Report

Specification-based firewall testing
Final project report for armasuisse

Author(s):
Bidder-Senn, Diana von

Publication Date:
2006

Permanent Link:
https://doi.org/10.3929/ethz-a-006782441

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Specification-based Firewall Testing

Diana von Bidder-Senn

final project report for armasuisse
ETH Technical Report 533
October 2006

Abstract

Firewalls are a central component in network security. Nowadays everybody is using firewalls, unfortunately without having good means for determining if they are accomplishing their job correctly. The aim of this project was to develop a specification-based firewall testing methodology. The phrase “specification-based” is important here: firewalls in a bank have to accomplish a different goal than those in a university, thus we cannot use the same tests. Or said otherwise, for every environment we have to generate tests separately, based on the security policy of the respective environment.

During the project, we designed a language to formally specify network security policies. Further we proposed a method for checking if a given firewall, which we treat as a black box, correctly implements such a policy. Thus only now it is possible to automatically check if the security one relies on really is in place.
Contents

1 Introduction .................................................. 5
   1.1 The Need for Firewalls .................................. 5
   1.2 Firewall in a Nutshell .................................. 5
   1.3 The Need for Firewall Testing .......................... 7
   1.4 State of the Art in Firewall Testing .................... 7
   1.5 Goal .................................................. 7
   1.6 Possible Approaches .................................... 8
   1.7 Overview of our Approach ............................... 9
   1.8 Summary of our Results ................................ 10
   1.9 Organisation of this Report ............................. 10

2 Background .................................................. 11
   2.1 Testing in General ...................................... 11
   2.2 Test Case Generation for Mealy Machines ............ 11
      2.2.1 General Introduction .............................. 11
      2.2.2 Verifying the State of a FSM ...................... 12
   2.3 Time of Generation: adaptive vs. preset ............. 14

3 Specification ................................................. 15
   3.1 Network Layout ........................................ 15
   3.2 Formal Policy and Keyword Definitions ............... 16

4 Test Methodology ........................................... 18
   4.1 The System under Test .................................. 18
   4.2 Test Case Generation (Technical Details) .......... 19
      4.2.1 Abstract Test Cases ............................... 19
      4.2.2 Test Tuples .................................... 19
      4.2.3 Concrete Test Cases .............................. 21
      4.2.4 Remarks ....................................... 21
   4.3 Practical Considerations ............................... 22
   4.4 Summary ............................................. 22

5 Endpoints versus Midpoints ............................... 24
   5.1 Motivation ............................................ 24
   5.2 Observations of Existing Systems .................... 24
   5.3 The Source of the Problem ............................ 28
   5.4 Construction .......................................... 29
      5.4.1 Setting ......................................... 29
      5.4.2 Idea .......................................... 30
      5.4.3 Technical Details ................................ 33
      5.4.4 Correctness .................................... 35
      5.4.5 Discussion ..................................... 39

6 Tools ......................................................... 40
   6.1 NetMate .............................................. 40
   6.2 HOL-TestGen++ ....................................... 40
   6.3 TCGTool ............................................... 40
   6.4 Glue .................................................. 41
      6.4.1 gen-abs-test ................................... 41
      6.4.2 gen-conc-test .................................. 41
6.5 ftest: ................................................................. 41
6.6 Analysis of the Result ........................................... 41

7 An Example Test Run .............................................. 43
7.1 Policy ............................................................... 43
7.2 Setup ............................................................... 43
  7.2.1 Firewall-Setup .................................................. 43
  7.2.2 Tester-Setup .................................................... 44
7.3 Test Case Generation ........................................... 45
  7.3.1 Generation of test tuples (approach 1) ...................... 45
  7.3.2 Generation of test tuples (approach 2) ...................... 46
  7.3.3 Generation of abstract test cases ......................... 48
  7.3.4 Generation of concrete test cases ......................... 51
7.4 Execution .......................................................... 52

8 Related Work ....................................................... 55
8.1 Security Policy .................................................. 55
8.2 Firewalls ........................................................ 55
  8.2.1 Product Testing ............................................... 55
  8.2.2 Penetration Testing ......................................... 55
  8.2.3 Analysis / Simulation ...................................... 55
  8.2.4 Specification-Based Testing .............................. 56
  8.2.5 Other .......................................................... 56
8.3 Other ............................................................... 57

9 Conclusion ........................................................ 58

10 Future Work ...................................................... 58

A Fundamentals / Background - some Definitions ................. 59
  A.1 (Security) Policy ............................................... 59
  A.2 Testing .......................................................... 59
  A.3 Formal Verification ........................................... 60
  A.4 Model Checking ................................................ 60
  A.5 Automata theory ............................................... 60

B Grammars .......................................................... 62
  B.1 EBNF ............................................................. 62
  B.2 General ........................................................ 62
  B.3 Abstract Firewall Rules ...................................... 63
  B.4 Network Layout ............................................... 63
  B.5 Formal Network Policy ....................................... 64

C A small iptables HOWTO ........................................ 65

D ftest v1.0 README ............................................... 67

E TCP End- and Midpoint Automata in CSP ....................... 76
  E.1 A short introduction to CSP and FDR ....................... 76
  E.2 TCP Automaton for Endpoints ............................... 77
    E.2.1 Specification [CSP] ....................................... 77
    E.2.2 Incorrect Specification ................................ 78
    E.2.3 iptables (in contrack 2,1) .............................. 78
List of Figures

1. TCP three-way-handshake .................................................. 5
2. An example of a firewall ruleset ........................................... 6
3. An example of an iptables ruleset ......................................... 6
4. Policy versus rules ............................................................ 8
5. Overview ............................................................................. 9
6. adaptive vs. preset test case generation .............................. 14
7. A typical network layout ..................................................... 15
8. A sample textual network layout ......................................... 16
9. A sample formal network policy ......................................... 17
10. Keyword definitions ............................................................ 17
11. System under Test ............................................................... 18
12. Policy for https with test points ......................................... 20
13. TCP specification for a Midpoint ......................................... 25
14. TCP automaton in iptables (ip, contrack 2.1) ...................... 26
15. TCP automaton in Checkpoint R35W .................................. 27
16. TCP automaton in ISA Server ............................................. 28
17. TCP Scenario 1 .................................................................. 28
18. TCP Scenario 2 .................................................................. 29
19. A message \( m \) from endpoint E0 to endpoint E1, forwarded by the midpoint \( M \) .......................... 30
20. A message \( m \) from endpoint E0 to endpoint E1, dropped by the midpoint \( M \) .......................... 30
21. A message \( m \) from endpoint E0 to endpoint E1, lost by the network ........................................ 30
22. Part of an Endpoint Automaton \( A_e \) .................................... 31
23. Two consecutive endpoint transitions .................................. 32
24. A transition in an Endpoint – a Midpoints view ................. 34
25. Our Tools – The Big Picture .................................................. 40
26. fwtest v0.3 ....................................................................... 41
27. fwtest v1.0 ....................................................................... 42
28. Demo – Formal Policy ........................................................... 43
29. Demo – Network Layout ....................................................... 43
30. Keyword definitions ............................................................ 43
31. Network Setup ................................................................... 44
32. Firewall Rules ................................................................... 45
33. Firewall Rules – Improved Version ..................................... 53
34. EBNF grammar ................................................................. 62
35. General non-terminals ........................................................ 62
36. Grammar of abstract firewall rules ..................................... 63
37. Grammar of logical firewall rules ....................................... 63
38. Grammar for textual network layout .................................. 63
39. Grammar for network policies ............................................ 64
40. Grammar for keyword definitions ...................................... 64
41. CSP Operators .................................................................. 76
42. Communication via a Firewall ............................................. 91
1 Introduction

1.1 The Need for Firewalls

We live in a world where private networks are connected to the Internet. As everybody can access the Internet, and there is no central entity controlling all of the Internet, a company has to protect its data from unauthorised access through the Internet. This is done by firewalls whose analogon in the physical world are doors. Everybody understands that buildings need to be locked to prevent unauthorised access. It is the same in the digital world: unauthorised access to a company’s network must be prevented, and this can be done by one or several firewalls. Why several firewalls? Let us explain this at the example of a bank. There we need one main door (door 1). This could lead to an anteroom containing an ATM. From there another door (door 2) could lead to the counter area, This is normally separated from the offices by another door (door 3) and we need even another door (door 4) to separate the vault from the offices. In this example it is clear why we need so many doors: because not everybody is (at all times) allowed to enter all rooms. Thus we have an access policy for our rooms which is implemented by the different doors. Such a policy could for example 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.

1.2 Firewall in a Nutshell

The computing world uses the term firewall for a hardware or software put between two (or more) networks to prevent communication forbidden by the network policy (see appendix A. for a definition).

A traditional firewall normally runs on a dedicated network device or computer positioned on the boundary of two or more networks. Such a firewall filters all traffic entering or leaving the connected networks. In the simplest case, filtering means deciding for every packet if it should be forwarded or blocked. More sophisticated filtering will also modify packets. Network address translation (NAT), for example, can be done by firewalls. There, the private addresses (as defined in RFC 1918) of hosts behind a firewall are translated to routable addresses.

One can distinguish between two major categories of firewalls:

- **network layer firewalls**, also called **packet filters**, which work at the TCP-level (and UDP)
- **application layer firewalls**, which work at the application level (and thus “understand” the content of traffic)

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

When speaking of packet filters, two types have to be distinguished: **stateless** and **stateful packet filters**. Hereby stateless and stateful refer to the kind of protocol inspection made by the packet filter, Let us explain the difference between the two at the example of the stateful TCP protocol (UDP is stateless). If

<table>
<thead>
<tr>
<th>Alice</th>
<th>message sent</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>—</td>
<td>LISTEN</td>
</tr>
<tr>
<td>SYN-SENT</td>
<td>← SYN</td>
<td>→ SYN-RECEIVED</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>← SYN &amp; ACK</td>
<td>→ ESTABLISHED</td>
</tr>
</tbody>
</table>

Figure 1: TCP three-way-handshake
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 \[1\] on the preceding page. We can see the protocol states of the endpoints in the figure. Having a stateless packet filter, every 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 (i.e., the state of the connection \[2\]). Thus with stateless packet filters we only have the possibility to allow a TCP connection between some endpoints A and B in both directions, whereas with stateful packet filters we can restrict it to one direction, Restricting a connection to one direction here means that connection initiation is only allowed from one endpoint to the other. Restricting the connection initiation from A to B (i.e., B can only communicate with A if A starts the connection) can be done by specifying one rule for allowing packets with the SYN flag set only from A to B, and another rule allowing packets between A and B for any non-new connections (i.e., at least a SYN-packet has been seen). The consequences are that with stateless packet filters more packets have to be allowed than with stateful packet filters. This means that stateful packet filters are in general more secure than stateless packet-filters. Due to this, stateless packet filters are not any longer used in practice, and we therefore do not consider them in our work.

What is important is to understand the difference between the firewall configuration, by a so-called ruleset, and the firewall implementation. The firewall implementation is the soft- or hardware delivered by the firewall vendor. The firewall configuration consists of rules which tell the firewall what to do. So, if a company buys three firewalls from the same vendor, the implementation will be the same for all of them, but the configuration most probably will be different. 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. Faith in misconfigured firewalls is misplaced indeed.

<table>
<thead>
<tr>
<th>source</th>
<th>destination</th>
<th>protocol</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>129.132.11.2</td>
<td>129.132.178.26</td>
<td>imap</td>
<td>accept</td>
</tr>
<tr>
<td>129.132.178.26</td>
<td>129.132.11.2</td>
<td>smtp</td>
<td>accept</td>
</tr>
<tr>
<td>...</td>
<td>any</td>
<td>any</td>
<td>deny</td>
</tr>
</tbody>
</table>

Figure 2: An example of a firewall ruleset

```
# delete everything
/sbin/iptables -F
/sbin/iptables -X

# allow all outgoing
/sbin/iptables -P OUTPUT ACCEPT

# allow only incoming ssh (and related or established connections)
/sbin/iptables -P INPUT DROP
/sbin/iptables -A INPUT -p tcp --syn --dport ssh -j ACCEPT
/sbin/iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

# allow connections from localhost to localhost
/sbin/iptables -A INPUT -d 127.0.0.1 -j ACCEPT
```

Figure 3: An example of an iptables ruleset

\(^1\)It does this by having an internal automaton for the TCP protocol.
Two examples of a ruleset for a packet filter are shown in figures 2 and 3. The problem is that every firewall hard-/software has its own rule-language. Also, important details like the order of rule-matching (first vs. last rule that matches is used) differ between the different firewalls. Sometimes there are also “hidden” default rules one might not want. All this makes the task of proper firewall configuration even more difficult and is a prominent source of errors.

1.3 The Need for Firewall Testing

Returning to the analogon of the door, everybody understands that it is not enough to have a door. A door only does its job if it is locked properly and if only authorised people have a key to unlock it. Similarly, it is not enough to have a firewall. We can only be satisfied if the firewall is doing what we expect from it.

An important point is that we have to be clear about our expectations. This sounds simple, but looking at current practice we see that this point does not get enough attention. Most people think they know the expectations, by having their own view (in their mind) of the situation. But the expectations are only clear if they are written down, in a so-called policy, and acknowledged. People often think they speak of the same thing until they write it down and realise that they meant completely different things.

Only if there is a policy, its coverage can clearly be analysed, and its implementation by the firewall be tested. This is important, as a policy without proper implementation is worth nothing. Thus the proper implementation of the policy by the firewall needs to be tested. Or said in the door analogon: We have to test if only authorised people can enter the door. This would, for example, not be satisfied if the door is left wide open.

1.4 State of the Art in Firewall Testing

This is just a short overview, for more detailed information on current work in this area please refer to section 8.

Currently, Firewall testing is mostly penetration testing. Penetration testing consists of scanning all the ports to find out which ones are open. It also includes running known attacks against the firewall to see if the firewall is vulnerable to them. The problems with penetration testing are various, 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) In a world where new attacks occur every day, resistance against known attacks is only part of the story. The more important part – if a firewall would resist new attacks – cannot be tested by penetration testing, 3) With penetration testing we cannot find out if a firewall does what we want it to do, we can just test it for resistance against (known) attacks. Let us take the real world analogue once more. When nobody has a key to a door, we can be sure that no unauthorised persons can enter the room. In this example we would do penetration testing by sending a lot of unauthorised persons and checking if they can enter the room, This will not be the case and we would say the room is secure. But in this setting also all the authorised persons cannot enter the room. And that is definitely not what we want.

In today’s world, where more and more depends on the internet, the shortcomings of penetration testing are simply too many. There is the urgent need for a method which is more powerful than penetration testing and less time-consuming (and error-prone) than the analysis of firewall rules by hand.

