10.20.30.128, 172.20.13.2, https, DROP)
885 (10.20.30.254, 172.20.13.2, https, DROP)
886 (10.20.30.2, 172.20.13.128, https, DROP)
887 (10.20.30.128, 172.20.13.128, https, DROP)
888 (10.20.30.254, 172.20.13.128, https, DROP)
889 (10.20.30.2, 172.20.13.254, https, DROP)
890 (10.20.30.128, 172.20.13.254, https, DROP)
891 (10.20.30.254, 172.20.13.254, https, DROP)
892         (10.20.29.254, 192.168.33.2, https, ACCEPT)
893         (10.20.31.2, 192.168.33.2, https, ACCEPT)
894         (10.20.29.254, 192.168.33.128, https, ACCEPT)
895         (10.20.31.2, 192.168.33.128, https, ACCEPT)
896         (10.20.29.254, 192.168.33.254, https, ACCEPT)
897         (10.20.31.2, 192.168.33.254, https, ACCEPT)
898
899 (10.20.30.2, 192.168.66.128, https, ACCEPT)
900 (10.20.30.128, 192.168.66.128, https, ACCEPT)
901 (10.20.30.254, 192.168.66.128, https, ACCEPT)
902         (10.20.29.254, 192.168.66.128, https, DROP)
903         (10.20.31.2, 192.168.66.128, https, DROP)
904 (10.20.30.2, 172.100.200.19, https, ACCEPT)
905 (10.20.30.128, 172.100.200.19, https, ACCEPT)
906 (10.20.30.254, 172.100.200.19, https, ACCEPT)
907         (10.20.29.254, 172.100.200.19, https, DROP)
908         (10.20.31.2, 172.100.200.19, https, DROP)
909 (10.20.30.2, 172.100.100.97, https, ACCEPT)
910 (10.20.30.128, 172.100.100.97, https, ACCEPT)
911 (10.20.30.254, 172.100.100.97, https, ACCEPT)
912         (10.20.29.254, 172.100.100.97, https, DROP)
913         (10.20.31.2, 172.100.100.97, https, DROP)
914 (10.20.30.2, 10.33.0.99, https, ACCEPT)
915 (10.20.30.128, 10.33.0.99, https, ACCEPT)
916 (10.20.30.254, 10.33.0.99, https, ACCEPT)
917         (10.20.29.254, 10.33.0.99, https, DROP)
918         (10.20.31.2, 10.33.0.99, https, DROP)
919 (10.20.30.2, 10.20.99.128, https, ACCEPT)
920 (10.20.30.128, 10.20.99.128, https, ACCEPT)
921 (10.20.30.254, 10.20.99.128, https, ACCEPT)
922         (10.20.29.254, 10.20.99.128, https, DROP)
923         (10.20.31.2, 10.20.99.128, https, DROP)
924 (10.20.30.2, 10.20.71.60, https, ACCEPT)
925 (10.20.30.128, 10.20.71.60, https, ACCEPT)
926 (10.20.30.254, 10.20.71.60, https, ACCEPT)
927         (10.20.29.254, 10.20.71.60, https, DROP)
928         (10.20.31.2, 10.20.71.60, https, DROP)
929 (10.20.30.2, 10.20.60.231, https, ACCEPT)
930 (10.20.30.128, 10.20.60.231, https, ACCEPT)
931 (10.20.30.254, 10.20.60.231, https, ACCEPT)
932         (10.20.29.254, 10.20.60.231, https, DROP)
933         (10.20.31.2, 10.20.60.231, https, DROP)
934 (10.20.30.2, 10.20.51.128, https, ACCEPT)
935 (10.20.30.128, 10.20.51.128, https, ACCEPT)
936 (10.20.30.254, 10.20.51.128, https, ACCEPT)
937         (10.20.29.254, 10.20.51.128, https, DROP)
938         (10.20.31.2, 10.20.51.128, https, DROP)
939 (10.20.30.2, 10.20.48.231, https, ACCEPT)
940 (10.20.30.128, 10.20.48.231, https, ACCEPT)
941 (10.20.30.254, 10.20.48.231, https, ACCEPT)
```

```
942          (10.20.29.254, 10.20.48.231, https, DROP)
943          (10.20.31.2, 10.20.48.231, https, DROP)
944  (10.20.30.2, 10.20.30.254, https, ACCEPT)
945  (10.20.30.128, 10.20.30.254, https, ACCEPT)
946  (10.20.30.254, 10.20.30.254, https, ACCEPT)
947          (10.20.29.254, 10.20.30.254, https, DROP)
948          (10.20.31.2, 10.20.30.254, https, DROP)
949  (10.20.30.2, 10.20.30.45, https, ACCEPT)
950  (10.20.30.128, 10.20.30.45, https, ACCEPT)
951  (10.20.30.254, 10.20.30.45, https, ACCEPT)
952          (10.20.29.254, 10.20.30.45, https, DROP)
953          (10.20.31.2, 10.20.30.45, https, DROP)
954  (10.20.30.2, 10.20.30.26, https, ACCEPT)
955  (10.20.30.128, 10.20.30.26, https, ACCEPT)
956  (10.20.30.254, 10.20.30.26, https, ACCEPT)
957          (10.20.29.254, 10.20.30.26, https, DROP)
958          (10.20.31.2, 10.20.30.26, https, DROP)
959  (10.20.30.2, 10.20.30.16, https, ACCEPT)
960  (10.20.30.128, 10.20.30.16, https, ACCEPT)
961  (10.20.30.254, 10.20.30.16, https, ACCEPT)
962          (10.20.29.254, 10.20.30.16, https, DROP)
963          (10.20.31.2, 10.20.30.16, https, DROP)
964  (10.20.30.2, 10.20.30.11, https, ACCEPT)
965  (10.20.30.128, 10.20.30.11, https, ACCEPT)
966  (10.20.30.254, 10.20.30.11, https, ACCEPT)
967          (10.20.29.254, 10.20.30.11, https, DROP)
968          (10.20.31.2, 10.20.30.11, https, DROP)
969  (10.20.30.2, 10.20.30.2, https, ACCEPT)
970  (10.20.30.128, 10.20.30.2, https, ACCEPT)
971  (10.20.30.254, 10.20.30.2, https, ACCEPT)
972          (10.20.29.254, 10.20.30.2, https, DROP)
973          (10.20.31.2, 10.20.30.2, https, DROP)
974  (10.20.30.2, 10.20.24.231, https, ACCEPT)
975  (10.20.30.128, 10.20.24.231, https, ACCEPT)
976  (10.20.30.254, 10.20.24.231, https, ACCEPT)
977          (10.20.29.254, 10.20.24.231, https, DROP)
978          (10.20.31.2, 10.20.24.231, https, DROP)
979
980
981  (10.20.30.2, 192.168.33.2, printer, DROP)
982  (10.20.30.128, 192.168.33.2, printer, DROP)
983  (10.20.30.254, 192.168.33.2, printer, DROP)
984  (10.20.30.2, 192.168.33.128, printer, DROP)
985  (10.20.30.128, 192.168.33.128, printer, DROP)
986  (10.20.30.254, 192.168.33.128, printer, DROP)
987  (10.20.30.2, 192.168.33.254, printer, DROP)
988  (10.20.30.128, 192.168.33.254, printer, DROP)
989  (10.20.30.254, 192.168.33.254, printer, DROP)
990  (10.20.30.2, 172.20.13.2, printer, DROP)
991  (10.20.30.128, 172.20.13.2, printer, DROP)
992  (10.20.30.254, 172.20.13.2, printer, DROP)
993  (10.20.30.2, 172.20.13.128, printer, DROP)
994  (10.20.30.128, 172.20.13.128, printer, DROP)
995  (10.20.30.254, 172.20.13.128, printer, DROP)
996  (10.20.30.2, 172.20.13.254, printer, DROP)
997  (10.20.30.128, 172.20.13.254, printer, DROP)
998  (10.20.30.254, 172.20.13.254, printer, DROP)
999          (10.20.29.254, 192.168.33.2, printer, ACCEPT)
1000         (10.20.31.2, 192.168.33.2, printer, ACCEPT)
1001         (10.20.29.254, 192.168.33.128, printer, ACCEPT)
1002         (10.20.31.2, 192.168.33.128, printer, ACCEPT)
1003         (10.20.29.254, 192.168.33.254, printer, ACCEPT)
1004         (10.20.31.2, 192.168.33.254, printer, ACCEPT)
1005
```

```
(10.20.30.2, 192.168.66.128, printer, ACCEPT)
(10.20.30.128, 192.168.66.128, printer, ACCEPT)
(10.20.30.254, 192.168.66.128, printer, ACCEPT)
        (10.20.29.254, 192.168.66.128, printer, DROP)
        (10.20.31.2, 192.168.66.128, printer, DROP)
(10.20.30.2, 172.100.200.19, printer, ACCEPT)
(10.20.30.128, 172.100.200.19, printer, ACCEPT)
(10.20.30.254, 172.100.200.19, printer, ACCEPT)
        (10.20.29.254, 172.100.200.19, printer, DROP)
        (10.20.31.2, 172.100.200.19, printer, DROP)
(10.20.30.2, 172.100.100.97, printer, ACCEPT)
(10.20.30.128, 172.100.100.97, printer, ACCEPT)
(10.20.30.254, 172.100.100.97, printer, ACCEPT)
        (10.20.29.254, 172.100.100.97, printer, DROP)
        (10.20.31.2, 172.100.100.97, printer, DROP)
(10.20.30.2, 10.33.0.99, printer, ACCEPT)
(10.20.30.128, 10.33.0.99, printer, ACCEPT)
(10.20.30.254, 10.33.0.99, printer, ACCEPT)
        (10.20.29.254, 10.33.0.99, printer, DROP)
        (10.20.31.2, 10.33.0.99, printer, DROP)
(10.20.30.2, 10.20.99.128, printer, ACCEPT)
(10.20.30.128, 10.20.99.128, printer, ACCEPT)
(10.20.30.254, 10.20.99.128, printer, ACCEPT)
        (10.20.29.254, 10.20.99.128, printer, DROP)
        (10.20.31.2, 10.20.99.128, printer, DROP)
(10.20.30.2, 10.20.71.60, printer, ACCEPT)
(10.20.30.128, 10.20.71.60, printer, ACCEPT)
(10.20.30.254, 10.20.71.60, printer, ACCEPT)
        (10.20.29.254, 10.20.71.60, printer, DROP)
        (10.20.31.2, 10.20.71.60, printer, DROP)
(10.20.30.2, 10.20.60.231, printer, ACCEPT)
(10.20.30.128, 10.20.60.231, printer, ACCEPT)
(10.20.30.254, 10.20.60.231, printer, ACCEPT)
        (10.20.29.254, 10.20.60.231, printer, DROP)
        (10.20.31.2, 10.20.60.231, printer, DROP)
(10.20.30.2, 10.20.51.128, printer, ACCEPT)
(10.20.30.128, 10.20.51.128, printer, ACCEPT)
(10.20.30.254, 10.20.51.128, printer, ACCEPT)
        (10.20.29.254, 10.20.51.128, printer, DROP)
        (10.20.31.2, 10.20.51.128, printer, DROP)
(10.20.30.2, 10.20.48.231, printer, ACCEPT)
(10.20.30.128, 10.20.48.231, printer, ACCEPT)
(10.20.30.254, 10.20.48.231, printer, ACCEPT)
        (10.20.29.254, 10.20.48.231, printer, DROP)
        (10.20.31.2, 10.20.48.231, printer, DROP)
(10.20.30.2, 10.20.30.254, printer, ACCEPT)
(10.20.30.128, 10.20.30.254, printer, ACCEPT)
(10.20.30.254, 10.20.30.254, printer, ACCEPT)
        (10.20.29.254, 10.20.30.254, printer, DROP)
        (10.20.31.2, 10.20.30.254, printer, DROP)
(10.20.30.2, 10.20.30.45, printer, ACCEPT)
(10.20.30.128, 10.20.30.45, printer, ACCEPT)
(10.20.30.254, 10.20.30.45, printer, ACCEPT)
        (10.20.29.254, 10.20.30.45, printer, DROP)
        (10.20.31.2, 10.20.30.45, printer, DROP)
(10.20.30.2, 10.20.30.26, printer, ACCEPT)
(10.20.30.128, 10.20.30.26, printer, ACCEPT)
(10.20.30.254, 10.20.30.26, printer, ACCEPT)
        (10.20.29.254, 10.20.30.26, printer, DROP)
        (10.20.31.2, 10.20.30.26, printer, DROP)
(10.20.30.2, 10.20.30.16, printer, ACCEPT)
(10.20.30.128, 10.20.30.16, printer, ACCEPT)
(10.20.30.254, 10.20.30.16, printer, ACCEPT)
        (10.20.29.254, 10.20.30.16, printer, DROP)
```

```
1070          (10.20.31.2, 10.20.30.16, printer, DROP)
1071  (10.20.30.2, 10.20.30.11, printer, ACCEPT)
1072  (10.20.30.128, 10.20.30.11, printer, ACCEPT)
1073  (10.20.30.254, 10.20.30.11, printer, ACCEPT)
1074          (10.20.29.254, 10.20.30.11, printer, DROP)
1075          (10.20.31.2, 10.20.30.11, printer, DROP)
1076  (10.20.30.2, 10.20.30.2, printer, ACCEPT)
1077  (10.20.30.128, 10.20.30.2, printer, ACCEPT)
1078  (10.20.30.254, 10.20.30.2, printer, ACCEPT)
1079          (10.20.29.254, 10.20.30.2, printer, DROP)