1.5 Goal

As mentioned in section 1.4 penetration testing needs to be replaced by a more powerful, less time-consuming (at least in time spent by experts) method. The goal of this project is to design such a

2Please refer to Appendix C for details on the iptables firewall, which is the Linux default.
method. In more detail, a method shall be designed which can, as automatic as possible, test several
firewalls for conformance to a given policy. The ‘several’ 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.

The important point is that it is not the goal to find vulnerabilities but to test for conformance. If
the policy for example 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.

1.6 Possible Approaches

It is very important that firewalls actually implement a stated security policy. Figure 4 shows five different
approaches of how this could be tested.

![Policy versus rules diagram]

Figure 4: Policy versus rules

Let us first explain the different levels / terms used, before explaining the different approaches. 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 (as explained in section 1.2), whereas the term abstract firewall rules is used to denote
firewall configuration in a vendor-independent format. Finally, the term firewall is here used to denote
the behaviour of the firewall software / hardware implementing given firewall rules. Thus the firewall is
“equivalent” to the firewall rules, if the firewall implementation is correct.

Now let us talk about the different approaches. They 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.
5. generate test cases from the formal policy and run them against the actual firewall.

As we want to show the equivalence of the formal policy and the actual firewall rules, approaches 2 and
3 can only be used together,

As actual firewall rules are vendor- and release-dependent, all approaches using the actual firewall
rules would need considerable (re-)engineering. This means that this approach would only be useful if
the reengineering is repeated every time a new firewall software (or release) is released. This is not very practical and also highly error-prone.

The 5th approach shown is the most general, as it handles the firewall as black box, it can be used independently of the kind of firewall installed. Also it is the only approach that gives us the possibility of finding bugs in the firewall implementation.

As the 5th approach is the most general, and the only one really testing conformance at the level it is needed, 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 Subsection.

1.7 Overview of our Approach

Figure 5 shows our approach to firewall testing and will be explained in this section.

Figure 5: Overview

Most companies do not have a security policy, and if they have one it is normally informal and can therefore not be used to reason about it in a formal manner. If we want to determine if a firewall really implements a stated security policy, at least the network policy (which is part of the security policy) should be formally stated. This formal network policy (formal policy) should specify what shall be allowed (but not how this should be done).

If we now want to test some firewalls against a given formal network policy, we need to know some low-level details. If our formal network policy for example states that only secure connections are allowed from the Internet to our corporate network, we have to define what secure connections are (keyword definitions) and how our network looks like (network layout). This low-level information may be stored separate from the policy.

Together the formal network policy, the network layout and the keyword definitions provide all the information we need for automatically generating test cases. These test cases can then be run directly on the network. If a test outcome is not as expected then there are three possible sources for the error: either the firewall rules do not correspond to the formal network policy (in this case, one has to decide if the rules are wrong or if the policy is wrong), there is a bug in the firewall implementation (the firewall is not doing what it is told by the firewall rules), packets were changed or lost due to heavy network traffic, or our test environment has bugs.

---

3 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)?
1.8 Summary of our Results

During the time of the project we designed a language for the formal specification of security policies, we developed a method for the automatic generation of test cases from such a policy and formally analysed the job of a firewall, Furthermore we implemented a prototype tool suite for the validation of our approach.

As most important future work we see conducting case studies with industry to get feedback, and extending the approach to application level firewalls.

1.9 Organisation of this Report

First we give some testing basics, needed to understand our work, in section 2. We then start by defining Network Layout, Keyword Definitions and Formal Policy, introduced above, in section 3. The actual aim of our work, the test methodology, is then defined in section 4, before discussing an important problem, namely how a firewall should handle some protocol, which we faced in section 5. We then go over to explain how all this is implemented in several tools in section 6, before showing an example test run in section 7. To finish the report, we provide a comparison to related work in section 8, before drawing conclusions in section 9 and pointing out some possible future directions in section 10.
2 Background

In this section we want to elaborate on some basic concepts on which we build our work. Even more fundamentals can be found in Appendix A.

2.1 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 (code-based) and black box (specification-based) testing. They differ in how *test cases*, consisting of *test data/input* and expected test output / results, are generated. With white box testing, test cases are generated from the code, while with black box testing they are generated from specifications. Therefore white box testing tests things like if all the code is executed or if copies of the code with inserted faults can be distinguished from the original. Black box testing tests if the code implements the specification. For more details please refer to [Unit](#). We are doing black box testing.

The difficult part in (automated) testing is to find good test cases. They must be reliable, adequate, and valid. Some of the first definitions in this direction can be found in a paper by J. Goodenough and S. Gerhart from 1975 [CG75]. 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 must be *reliable* and *valid*. In short, to be reliable C must insure selection of tests that are consistent in their ability to reveal errors, as opposed to necessarily being able to detect all errors. 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 criteria C, if C is reliable and valid. A test set is called successful if the result of every test is as expected. This leads to the fundamental theorem of testing, which states: 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)** [How79]: If P is a program to implement function F on domain D, then a test set $T \subseteq D$ is reliable for P and F if $\forall t \in T$, $P(t) = F(t)$, then a test set $T \subseteq D$ is reliable for P and F if $\forall t \in T$, $P(t) = F(t)$.

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

2.2 Test Case Generation for Mealy Machines

A Mealy Machine / Automaton is a Finite State Machine / Automaton (FSM) with output. For a precise definition please refer to Appendix A.5.

2.2.1 General Introduction

There exist many methods to generate test cases for automatons. We want to generate test cases for Machine Verification / Conformance Testing, i.e. we wish to test whether an implementation machine P conforms (is equivalent) to the specification machine F. Two 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 will produce
the same output sequence. The following assumptions are usually made in conformance testing:

- Specification machine $F$ (its transition diagram) is strongly connected.\footnote{Said in \textit{other} words: If we have two\,completely specified, minimal automata with the same number of states, and them all being equal (i.e. having the same outgoing transitions), they must be the same.}
- Machine $F$ is reduced.
- Implementation machine $P$ does not change during the experiment and has the same input alphabet as $F$.
- Machine $P$ has no more states than $F$.

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 $F$ is correctly implemented in FSM $P$. This is achieved as follows, for every transition of $F$, say from state $s_i$ to state $s_j$, do the following:

1) Bring machine $P$ to the initial state $s_i$.
2) Transfer machine $P$ 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 $s_j$.

\textbf{Step one} is easy if there is a reliable reset: Just apply the reset input to go back to the initial state.
\textbf{Steps two and three} can be solved by building a \textit{test tree} $T$ according to the following rules\footnote{A machine $M$ is strongly connected if for any two states $s$ and $s'$ of $M$, $s'$ is reachable from $s$.} and then walking along all the paths:

a) Label the root of $T$ with the initial state of $F$. 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 $F_i$ denote its label. If on input $x$, machine $F$ goes from state $F_i$ to state $F_j$, we attach a branch and a successor node to the node labeled $F_i$ in $T$.

The branch and the successor node are labeled with $x$ and $F_j$, respectively.

\textbf{Step four} is the one where the test methods differ. In the following we will present the most common methods, namely \textit{distinguishing sequences} \cite{Gil61, Gil62}, the \textit{W-method} \cite{Cho78}, \textit{UIO sequences} \cite{SD88}, the \textit{partial W method} \cite{Fy91}, and the \textit{UIOv method} \cite{CV89}. For more information about these methods please refer to the original papers or to the overview paper of Lee and Yannakakis \cite{LY96}.

### 2.2.2 Verifying the State of a FSM

\textbf{W-Method} \cite{Cho78}

Assumptions on the FSMs: completely specified, minimal, start with a fixed initial state, every state is reachable.

The W-method uses the \textit{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:

$$\forall (s_i, s_j) \neq j \exists x \in W : \lambda(s_i, x) \neq \lambda(s_j, x)$$

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
\[ Z = W \cup X \cdot W \cup \cdots \cup X^{m-n} \cdot W \quad X = \text{input alphabet, } n = \text{number of states of the specification}
\]

Instead of \( W \), the W-Method will find all faults (operation errors, transfer errors, extra states) in the implementation if it has at most \( m \) states.

For the sake of completeness: all partial paths (including \{\}) in the testing tree (used for steps two and three) are called the P set in the W-Method.

**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. It consists of two phases:

Phase 1: \( Q \cdot Z \quad Q = \text{state cover set, } Z = \text{see W-Method} \)

Phase 2: \( R \cap W = \bigcup_{p \in R} \{p\} \cdot W \quad R = P \setminus Q, P = \text{see W-Method} \)

**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 (input for which no transition exists at the current state) 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:

\[ \forall s_i \neq s : \lambda(s_i, x) \neq \lambda(s, x) \]

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 [CV189]**

Assumptions on the FSMs: as with UIO plus completely specified

The problem of the UIO method is solved by the UIOv method, by adding a verification step \( \sim \text{Uv} \) before the testing. The idea of \( \sim \text{Uv} \) 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 [Gh61, Gh62]**

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:

\[ \lambda(s_i, x) \neq \lambda(s_j, x) \text{ for every pair of states } s_i, s_j, \text{ where } i \neq j \]

Therefore DSs can also be used for state identification and a reliable reset is not required. The problem is that not every FSM has a DS.

**Comparison of the methods**

The nice thing about the UIO method is that it has shorter test cases than the W and the DS method. Also UIO sequences nearly always exist (and if they do not, a special signature can be used instead). So, 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 from the Wp method, and the DS method is a special case from 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 as the W method but with shorter test sequences, and as the UIO
method has no full fault coverage, we think it is in general the best to choose the Wp method. In special cases where we have automata without reset, the DS method can be tried.

### 2.3 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. Both approaches have advantages and disadvantages (for an overview see Figure 6, which we like to give in this section.

<table>
<thead>
<tr>
<th>preset experiment</th>
<th>advantages</th>
<th>disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(determined in advance)</td>
<td>• generation not time critical</td>
<td>• many test cases needed (for all possible reactions)</td>
</tr>
<tr>
<td></td>
<td>• easy generation</td>
<td>• no correction possible (during testing)</td>
</tr>
<tr>
<td></td>
<td>• generate once, use several times</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• interleaving of test cases can be computed in advance</td>
<td></td>
</tr>
<tr>
<td>adaptive experiment</td>
<td>• corrections possible (timing, ...)</td>
<td>• longer testing time</td>
</tr>
<tr>
<td>(on the fly test case generation)</td>
<td>• less test cases (test cases are based on current knowledge)</td>
<td>• storage overhead (all possible scenarios have to be carried along)</td>
</tr>
</tbody>
</table>
3 Specification

3.1 Network Layout

![Network Diagram](image)

Figure 7: A typical network layout

A typical network with firewalls can be found in figure[4] It consists of the Internet, a demilitarized zone (DMZ) and an Intranet. 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 important / confidential data) and network infrastructure from unauthorised outside access.

To be able to test if a formal policy is correctly implemented by some firewalls, we need some information about the firewalls under test and the networks adjacent to them. Or in more detail, we have to know where the firewalls are, what capabilities they have, and which networks are attached to them on which interface. Also we have to know where the servers are located and what services they offer (this is only for servers within our network, and the servers on the Internet we actively use). We do not need any specific knowledge about the clients — we are not interested which IP’s in a network are used and which are not. As this information is very low-level, frequently changing and only needed for the actual crafting of test packets, it should be specified separate from the policy. We called this specification the network layout.

There are different ways to gather this information: discovery⁶ and configuration. We do not have preferences in how to learn this data. Our testing methodology is independent of how this information is gathered. It only has to be valid and (preferably) complete.

For the network in figure[4] the result could be saved (textually) as given in figure[8] on the next page (for a grammar see appendix[2]). For the automatic generation of test cases we need such a textual representation. But humans understand graphical representations better. Therefore it would be the best to use a graphical network representation, for example as given in figure[7] for the users to specify their formal policy. This can then be converted to a textual one by a tool before being used for test generation. For this conversion, a graphical network layout to a textual one and vice versa, a tool called NetMap⁷ was developed by Markus Frauenfelder [Fra95]. Unfortunately the possibility of directly specifying a formal policy in the tool has not yet been implemented.

---

⁶To gather this data by discovery, the UNIX-commands /sbin/ifconfig and /sbin/route can be used on each firewall.
⁷For more information see Section 6
3.2 Formal Policy and Keyword Definitions

Note: The grammars for formal (network) policies and keyword definitions, explained in this section, can be found in figures 39 on page 62 and 40 on page 63 (Appendix B) respectively.

A security policy should express what should be achieved and not how it should be achieved. Therefore low-level (implementation) details do not belong in a security policy. A security policy should not change too often. A security policy consists of different parts (see appendix A.D). The focus of our work is only the network policy. All the other parts are outside of the scope and can be kept informal. There are three different ways how network policies are expressed in practice:

1) Informal: just some text.

2) Access models: Who (machine), on what (machine), how (allowed access mode).

3) Firewall rules: Which protocols, from where, to where, action (accept, reject,...).

Our aim is to test if a given network policy is correctly implemented. To do that, it must be possible to formally reason about the given policy. Therefore the network policy must be stated in a formal specification language. As there is no such language, we have to define our own.

The problem we face is that the demands on such a language are not known to us: The few companies using security policies do not want to show them 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.

So we decided to start with a simple policy language we feel accurate. Our hope is that it is possible to get comments on the suitability of this language at a time where users can be given a prototype tool, for firewall conformance testing, using this language as input. These comments could then be used to find out about the needs of the people and to improve the language.

So, what has to be specified in a policy? For every pair of subnetworks we need to know how they should communicate. For the network given in figure 7 on the page before, this could look like the following (which is the form we chose): A sample policy is shown in figure 10 on the facing page.

The “keywords” used (webtraffic, ...) help keep the policy high-level. These keywords can then be defined separately (see figure 11 on the next page for some example) and changes can be made in these keyword definitions instead of in the policy. This helps us achieve a high-level stable policy. The same arguments apply to the network details. Their specification was already explained in section 3.1. Please note that it depends on the testing level (packet or application level) of how a protocol in the keyword definitions is
3 SPECIFICATION 3.2 Formal Policy and Keyword Definitions

@ Connections to Private
Private → Private: — (not enforceable by a firewall — can be specified informal)
DMZ → Private: ACCEPT securetraffic
Internet → Private: DENY *

@ Connections to the DMZ
* → Webserver: ACCEPT webtraffic
* → Mailserver: ACCEPT mailtraffic
DMZ → DMZ: — (not enforceable by a firewall — can be specified informal)

@ Connections to the Internet
Private → Internet: ACCEPT *
DMZ → Internet: DENY *
Internet → Internet: — (not our business)

Figure 9: A sample formal network policy

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

Figure 10: Keyword definitions

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.
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 find out what they are all about)
• to make it stable.