1080          (10.20.31.2, 10.20.30.2, printer, DROP)
1081  (10.20.30.2, 10.20.24.231, printer, ACCEPT)
1082  (10.20.30.128, 10.20.24.231, printer, ACCEPT)
1083  (10.20.30.254, 10.20.24.231, printer, ACCEPT)
1084          (10.20.29.254, 10.20.24.231, printer, DROP)
1085          (10.20.31.2, 10.20.24.231, printer, DROP)
1086
1087
1088  (10.20.30.2, 192.168.33.2, icmp, DROP)
1089  (10.20.30.128, 192.168.33.2, icmp, DROP)
1090  (10.20.30.254, 192.168.33.2, icmp, DROP)
1091  (10.20.30.2, 192.168.33.128, icmp, DROP)
1092  (10.20.30.128, 192.168.33.128, icmp, DROP)
1093  (10.20.30.254, 192.168.33.128, icmp, DROP)
1094  (10.20.30.2, 192.168.33.254, icmp, DROP)
1095  (10.20.30.128, 192.168.33.254, icmp, DROP)
1096  (10.20.30.254, 192.168.33.254, icmp, DROP)
1097  (10.20.30.2, 172.20.13.2, icmp, DROP)
1098  (10.20.30.128, 172.20.13.2, icmp, DROP)
1099  (10.20.30.254, 172.20.13.2, icmp, DROP)
1100  (10.20.30.2, 172.20.13.128, icmp, DROP)
1101  (10.20.30.128, 172.20.13.128, icmp, DROP)
1102  (10.20.30.254, 172.20.13.128, icmp, DROP)
1103  (10.20.30.2, 172.20.13.254, icmp, DROP)
1104  (10.20.30.128, 172.20.13.254, icmp, DROP)
1105  (10.20.30.254, 172.20.13.254, icmp, DROP)
1106          (10.20.29.254, 192.168.33.2, icmp, ACCEPT)
1107          (10.20.31.2, 192.168.33.2, icmp, ACCEPT)
1108          (10.20.29.254, 192.168.33.128, icmp, ACCEPT)
1109          (10.20.31.2, 192.168.33.128, icmp, ACCEPT)
1110          (10.20.29.254, 192.168.33.254, icmp, ACCEPT)
1111          (10.20.31.2, 192.168.33.254, icmp, ACCEPT)
1112
1113  (10.20.30.2, 192.168.66.128, icmp, ACCEPT)
1114  (10.20.30.128, 192.168.66.128, icmp, ACCEPT)
1115  (10.20.30.254, 192.168.66.128, icmp, ACCEPT)
1116          (10.20.29.254, 192.168.66.128, icmp, DROP)
1117          (10.20.31.2, 192.168.66.128, icmp, DROP)
1118  (10.20.30.2, 172.100.200.19, icmp, ACCEPT)
1119  (10.20.30.128, 172.100.200.19, icmp, ACCEPT)
1120  (10.20.30.254, 172.100.200.19, icmp, ACCEPT)
1121          (10.20.29.254, 172.100.200.19, icmp, DROP)
1122          (10.20.31.2, 172.100.200.19, icmp, DROP)
1123  (10.20.30.2, 172.100.100.97, icmp, ACCEPT)
1124  (10.20.30.128, 172.100.100.97, icmp, ACCEPT)
1125  (10.20.30.254, 172.100.100.97, icmp, ACCEPT)
1126          (10.20.29.254, 172.100.100.97, icmp, DROP)
1127          (10.20.31.2, 172.100.100.97, icmp, DROP)
1128  (10.20.30.2, 10.33.0.99, icmp, ACCEPT)
1129  (10.20.30.128, 10.33.0.99, icmp, ACCEPT)
1130  (10.20.30.254, 10.33.0.99, icmp, ACCEPT)
1131          (10.20.29.254, 10.33.0.99, icmp, DROP)
1132          (10.20.31.2, 10.33.0.99, icmp, DROP)
1133  (10.20.30.2, 10.20.99.128, icmp, ACCEPT)
```

```
1134 (10.20.30.128, 10.20.99.128, icmp, ACCEPT)
1135 (10.20.30.254, 10.20.99.128, icmp, ACCEPT)
1136         (10.20.29.254, 10.20.99.128, icmp, DROP)
1137         (10.20.31.2, 10.20.99.128, icmp, DROP)
1138 (10.20.30.2, 10.20.71.60, icmp, ACCEPT)
1139 (10.20.30.128, 10.20.71.60, icmp, ACCEPT)
1140 (10.20.30.254, 10.20.71.60, icmp, ACCEPT)
1141         (10.20.29.254, 10.20.71.60, icmp, DROP)
1142         (10.20.31.2, 10.20.71.60, icmp, DROP)
1143 (10.20.30.2, 10.20.60.231, icmp, ACCEPT)
1144 (10.20.30.128, 10.20.60.231, icmp, ACCEPT)
1145 (10.20.30.254, 10.20.60.231, icmp, ACCEPT)
1146         (10.20.29.254, 10.20.60.231, icmp, DROP)
1147         (10.20.31.2, 10.20.60.231, icmp, DROP)
1148 (10.20.30.2, 10.20.51.128, icmp, ACCEPT)
1149 (10.20.30.128, 10.20.51.128, icmp, ACCEPT)
1150 (10.20.30.254, 10.20.51.128, icmp, ACCEPT)
1151         (10.20.29.254, 10.20.51.128, icmp, DROP)
1152         (10.20.31.2, 10.20.51.128, icmp, DROP)
1153 (10.20.30.2, 10.20.48.231, icmp, ACCEPT)
1154 (10.20.30.128, 10.20.48.231, icmp, ACCEPT)
1155 (10.20.30.254, 10.20.48.231, icmp, ACCEPT)
1156         (10.20.29.254, 10.20.48.231, icmp, DROP)
1157         (10.20.31.2, 10.20.48.231, icmp, DROP)
1158 (10.20.30.2, 10.20.30.254, icmp, ACCEPT)
1159 (10.20.30.128, 10.20.30.254, icmp, ACCEPT)
1160 (10.20.30.254, 10.20.30.254, icmp, ACCEPT)
1161         (10.20.29.254, 10.20.30.254, icmp, DROP)
1162         (10.20.31.2, 10.20.30.254, icmp, DROP)
1163 (10.20.30.2, 10.20.30.45, icmp, ACCEPT)
1164 (10.20.30.128, 10.20.30.45, icmp, ACCEPT)
1165 (10.20.30.254, 10.20.30.45, icmp, ACCEPT)
1166         (10.20.29.254, 10.20.30.45, icmp, DROP)
1167         (10.20.31.2, 10.20.30.45, icmp, DROP)
1168 (10.20.30.2, 10.20.30.26, icmp, ACCEPT)
1169 (10.20.30.128, 10.20.30.26, icmp, ACCEPT)
1170 (10.20.30.254, 10.20.30.26, icmp, ACCEPT)
1171         (10.20.29.254, 10.20.30.26, icmp, DROP)
1172         (10.20.31.2, 10.20.30.26, icmp, DROP)
1173 (10.20.30.2, 10.20.30.16, icmp, ACCEPT)
1174 (10.20.30.128, 10.20.30.16, icmp, ACCEPT)
1175 (10.20.30.254, 10.20.30.16, icmp, ACCEPT)
1176         (10.20.29.254, 10.20.30.16, icmp, DROP)
1177         (10.20.31.2, 10.20.30.16, icmp, DROP)
1178 (10.20.30.2, 10.20.30.11, icmp, ACCEPT)
1179 (10.20.30.128, 10.20.30.11, icmp, ACCEPT)
1180 (10.20.30.254, 10.20.30.11, icmp, ACCEPT)
1181         (10.20.29.254, 10.20.30.11, icmp, DROP)
1182         (10.20.31.2, 10.20.30.11, icmp, DROP)
1183 (10.20.30.2, 10.20.30.2, icmp, ACCEPT)
1184 (10.20.30.128, 10.20.30.2, icmp, ACCEPT)
1185 (10.20.30.254, 10.20.30.2, icmp, ACCEPT)
1186         (10.20.29.254, 10.20.30.2, icmp, DROP)
1187         (10.20.31.2, 10.20.30.2, icmp, DROP)
1188 (10.20.30.2, 10.20.24.231, icmp, ACCEPT)
1189 (10.20.30.128, 10.20.24.231, icmp, ACCEPT)
1190 (10.20.30.254, 10.20.24.231, icmp, ACCEPT)
1191         (10.20.29.254, 10.20.24.231, icmp, DROP)
1192         (10.20.31.2, 10.20.24.231, icmp, DROP)
1193
1194
1195 (10.20.30.2, 192.168.33.2, telnet, DROP)
1196 (10.20.30.128, 192.168.33.2, telnet, DROP)
1197 (10.20.30.254, 192.168.33.2, telnet, DROP)
```

```
1198 (10.20.30.2, 192.168.33.128, telnet, DROP)
1199 (10.20.30.128, 192.168.33.128, telnet, DROP)
1200 (10.20.30.254, 192.168.33.128, telnet, DROP)
1201 (10.20.30.2, 192.168.33.254, telnet, DROP)
1202 (10.20.30.128, 192.168.33.254, telnet, DROP)
1203 (10.20.30.254, 192.168.33.254, telnet, DROP)
1204 (10.20.30.2, 172.20.13.2, telnet, DROP)
1205 (10.20.30.128, 172.20.13.2, telnet, DROP)
1206 (10.20.30.254, 172.20.13.2, telnet, DROP)
1207 (10.20.30.2, 172.20.13.128, telnet, DROP)
1208 (10.20.30.128, 172.20.13.128, telnet, DROP)
1209 (10.20.30.254, 172.20.13.128, telnet, DROP)
1210 (10.20.30.2, 172.20.13.254, telnet, DROP)
1211 (10.20.30.128, 172.20.13.254, telnet, DROP)
1212 (10.20.30.254, 172.20.13.254, telnet, DROP)
1213         (10.20.29.254, 192.168.33.2, telnet, ACCEPT)
1214         (10.20.31.2, 192.168.33.2, telnet, ACCEPT)
1215         (10.20.29.254, 192.168.33.128, telnet, ACCEPT)
1216         (10.20.31.2, 192.168.33.128, telnet, ACCEPT)
1217         (10.20.29.254, 192.168.33.254, telnet, ACCEPT)
1218         (10.20.31.2, 192.168.33.254, telnet, ACCEPT)
1219
1220 (10.20.30.2, 192.168.66.128, telnet, ACCEPT)
1221 (10.20.30.128, 192.168.66.128, telnet, ACCEPT)
1222 (10.20.30.254, 192.168.66.128, telnet, ACCEPT)
1223         (10.20.29.254, 192.168.66.128, telnet, DROP)
1224         (10.20.31.2, 192.168.66.128, telnet, DROP)
1225 (10.20.30.2, 172.100.200.19, telnet, ACCEPT)
1226 (10.20.30.128, 172.100.200.19, telnet, ACCEPT)
1227 (10.20.30.254, 172.100.200.19, telnet, ACCEPT)
1228         (10.20.29.254, 172.100.200.19, telnet, DROP)
1229         (10.20.31.2, 172.100.200.19, telnet, DROP)
1230 (10.20.30.2, 172.100.100.97, telnet, ACCEPT)
1231 (10.20.30.128, 172.100.100.97, telnet, ACCEPT)
1232 (10.20.30.254, 172.100.100.97, telnet, ACCEPT)
1233         (10.20.29.254, 172.100.100.97, telnet, DROP)
1234         (10.20.31.2, 172.100.100.97, telnet, DROP)
1235 (10.20.30.2, 10.33.0.99, telnet, ACCEPT)
1236 (10.20.30.128, 10.33.0.99, telnet, ACCEPT)
1237 (10.20.30.254, 10.33.0.99, telnet, ACCEPT)
1238         (10.20.29.254, 10.33.0.99, telnet, DROP)
1239         (10.20.31.2, 10.33.0.99, telnet, DROP)
1240 (10.20.30.2, 10.20.99.128, telnet, ACCEPT)
1241 (10.20.30.128, 10.20.99.128, telnet, ACCEPT)
1242 (10.20.30.254, 10.20.99.128, telnet, ACCEPT)
1243         (10.20.29.254, 10.20.99.128, telnet, DROP)
1244         (10.20.31.2, 10.20.99.128, telnet, DROP)
1245 (10.20.30.2, 10.20.71.60, telnet, ACCEPT)
1246 (10.20.30.128, 10.20.71.60, telnet, ACCEPT)
1247 (10.20.30.254, 10.20.71.60, telnet, ACCEPT)
1248         (10.20.29.254, 10.20.71.60, telnet, DROP)
1249         (10.20.31.2, 10.20.71.60, telnet, DROP)
1250 (10.20.30.2, 10.20.60.231, telnet, ACCEPT)
1251 (10.20.30.128, 10.20.60.231, telnet, ACCEPT)
1252 (10.20.30.254, 10.20.60.231, telnet, ACCEPT)
1253         (10.20.29.254, 10.20.60.231, telnet, DROP)
1254         (10.20.31.2, 10.20.60.231, telnet, DROP)
1255 (10.20.30.2, 10.20.51.128, telnet, ACCEPT)
1256 (10.20.30.128, 10.20.51.128, telnet, ACCEPT)
1257 (10.20.30.254, 10.20.51.128, telnet, ACCEPT)
1258         (10.20.29.254, 10.20.51.128, telnet, DROP)
1259         (10.20.31.2, 10.20.51.128, telnet, DROP)
1260 (10.20.30.2, 10.20.48.231, telnet, ACCEPT)
1261 (10.20.30.128, 10.20.48.231, telnet, ACCEPT)
```

```
1262 (10.20.30.254, 10.20.48.231, telnet, ACCEPT)
1263         (10.20.29.254, 10.20.48.231, telnet, DROP)
1264         (10.20.31.2, 10.20.48.231, telnet, DROP)
1265 (10.20.30.2, 10.20.30.254, telnet, ACCEPT)
1266 (10.20.30.128, 10.20.30.254, telnet, ACCEPT)
1267 (10.20.30.254, 10.20.30.254, telnet, ACCEPT)
1268         (10.20.29.254, 10.20.30.254, telnet, DROP)
1269         (10.20.31.2, 10.20.30.254, telnet, DROP)
1270 (10.20.30.2, 10.20.30.45, telnet, ACCEPT)
1271 (10.20.30.128, 10.20.30.45, telnet, ACCEPT)
1272 (10.20.30.254, 10.20.30.45, telnet, ACCEPT)
1273         (10.20.29.254, 10.20.30.45, telnet, DROP)
1274         (10.20.31.2, 10.20.30.45, telnet, DROP)
1275 (10.20.30.2, 10.20.30.26, telnet, ACCEPT)
1276 (10.20.30.128, 10.20.30.26, telnet, ACCEPT)
1277 (10.20.30.254, 10.20.30.26, telnet, ACCEPT)
1278         (10.20.29.254, 10.20.30.26, telnet, DROP)
1279         (10.20.31.2, 10.20.30.26, telnet, DROP)
1280 (10.20.30.2, 10.20.30.16, telnet, ACCEPT)
1281 (10.20.30.128, 10.20.30.16, telnet, ACCEPT)
1282 (10.20.30.254, 10.20.30.16, telnet, ACCEPT)
1283         (10.20.29.254, 10.20.30.16, telnet, DROP)
1284         (10.20.31.2, 10.20.30.16, telnet, DROP)
1285 (10.20.30.2, 10.20.30.11, telnet, ACCEPT)
1286 (10.20.30.128, 10.20.30.11, telnet, ACCEPT)
1287 (10.20.30.254, 10.20.30.11, telnet, ACCEPT)
1288         (10.20.29.254, 10.20.30.11, telnet, DROP)
1289         (10.20.31.2, 10.20.30.11, telnet, DROP)
1290 (10.20.30.2, 10.20.30.2, telnet, ACCEPT)
1291 (10.20.30.128, 10.20.30.2, telnet, ACCEPT)
1292 (10.20.30.254, 10.20.30.2, telnet, ACCEPT)
1293         (10.20.29.254, 10.20.30.2, telnet, DROP)
1294         (10.20.31.2, 10.20.30.2, telnet, DROP)
1295 (10.20.30.2, 10.20.24.231, telnet, ACCEPT)
1296 (10.20.30.128, 10.20.24.231, telnet, ACCEPT)
1297 (10.20.30.254, 10.20.24.231, telnet, ACCEPT)
1298         (10.20.29.254, 10.20.24.231, telnet, DROP)
1299         (10.20.31.2, 10.20.24.231, telnet, DROP)
1300
1301
1302
1303 /*
1304 ADMIN           ->      FW              ACCEPT *
1305 */
```

Appendix B

Haskell Code for End to Mid

```
1  ------------------------------------------------------------------------------
2  -- Conversion of an endpoint automaton to a midpoint automaton
3  -- Diana von Bidder, 2006
4  -- $Id: End-Mid.hs 32815 2006-11-30 12:24:03Z dsenn $
5  ------------------------------------------------------------------------------
6
7  ------------------------------------------------------------------------------
8  -- example usage:
9  ------------------------------------------------------------------------------
10 -- EndMid> deltaM [(CLOSED, LISTEN, ([],[],[],[]))] ("E0", "M", "S")
11 -- [(SYNSENT,LISTEN,([],[],[("S",1)],[]))]
12 -- EndMid> lambdaM [(CLOSED, LISTEN, ([],[],[],[]))] ("E0", "M", "S")
13 -- "S"
14 -- EndMid>
15 ------------------------------------------------------------------------------
16 module EndMid where
17 import MultiSet
18
19
20
21 ------------------------------------------------------------------------------
22 -- Mealy machines
23 ------------------------------------------------------------------------------
24 -- a Mealy machine
25 data Nfa a = NFA (Bag a)
26                  (Bag (Move a))
27                  a
28                  (Bag a)
29               deriving (Eq, Show)
30
31 -- lambda & delta: start-state input output end-state
32 data Move a = MOVE a String String a
33               deriving (Eq, Ord, Show)
34
35
36
```

```
37 ------------------------------------------------------------------------------
38 -- End - Mid
39 ------------------------------------------------------------------------------
40 -- q \in Q_M, net_M = (M -> E0, E0 -> M, M -> E1, E1 -> M)
41 type Mtr a = (a, a, (Bag String, Bag String, Bag String, Bag String))
42
43 compare :: Ord a => Mtr a -> Mtr a -> Ordering
44 compare (a, b, (c, d, e, f)) (g, h, (i, j, k, l))
45       | (a == g) && (b == h) && (c == i) && (d == j) && (e == k)
46         && (f == l)   =  EQ
47       | otherwise   =  Prelude.compare a g
48
49 mequal (a, b, (c, d, e, f)) (g, h, (i, j, k, l)) = (a == g) && (b == h)
50         && (toList c == toList i) && (toList d == toList j)
51         && (toList e == toList k) && (toList f == toList l)
52
53
54 -- succ(q_M) as defined in my thesis
55 succS1 :: [Mtr TCPstate] -> [Mtr TCPstate]
56 succS1 q = list_to_set (foldl (++) [] [succS2 t | t <- q])
57
58 succS2 :: Mtr TCPstate -> [Mtr TCPstate]
59 succS2 (q0, q1, (net1, net2, net3, net4)) =
60     [(q0, q1, (net1, net2, net3, net4))]
61     ++ [(q0, q1, (delete msg net1, net2, net3, net4)) | (msg,_) <- net1]
62     ++ [(q0, q1, (net1, delete msg net2, net3, net4)) | (msg,_) <- net2]
63     ++ [(q0, q1, (net1, net2, delete msg net3, net4)) | (msg,_) <- net3]
64     ++ [(q0, q1, (net1, net2, net3, delete msg net4)) | (msg,_) <- net4]
65     ++ [(delta0 q0 msg, q1, (delete msg net1, insert2 (lambda0 q0 msg) net2,
66           net3, net4)) | (msg, _) <- net1]
67     ++ [(delta0 q0 "-", q1, (net1, insert2 (lambda0 q0 "-") net2, net3, net4))]
68     ++ [(q0, delta1 q1 msg, (net1, net2, delete msg net3,
69           insert2 (lambda1 q1 msg) net4)) | (msg, _) <- net3]
70     ++ [(q0, delta1 q1 "-", (net1, net2, net3, insert2 (lambda1 q1 "-") net4))]
71
72
73 -- closure as defined in my thesis
74 cl f x = if (setequal (f x) (x)) then x else (cl f (f x))
75
76
77 -- "-" should not be sent
78 insert2 m net
79     | m == "-"   =  net
80     | otherwise  =  insert m net
81
82
83
84 -- deltaM as defined in my thesis,  ("E0", "M", m) == E0 -> M: m
85 deltaM q ("E0", "M", m) = [(q0, q1, (net1, delete msg net2,
86     insert2 (lambdaM q ("E0", "M", msg)) net3, net4)) | (q0, q1, (net1, net2,
87     net3, net4)) <- (cl succS1 q), (msg, _) <- net2, msg == m]
```

```
88  deltaM q ("E1", "M", m) = [(q0, q1, (insert2 (lambdaM q ("E1", "M", m)) net1,
89      net2, net3, delete m net4)) | (q0, q1, (net1, net2, net3, net4)) <- (cl
90      succS1 q), (msg, _) <- net4, msg == m]
91
92
93  -- lambdaM as defined in my thesis
94  lambdaM q (e, "M", m) = my_out [out (q0, q1, (net1, net2, net3, net4)) (e, m)
95      | (q0, q1, (net1, net2, net3, net4)) <- (cl succS1 q)]
96
97
98  -- out as defined in my thesis
99  out (q0, q1, (net1, net2, net3, net4)) ("E0", m) = if (element m net2)
100                                                     then m else "-"
101 out (q0, q1, (net1, net2, net3, net4)) ("E1", m) = if (element m net4)
102                                                     then m else "-"
103
104 -- the first non-"-" element of a list
105 my_out [] = "-"
106 my_out (x:xs)
107     | x == "-"  = my_out xs
108     | otherwise = x
109
110
111
112 ------------------------------------------------------------------------------
113 -- set- and list-specific extras
114 ------------------------------------------------------------------------------
115 -- convert a list to a set (every element only once)
116 list_to_set []     = []
117 list_to_set (x:xs) = if ([m | m <- xs, mequal x m] == [])
118                      then x:(list_to_set xs)
119                      else list_to_set xs
120
121 -- delete y from (x:xs)
122 mtrminus [] _ = []
123 mtrminus (x:xs) y
124         | x == y     =  xs
125         | otherwise  =  x:(mtrminus xs y)
126
127
128 -- compare two sets (midpoint states) for equality
129 setequal [] [] = True
130 setequal [] _ = False
131 setequal _ [] = False
132 setequal (x:xs) (y:ys)
133     | mequal x y = setequal xs ys
134     | otherwise = if (temp == [])
135                   then False
136                   else setequal xs (y:(mtrminus ys (head temp)))
137         where temp = [m | m <- ys, mequal x m]
138
```

```
139
140 -- head does not work for empty lists...
141 my_head :: [String] -> String
142 my_head [] = "-"
143 my_head (x:_) = x
144
145
146
147 ------------------------------------------------------------------------------
148 -- TCP Specification for Endpoints
149 ------------------------------------------------------------------------------
150 -- states
151 -- only one run => CLOSED is the start-state, CLOSED2 is the endstate
152 data TCPstate = CLOSED | CLOSED2 | LISTEN | SYNSENT | SYNRCVD | ESTAB
153               | FINWAIT1 | FINWAIT2 | CLOSING | CLOSEWAIT | LASTACK | TIMEWAIT
154               deriving (Enum, Ord, Eq, Show)
155
156
157 -- transitions
158 -- "A1" is the ACK of a SYN, "A2" is the ACK of a FIN
159 -- no symultaneous open => LISTEN --> SYN_SENT is missing
160 tcp :: Nfa TCPstate
161 tcp = NFA
162     (fromList [CLOSED .. TIMEWAIT])
163     (fromList [(MOVE CLOSED "-" "S" SYNSENT),
164                (MOVE CLOSED "-" "-" LISTEN),
165                (MOVE LISTEN "S" "SA" SYNRCVD),
166                (MOVE SYNSENT "S" "A1" SYNRCVD),
167                (MOVE SYNSENT "SA" "A1" ESTAB),
168                (MOVE SYNRCVD "A1" "-" ESTAB),
169                (MOVE SYNRCVD "-" "F" FINWAIT1),
170                (MOVE ESTAB "-" "F" FINWAIT1),
171                (MOVE ESTAB "F" "A2" CLOSEWAIT),
172                (MOVE FINWAIT1 "F" "A2" CLOSING),
173                (MOVE FINWAIT1 "A2" "-" FINWAIT2),
174                (MOVE CLOSEWAIT "-" "F" LASTACK),
175                (MOVE CLOSING "A2" "-" TIMEWAIT),
176                (MOVE FINWAIT2 "F" "A2" TIMEWAIT),
177                (MOVE LASTACK "A2" "-" CLOSED2),
178                (MOVE TIMEWAIT "-" "-" CLOSED2)])
179     CLOSED
180     (fromList [CLOSED2])
181
182
183 -- delta and lambda of E0 and E1
184 delta0 q m = if (temp == []) then q else head temp
185         where temp = [qend | NFA states moves start term <- [tcp],
186                              (MOVE r n _ qend, _) <- moves, r == q, n == m]
187
188 lambda0 q m = my_head [out | (NFA states moves start term) <- [tcp],
189                              (MOVE r n out _, _) <- moves, r == q, n == m]
```

```
190
191  delta1 = delta0
192  lambda1 = lambda0
```

Appendix C

Abstract Test Cases for TCP

The following represent abstract test cases for the TCP Midpoint automaton in Figure 10.1. They were generated with the help of the TCGTool (see section 5.1). Note that every line represents a test case, where every (input/expected output)-tuple represents a test packet.