Why has a policy to be stable? If you look at the example in figure there is a rule DMZ → Private: ACCEPT securetraffic. This means that we only want to allow traffic from the DMZ to our private network which is secure. What we consider as secure can change every hour, for example when new protocol flaws or attacks are found, but our overall goal will stay the same.

To illustrate this important point, let us give an example from a related field. Which of the following rules would you prefer?
‘‘Only company employees are allowed to possess a key to the company’s main entrance.’’
or ‘‘Only Fred, Bob, ... are allowed to possess a key to the company’s main entrance.’’,
where Fred, Bob, ... are all employees of the company.
We think everybody would prefer the first one 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 not having to know anything about this policy).

The same arguments apply to our formal policy and the keyword definitions,
4 Test Methodology

Recalling figure 3 on page 9 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 the last section, we now want to focus on the main part, the generation.

4.1 The System under Test

![System under Test Diagram]

Figure 11: System under Test

Before giving a test methodology, we first have to state how our system under test looks like. We will do this for the example of the smallest possible system, which is given in figure 4. In this example we have two networks with one machine each, Alice and Bob respectively, and a firewall. In this setting, 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). Filtering means deciding whether or not each network packet is allowed to cross the boundary between the two networks. But how does the firewall decide if some packet is allowed to pass it? In short, it is allowed if it belongs to a connection which is allowed by the rules. But to really explain what this means, we have to switch to a lower level.

At a low-level, a connection between Alice and Bob consists of several consecutive network packets having Alice as source and Bob as destination or vice-versa. Hereby the source of the first packet is called the initiator of the connection, whereas the other principal is called the responder. The order of the packets in a connection is determined by the protocol automata of the endpoints. These endpoint automata, as we will call them, are Mealy machines. In short, a Mealy machine is an automaton taking inputs and returning outputs based on its current state.

To test this system, at a high-level means to simulate connections between Alice and Bob to see if the firewall correctly implements the policy. At a low-level this means that we have to simulate the endpoint automata. Thus we need 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.

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

1. are the protocol automata of the firewall correct?
2. are only connections allowed by the policy accepted by the firewall?

---

8 In practice, most protocols are two-party protocols, but also protocols with three and more parties exist.
9 For a complete definition of Mealy machines, please refer to appendix A.
10 i.e. the firewall lets all the connections pass which are accepted by the policy, and blocks all other connections.
11 Actually we simulate the firewall’s behaviour in a protocol run between two endpoints.
3. are all connections allowed by the policy accepted?

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 abstract test cases from these is explained in section 4.2.1.

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. This results in what we call concrete test cases. How test tuples are generated is shown in section 4.2.2.

4.2 Test Case Generation (Technical Details)

4.2.1 Abstract Test Cases

The behaviour of a Firewall can be described in terms of input and output: A packet reaches the firewall (input) and is either forwarded (output) or dropped (no output) by the firewall. Such a behaviour can be directly specified using a Mealy machine / automaton.\(^{12}\)

If, for every protocol \( p \), we specify the expected behaviour of the firewall in a Mealy machine \( M_{Mp} \) – we will call this midpoint protocol automaton from now on – we can then generate abstract test cases (testing the correct handling of protocol \( p \) by a firewall) by using the WP-method described in section 2.2.

We will later see that it is not a priori clear how such a midpoint protocol automaton for some protocol \( p \) should look like. We will investigate this problem in section 3 and present a solution there.

4.2.2 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.\(^{14}\) 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 \( DROP \). This means that the test tuples are policy-specific and thus must be generated for every policy. In this section we will present two approaches for the generation of test tuples. Please note that the statefulness of a connection is not modelled by these test tuples, but rather by the abstract test cases.

**Approach 1**

We generate test tuples in two steps. First 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

\[
\text{source} \rightarrow \text{destination: action keyword}
\]

from the formal policy into \( n \) low-level rules, where \( n \) is the number of protocols named in the keyword definitions. In these low-level rules, the names of \textit{source} and \textit{destination} are replaced with the corresponding IP ranges.

\(^{12}\)At the moment we do not look at the case where a packet is modified by the firewall, which for example is the case when using NAT, before being forwarded.

\(^{13}\)An introduction to Mealy machines is given in section A.5.

\(^{14}\)We do not consider other firewall actions, like sending ICMP error codes, here.
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 dIPr protocol action

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). In a second step, we choose our test tuples from these 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 9 on page 16, the network layout in figure 8 on page 16, the keyword definitions in figure 10 on page 17, and the https protocol) is given in figure 12.

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 (see figure 12 for an example). 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 uncoloured areas there exists no explicit policy statement. Note that a part of the yet uncoloured area is not testable since, as we stated earlier, policies for traffic within zones cannot be enforced by firewalls (this part is marked in yellow). But the rest of the uncoloured areas can be partitioned into rectangles and then test tuples can be chosen,
analagous to the procedure given above, where the expectation is set to DROP.

Approach 2

HOL-TestGen [BW] is a tool, built from members of our group, for the interactive generation of test cases (see [BW05] for more information). It was primarily used in Software Testing. The idea behind this second approach was to extend HOL-TestGen to be able to generating test tuples for our scenario.

This extension was done by Lukas Brügger in his semester thesis [Br06]. It consists of defining how TCP packets, networks and policies look like and what the functioning of a firewall is. This knowledge is stored in a so-called theory. Thus we only have to model the setting – policy, network – when we want to generate test data, All the rest is already done.

The benefits of this approach, with respect to the first one, are that it is automated and that it is based on a generic, found underlying theory. Unfortunately there are some shortcomings here. One is that HOL-TestGen always chooses small numbers, But IPs like 1.6.0.9 and 2.6.2.5 may not be a good representation for the Internet. Another is that even more firewall-specific knowledge is needed: In a small case study, only 12 from 288 generated test cases represented connections across the firewall under test. This means that HOL-TestGen must be learnt the fact that the policy can only be enforced between networks. Thus at the moment we get the better test cases with the first approach, and we are even faster.

4.2.3 Concrete Test Cases

In the last two Subsections, 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 and thereby generate concrete test cases that test if the policy is correctly implemented in a stateful manner. 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 == DENY then replace the expected output in every a_i with “-”.

The resulting test data represents network packets. These packets can then be built and injected into the actual network and the results can be compared to the expectations of the given test cases. Please refer to section 4 for a description of futest which we implemented for this purpose.

4.2.4 Remarks

For the rest of this report we will restrict ourselves to the packet level. 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. As the complexity at packet level is immense, and exponentially increases when switching to application level, we concentrated our efforts on the packet level.

In the following we will give some level-specific remarks to extend the general explanations of this section.

\[\text{More precisely, we will just look at TCP from now on. But as UDP (stateless) is much simpler than TCP (stateful), all the results can be applied to UDP as well.}\]
4.3 Practical Considerations

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.

application level
If we have got application level firewalls, answering the questions given in the introduction gets far more complex, as we have to test the application data as well. i.e. we now have to check things like are http connections from A to B not containing the words x, y, z able to pass the firewall and not just checking if any http connection between these endpoints can pass.

To get the complexity under control, we need to be able to specify at least the following things:

• contents, their possible values, and their ordering
• 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.

This means that our formal policy has to be adapted. It also means that we need a way to be able to describe the packet structure of a protocol, to be able to correctly generate test packets.

4.3 Practical Considerations

Preset test cases are easier to construct than adaptive ones (for a definition please refer to section 2.3), however in our setting it is not possible to get them a 100% correct. For example the correct acknowledgement number of a packet is known only at the time the packet is sent, because it depends on which previous packets reached their destination. On the other hand adaptive test cases would create a too big overhead, as we would need an automaton for every connection we are testing. Thus, we decided to take the best of both approaches.

• pregenerate test cases (preset) as far as possible
• set sequence and acknowledgement number on the fly (adaptive)
• when using NAT: set port (and IP) after having seen the first packet (adaptive)
• stop when expectation is wrong

4.4 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 one hand and test tuples are generated for the policy on the other hand. The abstract test cases for a protocol

\[^{16}\text{Unfortunately, as you can see in section 6, the adaptive part is not yet implemented.}\]

\[^{17}\text{It could be useful to repeat such a test case with different timings, e.g. longer breaks between the individual packets, to see if there is another reaction.}\]
represent correct (and incorrect) runs of that protocol. They only need to be generated once. The test
tuples represent the policy in the form of connections to be tested and thus need to be regenerated after
every change of the policy.

When instantiating the abstract test cases with the test tuples in the second phase, we get concrete test
cases specifying test packets that can directly be fed into the network to test if the policy is implemented
correctly by the firewalls. If the outcome of a test is not as expected, this can have several causes: Either
the firewall rules are incorrect, the firewall implementation is incorrect, our method is incorrect or different
network problems exist. Searching the cause of an unexpected test outcome has, at least for the moment,
to be done by hand. But compared to the amount of work without our approach, this is only a small part
left. syn keyword GoodWord transparent protocol syn keyword GoodWord transparent protocol
5 Endpoints versus Midpoints

This section examines the differences between endpoints and midpoints. We use the terms endpoint and midpoint as follows: An endpoint is an agent executing one part of a communication protocol. A midpoint is an agent observing or filtering traffic between two or more endpoints.

5.1 Motivation

A midpoint in its simplest form – forwarding traffic to the correct destination – is straightforward to implement. But as soon as stateful filtering comes into play, a midpoint needs to 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 handle the given protocol. This is due to the inability of midpoints to correctly track the correct protocol states of endpoints (see section 5.3 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 certain protocol, which means that there is no right and wrong. But if we want to test a midpoint, we need one specification to test against.

The implications of the lack of protocol specifications for midpoints are that manufacturers of midpoints have no guidelines on how they should implement a protocol. In practice, midpoint manufacturers implement the same protocol differently, based on how they think the midpoint should handle the endpoint data. This implementation is then, over the years, adapted to the practical experience gained using it. To show how this looks in practice, we give the TCP automata of three different firewalls in section 5.2 and discuss the differences. We then give, in section 5.3, several examples of situations, using the TCP protocol, where the correct behaviour of a firewall (as midpoint) cannot be defined from the TCP specification for endpoints.

We show how it is possible to generate midpoint specifications from endpoint specifications systematically. We propose an algorithm that, given protocol automata for the endpoints, generates a protocol automaton for the midpoint. We also show how such an automaton can be modified, in a clearly defined way, to suit different user needs.

5.2 Observations of Existing Systems

We wanted to see how substantial the differences between TCP automata of different firewalls are, due to the lack of specification. We started by writing down our expectations of such an automaton. These expectations, in the form of a Mealy machine, can be found in figure 18 on the next page. The figure can be read as follows: The start is in state NEW. If the firewall now sees a packet, from an agent A to another agent B, with the SYN-flag set, it forwards this packet and changes its state to SYN. A. Note that x / - means that packet x is dropped by the firewall, and y & z means that both the y- and the z-flag are set.

As firewalls we took three commonly used ones: Checkpoint R55/W,19 netfilter/iptables (ip_conntrack 2.1)20 and ISA Server v4.0.2161.50.21. We reverse-engineered their TCP automata with the help of ftest v0.6, Zanzu, and test-cases based on our specification (figure 18 on the facing page). The results are given in figure 19 on page 34, figure 16 on page 37 and figure 10 on page 38, with the differences to our specification marked in red.

19 For a formal definition of Mealy machines please refer to appendix A.
20 http://www.checkpoint.com/support/technical/documents/docs_r55w.html
21 http://www.netfilter.org/
22 http://www.microsoft.com/isaserver/default.mspx
5 ENDPOINTS VERSUS MIDPOINTS

5.2 Observations of Existing Systems

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

Now, we look into the differences between these three TCP automata and also between these automata and our expectation. It may look as there are not many differences, but in a security-critical device, every difference (i.e., possible fault) is one too much. Also, we have not taken into account sequence numbers and fragmentation, where we could have found other differences.

The strictest firewall was Checkpoint.

On one hand Checkpoint R55W blocks more packets than the other two firewall. On the other hand, some packets are already blocked by an attack defence tool, called SmartDefense, before actually reaching the TCP automaton of the firewall.

A ‘clean’ three-way-handshake is not enforced.

For initiating a TCP connection, the so-called three-way-handshake is used (see figure 11 on page 25). So let us assume the firewall has accepted a SYN from an agent A to another agent B. If there is now a SYN & ACK from B to A, then everything works as expected, the packet should be let through and the
firewall should go to the next state. If there is another SYN from A to B then this will be a retransmission (it could be that the first SYN was lost between the firewall and B) and should be allowed as well. If there is a RST from B to A then B does not want this connection, the packet should be let through and the TCP automaton 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. The most unexplainable to us is why a FIN from B to A is allowed during connection initiation as done in netfilter. There is no connection to be closed, if B does not want to have the connection then it should send a RST.

Netfilter allows SYN&FIN for connection initiation.

SYN is intended for connection initiation, FIN is intended for connection tear down, sending them together makes no sense. It is unclear why netfilter allows this packet to initiate a new connection. All we can say is that the netfilter team is aware of doing it (it is stated in the manpage) but until now we got no explanation on why this is done.\footnote{Harald Weite from the netfilter team promised us on 8/2005 to discuss this issue with his colleagues and tell us the cause for it.}

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 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 allows SYN & ACK all the time, which is not much better.

Consider the following, rare, scenario: A was rebooted, does not know of his previous unclosed connection to B, and now wants to initiate a connection (with the same source port) to B. If the firewall does not allow this SYN because it still thinks that there is an established connection from A to B and
Therefore a SYN should not occur (and therefore drops the SYN), then a legitimate connection cannot be initiated.

This scenario can be taken as explanation why, in practice, \textit{syn} : \textit{A} → \textit{B} and \textit{syn\&ack} : \textit{B} → \textit{A} are accepted for already established connections as well. But accepting \textit{syn} : \textit{B} → \textit{A} as netfilter does, does not make any sense and may be a security problem.

\textbf{After a FIN, data from both sides is still accepted.}

If \textit{A} sends a FIN to \textit{B} then this means that \textit{A} wants to close the connection. After this FIN, \textit{B} is still allowed to send data, but \textit{A} is not, except for the ACK belonging to \textit{B}'s FIN (actually it would not make sense to send data after having requested to close the connection). As packets may not arrive in their correct ordering at the firewall, the firewall cannot just drop all packets from \textit{A} after having seen a FIN. But the firewall should just let older packets (based on the sequence number), a retransmission of the FIN, and the ACK to \textit{B}'s FIN\&ACK (after having \textit{B}'s FIN\&ACK) through. For accomplishing this task the firewall has to keep track of the sequence numbers. We think that this is not done, and therefore more than the needed packets are let through. But to be sure that this is the real source of the problem, more tests would be needed.