1 (2/2)(5/5)(6/6)(8/8)(7/7)(11/−)(7/−)
2 (2/2)(9/9)(5/5)(6/6)(6/6)(2/−)(9/−)
3 (2/2)(9/9)(5/5)(6/6)(1a/−)(7/−)
4 (6/−)(9/−)
5 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/−)(6/−)
6 (2/2)(6/−)(9/9)
7 (2/2)(5/5)(6/6)(8/8)(6/−)
8 (2/2)(9/9)(7/−)(6/−)
9 (2/2)(9/9)(5/5)(8/8)(7/7)(9/−)(9/−)
10 (2/2)(5/5)(6/6)(8/8)(4/−)(7/7)
11 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(7/−)(6/6)(6/−)
12 (2/2)(5/5)(8/8)(7/7)(6/6)(8/−)
13 (2/2)(5/5)(6/6)(8/8)(8/−)(9/9)
14 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(5/−)(8/−)
15 (2/2)(5/5)(8/8)(10/−)(8/−)
16 (2/2)(9/9)(5/5)(8/8)(2/−)(7/7)
17 (2/2)(5/5)(5/−)(8/8)
18 (2/2)(5/5)(8/8)(11/−)(7/7)
19 (2/2)(5/5)(8/8)(7/7)(6/6)(6/−)
20 (2/2)(5/5)(8/8)(7/7)(1b/−)(8/−)
21 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)
22 (2/2)(9/9)(5/5)(6/6)(6/6)(4/−)(9/−)
23 (2/2)(9/9)(5/5)(3/−)(8/8)
24 (2/2)(9/9)(5/5)(8/8)(6/−)(7/7)(9/−)
25 (2/2)(5/5)(6/6)(9/9)
26 (2/2)(9/9)(5/5)(8/8)(4/−)(6/−)
27 (2/2)(5/5)(6/6)(7/−)(8/8)
28 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(4/−)(9/−)
29 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(8/−)(7/7)
30 (2/2)(5/5)(6/6)(8/8)(7/7)(1b/−)(7/−)
31 (2/2)(9/9)(5/5)(8/8)(7/7)(8/−)(8/−)

32 (2/2)(9/9)(5/5)(8/8)(6/−)(1a/−)(5/−)
33 (2/2)(5/5)(8/8)(7/7)(6/6)(5/−)
34 (1a/−)(9/−)
35 (2/2)(5/5)(6/6)(8/8)(7/7)(2/−)(6/−)
36 (3/−)(7/−)
37 (2/2)(5/5)(6/6)(8/8)(3/−)(7/7)
38 (2/2)(9/9)(5/5)(8/8)(1a/−)(7/−)
39 (6/−)(6/−)
40 (2/2)(5/5)(6/6)(8/8)(6/−)(9/9)
41 (2/2)(5/5)(8/8)(9/9)(9/−)
42 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)(6/−)
43 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/−)(5/−)
44 (2/2)(5/5)(4/−)(6/6)
45 (2/2)(5/5)(6/6)(2/−)(8/8)
46 (6/−)(8/−)
47 (2/2)(9/9)(1a/−)(6/−)
48 (2/2)(9/9)(2/−)(9/−)
49 (2/2)(9/9)(5/5)(6/6)(6/6)(3/−)(8/8)
50 (2/2)(11/−)(6/−)
51 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)(9/9)
52 (2/2)(5/5)(8/8)(6/6)(6/−)
53 (2/2)(5/5)(8/8)(7/7)(4/−)(8/−)
54 (2/2)(3/−)(9/9)
55 (2/2)(8/−)(8/−)
56 (2/2)(9/9)(5/5)(11/−)(8/8)
57 (2/2)(5/5)(6/6)(8/8)(1a/−)(7/−)
58 (2/2)(9/9)(5/5)(8/8)(7/7)(5/−)(9/−)
59 (2/2)(5/5)(6/6)(8/8)(7/7)(10/−)(6/−)
60 (9/−)(9/−)
61 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(2/−)(7/7)
62 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(11/−)(9/−)
63 (2/2)(9/9)(5/5)(8/8)(5/−)(7/7)
64 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)(9/9)
65 (2/2)(5/5)(8/8)(7/7)(6/6)(7/−)
66 (2/2)(9/9)(5/5)(8/8)(7/7)(4/−)(9/−)
67 (7/−)(7/−)
68 (2/2)(5/5)(6/6)(9/9)(9/−)
69 (2/2)(9/9)(5/5)(1a/−)(8/−)
70 (2/2)(9/9)(5/5)(6/6)(9/−)
71 (2/2)(4/−)(7/−)
72 (2/2)(5/5)(8/8)(1a/−)(8/−)
73 (2/2)(5/5)(6/6)(7/−)(6/−)
74 (2/2)(9/9)(5/5)(6/6)(3/−)(8/8)
75 (2/2)(9/9)(5/5)(1b/−)(6/−)
76 (2/2)(5/5)(6/6)(8/8)(9/9)(7/7)
77 (2/2)(5/5)(8/8)(7/7)(1a/−)(6/−)
78 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(5/−)
79 (2/2)(9/9)(5/5)(8/8)(7/7)(1b/−)(7/−)
80 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(2/−)(6/6)(6/−)
81 (7/−)(6/−)
82 (2/2)(9/9)(5/5)(8/8)(6/−)(1a/−)(9/−)

83 (2/2)(9/9)(7/−)(5/5)
84 (2/2)(5/5)(6/6)(5/−)
85 (2/2)(9/9)(7/−)(8/−)
86 (2/2)(5/5)(6/6)(8/8)(7/7)(2/−)(5/−)
87 (2/2)(9/9)(5/5)(8/8)(6/−)(5/−)(7/7)
88 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(8/−)(1a/−)
89 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(10/−)(9/−)
90 (10/−)(6/−)
91 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)(6/−)
92 (2/2)(5/5)(8/8)(2/−)(8/−)
93 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(11/−)(7/−)
94 (2/2)(5/5)(6/6)(5/−)(8/8)
95 (2/2)(5/5)(8/8)(1a/−)(5/−)
96 (2/2)(9/9)(4/−)(8/−)
97 (2/2)(5/5)(1a/−)
98 (2/2)(5/5)(6/6)(8/8)(7/7)(3/−)(6/−)
99 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)(5/−)
100 (2/2)(9/9)(5/5)(6/6)(1b/−)(7/−)
101 (2/2)(9/9)(9/−)(7/−)
102 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(6/−)(9/−)
103 (2/2)(5/5)(6/6)(8/8)(5/−)(6/−)
104 (2/2)(5/5)(8/8)(7/7)(7/−)(6/6)
105 (2/2)(6/−)(6/−)
106 (2/2)(9/9)(5/5)(6/6)(1a/−)(9/−)
107 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(11/−)(6/6)(6/−)
108 (7/−)(5/−)
109 (2/2)(5/5)(2/−)(8/8)
110 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(5/−)(1a/−)
111 (2/2)(9/9)(8/−)(6/−)
112 (2/2)(5/5)(6/6)(5/−)(6/−)
113 (2/2)(5/5)(6/6)(8/8)(7/7)(2/−)(7/−)
114 (11/−)(8/−)
115 (2/2)(9/9)(1b/−)(9/−)
116 (2/2)(5/5)(8/8)(10/−)(7/7)
117 (2/2)(9/9)(5/5)(6/6)(5/−)(8/8)
118 (2/2)(5/5)(4/−)(8/8)
119 (2/2)(9/9)(5/5)(8/8)(4/−)(1a/−)
120 (2/2)(5/5)(8/8)(7/7)(5/−)(9/9)
121 (9/−)
122 (2/2)(9/9)(5/5)(6/6)(6/6)(9/−)(8/8)
123 (2/2)(5/5)(2/−)(6/6)
124 (2/2)(9/9)(5/5)(8/8)(7/7)(10/−)(7/−)
125 (2/2)(5/5)(9/9)(8/8)
126 (2/2)(9/9)(2/−)(8/−)
127 (2/2)(9/9)(5/5)(8/8)(6/−)(8/−)(7/7)
128 (2/2)(5/5)(6/6)(8/8)(3/−)(6/−)
129 (2/2)(9/9)(5/5)(6/6)(8/8)(8/−)
130 (2/2)(5/5)(6/6)(11/−)(7/−)
131 (2/2)(9/9)(5/5)(8/8)(7/7)(11/−)(8/−)
132 (1b/−)(6/−)
133 (2/2)(5/5)(6/6)(8/8)(7/7)(6/−)(7/−)

134 (2/2)(9/9)(5/5)(6/6)(6/6)(6/−)(8/8)
135 (2/2)(9/9)(5/5)(8/8)(6/−)(10/−)(6/6)
136 (2/2)(9/9)(5/5)(8/8)(5/−)(9/−)
137 (2/2)(9/9)(5/5)(8/8)(6/−)(2/−)(6/6)
138 (2/2)(9/9)(10/−)(6/−)
139 (2/2)(9/9)(4/−)(5/5)
140 (2/2)(9/9)(2/−)(5/5)
141 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)(7/−)
142 (2/2)(9/9)(5/5)(8/8)(6/−)(5/−)
143 (11/−)(5/−)
144 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/−)(9/−)
145 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)(8/−)
146 (7/−)(9/−)
147 (2/2)(9/9)(11/−)(5/5)
148 (2/2)(5/5)(8/8)(8/−)(6/6)
149 (2/2)(5/5)(8/8)(7/7)(5/−)
150 (2/2)(5/5)(6/6)(8/8)(7/7)(11/−)(8/−)
151 (2/2)(9/9)(5/5)(5/−)(6/6)(6/6)
152 (2/2)(9/9)(5/5)(1b/−)(8/−)
153 (2/2)(9/9)(10/−)(7/−)
154 (2/2)(9/9)(5/5)(8/8)(7/7)(5/−)(8/−)
155 (2/2)(5/5)(1a/−)(5/−)
156 (2/2)(5/5)(8/8)(5/−)(6/6)
157 (2/2)(9/9)(5/5)(10/−)(8/8)
158 (2/2)(9/9)(5/5)(8/8)(7/7)(10/−)(6/6)(6/6)
159 (2/2)(5/5)(6/6)(7/−)(9/9)
160 (2/2)(5/5)(6/6)(8/8)(8/−)
161 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)(7/−)
162 (2/2)(9/9)(2/−)(6/−)
163 (2/2)(5/5)(1b/−)(9/−)
164 (2/2)(5/5)(11/−)(9/9)
165 (2/2)(4/−)(8/−)
166 (2/2)(5/5)(6/6)(8/8)(5/−)(9/9)
167 (2/2)(11/−)(9/9)
168 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(8/−)(7/−)
169 (2/2)(9/9)(5/5)(8/8)(6/−)(1b/−)(7/−)
170 (2/2)(9/9)(5/5)(8/8)(11/−)(9/−)
171 (2/2)(5/5)(7/−)(8/8)
172 (4/−)(7/−)
173 (2/2)(5/5)(8/8)(7/7)(8/−)(8/−)
174 (2/2)(5/5)(6/6)(8/8)(4/−)(9/9)
175 (2/2)(5/5)(8/8)(1a/−)
176 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(4/−)(7/7)
177 (2/2)(9/9)(5/5)(8/8)(6/−)(8/−)(8/−)
178 (2/2)(5/5)(8/8)(7/7)(10/−)(6/6)
179 (2/2)(9/9)(5/5)(6/6)(6/6)(5/−)(8/8)
180 (2/2)(5/5)(1b/−)(7/−)
181 (3/−)(8/−)
182 (2/2)(9/9)(5/5)(6/6)(9/−)(6/6)(6/−)
183 (2/2)(9/9)(5/5)(4/−)(6/6)(6/6)
184 (2/2)(9/9)(10/−)(5/5)

185 (2/2)(5/5)(8/8)(7/7)(3/−)(6/6)
186 (2/2)(5/5)(8/8)(7/7)(4/−)(7/−)
187 (2/2)(9/9)(5/5)(8/8)(2/−)(6/−)
188 (2/2)(9/9)(5/5)(6/6)(6/6)(7/−)(7/−)
189 (8/−)(7/−)
190 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/−)(9/−)
191 (2/2)(5/5)(8/8)(7/7)(6/6)
192 (2/2)(5/5)(8/8)(7/7)(10/−)(9/9)
193 (2/2)(9/9)(5/5)(8/8)(6/−)(5/−)(6/6)
194 (2/2)(9/9)(9/−)(8/−)
195 (2/2)(10/−)(7/−)
196 (2/2)(5/5)(6/6)(8/8)(9/9)(6/6)
197 (2/2)(9/9)(5/5)(8/8)(11/−)(7/7)
198 (2/2)(5/5)(1a/−)(8/−)
199 (2/2)(9/9)(5/5)(8/8)(7/7)(11/−)(9/−)
200 (2/2)(5/5)(8/8)(4/−)(6/6)
201 (2/2)(9/9)(5/5)(6/6)(11/−)(6/6)(6/−)
202 (2/2)(5/5)(6/6)(1a/−)(7/−)
203 (2/2)(5/5)(7/−)
204 (2/2)(6/−)(7/−)
205 (2/2)(9/9)(5/5)(4/−)(9/−)
206 (2/2)(5/5)(6/6)(6/−)(6/−)
207 (2/2)(9/9)(3/−)(5/5)
208 (2/2)(9/9)(6/−)(8/−)
209 (2/2)(5/5)(6/6)(8/8)(7/7)(11/−)(6/−)
210 (2/2)(5/5)(8/8)(7/7)(9/9)(9/−)
211 (2/2)(5/5)(8/8)(7/7)(9/9)(6/6)(6/6)
212 (2/2)(9/9)(5/5)(8/8)(7/7)(4/−)(6/6)(6/6)
213 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(5/−)(7/−)
214 (2/2)(5/5)(1a/−)(6/−)
215 (2/2)(10/−)(5/5)
216 (2/2)(9/9)(5/5)(8/8)(6/−)(8/−)
217 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(4/−)(8/−)
218 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(11/−)(7/7)
219 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)(7/−)
220 (2/2)(5/5)(8/8)(7/7)(1b/−)(9/−)
221 (2/2)(9/9)(5/5)(6/6)(1b/−)(6/−)
222 (2/2)(9/9)(5/5)(8/8)(6/−)(4/−)(9/−)
223 (2/2)(5/5)(8/8)(4/−)(7/7)
224 (5/−)(7/−)
225 (2/2)(9/9)(5/5)(8/8)(1b/−)(5/−)
226 (2/2)(9/9)(5/5)(8/8)(5/−)
227 (2/2)(5/5)(8/8)(7/7)(7/−)
228 (2/2)(5/5)(6/6)(8/8)(1b/−)(7/−)
229 (10/−)(7/−)
230 (2/2)(9/9)(5/5)(6/6)(6/6)(8/8)(1a/−)
231 (2/2)(5/5)(8/8)(8/−)
232 (2/2)(9/9)(5/5)(8/8)(7/7)(7/−)(6/6)(6/6)
233 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(3/−)(6/6)(6/−)
234 (2/2)(9/9)(5/5)(8/8)(1a/−)(9/−)
235 (2/2)(5/5)(6/6)(1b/−)(7/−)

236 (2/2)(8/−)
237 (2/2)(9/9)(5/5)(6/6)(10/−)(6/6)(6/−)
238 (2/2)(5/5)(8/8)(7/7)(8/−)
239 (2/2)(2/−)(5/5)
240 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)
241 (2/2)(1b/−)(9/−)
242 (2/2)(5/5)(6/6)(8/8)(2/−)(6/−)
243 (2/2)(5/5)(6/6)(8/8)(7/7)(3/−)(7/−)
244 (2/2)(5/5)(8/8)(7/7)(10/−)(7/−)
245 (2/2)(5/5)(1a/−)(7/−)
246 (2/2)(5/5)(6/6)(6/−)
247 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(5/−)(9/−)
248 (2/2)(5/5)(6/6)(4/−)(7/−)
249 (2/2)(9/9)(5/5)(8/8)(6/−)(1b/−)(5/−)
250 (2/2)(9/9)(5/5)(5/−)
251 (2/2)(9/9)(5/5)(6/6)(6/6)(4/−)(8/8)
252 (2/2)(9/9)(5/5)(8/8)(4/−)(9/−)
253 (3/−)(6/−)
254 (2/2)(9/9)(5/5)(8/8)(8/−)(7/7)
255 (2/2)(9/9)(5/5)(8/8)(7/7)(1b/−)(6/−)
256 (2/2)(9/9)(5/5)(6/6)(6/6)(10/−)(6/−)
257 (2/2)(7/−)(5/5)
258 (2/2)(5/5)(8/8)(5/−)(9/9)
259 (2/2)(9/9)(8/−)(5/5)
260 (2/2)(9/9)(5/5)(6/6)
261 (2/2)(5/5)(7/−)(9/9)
262 (2/2)(5/5)(6/6)(6/−)(8/8)
263 (2/2)(5/5)(6/6)(1a/−)(9/−)
264 (2/2)(5/5)(4/−)(9/9)
265 (2/2)(9/9)(5/5)(6/6)(6/6)(5/−)(9/−)
266 (2/2)(5/5)(3/−)(8/8)
267 (2/2)(5/5)(8/8)(1b/−)(8/−)
268 (2/2)(11/−)(7/−)
269 (2/2)(9/9)(5/5)(6/6)(6/6)(11/−)(7/−)
270 (4/−)(9/−)
271 (2/2)(1b/−)(8/−)
272 (2/2)(9/9)(5/5)(6/6)(5/−)(6/6)(6/−)
273 (2/2)(9/9)(5/5)(6/6)(6/6)(11/−)(9/−)
274 (2/2)(9/9)(5/5)(8/8)(7/7)(7/−)(8/−)
275 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/−)
276 (2/2)(9/9)(5/5)(8/8)(6/−)(10/−)(9/−)
277 (2/2)(9/9)(5/5)(8/8)(7/7)(11/−)(7/−)
278 (2/2)(9/9)(5/5)(6/6)(6/6)(2/−)(8/8)
279 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(7/−)(7/−)
280 (2/2)(9/9)(5/5)(6/6)(6/6)(5/−)(7/−)
281 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(5/−)
282 (2/2)(9/9)(1b/−)(6/−)
283 (3/−)(5/−)
284 (2/2)(5/5)(6/6)(8/8)(7/7)(9/9)(7/−)
285 (2/2)(2/−)(7/−)
286 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)(9/9)

287 (2/2)(9/9)(5/5)(8/8)(3/−)(7/7)
288 (2/2)(9/9)(5/5)(6/6)(4/−)(8/8)
289 (2/2)(5/5)(6/6)(11/−)(8/8)
290 (2/2)(9/9)(5/5)(8/8)(6/−)(5/−)(8/−)
291 (2/2)(5/5)(1a/−)(9/−)
292 (2/2)(5/5)(8/8)(11/−)(9/9)
293 (2/2)(9/9)(5/5)(8/8)(10/−)(1a/−)
294 (1b/−)(7/−)
295 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(9/−)(8/−)
296 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)(7/−)
297 (2/2)(9/9)(5/5)(8/8)(7/7)(4/−)(8/−)
298 (2/2)(5/5)(6/6)(1a/−)(6/−)
299 (2/2)(9/9)(11/−)(6/−)
300 (2/2)(9/9)(5/5)(8/8)(5/−)(1a/−)
301 (1b/−)(8/−)
302 (2/2)(9/9)(5/5)(6/6)(6/6)(8/8)(6/−)
303 (1a/−)(8/−)
304 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(9/−)(1a/−)
305 (2/2)(9/9)(5/5)(8/8)(2/−)(9/−)
306 (9/−)(5/−)
307 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)(9/−)
308 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(8/−)(6/−)
309 (2/2)(9/9)(5/5)(6/6)(6/6)(5/−)
310 (2/2)(9/9)(1a/−)(7/−)
311 (2/2)(5/5)(6/6)(2/−)(9/9)
312 (2/2)(5/5)(3/−)(6/6)
313 (3/−)(9/−)
314 (2/2)(9/9)(5/5)(8/8)(6/−)(3/−)(8/−)
315 (2/2)(9/9)(5/5)(8/8)(6/−)(9/−)(6/6)
316 (2/2)(5/5)(8/8)(4/−)(9/9)
317 (2/2)(9/9)(5/5)(6/6)(6/6)(7/−)(8/8)
318 (2/2)(9/9)(5/5)(6/6)(1b/−)(5/−)
319 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/−)(8/−)
320 (2/2)(5/5)(6/6)(8/8)(7/7)(11/−)(9/9)
321 (2/2)(5/5)(6/6)(9/9)(8/8)
322 (2/2)(9/9)(5/5)(8/8)(1b/−)(8/−)
323 (2/2)(5/5)(6/6)(8/8)(1a/−)(6/−)
324 (2/2)(9/9)(5/5)(6/6)(6/6)(1b/−)(5/−)
325 (2/2)(5/5)(6/6)(8/8)(7/7)(3/−)(8/−)
326 (2/2)(9/9)(5/5)(8/8)(7/7)(5/−)
327 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(5/−)(9/−)
328 (2/2)(9/9)(5/5)(8/8)(6/−)(9/−)(7/7)
329 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(7/−)
330 (2/2)(9/9)(5/5)(8/8)(6/−)(11/−)(6/6)
331 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)(6/−)
332 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(9/−)(9/−)
333 (2/2)(5/5)(8/8)(10/−)(6/6)
334 (2/2)(9/9)(5/5)(6/6)(6/6)(7/−)
335 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(9/−)(6/−)
336 (2/2)(5/5)(8/8)(3/−)(9/9)
337 (2/2)(7/−)(6/−)

338 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/-)(5/-)
339 (8/-)(6/-)
340 (2/2)(9/9)(5/5)(3/-)(6/6)(6/6)
341 (2/2)(5/5)(6/6)(5/-)(9/9)
342 (2/2)(9/9)(5/5)(8/8)(6/-)(2/-)(8/-)
343 (2/2)(5/5)(8/8)(1b/-)(7/-)
344 (2/2)(5/5)(6/6)(3/-)(9/9)
345 (2/2)(5/5)(6/6)(8/8)(5/-)(7/7)
346 (2/2)(9/9)(5/5)(8/8)(8/-)
347 (2/2)(9/9)(5/5)(9/-)(6/6)(6/6)
348 (2/2)(5/5)(6/6)(8/8)(7/7)
349 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/-)(7/-)
350 (2/2)(5/5)(6/6)(8/8)(6/-)(6/-)
351 (2/2)(9/9)(8/-)(9/-)
352 (2/2)(5/5)(6/6)(8/8)(10/-)(7/7)
353 (2/2)(9/9)(1b/-)(8/-)
354 (2/2)(5/5)(8/8)(1a/-)(6/-)
355 (4/-)(6/-)
356 (2/2)(9/9)(5/5)(8/8)(6/-)(1a/-)(6/-)
357 (2/2)(9/9)(5/5)(8/8)
358 (2/2)(5/5)(6/6)(8/8)(7/7)(6/-)(8/-)
359 (2/2)(9/9)(5/5)(6/6)(6/6)
360 (2/2)(9/9)(5/5)(11/-)(9/-)
361 (2/2)(9/9)(5/5)(8/8)(7/7)(5/-)(7/-)
362 (2/2)(3/-)(8/-)
363 (2/2)(9/9)(7/-)(9/-)
364 (2/2)(5/5)(8/8)(7/7)(11/-)(8/-)
365 (2/2)(4/-)(6/-)
366 (2/2)(1a/-)(8/-)
367 (2/2)(5/5)(8/8)(7/7)(7/-)(9/9)
368 (2/2)(9/9)(5/5)(8/8)(6/-)(2/-)(7/7)
369 (2/2)(9/9)(5/5)(8/8)(6/-)(6/6)(6/-)(7/7)
370 (2/2)(9/9)(5/5)(8/8)(7/7)
371 (2/2)(9/9)(5/5)(6/6)(6/6)(1b/-)(7/-)
372 (2/2)(9/9)(5/5)(8/8)(6/-)(6/6)(9/-)(7/7)
373 (2/2)(5/5)(6/6)(8/8)(7/7)(6/-)(5/-)
374 (2/2)(9/9)(5/5)(8/8)(1a/-)(8/-)
375 (2/2)(9/9)(5/5)(8/8)(7/7)(5/-)(6/6)(6/6)
376 (2/2)(7/-)(8/-)
377 (2/2)(5/5)
378 (2/2)(9/9)(5/5)(6/6)(6/6)(9/-)
379 (2/2)(5/5)(6/6)(3/-)(7/-)
380 (2/2)(11/-)(8/-)
381 (2/2)(9/9)(5/5)(6/6)(2/-)(8/8)
382 (2/2)(9/9)(5/5)(6/6)(6/6)(6/-)(6/-)
383 (2/2)(9/9)(5/5)(6/6)(6/6)(5/-)(6/-)
384 (2/2)(5/5)(8/8)(7/7)(1a/-)(7/-)
385 (2/2)(9/9)(5/5)(8/8)(6/-)(11/-)(9/-)
386 (2/2)(5/5)(6/6)(11/-)(6/-)
387 (2/2)(5/5)(5/-)(6/6)
388 (2/2)(5/5)(6/6)(8/8)(7/7)(9/9)(8/-)

389 (7/−)(8/−)
390 (2/2)(9/9)(5/5)(6/6)(1a/−)(8/−)
391 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)(6/−)
392 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(10/−)(9/−)
393 (2/2)(5/5)(6/6)(8/8)(2/−)(7/7)
394 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(10/−)(6/−)
395 (2/2)(5/5)(9/9)(9/−)
396 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(7/−)(8/−)
397 (2/2)(3/−)(7/−)
398 (2/2)(9/9)(5/5)(8/8)(6/−)(8/−)(9/−)
399 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(4/−)(7/−)
400 (2/2)(9/9)(1a/−)(8/−)
401 (5/−)(9/−)
402 (2/2)(9/9)(5/5)(8/8)(7/7)(1b/−)(5/−)
403 (2/2)(9/9)(7/−)
404 (2/2)(9/9)(5/5)(8/8)(11/−)(1a/−)
405 (2/2)(9/9)(5/5)(8/8)(6/−)(9/−)(9/−)
406 (2/2)(5/5)(6/6)(8/8)(11/−)(6/−)
407 (2/2)(2/−)(8/−)
408 (2/2)(5/5)(6/6)(8/8)(7/7)(4/−)(7/−)
409 (2/2)(5/5)(8/8)(2/−)(9/9)
410 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(4/−)(6/−)
411 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(3/−)(9/−)
412 (2/2)(5/5)(8/8)(9/9)
413 (2/2)(5/5)(6/6)(8/8)(7/7)(9/9)
414 (2/2)(9/9)(11/−)(7/−)
415 (2/2)(9/9)(5/5)(8/8)(7/7)(2/−)(6/6)(6/6)
416 (2/2)(9/9)(5/5)(8/8)(6/−)(1a/−)(8/−)
417 (2/2)(5/5)(6/6)(9/9)(6/6)(6/−)
418 (2/2)(9/9)(5/5)(2/−)(8/8)
419 (2/2)(5/5)(8/8)(3/−)(6/6)
420 (2/2)(9/9)(5/5)(10/−)(9/−)
421 (2/2)(5/5)(6/6)(8/8)(1b/−)(9/−)
422 (2/2)(9/9)(5/5)(6/6)(6/6)(2/−)(7/−)
423 (2/2)(9/9)(5/5)(8/8)(3/−)(6/−)
424 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(2/−)(7/−)
425 (2/2)(1a/−)(7/−)
426 (2/2)(6/−)(8/−)
427 (2/2)(5/5)(6/6)(8/8)(8/−)(7/7)
428 (2/2)(9/9)(5/5)(6/6)(9/−)(9/−)
429 (2/2)(9/9)(10/−)(8/−)
430 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)(5/−)
431 (2/2)(5/5)(6/6)(8/8)(1a/−)(5/−)
432 (2/2)(9/9)(5/5)(8/8)(6/−)(9/−)
433 (2/2)(8/−)(5/5)
434 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(7/7)(9/−)
435 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)(5/−)
436 (2/2)(9/9)(1b/−)(5/−)
437 (2/2)(5/5)(8/8)(7/7)(6/6)(9/9)
438 (2/2)(9/9)(8/−)(7/−)
439 (2/2)(9/9)(5/5)(8/8)(7/7)(3/−)(7/−)

440 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(3/−)(6/−)
441 (2/2)(5/5)(8/8)(7/7)(3/−)(9/9)
442 (2/2)(9/9)(5/5)(8/8)(1b/−)(6/−)
443 (2/2)(9/9)(3/−)(7/−)
444 (2/2)(9/9)(5/5)(7/−)
445 (2/2)(5/5)(6/6)(7/−)
446 (2/2)(9/9)(5/5)(8/8)(7/7)(3/−)(6/6)(6/6)
447 (1b/−)(5/−)
448 (2/2)(5/5)(6/6)(8/8)(1b/−)(6/−)