\textbf{Checkpoint interprets FIN (without ACK) as attacks.}

Packets with only the FIN flag set do not occur in practice. In a correct, real world protocol run, packets having the FIN flag set will have the ACK flag set as well. Therefore FIN-only packets will probably belong to attacks and are therefore dropped by Checkpoint’s SmartDefense. We think this is a good idea and it should be done by other firewalls as well.
5.3 The Source of the Problem

In the last section we looked at the differences between TCP automata of different firewalls. In this section we now want to explain why such differences exist. More precisely, we explain why midpoints are different from endpoints and thus need a different protocol specification. In the next section we then show how to generate a midpoint protocol specification from endpoint protocol specifications.

The problem arises with the filtering midpoints: They base their decisions – basically drop or forward – on the protocol states the endpoints are in. Unfortunately two endpoints of a connection can be in different states and not all of these are observable by the midpoint.

Consider the following example: the TCP connection initiation (three-way-handshake) shown in figure 5 on page 5 Imagine the second packet gets lost after the midpoint (figure 17). Alice is now in state SYN-SENT, whereas Bob is in state SYN-RECEIVED. To the midpoint, this situation looks the same as the situation where the second packet reaches Alice, but the third packet gets lost before the midpoint (figure 16 on the next page).

<table>
<thead>
<tr>
<th>Alice</th>
<th>Midpoint</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td></td>
<td>LISTEN</td>
</tr>
<tr>
<td>SYN-SENT</td>
<td>SYN</td>
<td>SYN</td>
</tr>
<tr>
<td></td>
<td>← SYN &amp; ACK</td>
<td>← SYN &amp; ACK</td>
</tr>
</tbody>
</table>

Figure 17: TCP Scenario 1

Given that the midpoint cannot differentiate the scenario in figure 16 from the scenario in figure 16 on the next page 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 retransmit and should be forwarded, whereas in scenario 2 a SYN would not be conforming 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 as long as he has not 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 saw that one reason why midpoints cannot always know the protocol states of the endpoints lies in packet loss. But packet loss is only part of the problem. Another part lies in certain properties of endpoint automata. These are:

1. multiple transitions with the same output
   - If there are two transitions - a and b / a starting at one endpoint state, a midpoint cannot know from which transition an a it gets originated if it has previously seen a b.

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. Or at least not as long as it has not seen other, unambiguous output of the endpoint. A special case is hidden states – states where all incoming and outgoing transitions have no output – they cannot in any case be identified by a midpoint.

3. a packet can be sent in different states
   - This makes an unambiguous mapping between packet and state impossible. This is a problem when the midpoint already has a tracking problem.

If the actions of a midpoint are security relevant, it matters how it decides. The problem is that we always have a tradeoff between security and convenience. The stricter a midpoint acts, the more secure. But unfortunately this is often inconvenient for the users.

5.4 Construction of a Midpoint Automaton from Endpoint Automata

In the proceeding sections we saw why there is a need for protocol specifications for midpoints. Basically there are two ways to get such specifications: either somebody has to write the specification, or we generate it from the endpoint specification. The first alternative has a major drawback: consistency with the endpoint specification will most probably not be achieved. There are also minor drawbacks: 1) being additional work for the people designing new protocols and 2) being unclear who should do the work for already existing protocols. Due to these drawbacks we decided to pursue the second alternative.

5.4.1 Setting

Most communication protocols work with two endpoints. But there are also some protocols, e.g. Voice Over IP protocols, with three and more endpoints. For the sake of simplicity we only look at two-party protocols for the moment. We are sure that our ideas can easily be adapted to other protocols as well.
We model a two-party protocol as follows: We have two endpoints $E_0$ and $E_1$ and a midpoint $M$. All the communication between $E_0$ and $E_1$ has to pass $M$. Communication takes place in the form of messages, where the endpoint specification of the communication protocol determines when an endpoint is allowed to send which kind of message. For every message arriving at the midpoint, the midpoint has two choices: it can either forward the message (figure 19), or it can drop it (figure 20).

![Figure 19: A message $m$ from endpoint $E_0$ to endpoint $E_1$, forwarded by the midpoint $M$.](image1)

![Figure 20: A message $m$ from endpoint $E_0$ to endpoint $E_1$, dropped by the midpoint $M$.](image2)

We denote a message $m$ sent from one endpoint to the other (i.e. in transit on the network) as $X \to Y : m$, where $X \in \{E_0, E_1\}$ is the sender (= source) of the message and $Y \in \{E_0, E_1\}, Y \neq X$ is the intended recipient (= destination) of the message. When talking about midpoint actions, we will talk about such a message $X \to Y : m$ as $X \to M : m$ and $M \to Y : m'$, This helps us to know on which side of the midpoint a message is and also to argue about the midpoint actions (forwarding or dropping a message). Note that $m' = m$ if the midpoint forwards the message unchanged, $m = -$ if the midpoint drops the message, and $m' \in \Sigma_Y \setminus \{m\}$ if the midpoint changes the message before forwarding it\(^ {23}\).

![Figure 21: A message $m$ from endpoint $E_0$ to endpoint $E_1$, lost by the network.](image3)

As network model we assume a network that either delivers messages (in an arbitrary order) or looses them (figure 21).

5.4.2 Idea

Before giving the (technical) construction of a midpoint automaton from endpoint automata in section 5.4.3 we first sketch the ideas behind it at a high-level.

We model the global state of a system (endpoints, midpoint, network) at some time $t$ using a global state $s^t = (q_0^t, q_M^t, q_1^t, net^t)$\(^ {24}\), where $q_0^t$ is the state of endpoint $E_0$ at time $t$, $q_1^t$ is the state of endpoint $E_1$ 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 $t$.

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 $M$ knows the correct states of the endpoints and the correct contents of the network at all times.

---

\(^{23}\)This is for example done when using Network Address Translation (NAT) in the firewall. For the sake of simplicity we will not consider this case.

\(^{24}\)There are infinitely many global states. We will later model them as traces, not as automata.
This would be perfect. But as we have midpoints that cannot at all times determine the correct values of these three states, we will let the state of our midpoint be a set of such triples. The intuition behind this is that each of these triples represents a possibly correct view of the world. 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. To have a correctly functioning midpoint, two properties about a midpoint state need to be satisfied: 1) one of the triples is the correct one; and 2) only “correct” messages are forwarded.

To give intuition about the functioning of a midpoint (how we envision it), let us give an example. Suppose the system starts in the global state \( st^1 = (q_{A5}, \{(q_{A5}, q_{B3}, nett)\}, q_B3, nett) \). Then the following actions occur:

1. \( M \) sends (forwards) a message \( x \) to \( E0 \)
2. \( x \) is received by \( E0 \) and used as input to his automaton (suppose there are the two transitions illustrated in figure 22
3. a message \( y \) reaches \( M \)

![Figure 22: Part of an Endpoint Automaton \( A_0 \)](image)

Now we want to explain, step by step, how our midpoint should act in this example:

- **start** \( q_M = \{(q_{A5}, q_{B3}, nett)\} \)
- **action 1** \( q_M' = \{(q_{A5}, q_{B3}, net')\}, net' = net \cup \{M \rightarrow E0 : x\} \)
- **action 2** \( M \) calculates all possible successor states:
  \[ q_M'' = \{(q_{A5}, q_{B3}, net''), (q_{A6}, q_{B3}, net''), (q_{A7}, q_{B3}, net' \cup \{E0 \rightarrow M : z\})\} \]
  \[ net''' = net'' \setminus \{M \rightarrow E0 : x\} \cup \{E0 \rightarrow M : y\} \]
- **action 3** \( M \) determines its reaction on \( q_M'' \):
  \[ q_M''' = \{(q_{A6}, q_{B3}, net''\prime)\} \]
  \[ net'''' = net'''' \setminus \{E0 \rightarrow M : y\} \cup \{M \rightarrow E1 : y\}\]}

Thus our global state changes as follows during the example:

- \( st^1 = (q_{A5}, (q_{A5}, q_{B3}, net), q_{B3}, net) \)
- \( st^2 = (q_{A5}, (q_{A5}, q_{B3}, net'), q_{B3}, net') \)
- \( st^3 = (q_{A6}, (q_{A5}, q_{B3}, net''), q_{B3}, net'') \)
- \( st^4 = (q_{A6}, (q_{A6}, q_{B3}, net''\prime), q_{B3}, net''\prime) \)

with action 1 leading from \( st^1 \) to \( st^2 \), action 2 leading from \( st^2 \) to \( st^3 \), and action 3 leading from \( st^3 \) to \( st^4 \). Note that \( q_M'' \) does not appear here as it is only a temporary, internal state.

In the example above, we noted that to get from \( q_M' \) to \( q_M'' \), \( M \) needs to calculate all possible successor states. To keep the example simple, we did compute only one step with respect to \( E0 \). This was enough

---

\(^22\)If there is one or more triple in its state having \( y \) in its net, \( y \) is forwarded and the next state of \( M \) will consist of all these matching triples with their nets updated, i.e. all the still possible scenarios.

31
as there was only one transition of $E0$ between the two steps of the midpoint, But for the real world, where more than one endpoint transition can take place between two consecutive midpoint transitions, all possible successor states are needed, as messages resulting from endpoint transitions need not reach the midpoint in the order they have been sent. Let us illustrate this with another simple example. In figure 23, a setting is shown where two endpoint transitions happen consecutively. After this, it can happen that the message ($E0 \rightarrow M : z2$) reaches the midpoint first. Thus it would be not enough if the midpoint adapted its state, as in the example above, only with the next possible state, but all possible successor states are needed, as computing all possible successor states would lead to the temporary midpoint state

\[\{(q1, qB, \{M \rightarrow E0 : x, M \rightarrow E0 : y\}), (q2, qB, \{M \rightarrow E0 : y, E0 \rightarrow M : z1\}), (q3, qB, \{E0 \rightarrow M : z1, E0 \rightarrow M : z2\}\}\]

and thus leading to the correct action – forwarding $z2$ – and the correct next state

\[\{(q3, qB, \{E0 \rightarrow M : z1, M \rightarrow E1 : z2\}\}\}.

Let us return to our first example, to illustrate why all possibly correct messages need to be forwarded. Imagine the actions look like the following:

1. $M$ sends (forwards) a message $x$ to $E0$
2. $x$ is lost by the network
3. $E0$ takes the transition to state $qA7$
4. somebody incorrectly sends message $y$
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 this $y$ here (it could be the correct one from above). Thus we end with accepting an incorrect message, but only in the cases where there is a possible correct scenario in which this message could occur.
5.4.3 Technical Details

The handling of some protocol \( p \) by the endpoints can be specified as Mealy automata:

\[
A_0 = (Q_0, \Sigma_0, \Gamma_0, \delta_0, \lambda_0, s_0) \\
A_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, \lambda_1, s_1)
\]

The network can be modelled as a multiset (also called bag) used to store all messages in transit (i.e., on the network) between the midpoint and the endpoints:

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

where \( S \) is the set of messages allowed by the protocol, and \( =_s \) means set equality (the same elements, ignoring repetition).

As we cannot handle a network of infinite size, as defined above, we have to restrict the actions of endpoints and network. The actions that need restriction are those that add messages to the network that are already present. These actions are retransmission (−/x-loops) in the endpoints and duplication in the network. For the sake of simplicity we just forbid these. After having solved the problem with this restriction, the problem should be tackled again for a setting where a certain number of retransmissions and duplications are allowed. This is left as future work.

Based on \( A_0 \) and \( A_1 \) we will now construct a Mealy automaton \( A_M \) for the handling of protocol \( p \) by the midpoint:

\[
A_M = (Q_M, \Sigma_M, \Gamma_M, \delta_M, \lambda_M, s_M)
\]

\[
Q_M = \mathcal{P}(Q_0 \times Q_1 \times net)
\]

\[
\Sigma_M = \{E0 \rightarrow M : a \mid a \in (\Gamma_0 \setminus \{-\})\} \cup \{E1 \rightarrow M : a \mid a \in (\Gamma_1 \setminus \{-\})\}
\]

\[
\Gamma_M = \{M \rightarrow E0 : a \mid a \in (\Sigma_0 \setminus \{-\})\} \cup \{M \rightarrow E1 : a \mid a \in (\Sigma_1 \setminus \{-\})\} \cup \{-\}
\]

\[
s_M = \{(s_0, s_1, \{\})\}
\]

To be able to define \( \delta_M \), we first have to analyse the different scenarios. We do this with the help of figure 24 on the following page. There we look at the relationship between the actions of an endpoint \( A \), the network and the midpoint \( M \). In particular, we look how the four different kinds of transitions an endpoint can make (\( x/-x, y/-y \), \( x/y, -/y \) for \( x \in \Sigma_A, y \in \Gamma_A \)) look from the endpoints’, the network’s and the midpoint’s point of view respectively. These are shown in columns 1 - 3, where one row represents one case. Note that one “view” of one principal can belong to several “views” of another principal. In the fourth column then the correct midpoint transition is shown, i.e., the transition the midpoint must take if it correctly wants to track the endpoint’s state and the messages in the network. For this we assume \( q_M = \{(q_0, q_{E1}, net_M)\} \) to be the state of the midpoint after it’s last transition (this is, in some cases, forwarding the \( x \)). Important to understand is 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 end- or midpoint, or lost by the network.