449 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(8/−)(6/6)(6/−)
450 (2/2)(5/5)(8/8)(7/7)(1a/−)(8/−)
451 (2/2)(5/5)(6/6)(8/8)(1a/−)
452 (2/2)(9/9)(5/5)(8/8)(6/−)(3/−)(7/7)
453 (2/2)(4/−)(5/5)
454 (2/2)(9/9)(4/−)(9/−)
455 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(3/−)(8/−)
456 (2/2)(9/9)(5/5)(6/6)(6/6)(6/−)
457 (2/2)(5/5)(8/8)(6/6)(9/9)
458 (5/−)(5/−)
459 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)(8/−)
460 (2/2)(5/5)(8/8)(7/7)(9/9)(8/−)
461 (2/2)(5/5)(8/8)(9/9)(1a/−)
462 (2/2)(5/5)(6/6)(8/8)(11/−)(7/7)
463 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1b/−)(7/−)
464 (2/2)(9/9)(6/−)(7/−)
465 (1b/−)(9/−)
466 (2/2)(9/9)(5/5)(6/6)(1b/−)(8/−)
467 (2/2)(5/5)(8/8)(7/7)(8/−)(6/6)
468 (2/2)(2/−)(6/−)
469 (2/2)(5/5)(8/8)(7/7)(1a/−)
470 (2/2)(9/9)(5/5)(8/8)(6/−)(3/−)(9/−)
471 (2/2)(9/9)(5/5)(6/6)(2/−)(6/6)(6/−)
472 (2/2)(5/5)(8/8)(1b/−)(5/−)
473 (2/2)(9/9)(5/5)(6/6)(6/6)(8/8)(7/7)
474 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1b/−)(8/−)
475 (2/2)(5/5)(8/8)(1a/−)(7/−)
476 (2/2)(9/9)(5/5)(6/6)(6/6)(10/−)(9/−)
477 (2/2)(5/5)(6/6)(8/8)(7/7)(1b/−)(5/−)
478 (2/2)(9/9)(5/5)(8/8)(6/−)(7/7)(8/−)
479 (2/2)(5/5)(6/6)(8/8)(7/7)(1b/−)(6/−)
480 (2/2)(9/9)(5/5)(6/6)(6/6)(1b/−)(8/−)
481 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(3/−)(9/−)
482 (9/−)(6/−)
483 (2/2)(5/5)(6/6)(8/8)(7/7)(3/−)(9/9)
484 (2/2)(5/5)(8/8)(4/−)(8/−)
485 (2/2)(1a/−)(5/−)
486 (2/2)(9/9)(8/−)
487 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(8/−)
488 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/−)
489 (1a/−)(5/−)
490 (2/2)(9/9)(5/5)(8/8)(7/7)(4/−)(7/−)

491 (2/2)(9/9)(5/5)(7/−)(9/−)
492 (2/2)(9/9)(3/−)(8/−)
493 (5/−)(6/−)
494 (2/2)(9/9)(5/5)(8/8)(7/7)(8/−)(6/6)(6/6)
495 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(5/−)(6/−)
496 (2/2)(5/5)(6/6)(2/−)(7/−)
497 (2/2)(9/9)(5/5)(6/6)(9/−)(8/8)
498 (2/2)(5/5)(8/8)(7/7)(1b/−)(7/−)
499 (2/2)(5/5)(6/6)(8/8)(1a/−)(9/−)
500 (2/2)(5/5)(8/8)(7/7)(8/−)(7/−)
501 (2/2)(5/5)(6/6)(1b/−)(6/−)
502 (2/2)(9/9)(5/5)(8/8)(9/−)(6/−)
503 (2/2)(9/9)(5/5)(6/6)(6/6)(10/−)(7/−)
504 (2/2)(5/5)(8/8)(1b/−)(6/−)
505 (2/2)(5/5)(6/6)(8/8)(1b/−)(5/−)
506 (2/2)(9/9)(5/5)(8/8)(7/7)(7/−)(7/−)
507 (2/2)(9/9)(5/5)(6/6)(5/−)
508 (2/2)(5/5)(8/8)(7/7)(9/9)
509 (2/2)(5/5)(6/6)(8/8)(8/−)(6/−)
510 (2/2)(9/9)(5/5)(8/8)(9/−)
511 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(3/−)(1a/−)
512 (2/2)(9/9)(5/5)(6/6)(6/6)(3/−)(7/−)
513 (2/2)(9/9)(5/5)(8/8)(6/−)(7/7)(7/−)
514 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(10/−)(7/−)
515 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/−)(7/−)
516 (1a/−)
517 (2/2)(10/−)(6/−)
518 (2/2)(3/−)(5/5)
519 (2/2)(5/5)(6/6)(1b/−)(5/−)
520 (2/2)(5/5)(6/6)(8/8)(10/−)(6/−)
521 (2/2)(5/5)(6/6)(4/−)(9/9)
522 (2/2)(9/9)(5/5)(8/8)(7/7)(9/−)(6/6)(6/6)
523 (2/2)(9/9)(5/5)(8/8)(7/7)(10/−)(8/−)
524 (2/2)(5/5)(8/8)(7/7)(2/−)(8/−)
525 (2/2)(8/−)(7/−)
526 (2/2)(5/5)(10/−)(8/8)
527 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(9/−)(7/−)
528 (2/2)(9/9)(10/−)(9/−)
529 (2/2)(9/9)(4/−)(6/−)
530 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)
531 (2/2)(9/9)(5/5)(1b/−)(5/−)
532 (2/2)(5/5)(8/8)(6/6)(7/7)
533 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(1a/−)
534 (2/2)(9/9)(9/−)
535 (2/2)(5/5)(6/6)(8/8)(7/7)(6/−)
536 (2/2)(5/5)(6/6)(8/8)(10/−)(9/9)
537 (11/−)(7/−)
538 (2/2)(5/5)(6/6)
539 (2/2)(9/9)(5/5)(8/8)(7/7)(1b/−)(8/−)
540 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(7/−)(9/−)
541 (2/2)(9/9)(5/5)(6/6)(3/−)(6/6)(6/−)

542 (2/2)(5/5)(8/8)(7/7)(8/−)(9/9)
543 (2/2)(9/9)(5/5)(6/6)(6/6)(7/−)(9/−)
544 (2/2)(5/5)(6/6)(8/8)(7/7)(4/−)(8/−)
545 (2/2)(9/9)(5/5)(8/8)(7/7)(2/−)(9/−)
546 (2/2)(9/9)(5/5)(6/6)(6/6)(1b/−)(9/−)
547 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)
548 (2/2)(9/9)(5/5)(8/8)(3/−)(1a/−)
549 (2/2)(5/5)(8/8)(7/7)(3/−)(8/−)
550 (2/2)(5/5)(6/6)(8/8)(7/7)(4/−)(9/9)
551 (2/2)(5/5)(6/6)(8/8)(7/7)(10/−)(7/−)
552 (2/2)(9/9)(5/5)(8/8)(6/−)(8/−)(6/6)
553 (2/2)(1a/−)(9/−)
554 (2/2)(9/9)(5/5)(8/8)(7/7)(11/−)(6/6)(6/6)
555 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1b/−)(9/−)
556 (2/2)(9/9)(5/5)(8/8)(7/7)(7/−)
557 (2/2)(6/−)
558 (10/−)(5/−)
559 (2/2)(4/−)(9/9)
560 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(8/−)(9/−)
561 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/−)(8/−)
562 (2/2)(9/9)(5/5)(8/8)(6/−)(1a/−)(7/−)
563 (2/2)(5/5)(1b/−)(5/−)
564 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(4/−)(6/6)(6/−)
565 (4/−)(5/−)
566 (2/2)(5/5)(6/6)(8/8)(7/7)(2/−)(8/−)
567 (2/2)(9/9)(5/5)(1a/−)(7/−)
568 (2/2)(5/5)(6/6)(8/8)(7/7)(4/−)(6/−)
569 (2/2)(5/5)(8/8)(8/−)(8/−)
570 (2/2)(5/5)(8/8)(3/−)(8/−)
571 (2/2)(5/5)(6/6)(1a/−)(8/−)
572 (2/2)(9/9)(5/5)(6/6)(10/−)(9/−)
573 (2/2)(9/9)(5/5)(6/6)(6/6)(6/−)(7/−)
574 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(4/−)(1a/−)
575 (2/2)(5/5)(6/6)(10/−)(9/9)
576 (2/2)(9/9)(5/5)(6/6)(3/−)(9/−)
577 (2/2)(5/5)(6/6)(8/8)(7/7)(1b/−)(8/−)
578 (2/2)(9/9)
579 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(11/−)(8/−)
580 (2/2)(11/−)(5/5)
581 (2/2)(5/5)(8/8)(7/7)(1b/−)(5/−)
582 (2/2)(9/9)(5/5)(7/−)(6/6)(6/6)
583 (9/−)(8/−)
584 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/−)(5/−)
585 (2/2)(5/5)(8/8)(7/7)(2/−)(7/−)
586 (2/2)(9/9)(5/5)(8/8)(9/−)(9/−)
587 (2/2)(9/9)(5/5)(8/8)(7/7)(8/−)(9/−)
588 (2/2)(5/5)(8/8)(2/−)(6/6)
589 (2/2)(5/5)(8/8)(10/−)(9/9)
590 (2/2)(5/5)(6/6)(8/8)(7/7)(6/−)(9/9)
591 (2/2)(5/5)(11/−)(6/6)
592 (2/2)(9/9)(3/−)(9/−)

593 (2/2)(9/9)(5/5)(8/8)(6/−)(4/−)(7/7)
594 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)
595 (2/2)(1b/−)(5/−)
596 (6/−)(7/−)
597 (2/2)(9/9)(5/5)(6/6)(6/6)(8/8)
598 (2/2)(9/9)(5/5)(8/8)(6/−)(3/−)(6/6)
599 (7/−)
600 (2/2)(9/9)(1a/−)(9/−)
601 (2/2)(5/5)(6/6)(11/−)(9/9)
602 (2/2)(5/5)(6/6)(8/8)(7/7)(3/−)(5/−)
603 (1a/−)(6/−)
604 (2/2)(9/9)(6/−)(9/−)
605 (10/−)(8/−)
606 (2/2)(5/5)(8/8)(6/6)
607 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(6/−)(1a/−)
608 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1a/−)(6/−)
609 (2/2)(9/9)(5/5)(8/8)(7/7)(2/−)(8/−)
610 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(6/−)(6/−)
611 (2/2)(5/5)(8/8)(7/7)(5/−)(7/−)
612 (2/2)(9/9)(9/−)(5/5)
613 (2/2)(6/−)(5/5)
614 (2/2)(9/9)(5/5)(8/8)(6/−)(1b/−)(6/−)
615 (2/2)(9/9)(5/5)(8/8)(7/7)(2/−)(7/−)
616 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(9/−)
617 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(5/−)(6/6)(6/−)
618 (2/2)(5/5)(8/8)(7/7)(11/−)(9/9)
619 (2/2)(9/9)(5/5)(9/−)
620 (2/2)(7/−)(9/9)
621 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)(7/−)
622 (2/2)(5/5)(6/6)(8/8)(7/7)(4/−)(5/−)
623 (2/2)(9/9)(5/5)(8/8)(7/7)(3/−)(9/−)
624 (2/2)(9/9)(5/5)(8/8)(10/−)(9/−)
625 (2/2)(9/9)(5/5)(6/6)(6/6)(9/−)(7/−)
626 (2/2)(9/9)(5/5)(6/6)(6/6)(11/−)(8/8)
627 (2/2)(5/5)(6/6)(8/8)(1a/−)(8/−)
628 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(11/−)(9/−)
629 (2/2)(5/5)(8/8)(7/7)(9/9)(7/−)
630 (2/2)(5/5)(6/6)(8/8)(6/−)(7/7)
631 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)(8/−)
632 (2/2)(5/5)(8/8)(7/7)(4/−)(9/9)
633 (2/2)(9/9)(5/5)(6/6)(4/−)(6/6)(6/−)
634 (2/2)(9/9)(5/5)(1a/−)(9/−)
635 (2/2)(9/9)(5/5)(6/6)(6/6)(8/8)(9/−)
636 (2/2)(9/9)(2/−)(7/−)
637 (2/2)(5/5)(8/8)
638 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(11/−)(6/−)
639 (2/2)(5/5)(8/8)(7/7)(5/−)(8/−)
640 (2/2)(5/5)(11/−)(8/8)
641 (2/2)(5/5)(6/6)(3/−)(8/8)
642 (2/2)(9/9)(5/5)(8/8)(7/7)(1a/−)(8/−)
643 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(8/−)(8/−)

644 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(2/-)(9/-)
645 (2/2)(9/9)(5/5)(6/6)(4/-)(9/-)
646 (2/2)(9/9)(5/5)(8/8)(6/-)(7/7)
647 (2/2)(9/9)(5/5)(6/6)(5/-)(9/-)
648 (2/2)(9/9)(5/5)(8/8)(6/-)(11/-)(8/-)
649 (2/2)(9/9)(5/5)(8/8)(6/-)(6/6)(2/-)(1a/-)
650 (2/2)(9/9)(5/5)(6/6)(7/-)
651 (2/2)(9/9)(1b/-)(7/-)
652 (2/2)(9/9)(5/5)(8/8)(11/-)(6/-)
653 (2/2)(5/5)(8/8)(5/-)(8/-)
654 (2/2)(9/9)(5/5)(1b/-)(7/-)
655 (2/2)(2/-)(9/9)
656 (2/2)(9/9)(5/5)(6/6)(6/6)(2/-)(6/-)
657 (2/2)(9/9)(5/5)(8/8)(6/-)(10/-)(8/-)
658 (2/2)(10/-)(9/9)
659 (2/2)(9/9)(5/5)(8/8)(6/-)
660 (2/2)(9/9)(5/5)(8/8)(6/-)(6/6)(8/-)
661 (2/2)(5/5)(6/6)(8/8)(11/-)(9/9)
662 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1b/-)(5/-)
663 (2/2)(9/9)(5/5)(8/8)(7/7)(3/-)(8/-)
664 (2/2)(9/9)(5/5)(8/8)(6/-)(6/6)(4/-)(9/-)
665 (2/2)(5/5)(6/6)(8/8)(4/-)(6/-)
666 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(10/-)(8/-)
667 (2/2)(9/9)(5/5)(8/8)(6/-)(1a/-)
668 (2/2)(9/9)(5/5)(8/8)(6/-)(7/7)(6/6)(6/-)
669 (2/2)(9/9)(5/5)(9/-)(8/8)
670 (2/2)(9/9)(5/5)(8/8)(6/-)(5/-)(9/-)
671 (2/2)(5/5)(7/-)(6/6)
672 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/-)(7/-)
673 (2/2)(5/5)(8/8)(7/7)(4/-)(6/6)
674 (2/2)(9/9)(5/5)(8/8)(2/-)(1a/-)
675 (2/2)(5/5)(9/9)
676 (2/2)(8/-)(6/-)
677 (2/2)(9/9)(5/5)(6/6)(1a/-)(6/-)
678 (2/2)(9/9)(5/5)(8/8)(1a/-)(5/-)
679 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)(6/-)
680 (2/2)(1a/-)
681 (2/2)(9/9)(5/5)(6/6)(6/6)(3/-)(9/-)
682 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/-)(6/-)
683 (2/2)(9/9)(5/5)(2/-)(6/6)(6/6)
684 (2/2)(9/9)(5/5)(6/6)(6/6)(10/-)(8/8)
685 (2/2)(9/9)(5/5)(8/8)(7/7)(9/-)(8/-)
686 (2/2)(9/9)(5/5)(3/-)(9/-)
687 (2/2)(9/9)(4/-)(7/-)
688 (2/2)(5/5)(6/6)(8/8)(7/7)(10/-)(8/-)
689 (2/2)(5/5)(8/8)(5/-)(7/7)
690 (8/-)(9/-)
691 (2/2)(9/9)(5/5)(8/8)(1a/-)
692 (2/2)(9/9)(1a/-)
693 (2/2)(5/5)(8/8)(11/-)(8/-)
694 (2/2)(9/9)(6/-)(6/-)

695 (2/2)(9/9)(5/5)(11/−)(6/6)(6/6)
696 (2/2)(9/9)(8/−)(8/−)
697 (2/2)(9/9)(5/5)(6/6)(8/8)
698 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(1b/−)(6/−)
699 (2/2)(9/9)(5/5)(8/8)(1b/−)(7/−)
700 (2/2)(5/5)(8/8)(7/7)(11/−)(7/−)
701 (2/2)(5/5)(6/6)(8/8)(7/7)(2/−)(9/9)
702 (8/−)(8/−)
703 (2/2)(9/9)(5/5)(8/8)(7/7)(1b/−)(9/−)
704 (2/2)(9/9)(5/5)(8/8)(6/−)(9/−)(8/−)
705 (10/−)(9/−)
706 (2/2)(9/9)(5/5)(6/6)(1a/−)
707 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)
708 (2/2)(5/5)(8/8)(3/−)(7/7)
709 (2/2)(5/5)(8/8)(7/7)(2/−)(6/6)
710 (2/2)(5/5)(6/6)(8/8)(5/−)
711 (2/2)(9/9)(5/5)(8/8)(10/−)(6/−)
712 (2/2)(5/5)(6/6)(8/8)(7/7)(1b/−)(9/−)
713 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(8/−)(9/−)
714 (2/2)(5/5)(8/8)(7/7)(3/−)(7/−)
715 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(3/−)(7/7)
716 (2/2)(5/5)(8/8)(7/7)(1b/−)(6/−)
717 (2/2)(5/5)(8/8)(7/7)(11/−)(6/6)
718 (2/2)(9/9)(5/5)(8/8)(6/−)(10/−)(7/7)
719 (2/2)(3/−)(6/−)
720 (2/2)(9/9)(5/5)(6/6)(2/−)(9/−)
721 (2/2)(9/9)(5/5)(1a/−)(6/−)
722 (2/2)(5/5)(8/8)(9/9)(7/7)
723 (2/2)(9/9)(5/5)(8/8)(5/−)(6/−)
724 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(6/−)
725 (2/2)(5/5)(6/6)(4/−)(6/−)
726 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(2/−)(6/−)
727 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(10/−)(1a/−)
728 (2/2)(9/9)(5/5)(6/6)(8/8)(6/6)
729 (2/2)(9/9)(5/5)(6/6)(6/6)(3/−)(6/−)
730 (2/2)(5/5)(6/6)(8/8)(9/9)(8/−)
731 (2/2)(5/5)(6/6)(3/−)(6/−)
732 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)(5/−)
733 (2/2)(5/5)(6/6)(4/−)(8/8)
734 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(2/−)(9/−)
735 (6/−)(5/−)
736 (2/2)(5/5)(6/6)(8/8)(7/7)(9/9)(9/−)
737 (2/2)(9/9)(5/5)(8/8)(6/−)(11/−)(7/7)
738 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/−)
739 (2/2)(9/9)(5/5)(9/−)(9/−)
740 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(10/−)(7/7)
741 (2/2)(9/9)(5/5)(8/8)(7/7)(8/−)
742 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(9/−)
743 (2/2)(5/5)(6/6)(8/8)(7/7)(7/−)(5/−)
744 (2/2)(9/9)(3/−)(6/−)
745 (2/2)(5/5)(8/8)(9/9)(6/−)

746 (2/2)(5/5)(6/6)(8/8)(7/7)(10/−)(9/9)
747 (1a/−)(7/−)
748 (2/2)(5/5)(10/−)(6/6)
749 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(10/−)(6/6)(6/−)
750 (2/2)(5/5)(6/6)(8/8)(7/7)(1a/−)(9/−)
751 (2/2)(9/9)(5/5)(10/−)(6/6)(6/6)
752 (2/2)(1b/−)(6/−)
753 (2/2)(9/9)(11/−)(8/−)
754 (2/2)(9/9)(5/5)(8/8)(1b/−)(9/−)
755 (5/−)(8/−)
756 (2/2)(5/5)(6/6)(10/−)(6/−)
757 (2/2)(9/9)(5/5)(1b/−)(9/−)
758 (2/2)(9/9)(5/5)(6/6)(7/−)(6/6)(6/−)
759 (2/2)(9/9)(5/5)(6/6)(6/6)(4/−)(7/−)
760 (2/2)(9/9)(5/5)(6/6)(11/−)(9/−)
761 (2/2)(5/5)(6/6)(8/8)(7/7)(11/−)(5/−)
762 (2/2)(9/9)(6/−)(5/5)
763 (2/2)(5/5)(6/6)(8/8)(1b/−)(8/−)
764 (2/2)(9/9)(5/5)(8/8)(7/7)(9/−)(7/−)
765 (2/2)(5/5)(6/6)(1a/−)
766 (2/2)(5/5)(6/6)(1b/−)(8/−)
767 (2/2)(5/5)(8/8)(7/7)(1a/−)(5/−)
768 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)(7/−)
769 (9/−)(7/−)
770 (2/2)(1a/−)(6/−)
771 (2/2)(9/9)(9/−)(6/−)
772 (2/2)(5/5)(5/−)
773 (2/2)(9/9)(5/5)(5/−)(9/−)
774 (2/2)(9/9)(5/5)(8/8)(7/7)(10/−)(9/−)
775 (2/2)(5/5)(6/6)(8/8)(3/−)(9/9)
776 (2/2)(9/9)(5/5)(8/8)(3/−)(9/−)
777 (2/2)(9/9)(5/5)(6/6)(6/6)(7/−)(6/−)
778 (2/2)(9/9)(5/5)(5/−)(8/8)
779 (2/2)(9/9)(5/5)(8/8)(9/−)(7/7)
780 (2/2)(9/9)(5/5)(1a/−)
781 (2/2)(5/5)(8/8)(7/7)(7/−)(7/−)
782 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(3/−)(7/−)
783 (2/2)(9/9)(5/5)(8/8)(8/−)(9/−)
784 (2/2)(5/5)(6/6)(2/−)(6/−)
785 (11/−)(9/−)
786 (2/2)(9/9)(5/5)(6/6)(7/−)(9/−)
787 (2/2)(9/9)(5/5)(6/6)(6/6)(1a/−)(8/−)
788 (2/2)(9/9)(5/5)(8/8)(8/−)(6/−)
789 (2/2)(5/5)(8/8)(5/−)
790 (2/2)(9/9)(5/5)(6/6)(6/6)(11/−)(6/−)
791 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)
792 (2/2)(9/9)(5/5)(8/8)(6/−)(4/−)(8/−)
793 (2/2)(9/9)(5/5)(8/8)(6/−)(1b/−)(9/−)
794 (2/2)(9/9)(5/5)(6/6)(8/8)(7/7)
795 (2/2)(5/5)(1b/−)(8/−)
796 (5/−)

797 (2/2)(5/5)(8/8)(7/7)(1a/−)(9/−)
798 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)(6/−)
799 (2/2)(9/9)(5/5)(6/6)(1b/−)(9/−)
800 (2/2)(9/9)(5/5)(6/6)(6/6)(1b/−)(6/−)
801 (2/2)(7/−)(7/−)
802 (2/2)(5/5)(6/6)(8/8)(9/9)(9/−)
803 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)
804 (2/2)(9/9)(5/5)(4/−)(8/8)
805 (2/2)(9/9)(5/5)(8/8)(7/7)(7/−)(9/−)
806 (2/2)(9/9)(5/5)(6/6)(11/−)(8/8)
807 (2/2)(9/9)(5/5)(8/8)(8/−)(1a/−)
808 (2/2)(9/9)(5/5)(1a/−)(5/−)
809 (2/2)(5/5)(6/6)(1a/−)(5/−)
810 (2/2)(9/9)(5/5)(6/6)(6/6)(9/−)(6/−)
811 (2/2)(5/5)(8/8)(7/7)(5/−)(6/6)
812 (2/2)(5/5)(8/8)(1b/−)(9/−)
813 (2/2)(5/5)(8/8)(7/7)(2/−)(9/9)
814 (6/−)
815 (2/2)(5/5)(6/6)(8/8)(7/7)(8/−)(8/−)
816 (2/2)(5/5)(10/−)(9/9)
817 (2/2)(9/9)(5/5)
818 (2/2)(9/9)(6/−)
819 (2/2)(5/5)(6/6)(8/8)(2/−)(9/9)
820 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)(5/−)
821 (2/2)(9/9)(5/5)(6/6)(6/6)(9/−)(9/−)
822 (2/2)(5/5)(9/9)(6/6)(6/6)
823 (2/2)(5/5)(6/6)(6/−)(9/9)
824 (2/2)(8/−)(9/9)
825 (2/2)(5/5)(5/−)(9/9)
826 (2/2)(9/9)(5/5)(6/6)(6/6)(6/−)(9/−)
827 (2/2)(5/5)(6/6)(8/8)(7/7)(10/−)(5/−)
828 (11/−)(6/−)
829 (2/2)(5/5)(3/−)(9/9)
830 (2/2)(9/9)(5/5)(6/6)(6/6)(4/−)(6/−)
831 (2/2)(5/5)(6/6)(10/−)(8/8)
832 (2/2)(9/9)(5/5)(7/−)(8/8)
833 (2/2)(5/5)(8/8)(7/7)(7/−)(8/−)
834 (2/2)(9/9)(5/5)(6/6)(7/−)(8/8)
835 (2/2)(5/5)(6/6)(10/−)(7/−)
836 (2/2)(9/9)(5/5)(6/6)(8/8)(9/−)
837 (2/2)(5/5)(6/6)(6/−)(7/−)
838 (2/2)(9/9)(5/5)(8/8)(7/7)(9/−)
839 (2/2)(9/9)(1a/−)(5/−)
840 (2/2)(5/5)(1b/−)(6/−)
841 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(6/6)(9/−)
842 (2/2)(5/5)(8/8)(7/7)(10/−)(8/−)
843 (2/2)(9/9)(5/5)(8/8)(1a/−)(6/−)
844 (2/2)(9/9)(5/5)(8/8)(7/7)(8/−)(7/−)
845 (2/2)(9/9)(5/5)(8/8)(6/−)(2/−)(9/−)
846 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(9/−)(6/6)(6/−)
847 (2/2)(9/9)(5/5)(2/−)(9/−)

848 (2/2)(5/5)(8/8)(1 a/−)(9/−)
849 (2/2)(5/5)(8/8)(8/−)(7/7)
850 (2/2)(9/9)(11/−)(9/−)
851 (2/2)(5/5)(8/8)(8/−)(9/9)
852 (2/2)(5/5)(8/8)(11/−)(6/6)
853 (2/2)(1b/−)(7/−)
854 (2/2)(10/−)(8/−)
855 (2/2)(9/9)(9/−)(9/−)
856 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(11/−)(1a/−)
857 (2/2)(9/9)(5/5)(8/8)(10/−)(7/7)
858 (2/2)(9/9)(5/5)(8/8)(9/−)(1a/−)
859 (2/2)(9/9)(5/5)(8/8)(7/7)(6/6)(2/−)(8/−)
860 (2/2)(5/5)(6/6)(8/8)(9/9)
861 (2/2)(5/5)(8/8)(7/7)
862 (2/2)(9/9)(5/5)(6/6)(10/−)(8/8)
863 (2/2)(5/5)(6/6)(8/8)
864 (2/2)(9/9)(5/5)(8/8)(4/−)(7/7)
865 (2/2)(5/5)(8/8)(2/−)(7/7)
866 (2/2)(9/9)(5/5)(8/8)(6/−)(1b/−)(8/−)
867 (2/2)(5/5)(6/6)(8/8)(7/7)(5/−)(8/−)
868 (2/2)(5/5)(6/6)(7/−)(7/−)
869 (2/2)(9/9)(5/5)(8/8)(6/−)(4/−)(6/6)
870 (2/2)(9/9)(7/−)(7/−)
871 (2/2)(5/5)(2/−)(9/9)
872 (2/2)(9/9)(5/5)(6/6)(1a/−)(5/−)
873 (2/2)(5/5)(6/6)(1b/−)(9/−)
874 (2/2)(5/5)(6/6)(8/8)(7/7)(9/9)(6/6)(6/−)
875 (8/−)
876 (8/−)(5/−)
877 (2/2)(7/−)
878 (4/−)(8/−)
879 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(5/−)(7/7)
880 (2/2)(5/5)(6/6)(5/−)(7/−)
881 (2/2)(5/5)(6/6)(8/8)(7/7)(6/−)(6/−)
882 (2/2)(9/9)(5/5)(8/8)(6/−)(6/6)(9/−)(9/−)

Appendix D

A small iptables HOWTO

This small HOWTO should help understanding the iptables examples in this thesis. For more information please refer to [And].