With the help of figure 24 on the next page we will now define the successor function, i.e. the function

\footnote{For the sake of simplicity we only look at one endpoint transition and network loss at the moment. This is enough to get the idea. Further transitions and reordering can then easily be added later on.}
5.4 Construction

<table>
<thead>
<tr>
<th>Endpoint E0</th>
<th>Network</th>
<th>Midpoint</th>
<th>correct Midpoint transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td></td>
<td>$\delta_M(q_{M,-}) = {(q_1, q_{E1}, \text{net}<em>M \setminus {M \rightarrow E0 : x})}$ $\lambda_M(q</em>{M,-}) = -$</td>
</tr>
<tr>
<td>$q_1$ - $x$ - $q_2$</td>
<td></td>
<td></td>
<td>$\delta_M(q_{M,-}) = {(q_2, q_{E1}, \text{net}<em>M \setminus {M \rightarrow E0 : x})}$ $\lambda_M(q</em>{M,-}) = -$</td>
</tr>
<tr>
<td>$q_1$ - $x/y$ - $q_1$</td>
<td></td>
<td></td>
<td>$\delta_M(q_{M,-}) = {(q_3, q_{E1}, \text{net}<em>M \setminus {M \rightarrow E0 : x})}$ $\lambda_M(q</em>{M,-}) = -$</td>
</tr>
<tr>
<td>$q_1$ - $-y$ - $q_4$</td>
<td></td>
<td></td>
<td>$\delta_M(q_{M,-}) = {(q_4, q_{E1}, \text{net}<em>M \setminus {M \rightarrow E0 : y})}$ $\lambda_M(q</em>{M,-}) = -$</td>
</tr>
<tr>
<td>$q_1$ - $-x$ - $q_5$</td>
<td></td>
<td></td>
<td>$\delta_M(q_{M,-}) = {(q_5, q_{E1}, \text{net}<em>M)}$ $\lambda_M(q</em>{M,-}) = -$</td>
</tr>
</tbody>
</table>

Figure 24: A transition in an Endpoint – a Midpoints view

which computes all possible successor states of a given state,

\[
succ((q_0, q_1, \text{net}_M)) = \bigcup_{(M \rightarrow E0 : x)} \bigcup_{(M \rightarrow E1 : x)} \bigcup_{m \in \text{net}_M} \bigcup_{m \in \text{net}_M} \bigcup_{m \in \text{net}_M} \bigcup_{m \in \text{net}_M}
\]

\[
\{q_0, q_1, \text{net}_M\}, \\
\{q_0, q_1, \text{net}_M \setminus \{msg\}\}, \\
(q_0, q_1, \text{net}_M \setminus \{m \} \cup \{m \} \cup \{m \} \cup \{m \}) \\
\{\delta_0(q_0, x, 1), q_1, \text{net}_M \setminus \{M \rightarrow E0 : x\} \cup m1\} \\
\{\delta_0(q_0, -, 1), q_1, \text{net}_M \cup m2\} \\
\{q_0, \delta_1(q_1, x), \text{net}_M \setminus \{M \rightarrow E1 : x\} \cup m3\} \\
\{q_0, \delta_1(q_1, -), \text{net}_M \cup m4\}
\]

where

\[
m_1 = \begin{cases} 
E0 \rightarrow M : \lambda_0(q_0, x, 1), & \lambda_0(q_0, x, 1) \neq - \\
\emptyset, & \text{otherwise}
\end{cases}
\]

\[
m_2 = \begin{cases} 
E0 \rightarrow M : \lambda_0(q_0, -), & \lambda_0(q_0, -) \neq - \\
\emptyset, & \text{otherwise}
\end{cases}
\]

\[
m_3 = \begin{cases} 
E1 \rightarrow M : \lambda_1(q_1, x), & \lambda_1(q_1, x) \neq - \\
\emptyset, & \text{otherwise}
\end{cases}
\]

\[
m_4 = \begin{cases} 
E1 \rightarrow M : \lambda_1(q_1, -), & \lambda_1(q_1, -) \neq - \\
\emptyset, & \text{otherwise}
\end{cases}
\]
succ computes all the states that are reachable in one step by an endpoint or the network. But we are interested in all possible successor states. We achieve this by computing the closure of succ where the closure is defined as:

\[ cl(f(x)) = f^*(x) = \bigcup_{i=0}^{\infty} f^i(x) \] (13)

Before going on, let us elaborate on why \( cl(\text{succ}(q)) \) is computable in finite time. For this we have to show that after a finite number of succ-steps, no new triples are added. With the triple from equation 2 this is easy, it always stays the same. The triple from equation 4 is also easy, here \( net_M \), which is finite, decrease in every step by one element. With the triples from equations 4 and 5 it gets more difficult. They only “relatively” decrease. Or said otherwise, what is added in these steps is of the form \( \{Ei \to M : \ldots\} \) and thus cannot be removed in a further succ-step. And last but not least, equations 6 and 7. These get bigger in every step. But as there is only a finite number of transitions in the endpoint automata, and we do not have retransmission and duplication, these additions can only last for a finite number of steps.

We now can define \( \delta_M \). 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.\(^{27}\)

\[ \delta_M(q_M, m) = \bigcup_{(q_i, q, net_M) \in cl(\text{succ}(q_M)) \text{ and } m \in net_M} \{ (q_i, q, (net_M \setminus \{ m \}) \cup \lambda_M(q_M, m)) \} \] (14)

\( \lambda_M \) is now trivial: If there is any triple where the input occurs (i.e. a possibly correct scenario having produced this message), we forward it. This is because we want our firewall to be permissive.

\[ out((q_i, q, net_M), Ai \to M : y) = \begin{cases} (M \to Aj : y), & \text{if } (\exists \{Ai \to M : y\} \in net_M) \text{ where } j = (i+1) \mod 2 \\ - & \text{otherwise} \end{cases} \]

\[ \lambda_M(q_M, m) = \begin{cases} out(q_i, m), & \text{if } (\exists q \in cl(\text{succ}(q_M))) \text{ with } (out(q_i, m) \neq -) \\ - & \text{otherwise} \end{cases} \] (16)

Note that as for a given \( m \), \( out(q_i, m) \) will be the same for any \( q \) if it is non-zero (i.e. if it is not -), \( \lambda_M \) is unambiguous. This is due the fact that our midpoint either drops or forwards a message. Thus it would need to be revised for a midpoint that changes messages (e.g. a firewall doing NAT).

5.4.4 Correctness

Let \( st^i = (q_0^i, q_M^i, q_1^i, net^i) \) with \( q_0^i \in Q_0, q_M^i \in Q_M, q_1^i \in Q_1 \), \( net^i \in \{ \Sigma_M \cup \Gamma_M \} \) be a global state of the system at time \( t \). We now define the possible transitions between a global state \( st^i \) and its successor state \( st^{i+1} \):

**Definition 1** \( st^i \vdash st^{i+1} \)

In the following we will use \( i \in \{0, 1\} \) and \( j = (i + 1) \mod 2. \)

a. (midpoint transition)

for any \( msg \in net \) with \( msg = (A_i \to M : y) \), \( q_0^{i+1} = q_0^i, q_1^{i+1} = q_1^i, q_M^{i+1} = \delta_M(q_M^i, msg), net^{i+1} = (net^i \setminus \{msg\}) \cup \lambda_M(q_M^i, msg) \)

b. (correct endpoint transition)

for any \( msg \in (net^i \cup \{-\}) \) with \( msg = (M \to A_i : x) \) or \( msg = - \),

\[ msg' = \begin{cases} (A_i \to M : \lambda_M(q_i^i, x)), & \text{if } (\lambda_M(q_i^i, x) \neq -) \\ - & \text{otherwise} \end{cases} \]

\[ net^{i+1} = (net^i \setminus \{msg\}) \cup \{msg'\}, q_i^{i+1} = \delta_i(q_i^i, x), q_j^{i+1} = q_j^i, q_M^{i+1} = q_M^i \]

\(^{27}\)Note that \( cl(\text{succ}(q_M)) \) represents all possible successor states of \( q_M \), whereas \( \delta_M(q_M, m) \) only contains those successor states of \( q_M \) which can be “reached” with an \( m. \)
5.4 Construction

5 ENDPOINTS VERSUS MIDPOINTS

c. (incorrect endpoint transition)
\[ \text{msg}' \in (\Gamma \setminus \{ \lambda \}) \setminus \{ s \in \text{net} \cup \{ - \} \}, \text{net}^{i+1} = \text{net}^i \cup \{ \text{msg}' \} \}, q_0^{i+1} = q_0^i, q_1^{i+1} = q_1^i, q_M^{i+1} = q_M^i \]

d. (network loss)
\[ \text{for any msg} \in \text{net}^i, q_0^{i+1} = d_0^i, q_1^{i+1} = d_1^i, q_M^{i+1} = d_M^i, \text{net}^{i+1} = \text{net}^i \setminus \{ \text{msg} \} \]

Note that, as there is no order in multisets, permutation is handled implicitly.

**Definition 2** Transitions \[ \text{\textbf{E}}, \text{\textbf{L}}, \text{\textbf{M}} \] represent correct transitions.

**Definition 3** A correct trace is a trace \[ s^1 \vdash s^2 \vdash \ldots \vdash s^n \], where every transition \[ s^i \vdash s^{i+1} \] with \[ 1 \leq i < n \] represents a correct transition.

**Definition 4** The midpoint message history of a trace \[ tr = s^1 \vdash s^2 \vdash \ldots \vdash s^n \] is a sequence of messages \( m_1, m_2, \ldots, m_s \), where \( s \) is the number of midpoint transitions in \( tr \), and \( m_i \) is the message seen by \( M \) at its \( i \)-th transition.

**Definition 5** Two traces are midpoint-equivalent if they have the same midpoint message history.

**Definition 6** A possibly correct state \( s^m \), with respect to the current system state \( s^n = (q_0^n, q_M^n, q_1^n, \text{net}^n) \), is the endstate of a trace \( s^1 \vdash s^2 \vdash \ldots \vdash s^m \) which is midpoint-equivalent to the current system trace \( s^1 \vdash s^2 \vdash \ldots \vdash s^n \).

**Definition 7** \( M \) tracks endpoints correctly if after every midpoint transition \( (q_0^{n-1}, q_M^{n-1}, q_1^{n-1}, \text{net}^{n-1}) \vdash (q_0^n, q_M^n, q_1^n, \text{net}^n) \) in a trace, \( q_M^n \) is neither too small nor too large. Not too small means that \( (q_0^n, q_1^n, \text{net}^n) \in q_M^n \) holds. Not too large means that all \( q \in q_M^n \) represent a possibly correct state.

**Lemma 1** \( M \) tracks endpoints correctly

**Proof** by induction on the number of midpoint transitions.
During the proof, we will denote the correct triple \( (q_0^i, q_1^i, \text{net}^i) \) as \( q_{\text{corr}}^i \).

**Basis:** 1 midpoint transition

For not running into a problem, we assume the first state \( s^0 \) to be a state after a midpoint transition. Here, \( q_0^0 \) is correct:

\[ s^0 = (q_0^0, q_M^0, q_1^0, \text{net}^0) \]
\[ = (s_0, (s_0, s_1, \{\}), s_1, \{\}) \]

\[ \square \]

**Step:** \( n \rightarrow n + 1 \) midpoint transitions

We assume the \( n \)-th midpoint transitions to be \( s^t \vdash s^{t+1} \) and the \( n + 1 \)-th midpoint transition to be \( s^t \vdash s^{t+1} \).

We can divide the proof into two parts:

- \( d(\text{succ}(q_{M}^{t+1})) \) is a correct tracking up to time \( t2 \)
- \( \delta_M(q_{M}^{t+1}) \) is a correct tracking up to time \( t2 + 1 \)
The second part is easy: As of definition 5.4, \( net^{t+1} = (net^t \setminus \{msg\}) \cup \lambda_M(d^t_M, msg) \), which is reflected by \( \delta_M \) (definition 5.4).

For the first part we need another induction over the size of \( t=t-1 \). We do this induction a bit unconventional. We first show that \( \text{succ}(q^{t+1}_M) \) computes one step correctly (“Basis”) and then show that \( \text{cl}(\text{succ}(q^{t+1}_M)) \) computes any number \( t_2-t_1 \) of steps (“Step”) correctly. Note that \( t_2-t_1 \) is finite, as the endpoints cannot make infinitely many steps without midpoint interaction.

We divide the inner proof into the following parts, where \( t_1 \leq t < t_2 \):

1. For a correct transition \( st^t \vdash st^{t+1} \), \( d^{t+1}_\text{corr} \in (\bigcup_{q \in \Delta_M} \text{succ}(q)) \).

2. A message from an incorrect transition is not added to \( net^t_M \) by \( \text{succ}() \), except in the case where it could also occur in a correct scenario.

3. Computing \( d(\text{succ}()) \) for successive non-midpoint steps of a trace yields the same result as computing \( \text{succ}() \) consecutively for every step of the trace.

After this sketch of the proof idea we now come to the proof.

1. \( \text{succ}() \) for correct transitions
   We consider the following cases:

   (a) correct endpoint transition
   
   i. \( msg = (M \rightarrow Ei : x) \in net^t \)
   As of Definition 5.4 on page 85 we have:
   \[
   \begin{align*}
   q^{t+1}_i &= \delta_i(q^t_i, x) \\
   q^{t+1}_j &= q^t_j \\
   q^{t+1}_M &= q^t_M \\
   net^{t+1} &= net^t \setminus \{M \rightarrow Ei : x\} \cup \{msg'\} \\
   msg' &= \text{if } ((y = \lambda_i(q^t_i, x)) \neq -) \text{ then } \{Ei \rightarrow M : y\} \text{ else } \{-\}
   \end{align*}
   \]

   \( j = (i + 1) \mod 2 \)

   From Equation 5.4 on page 84 it holds that
   \[
   \forall (M \rightarrow E0 : z1) \in net^t_M, (q^t_0, x1, q^t_1, net^t_M \setminus \{M \rightarrow E0 : x1\} \cup m1) \in \text{succ}(d^{t+1}_\text{corr})
   \]
   where \( m1 = (\text{if } ((z1 = \lambda_0(q^t_0, x1)) \neq -) \text{ then } \{E0 \rightarrow M : z1\} \text{ else } \{\}) \)

   and

   \[
   \forall (M \rightarrow E1 : x2) \in net^t_M, (q^t_0, \delta_1(q^t_0, x2), q^t_1, net^t_M \setminus \{M \rightarrow E1 : x2\} \cup m3) \in \text{succ}(d^{t+1}_\text{corr})
   \]
   where \( m3 = (\text{if } ((z3 = \lambda_1(q^t_0, x2)) \neq -) \text{ then } \{E1 \rightarrow M : z3\} \text{ else } \{\}) \)

   □

   ii. \( msg = - \)
   As of Definition 5.4 on page 85 we have:
   \[
   \begin{align*}
   q^{t+1}_0 &= \delta_0(q^t_0, -) \\
   q^{t+1}_i &= q^t_i \\
   q^{t+1}_M &= q^t_M \\
   net^{t+1} &= net^t \cup \{msg'\} \\
   msg' &= \text{if } ((y = \lambda_0(q^t_0, -)) \neq -) \text{ then } \{E0 \rightarrow M : y\} \text{ else } \{-\}
   \end{align*}
   \]

   As of Equation 5.4 on page 84 it holds that
   \[
   (\delta_0(q^t_0, -), q^t_1, net^t_M \cup m2) \in \text{succ}(d^{t+1}_\text{corr})
   \]
   where \( m2 = (\text{if } ((y = \lambda_0(q^t_0, -)) \neq -) \text{ then } \{E0 \rightarrow M : y\} \text{ else } \{\}) \)

   □

37
5.4 Construction

(b) network loss

As of Definition 14 on page 33, the following holds:

\[
\begin{align*}
q_{i+1}^t &= q_i^t \\
q_{i+1}^t &= q_i^t \\
q_{m+1}^t &= q_m^t \\
\text{net}_{i+1}^t &= \text{net}_i^t \setminus \{\text{msg}\}
\end{align*}
\]

where \( \text{msg} \) is an arbitrary element of \( \text{net}_i^t \)

As of Equation 15 on page 34, it holds that

\[
\forall \text{msg} \in \text{net}_i^t, (q_i^t, q_i^t, \text{net}_i^t \setminus \{\text{msg}\}) \in \text{succ}(q_i^t_{\text{corr}})
\]

\[
\square
\]

2. \( \text{succ}() \) for incorrect transitions

As of Definition 16 on page 35, a message from an incorrect transition has the following form:

\( \text{msg} \in (\Gamma_i \setminus \{\lambda_i(q_i, x) | x \in \text{net}_i^t\} \setminus \lambda_i(q_i, -)) \).