Every iptables rule looks like the following:
`iptables [-t table] command [match] [target/jump]`

In iptables, three different **tables** exist (nat, mangle, filter) of which we just use `filter` (the default) which is used for filtering rules. For every table there are a number of **built-in chains** (between 3 and 5 out of PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING) from which we only consider INPUT, FORWARD and OUTPUT which are the built-in chains of the filter table. The `INPUT` chain is for packets destined for the firewall (source → firewall), the `OUTPUT` chain is for packets originating from the firewall (firewall → destination), and the `FORWARD` chain is for packets passing the firewall (source → destination, via firewall). Other chains can be created by the user, but we will not say anything about this.

some commands:

	meaning	**example**
`-F`	flush (delete) all rules (from the given chain)	`iptables -F input`
`-X`	delete given chain (which must be empty) (if no argument given: delete all user-defined chains)	`iptables -X mychain`
`-P`	set given policy as default for given chain	`iptables -P INPUT DROP`
`-A`	append the following rule to the given chain	`iptables -A INPUT ...`

some matches:

match	**meaning** (matches packets ...)	**example**
`-p`	... with the given protocol (e.g. tcp, udp, icmp)	`iptables -p tcp ...`
`-s`	... with the given source IP	`iptables ... -s 129.132.178.26`
`-d`	... with the given destination IP	`iptables ... -d 127.0.0.1`
`--sport`	... with the given source port (only with '-p tcp' or '-p udp')	`iptables ... -p tcp --sport ssh ...`
`--dport`	... with the given destination port (only with '-p tcp' or '-p udp')	`iptables ... -p udp --dport 68 ...`
`--syn`	... with the syn flag set (only with '-p tcp')	`iptables ... -p tcp --syn ...`
`-m state --state`	... which are in the given state(s)	`iptables ... -m state --state NEW ...`

states (to be used with the state-match):

For every connection — identified by (source ip, destination ip, source port, destination port) — the corresponding state is remembered by the firewall.

state	**meaning**
NEW	as long as the connection is only one-way
ESTABLISHED	connections that have already seen packets in both directions
RELATED	new connections associated with an ESTABLISHED one (e.g. ftp data)
INVALID	most of the time faulty data or headers (should be rejected)

some targets:

There are a lot of possibilities to use targets/jumps. We only explain the simplest two:

target	**meaning**
`-j ACCEPT`	accept the packet (let it pass the firewall) and stop processing
`-j DROP`	drop the packet and stop processing

Bibliography

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. pages 159 – 192. ACM Press New York, NY, USA, June 1982.

[And] Oskar Andreasson. Iptables tutorial 1.1.19. `http://iptables-tutorial.frozentux.net/iptables-tutorial.html`.

[ASH03] Ehab Al-Shaer and Hazem Hamed. Management and translation of filtering security policies. In *Proc. 38th Int. Conf. Communications (ICC 2003), IEEE*, pages 256– 260, May 2003.

[Aud06] AuditMyPc.com. Firewall test. `http://www.auditmypc.com/firewall-test.asp`, 2006.

[BA82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. In *Acta Informatica*, volume 18, pages 31 – 45, November 1982.

[BCG+01] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A.V. Surendran, and D.M. Martin. Automatic management of network security policy. In *Proceedings of DISCEX II*, 2001.

[BCMG01] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–219, New York, NY, USA, 2001. ACM Press.

[BL73] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. MITRE Technical Report 2547, Volume I, March 1973.

[BMNW99] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[BMNW03] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. Technical report, Dept. Electrical Engineering Systems, Tel Aviv University, Ramat Aviv 69978 Israel, February 2003.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. In *IEEE Transactions on Software Engineering, Vol. SE-4, No 3*, pages 178–187, May 1978.

[CN98] S. Deering (Cisco) and R. Hinden (Nokia). RFC 2460: Internet protocol, version 6 (IPv6), December 1998.

[CVI89] Wendy Y. L. Chan, Son T. Vuong, and M. Robert Ito. An improved protocol test generation procedure based on UIOS. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 283–294, New York, NY, USA, 1989. ACM Press.

[Dij70] Edsger W. Dijkstra. Notes on structured programming. T.h.-report 70-wsk-03, Technological University Eindhoven, The Netherlands, August 1970.

[ea] Harald Welte et al. netfilter/iptables (ip_conntrack 2.1). `http://www.netfilter.org/`.

[EZ01] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, pages 100–107, Copenhagen, Denmark, November 2001. Technical Report IMM-TR-2001-14, Technical University of Denmark.

[FKSF01] M. Frantzen, F. Kerschbaum, E. Schultz, and S. Fahmy. A framework for understanding vulnerabilities in firewalls using a dataflow model of firewall internals. In *Computers & Security, vol. 20, no. 3*, pages 263–270, May 2001.

[Fra05] Markus Frauenfelder. Representation of a network. Semester thesis, Departement of Computer Science, ETH Zürich, 2005.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. volume 17, pages 591–603, Piscataway, NJ, USA, 1991. IEEE Press.

[GFi06] GFi. GFi LANguard network security scanner7. `http://www.gfi.com/lannetscan/`, 2006.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *IEEE Transactions on Software Engineering (TSE), Volume 1, Number 2*, pages 156–173, June 1975.

[Gil61] A. Gill. State-identification experiments in finite automata. In *Information and Control, vol. 4*, pages 132 – 154, 1961.

[Gil62] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[Gro97] Network Working Group. RFC 2196: Site security handbook, September 1997.

[Gut97] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *1997 IEEE Symposium on Security and Privacy*, pages 120–129, Oakland, CA, 1997. IEEE Computer Society Press.

[Haz00] Scott Hazelhurst. Algorithms for analysing firewall and router access lists. In *Proc. ICDSN*. Workshop on Dependable IP Systems and Platforms, June 2000.

[Hil05] Stefan Hildenbrand. Generation of test cases from automata. Semester thesis, Departement of Computer Science, ETH Zürich, 2005.

[How76] W.E. Howden. Reliability of the path analysis testing strategy. In *IEEE Transactions on Software Engineering, SE-2*, July 1976.

[HPL98] James Hoagland, Raju Pandey, and Karl Levitt. Security policy specification using a graphical approach. Technical report CSE-98-3, University of California, Davis Department of Computer Science, July 1998.

[ISI81a] University of Southern California Information Sciences Institute. RFC 791: Internet protocol, September 1981.

[ISI81b] University of Southern California Information Sciences Institute. RFC 793: Transmission control protocol, September 1981.

[ITU03] International Telecommunication Union ITU-T. H.323: Packet-based multimedia communications systems, July 2003.

[JW01] Jan Jürjens and Guido Wimmel. Specification-based testing of firewalls. In Andrei Ershov, editor, *4th International Conference Perspectives of System Informatics (PSI'01)*, LNCS. Springer, 2001.

[KFS+03] Seny Kamara, Sonia Fahmy, Eugene Schultz, Florian Kerschbaum, and Michael Frantzen. Analysis of vulnerabilities in internet firewalls. In *Computers and Security*, volume 22, pages 214–232, 2003.

[Ltd] Checkpoint Software Technologies Ltd. Checkpoint R55W. `http://www.checkpoint.com/`.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of The IEEE, Vol. 84, No. 8*, pages 1090 – 1123, August 1996.

[Ma04] Huibo Heidi Ma. Specification based firewall testing. Master's thesis, Texas State University-San Marcos, May 2004.

[Mea55] G.H. Mealy. Method for synthesizing sequential circuits. In *Bell System Technical Journal*, volume 34, pages 1045 – 1079, 1955.

[Mic] Microsoft. ISA server v4.0.2161.50. `http://www.microsoft.com/isaserver/default.mspx`.

[MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*, pages 177–187, May 2000.

[MWZ05] A. Mayer, A. Wool, and E. Ziskind. Offline firewall analysis. In *International Journal of Information Security*, pages 125–144, 2005.

[Mye04] Glenford J. Myers. *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc., 2004.

[oCS03] Polytechnic University Department of Computer and Information Science. Security policy. `http://cis.poly.edu/security-policy.html`, 2003.

[Org96] International Standardization Organization. ISO/IEC 7498-1: Information technology – open systems interconnection – basic reference model: The basic model (second edition), June 1996.

[oSN95] National Institute of Standards and Technology (NIST). An introduction to computer security: The NIST handbook, October 1995.

[RF] Susan H. Rodger and Thomas Finley. Jflap - java formal language and automata package. `http://www.jflap.org`.

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261 SIP: Session initiation protocol. `http://www.ietf.org/rfc/rfc3261.txt`, June 2002.

[SAN] SANS. The sans security policy project. `http://www.sans.org/resources/policies/`.

[Sch96] E. Schultz. How to perform effective firewall testing. In *Computer Security Journal, vol. 12, no. 1*, pages 47–54, 1996.

[Sch06] Adrian Schüpbach. Firewall testing with NAT. Semester thesis, Departement of Computer Science, ETH Zürich, 2006.

[SCR+96] Y. Rekhter (Cisco Systems), B. Moskowitz (Chrysler Corp.), D. Karrenberg (RIPE NCC), G. J. de Groot (RIPE NCC), and E. Lear (Silicon Graphics Inc.). RFC 1918: Address allocation for private internets, February 1996.

[SD88] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. In *Computer Networks and ISDN Systems 15*, pages 285–297, 1988.

[Sec06] Network Secure. Network-Standardverfahren. `http://www.network-secure.de/`, 2006.

[ST99] P. Srisuresh and M. Holdrege (Lucent Technologies). RFC 2663: IP network address translator (NAT) terminology and considerations, August 1999.

[Str06] Beat Strasser. Extending fwtest to handle udp and icmp. Semester thesis, Departement of Computer Science, ETH Zürich, 2006.

[Tan96] Andrew Tanenbaum. *Computer Networks, Third Edition*. Prentice-Hall International, 1996.

[Uni03] RMIT University. Security policy. `http://www.cs.rmit.edu.au/rules/computer-security.shtml`, 2003.

[UoI03] CS Department University of Idaho. Information assurance security plan. `http://www.cs.uidaho.edu/security.html`, 2003.

[Wal04] Jack Walsh. ICSA labs firewall testing: An in depth analysis, June 2004.

[Woo01] A. Wool. Architecting the lumeta firewall analyzer. In *Proceedings of the 10th USENIX Security Symposium*, pages 85–97, August 2001.

[WTS03] John Wack, Miles Tracy, and Murugiah Souppaya. Guideline on network security testing. NIST special publication 800-42, October 2003.

[Zau04] Gerry Zaugg. Firewall testing. Diploma thesis, Departement of Computer Science, ETH Zürich, 2004.

Index

Curriculum Vitae

Diana von Bidder - Senn
Forchstrasse 179, 8032 Zürich
diana.bidder@inf.ethz.ch

born 14.11.78
citizen of Zürich ZH, Obersiggenthal AG, Basel BS, Genève GE

Education

09.2003 – 04.2007	PhD in "Specification-based Firewall Testing"
10.2003 – 04.2006	Didaktikausweis (Höheres Lehramt) in Computer Science
10.1998 – 07.2003	Dipl. Informatik Ing. ETH
08.1993 – 01.1998	Matura Typus C, MNG Rämibühl, Zürich

Work Experience

09.2003 – 05.2007	research and teaching assistant, ETH Zürich
04.2002 – 09.2002	internship, open systems ag
10.2000 – 02.2003	teaching assistant (Hilfsassistent), ETH Zürich