\( \text{msg} \) could be added to the net of a \( q \in q_M^t \), if the following holds (see Equations 15 on page 35 and 16 on page 34):

\( \exists (q_i, q_i, \text{net}) \in q_M^t \) with \( \lambda_i(q_i, x) = \text{msg}, x \in \text{net} \) or \( \lambda_i(q_i, -) = \text{msg} \)

If \( q_M^t \) is a correct tracking, i.e. all \( q \in q_M^t \) represent possibly correct end scenarios at time \( t \), it follows that only those incorrect messages are added to the net of some triple which could also occur in a correct scenario.

\[
\square
\]

3. Using \( \text{cl}(\text{succ}()) \) yields the same result as computing \( \text{succ}() \) for every step

\( d(\text{succ}()) \) can be computed by finitely many \( \text{succ} \) steps. As every of these steps only adds possibly correct scenarios to the set (shown above), we end with a set containing only possibly correct scenarios. Thus it is not too large. As \( \text{succ}^n(q) \subseteq \text{cl}(\text{succ}(q)) \), with \( s \) denoting the number of steps between two consecutive midpoint transitions, holds, this set is also not too small.

\[
\square
\]

**Definition 8** The message history of a trace \( st^1 \vdash st^2 \vdash \ldots \vdash st^n \) is a series of messages \( m_1, m_2, \ldots, m_s \), where \( s \) is the number of non-midpoint transitions with “output” in \( tr \), and \( m_i \) is the message occurring at the \( i \)-th of these transitions.

**Definition 9** \( M \) computes outputs correctly if for every trace \( tr = st^0 \vdash \ldots \vdash st^n \) and every \( t, 0 \leq t \leq n \) we have:

\[
\lambda_M(q_M^t, Ai \rightarrow M : m) = \begin{cases} 
M \rightarrow Aj : m & \text{if } (Ai \rightarrow M : m) \text{ occurs in the message history of } tr \text{ or one of it's equivalents} \\
- & \text{otherwise}
\end{cases}
\]

where \( j = (i + 1) \text{mod} 2 \).

**Lemma 2** \( M \) computes outputs correctly.

**Proof**

We can divide the proof into two parts:
1. \( \text{msg} = (A_i \rightarrow M : m) \) was “inserted” by a correct transition
   As proven above, there is a triple \( q^{\text{corr}}_i \in q^i_M \) with \( q^{\text{corr}}_i = (q^i_0, q^i_1, \text{net}^i) \).
   The output of this triple, as of equation 15 on page 55, is correct, namely \( \text{msg}' = (M \rightarrow A_j : m) \).
   For every other triple \( q_i \), \( \text{out}(q_i, \text{msg}) \) is either \( \text{msg}' \) or \( - \). Thus, as of equation 16 on page 55, \( \lambda_M(q^i_M, A_i \rightarrow M : m) \) is correct.

2. \( \text{msg} \) was “inserted” by an incorrect transition
   As seen above, there can only be a \( (q_0, q_1, \text{net}) \in q^i_M \) with \( \text{msg} \in \text{net} \), if this represents a possibly correct scenario. But in this case forwarding \( \text{msg} \) is correct.\(^\text{28} \)

\[ \square \]

### 5.4.5 Discussion

Using our above construction, the resulting midpoint automaton \( A_{\text{Mp}} \) accepts all correct protocol flows between two endpoints E0 and E1 for a protocol \( p \). It might be the case that this is not what some midpoint vendors see as job of their midpoints. Thus let us show how \( A_{\text{Mp}} \) can be adapted to special needs:

1. \( A_{\text{Mp}} \) can be liberalised by adding more transitions. This will result in a midpoint automaton accepting more than the allowed protocol flows (as given by the endpoint specification). Note that it normally makes no sense to let a midpoint accept unallowed protocol flows. And if there is a good argument why such protocol flows should be allowed, it may be better to adapt the endpoint specifications of protocol \( p \).

2. \( A_{\text{Mp}} \) can be restricted by deleting transitions. This will result in a midpoint automaton which does not accept all allowed protocol flows (as given by the endpoint specification). One can have good reasons to restrict an automaton, e.g. some attacks.

Working this way, still every midpoint vendor can create midpoint automata of his liking. What we have achieved is that 1.) it now can clearly be said where the deviations are, and 2.) midpoints become testable (we can test on the basis of any automaton).

\(^{28}\)Note that in this case the endpoints might not be able to continue their run of the protocol, but that the incorrect endpoint is only able to continue his run like the possibly correct run which his message matched. This is what we have to live with when having a permissive firewall.
6 Tools

With the help of students we implemented an essential part of our research in a prototype test harness. Figure 25 gives an overview of the various tools and their relation to each other. In the following we want to give a short description of all the tools, before giving an example test run in the next section.

![Diagram of our tools](image)

**Figure 25: Our Tools – The Big Picture**

6.1 NetMap

As explained in section 3.1 we need to know the network under test to being able 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 and a useful extension, the specification of a formal policy in this tool, has not yet been implemented.

6.2 HOL-TestGen++

The basic functionality of HOL-TestGen with firewall extensions [Br06] has already been explained in section 4.2.2 (Approach 2).

6.3 TCGTool

The TCGTool (TCG standing for Test Case Generation) is an extension of JFLAP [RE] written by Stefan Hiklenbrand [Hik05]. The TCGTool can generate abstract test cases for protocols. Or in some more detail: In the GUI of the TCGTool one can enter a protocol specification as mealy automaton. The alphabet being used can be defined by the user. Then, by simply pressing a button, abstract test cases are generated within seconds, using the Wp-Method described in section 2.2.
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. How useful it is is also shown by the fact that already somebody (Dr Mark Utting from the University of Waikato, New Zealand) promised to write some bugfixes / extensions for the tool.

6.4 Glue

This section explains some small compilers which we wrote (with the help of flex and bison) to convert output from some tools to input for other tools.

6.4.1 gen-abc-test

gen-abc-test parses the output of the TCGTool (test cases and alphabet) and generates abstract test cases from it. The resulting abstract test cases are in fwtests input format, with names for IPs and ports.

6.4.2 gen-conc-test

gen-conc-test parses the output of HOL-TestGen++ (test tuples) and uses them to instantiate the abstract test cases generated by gen-abc-test. The resulting concrete test cases can then be directly used with fwtest.

6.5 fwtest

The first version of fwtest was written by Gerry Zaugg in his diploma thesis [Zau01]. This initial version (0.5) of fwtest was able to craft, inject, capture and analyse predefined test packets as shown in figure 26. The tool was written with TCP and UDP in mind, but at that time only TCP was supported.

![Diagram of fwtest v0.5](image)

Figure 26: fwtest v0.5

Fwtest was then extended (v1.0) by Beat Strasser [Str06] and Adrian Schüpbach [Sch06] to support UDP, ICMP and NATting. Also the test is now conducted by a single test instance (see figure on the next page) instead of two. How to use fwtest is explained in a short HOWTO given in appendix D. For more details please refer to the thesis reports.

6.6 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. What could be done to help the person doing this, but unfortunately is not implemented yet,

---

29With the agreement of the JFLAP authors, we put the TCGTool under GPL. Also with the hope that others in the field would invest their time in extending it – and thus helping us – instead of inventing the wheel once more (there already existed such a tool – see [http://www/site.uottawa.ca/ural/tsg/](http://www/site.uottawa.ca/ural/tsg/) – but unfortunately it was not available for use).

30Except in the figure there is only an unidirectional flow shown, whereas fwtest works bidirectional.
is to show the policy-rule corresponding to the failed test case and maybe also the corresponding part of the protocol specification.
7 An Example Test Run

In this section we want to show how a complete test run is conducted. We do this at a simple example, similar to the one in section 3. Please note that all the used tools and files can be found on the accompanying CD.

7.1 Policy

Our formal policy, network layout and keyword definitions can be found in figures 28, 29 and 30.

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

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

Figure 28: Demo – Formal Policy

<table>
<thead>
<tr>
<th>Name of the Firewall</th>
<th>Interface</th>
<th>Network behind</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>charly</td>
<td>eth0 (172.16.70.3)</td>
<td>DMZ</td>
<td>Packet filter</td>
</tr>
<tr>
<td>charly</td>
<td>eth1 (192.168.72.3)</td>
<td>Intranet</td>
<td>Packet filter</td>
</tr>
</tbody>
</table>

@ 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 29: Demo – Network Layout

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

Figure 30: Keyword definitions

7.2 Setup

To do firewall testing we need at least two machines: a firewall and a tester. These are then connected as given in figure 31 on the next page.

7.2.1 Firewall-Setup

As firewall we choose a Checkpoint product. We configure it as follows:

- Install Checkpoint R55W
- configure the interfaces
7.2 Setup

- 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 32 on the facing page)
- install the rules

7.2.2 Tester-Setup

On the tester we need at least fwtest installed. This means installing

flex, bison, gcc, make, libc, libc-dev, m4, iptables

as binaries, and

libpcap-0.7.2, libnet-1.1.2.1, libdnet-1.8

from source. Then we copy the fwtest-directory\[1\] to the machine and run /sbin/ldconfig.

If we also want to generate our test cases on the tester, we need to install all the other tools:

TCGTool copy the jar-file\[2\]
HOL-TestGen++ install Isabelle and HOL-TestGen\[3\] copy the firewall-specific theories
gen-abs-test copy directory, run make
gen-conc-test copy directory, run make

Configure the interfaces:

| ifconfig eth0 down |

\[1\]http://www.infsec.ethz.ch/education/projects/archive/fwtest_1.0.tar.gz
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

7.3 Test Case Generation

7.3.1 Generation of test tuples (approach 1)

Using approach 1 from Section 4 yields 75 test tuples. They are stored in the same format as for approach 2 (Small_Example.diana):

```xml
<!---------------------------------------------------------------------
--         Test-Data for Small_Example
--         generated by Diana von Bidder (by hand)
--         generated on 12.09.06
--------------------------------------------------------------------->
<testcase>
FUT (1, http, ((172, 16, 0, 0), 1025), ((192, 168, 0, 0), 80), content) =
    Some deny
</testcase>
<testcase>
FUT (1, http, ((172, 16, 70, 2), 1025), ((192, 168, 0, 0), 80), content) =
```
Some deny
</testcase>
<testcase>
PUT (1, http, ((172, 16, 255, 255), 1025), ((192, 168, 0, 0), 80), content) =
    Some deny
</testcase>
[..]

7.3.2 Generation of test tuples (approach 2)

To write input to HOL-TestGen, a lot of expert knowledge is needed. And even having this expert
knowledge, quite some time is needed to optimize the specification as such that the generation runs
through in a “feasible” time. The specification for our current example – which we will not explain
(please refer to [Br’06]) – is as follows (case_study2.thy):

theory case_study2
imports IPv4_Combinators Testing
begin

consts
defs
  intranet ::" ipv4 subnet"
  "intranet \equiv {\{(a,b,c,d),e\} . (a = 192) \land (b=168)}"

  dmz :: " ipv4 subnet"
  "dmz \equiv {\{(a,b,c,d),e\} . (a = 172) \land (b = 16)}"

  webserver :: " ipv4 subnet"
  "webserver \equiv {\{(a,b,c,d),e\} . (a = 172) \land (b = 16) \land (c = 70)}"

  mailserver :: " ipv4 subnet"
  "mailserver \equiv {\{(a,b,c,d),e\} . (a = 172) \land (b = 16) \land (c = 70)}"

lemmas network_def = webserver_def mailserver_def dmz_def intranet_def

text{* 
  DMZ --> Intranet any deny
  Intranet --> Webserver http, https accept
  Intranet --> Mailserver smtp, imap, imaps accept
  *}

casts
defs
  DMZ_Intranet :: "(ipv4,’b) Rule"
  "DMZ_Intranet \equiv \text{deny\_all\_from\_to\_dmz\_intranet}"

to_Webserver :: "(ipv4,’b) Rule"
  "to_Webserver \equiv \text{allow\_prot\_from\_to\_http\_intranet\_webserver}
  \text{allow\_prot\_to\_https\_webserver}"

46
To_Mailserver :: "(ipv4, 'b) Rule"
"To_Mailserver \<equiv> allow_prot_from_to smtp intranet mailserver
++ allow_prot_to imap mailserver
++ allow_prot_to imaps mailserver"

costdefs
test_policy :: "(ipv4, 'b) Policy"
"test_policy \<equiv> deny_all ++ DMZ_Intranet ++
To_Webserver ++ To_Mailserver"

lemmas ruleset = test_policy_def deny_all_def DMZ_Intranet_def
To_Webserver_def To_Mailserver_def

(* simple test *)

test_spec "FUT (x::(ipv4, DummyContent) packet) = ((deny_all_to_webserver)
++ (allow_all_from_dmz) ++ To_Webserver) x"
apply (unfold ruleset ipv4_rules stateless_rules)
apply (simp add: network_def)
apply (gen_test_cases FUT)
store_test_thm "test_to_WebServer"
gen_test_data "test_to_WebServer"
export_test_data "case_study2_to_webServer.dat" test_to_WebServer

test_spec "FUT (x::(ipv4, DummyContent) packet) = ((deny_all_to_mailserver)
++ (allow_all_from_dmz) ++ To_Mailserver) x"
apply (unfold ruleset ipv4_rules stateless_rules)
apply (simp add: network_def)
apply (gen_test_cases FUT)
store_test_thm "test_to_MailServer"
gen_test_data "test_to_MailServer"
export_test_data "case_study2_to_mailServer.dat" test_to_MailServer

test_spec "FUT (x::(ipv4, DummyContent) packet) = (deny_all
++ allow_prot_from_to http intranet webserver ) x"
apply (unfold ruleset ipv4_rules stateless_rules)
apply (simp add: network_def)
apply (gen_test_cases FUT)
store_test_thm "test_to_MailServer"
gen_test_data "test_to_MailServer"
thumb test_to_MailServer.test_data
export_test_data "t2.dat" test_to_MailServer

lemma [simp]: "(deny_all ++ DMZ_Intranet) = (deny_all)"
After 3 days of computation, HOL-TestGen delivers us the following test data (Demo/case\_study2\_policy.dat):

```xml
<!------------------------------------------------------------------
--    Test-Data (case_study2\_policy, 828 test cases)
--    generated by HOL-TestGen 1.1.1 (build: 29937M)
--    generated on Sun Sep 17 07:29:06 2006 (UTC)
------------------------------------------------------------------>
<testcase>
FUT (4, http, ((192, 168, 10, 7), 5), ((172, 16, 70, 4), 6), content) = Some
  (accept (4, http, ((192, 168, 10, 7), 5), ((172, 16, 70, 4), 6), content))
</testcase>
<testcase>
FUT (4, smtp, ((192, 168, 4, 4), 4), ((172, 16, 70, 5), 8), content) = Some
  (accept (4, smtp, ((192, 168, 4, 4), 4), ((172, 16, 70, 5), 8), content))
</testcase>
<testcase>
FUT (2, https, ((0, 5, 10, 6), 0), ((7, 1, 3, 8), 0), content) = Some deny
</testcase>
[..]
```

Unfortunately only 12 of these 828 test tuples represent connections across the firewall.

### 7.3.3 Generation of abstract test cases

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

```
< dsmm@gin:169 > java-1.4.2 -jar ../../../TCGTool/TCGTool.jar
```
Define the fields of a TCP packet\(^\text{34}\) (TCP.alpha.tcg):

Define the different TCP packets:

\(^{34}\)At the moment we are only interested in header fields.
Draw the automaton \( \text{TCP\_CheckpointR55W\_mm\_tgc} \):

Generate abstract test cases ("test case generation" → "Wp-method"):

\[
\begin{align*}
&\text{(SYN\_A\_B/SYN\_A\_B)} \text{(FINACK\_A\_B/FINACK\_A\_B)} \text{(SYN\_A\_B/SYN\_A\_B)} \\
&\text{(SYN\_A\_B/SYN\_A\_B)} \text{(SYNACK\_A\_B/SYNACK\_A\_B)} \text{(ACK\_A\_B/ACK\_A\_B)} \text{(FINACK\_A\_B/FINACK\_A\_B)} [..] \\
&\text{(SYN\_A\_B/SYN\_A\_B)} \text{(ACK\_A\_B/ACK\_A\_B)} \text{(ACK\_B\_A/-)} [..]
\end{align*}
\]

Convert the abstract test cases:

```
./gen-abs-test TCP_CheckpointR55W.tests.tcg TCP.alpha.tcg TCP_CheckpointR55W.tp
```

The resulting abstract test cases (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}
    packet 8 { TCP send {ipA ipB portA portB S 1 -} receive ok}
}
```

\( ^{35} \) as in figure \( ^{36} \) on page \( ^{47} \)
7.3.4 Generation of concrete test cases

Instantiating the abstract test cases for the Checkpoint R55W firewall with the test tuples from the last section

```
```

already yields 16'875 test cases (100'800 test packets). It makes no sense to run all of them. It would be better to conduct testing in two steps:

1. Instantiate the abstract test cases for Checkpoint R55W with some accepting test tuples. Use the resulting concrete test cases to test the firewall implementation.

2. Instantiate a single TCP connection initiation and tear down with all the test tuples. Use the resulting concrete test cases to test the firewall implementation.

The concrete test cases for the second step (Small.Example_simple_diana.tp) look as follows:

```c
/**
 * Small.Example_simple_diana.tp
 * concrete test cases for the use with fwtest v1.0
 * generated by ./gen-conc-test
 */
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 § 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 § 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 {}}
}
```
7.4 Execution

The concrete test cases, generated in the last Subsection, can directly be fed into fwtest. Following the instructions in the fwtest-README (see Appendix D) the following steps are needed:

1. Ping the firewall from the tester (this should either be allowed by the rules, as in our case, or be done before the policy is in place):

   ```
   ping -c 1 192.168.72.2
   ping -c 1 172.16.70.2
   ```

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

3. Run fwtest on the tester:

   ```
   ./run_fwtest.sh Small_Example_simple_diana.tp Small_Example_simple_diana_20060912.1 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 in 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
   - 64 - 66: 192.168.x.x → 172.16.70.5 imap ok

   For 46 - 48, 49 - 51, and 67 - 69 all packets were blocked instead of passed. For 55 - 57 all packets were passed instead of blocked, and for 64 - 66 the first two packets (SYN and SYN, ACK) were blocked instead of passed.

   Comparing the ruleset (figure 82) to the policy, having 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 83. 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]
   # 09/12/2006 13:28:55.133006
   # Timeout between time slots: 1 second(s)
   # [time] [type] [id] [packet]
   #
   # 13:30:08.563064 false_pos 69.46 --
   # 13:30:08.563147 false_pos 68.46 --
   # 13:30:08.563204 false_pos 67.46 --
   # 13:30:08.563260 false_pos 66.46 --
   # 13:30:08.563316 false_pos 65.46 --
   # 13:30:08.563497 false_pos 64.46 --
   # 13:30:08.563736 false_pos 51.46 --
   # 13:30:08.563768 false_pos 50.46 --
   # 13:30:08.563800 false_pos 49.46 --
   # 13:30:09.581246 false_pos 69.47 --
   # 13:30:09.581855 false_pos 68.47 --
   ```

   The log-file above shows the correct behavior of the firewall. The packets that were blocked are indicated as false_pos.

---

36 Actually we have to do some interleaving by hand.
We still have some entries, but fewer, and only false positives. We again group them into categories:

- 49 - 51: 192.168.x.x → 172.16.70.4 https
- 64 - 66: 192.168.x.x → 172.16.70.5 imap
- 67 - 69: 192.168.x.x → 172.16.70.5 imaps
For the first two categories, only the SYN and the SYN, ACK packets were blocked, but the other packets were let through. This looks like these test cases interfere with others, i.e. the packets are handled as belonging to another connection. To find out if this is the case, we test both of these categories alone. Now everything works as expected. So the problem was that the test cases were not correctly interleaved: a problem that will be tackled by a student next semester.

But with the third category it is different: There all packets are rejected. If we look at the problem in more detail, we find out that it is another error in the firewall rules: The second rule does not accept imaps as it should. This might be because Checkpoint does not seem to know imaps.
8 Related Work

The definitions of terms used in this section can be found in appendix A

8.1 Security Policy

There are some papers whose main focus is security policies. A paper which presents very similar ideas to ours is [BCG+01]. But it is not clear if there is more than just ideas. Also they rely on the help of device vendors. We doubt that device vendors will supply models for new devices and make existing devices conform to standards.

A Language for Security Constraints on Objects (LaSCO) is defined in [HPL]. 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. And this is not expressible in LaSCO.

There also exist guidelines [Gre97, SAN] on how to write a security policy, on security services, security incident handling, et cetera. But these discuss informal policies and are intended as a guideline for system administrators and as information for management.

8.2 Firewalls

8.2.1 Product Testing

Most people, and also 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. One of them is ICSA Labs, having a nice white paper describing their tests [Wal04]. Reading about all the bugs they found, we think that it is not safe to rely on the correctness of a firewall. Against this background, we find our decision to conduct the tests at the network level to be justified. The ICSA white paper also 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.”

8.2.2 Penetration Testing

Unfortunately penetration testing [Sche06, 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, it should not be the only test conducted. The usage of penetration testing as a ‘tool’ in fully manual conformance testing 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 clear specification. Thus we see this field partially (probing for known attacks) as complement to our method and partially (manual conformance testing) as predecessor of our work.

8.2.3 Analysis / Simulation

An approach orthogonal to ours is the analysis of existing vulnerabilities of firewalls [FKSF01, KFS]. The authors first define different stages of packet traversal in firewalls. Then they analyse the different stages for different firewalls to get a classification of the different types of errors and vulnerabilities occurring. They claim that the knowledge about known vulnerabilities can help to develop an intuition

---

37There are also other papers talking about security policies which we discuss somewhere else because security policies are not their main focus.
38Manual conformance testing: Humans using tools to find out what a firewall passes and then manually checking (very time intensive) if that is what should happen.
8.2 Firewalls

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 [BMNW03, BMNW03], Fang [MWZ05] and FA (earlier called LFA; Lumeta Firewall Analyzer) [Wor01] 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 also automatically retrieves the network layout from the firewall routing tables. Then all possible connections are analysed and the output is given as a bunch of HTML pages. 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). The problem we see here is that it is left to the user to find possible security problems in these reports.

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, but also helps the firewall rule editor to insert rules at the correct place. This approach has the same shortcomings as all on this level have: First, one has to know all possible firewall rule languages, and second, implementation errors cannot be found. Also this approach does not take into account if the rules express what they should. So, this approach does not solve our problem, but could be used complementary to ours.

Two other simple approaches are [EZ01] and [Haz99]. 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).

8.2.4 Specification-Based Testing

There are not many formal approaches to firewall testing yet.

The first, known to us, having ideas into this direction was Giovanni Vigna. Already in 1997 he wrote about formal firewall testing in a report [Vig97] which seems not to be published anywhere. Also his method only works with known attacks and relies on experts.

A work claiming to do formal firewall testing is [JWO1]. 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]. But 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. And last but not least the author claims to be the first to do analysis instead of injection, while citing at least three papers [MWZ05, BMNW03, EZ01] also doing analysis (the last of them even having analysis in its title).

8.2.5 Other

What does the best firewall solution help if your CIO does not think it is needed? Then we have [Ger97] which explains in an easy to understand way why one has to protect itself, why one needs a security policy, 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 [CMT97].
8.3 Other

[SHIJ+02] [Ron02] [OCA] have an aim similar to penetration testing. They also search for attacks but in another way: They define known attacks and then use a model checker to combine these such that, within a given network having an access policy, unallowed rights are achieved. The main problem we see here is that it only works with known attacks.
9 Conclusion

Said in one sentence, our contribution is the automation of firewall conformance testing. Doing this we solved one big security problem of companies, the one of not being able to check if their defense (by firewalls) is working as expected.

In more detail, we gave a formal specification language for network security policies, we solved the problem of determining a midpoint’s handling of a protocol (a.k.a. the first formal analysis of the job of a firewall), we presented a method for the generation of test cases from a formal policy, and we validated our approach using our prototype tool support.

Only now, using our approach, it is possible to formally argue about the correct implementation of a firewall and about the correct implementation of a security policy by some firewalls. And all this in a black-box way. Thus we no longer need experts searching hours to find a bug. We just need them to write a formal policy. Our approach is product-independent, thus works with any firewall soft- or hardware. And we are able to test conformance were one really wants it, at the network level.

10 Future Work

The expressive power of our formal specification language for network security policies might be not strong enough for a real world use. This is due to the unclear requirements; Unfortunately the few companies having (informal) security policies do not want to disclose them, because they fear that this will decrease their security level. We feel that if such companies could be won to give feedback on our prototype tool. This could then be used to improve the policy language.

Another weak point at the moment is the presentation of test results. Searching the source of an error, and representing this nicely to a user, poses many (not only technical) challenges. Generally, work is needed to transform the prototypical tool-support into an industrial strength tool.

Another direction that needs to be tackled is the application layer. In the near future, a firewall test method will only be useful if it can handle application level firewalls. One big problem there lies on the practical side: How to build a tool that can ‘speak’ any protocol. I.e. how to let a tool physically craft a protocol message of an 'unknown' protocol.

There are also a lot of interesting research questions. One is to revise the end- versus midpoint problem with retransmission und duplicates allowed.

---

39 Or only with a great effort of experts.
40 Conformance of the firewall rules to a policy does not help if the firewall implementation is buggy.
41 The specification of the protocols, and the generation of the test specification from it, can be done with our tools.
A  Fundamentals / Background - some Definitions

The following definitions are partly or fully taken from [WikiNetBar].

A.1  (Security) Policy

A security policy is a statement of a management strategy with regards to security. The policy statements are 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

A policy should outline a certain security topic, why it is needed/important, and explain what is allowed and what is not allowed. It should be general enough that changes are not required too often. It should contain general directives that are not architecture or system dependant. A policy should also tackle enforcement i.e. it should be clear what disciplinary measures are to be expected if 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, which may also be physical or administrative issues, that has to do with security at a specific site (or for a specific system). In contrast to this, a model is generally stated in a formal language (including proofs for correctness, consistency, ...) where characteristics of policies are explicitly stated. An example is the Bell LaPadula model. Therefore one can take an adequate model (or more than one) and use it to write his own policy.

Examples

The following are some excerpts of security policies:

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

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

“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.” [Goj]

A.2  Testing

The goal of testing is to find errors in the implementation. Errors are behaviours of a system that are not as expected (i.e. as specified).
Software Testing
Software testing is a process used to identify the correctness, completeness and quality of developed computer software. There are a number of different testing approaches that are used to do this ranging from the most informal ad hoc testing, to formally specified and controlled methods such as automated testing. In general, software engineers distinguish software faults and software failures. A fault is a programming error that does not actually manifest itself. In the case of a failure, the software does not do what the user expects. A fault can turn into a failure when the software is ported to a different hardware platform or a different compiler, or when the software gets extended.

Automated Testing
Generally speaking, software testing involves devising a test case (or, more likely, a set of test cases), running the program with the test case, and checking that the performance of the software with the test case as input is as expected. All three aspects of testing can be automated to a greater or lesser extent.

Penetration Testing
Penetration testing is security-oriented probing of a computer system or network to seek out vulnerabilities that an attacker could exploit. The testing process involves an exploration of all the security features of the system in question, followed by an attempt to breach security and penetrate the system. The tester, sometimes known as an ethical hacker, generally uses the same methods and tools as a real attacker. Afterwards, the penetration tester reports on the vulnerabilities and suggests steps that should be taken to make the system more secure.

A.3 Formal Verification
In the context of (software) systems, formal verification means the act of proving or disproving the correctness of a system with respect to a certain property, using mathematical methods.

A.4 Model Checking
Model checking is an approach to algorithmically checking whether a model (often derived from a hardware or software design) satisfies a logical specification.
The model is usually expressed as a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node. The nodes represent states of a program, the edges represent possible executions which alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.
A specification language, usually some kind of temporal logic, is used to express properties.

A.5 Automata theory
There are different types of automata. We only use Mealy automata / machines. A Mealy machine is a six-tuple
\[ M = (Q, \Sigma, \Gamma, \delta, \lambda, q_0) \]
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;
• \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function;
• \( \lambda : Q \times \Sigma \rightarrow \Gamma \) is the output function;
• \( q_1 \in Q \) is the initial state.

Automata will often be specified graphically. This will look the following for Mealy machines:

Here \( q_0 \), \( q_1 \) and \( q_2 \) are states, where \( q_0 \) is the start state and \( q_2 \) is an accepting state. The arrow from \( q_0 \) to \( q_1 \) represents a transition with input \( x \) and output \( y \).
B Grammars

B.1 EBNF

\[
\begin{align*}
\text{EBNF} & = \{ \text{stmt} \} . \\
\text{stmt} & = \text{nts} \ '=>' \ \text{expr} \ '>'. \\
\text{expr} & = \text{term} \ \{ \ '[' \ \text{term} \ ']'. \\
\text{term} & = \text{factor} \ \{ \ \text{factor} \} . \\
\text{factor} & = \text{nts} \ | \ \text{ts} \ | \ \text{expr} \ ')'. \ | \ '[' \ \text{expr} \ ']'. \ | \ '{' \ \text{expr} \ '}'. \\
\text{nts} & = \ \text{letter} \ \{ \ \text{letter} \ | \ \text{digit} \} . \\
\end{align*}
\]

Some explanations:

- **stmt**: A syntactic equation.
- **expr**: A list of alternative terms.
- **term**: A concatenation of factors.
- **factor**: A single syntactic entity or parenthesised expression.
- **nts**: Nonterminal symbol that denotes a syntactic entity. It consists of a sequence of letters and digits where the first character must be a letter.
- **ts**: Terminal symbol that belongs to the defined language’s vocabulary. Since the vocabulary depends on the language to be defined there is no production for ts. They are represented between single quotes.

\[
\begin{align*}
A & \quad B & \quad A \ \text{and} \ B \\
A & \ | \ B & \quad A \ \text{or} \ B \\
[A] & & \quad A \ \text{is optional} \quad (0 \ \text{or} \ 1 \ \text{times} \ A) \\
\{A\} & & \quad \text{any number of A’s} \quad (\text{including none}) \\
(A \ | \ B) \ (C \ | \ D) & = \quad AC \ | \ AD \ | \ BC \ | \ BD \quad (\text{parentheses are for separation})
\end{align*}
\]

B.2 General

\[
\begin{align*}
\text{IP} & = \text{DDD}.'\text{DDD}.'\text{DDD}.'\text{DDD}. \\
\text{D} & = \text{[digit]}. \\
\text{PROTO} & = \text{letter} \ \{ \text{letter} \ | \ '-' \ | \ '+' \ | \ digit \ | \ ':' \ | \ '.' \} \ | \ \text{NUM}. \\
\text{NUM} & = \text{[digit]}. \\
\text{NAME} & = \text{letter} \ \{ \text{letter} \ | \ \text{digit} \}. \\
\text{ACTION} & = \ '\text{accept}' \ | \ '\text{deny}'. \\
\text{PRE} & = \ '\text{pre}'. \\
\text{POST} & = \ '\text{post}'. \\
\text{COMMENT} & = \ '@' \ \text{TEXT} \ '\n' \\
\text{TEXT} & = \ \{ \text{letter} \ | \ \text{digit} ...\}
\end{align*}
\]

Figure 35: General non-terminals

62
B.3 Abstract Firewall Rules

RULE = 'from' HOSTS 'to' HOSTS 'service' SERVICES 'action' ACTIONS
HOSTS = IP | HOSTGROUP.NAME
HOSTGROUP = NAME = IP {, IP}
SERVICES = ['!'PROTO {, ['!'PROTO}
ACTIONS = ACTION {, ACTION}

Figure 36: Grammar of abstract firewall rules

RULES = {RULE} .
RULE = BEFORE '⇒' AFTER .
BEFORE = COND | COND '∧' COND | COND '∨' COND | '('COND') | '!'COND .
AFTER = ACTION ∧ NEWSTATE .
COND = '('proto '⇒' PROTO') | '('src '⇒' IP') | '('dst '⇒' IP') | '('sport '⇒' NUM')
| '('dport '⇒' NUM') | '('flag '⇒' FLAG {, 'FLAG}')' | OLDSTATE .
OLDSTATE = '('EXTSTATE 'ε' PRE') .
NEWSTATE = '('POST '⇒' PRE') | '('EXTSTATE 'ε' POST') .
EXTSTATE = '<'PROTO', 'IP', 'IP', NUM', NUM', STATE>' .
STATE = 'new' | 'syn' | 'established' | 'closing' | 'related' .
FLAG = 'syn' | 'ack' | 'fin' | 'rst' | 'psh' | 'urg' .

Figure 37: Grammar of logical firewall rules

Some explanations for figure 37:

state, state' They contain all the states of all the connections before and after having handled the matching packet respectively.

... (x ∈ state) ⇒ ... (y ∈ state') The poststate (state') is obtained from the prestate (state) by replacing x with y.

B.4 Network Layout

NETLAYOUT = NETWORKS '***' FIREWALLS '***' SERVERS
NETWORKS = {NET | COMMENT}
NET = NAME ':' RANGE {',', RANGE}
RANGE = IP '/' DD | '!'NAME
FIREWALLS = {FIREWALL | COMMENT}
FIREWALL = FW IF NAME PROP
FW = NAME
IF = ['eth0' | 'eth1' ... ] '('IP')
PROP = '[' packet filter' [ ... ]
SERVERS = {SERVER | COMMENT}
SERVER = NAME IP PROTO

Figure 38: Grammar for textual network layout
B.5 Formal Network Policy

\[
\begin{align*}
\text{POLICY} & \quad = \quad \{\text{RULE} \mid \text{COMMENT}\} \\
\text{RULE} & \quad = \quad \text{SOURCE} \ '\rightarrow'\ \text{DEST} : \text{ACTION}\ \text{KEYWORDS} \\
\text{SOURCE} & \quad = \quad \text{NETWORK} \\
\text{DEST} & \quad = \quad \text{NETWORK} \\
\text{NETWORK} & \quad = \quad \text{NAME} \\
\text{KEYWORDS} & \quad = \quad ('*$ | \text{NAME}) \ '{',\ \text{KEYWORDS}\}
\end{align*}
\]

Figure 39: Grammar for network policies

\[
\begin{align*}
\text{KEYWORD-DEFINITIONS} & \quad = \quad \{\text{DEFINITION} \mid \text{COMMENT}\} \\
\text{DEFINITION} & \quad = \quad \text{NAME} \ '=' \ \text{PROTO} \ '{',\ \text{PROTO}\}
\end{align*}
\]

Figure 40: Grammar for keyword definitions
C A small iptables HOWTO

This small HOWTO should help understanding the iptables examples in this document. 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 will 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 will 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:**

<table>
<thead>
<tr>
<th>command</th>
<th>meaning</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-F</td>
<td>flush (delete) all rules (from the given chain)</td>
<td><code>iptables -F input</code></td>
</tr>
<tr>
<td>-X</td>
<td>delete given chain (which must be empty)</td>
<td><code>iptables -X mychain</code></td>
</tr>
<tr>
<td></td>
<td>(if no argument given: delete all user-defined chains)</td>
<td></td>
</tr>
<tr>
<td>-P</td>
<td>set given policy as default for given chain</td>
<td><code>iptables -P INPUT DROP</code></td>
</tr>
<tr>
<td>-A</td>
<td>append the following rule to the given chain</td>
<td><code>iptables -A INPUT ...</code></td>
</tr>
</tbody>
</table>

**some matches:**

<table>
<thead>
<tr>
<th>match</th>
<th>meaning (matches packets ...)</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p</td>
<td>... with the given protocol</td>
<td><code>iptables -p tcp ...</code></td>
</tr>
<tr>
<td></td>
<td>(e.g. tcp, udp, icmp)</td>
<td></td>
</tr>
<tr>
<td>-s</td>
<td>... with the given source IP</td>
<td><code>iptables ... -s 129.132.178.26 ...</code></td>
</tr>
<tr>
<td>-d</td>
<td>... with the given destination IP</td>
<td><code>iptables ... -d 127.0.0.1 ...</code></td>
</tr>
<tr>
<td>--sport</td>
<td>... with the given source port</td>
<td><code>iptables ... -p tcp --sport ssh ...</code></td>
</tr>
<tr>
<td></td>
<td>(only with <code>-p tcp</code> or <code>-p udp</code>)</td>
<td></td>
</tr>
<tr>
<td>--dport</td>
<td>... with the given destination port</td>
<td><code>iptables ... -p udp --dport 88 ...</code></td>
</tr>
<tr>
<td></td>
<td>(only with <code>-p tcp</code> or <code>-p udp</code>)</td>
<td></td>
</tr>
<tr>
<td>--syn</td>
<td>... with the syn flag set</td>
<td><code>iptables ... -p tcp --syn ...</code></td>
</tr>
<tr>
<td></td>
<td>(only with <code>-p tcp</code>)</td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>state --state</td>
<td><code>iptables ... -m state --state NEW ...</code></td>
</tr>
<tr>
<td>state</td>
<td>... which are in the given state(s)</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>state</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>as long as the connection is only one-way</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>connections that have already seen packets in both directions</td>
</tr>
<tr>
<td>RELATED</td>
<td>new connections associated with an ESTABLISHED one (e.g. ftp data)</td>
</tr>
<tr>
<td>INVALID</td>
<td>most of the time faulty data or headers (should be rejected)</td>
</tr>
</tbody>
</table>
some targets:
There are a lot of possibilities to use targets/jumps. We will only explain the simplest two:

<table>
<thead>
<tr>
<th>target</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-j ACCEPT</td>
<td>accept the packet (let it pass the firewall) and stop processing</td>
</tr>
<tr>
<td>-j DROP</td>
<td>drop the packet and stop processing</td>
</tr>
</tbody>
</table>
fwtest v1.0 README

FWTEST v1.0 (February 2006)
Author: Gerry Zaugg <zauggge@gmail.com>
Contributors: Adrian Schuepbach <scadrian@student.ethz.ch>
             Beat Strasser <b8@student.ethz.ch>
Maintainer: Diana von Bidder <diana.bidder@inf.ethz.ch>

0. Content

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

1. Description

Fwtest v1.0 provides a simple, portable interface to perform firewall testing. It is executed on a so-called testing host that sends test packets through a firewall. Fwtest crafts and injects the test packets and captures them if they pass the firewall. When receiving packets, fwtest performs an exhaustive analysis revealing irregularities (i.e. packets that are accepted by the firewall although they were expected to be dropped or packets that are discarded/changed although they were expected to be passed). The irregularities are logged and serve as source of information to identify failures.

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

Fwtest v1.0 is known to compile and run under Linux Debian 3.0 with the 2.4.24 kernel, Linux Red Hat 7.3 with the 2.4.18-3 kernel and Gentoo Linux 2005.1 with the 2.6.14 kernel. Probably, it also runs under OpenBSD, FreeBSD and NetBSD.

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

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

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

NOTE: Run /sbin/ldconfig before using libdnet. ldconfig creates the necessary links and cache (for use by the run-time linker, ld.so) to the most recent shared libraries (for more information: man ldconfig).

3. How To Perform a Test Run

Disclaimer: Do not use fwtest in a productive environment.

3.1 Test Environment

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

```
+----------+          +----------+
|  eth0    |          |  eth1    |
+----------+          +----------+
|  Testing Host |      |  Firewall |
```

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

3.2 Mandatory Files

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

Copy the following files into the test directory:

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

3.3 run_fwtest.sh

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

run_fwtest.sh requires at least six arguments:
1. Test packets file
   File containing the test packets
   Read more about the file format below.
2. Log file
   File where the irregularities are reported
3. Network 1: IP and mask in CIDR notation
   Specifies the first network that the testing host represents.
   (e.g. 192.168.1.0/29)
4. Network 1: interface
   The network interface to capture the packets from network 1 (e.g. eth0)
5. Network 2: IP and mask in CIDR notation
   Specifies the second network that the testing host represents.
   (e.g. 172.16.70.0/29)
6. Network 2: interface
   The network interface to capture the packets from network 2 (e.g. eth1)
7. Option -p
   Transmit packet id in payload (TCP/UDP only*)
8. Option -u
   Interpret TCP/UDP packets as UDP instead of TCP

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

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

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

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

To make this point clear: fptest is the firewall testing tool that performs testing whereas run_fptest.sh is only a shell script that prepares the testing host for the test run and compiles and runs fptest for you. Read more about invoking fptest in section 3.5.

6. Remove the local firewall rules.
7. Exit successfully.

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

3.4 Test Packets File

3.4.1 General layout

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

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

70
I, J, K, L, M and N are integers which define the testcase/packet id. Every testcase is run in parallel. The packet id stands for a global timeslot, so for example the packet 1.2 (testcase 1, packet 2) is sent at the same time as packet 3.2.

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

3.4.2 Protocol specific fields (PROT_FIELDs)

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

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

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

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

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

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

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

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

Fwtest v1.0 filters the given test packet file with a preprocessor (m4). So you may define constants for often used IPs or ports. Fwtest includes by default the file defined by the environment variable FWTEST_MACROS. run_fwtest.sh sets this to macros.m4 which contains constants for possible ICMP types and codes. It's possible to disable the preprocessor by setting the environment variable FWTEST_USEM4 to 'no'.

3.4.4 Variables

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

example:

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

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

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

3.4.5 Parse errors

Whenever you get parse errors on a test packet file, the line number will be displayed. Whereas the number is correct for syntax errors, Fwtest v1.0 errs mostly on a semantical error e.g. the discovery of a duplicate packet id. Because packet specifications are internalized only at the end of a testcase, such errors are not detected until the end of the actual testcase.
Since the order of the packets is not relevant (only the packet id specifies the
time slot), Fwtest reverses the packets to simplify matters because of the
internal data structure; so, errors may be detected in the opposite order than
they appear in the specification.

3.5 Running Fwtest

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

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

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

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

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

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

3.6 Example

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

Test host setup:

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

Firewall setup:

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

73
Ping the firewall from the test host (the firewall is still accepting all incoming packets):

    ping -c 1 172.16.70.1
    ping -c 1 192.168.72.1

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

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

Create a test file named ‘test.tp’ like this:

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

Now fire off fptest by calling run_fptest.sh:

    ./run_fptest.sh test.tp test.log 172.16.70.0/24 eth0 192.168.72.0/24 eth1

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

4. Timeout

A timer is started by fptest after the packets for a given timeslot are sent out. When the timer expires after a specified timeout value, fptest steps to the next timeout. The timeout value is stored in the header file main.h. After changing the timeout value, fptest has to be recompiled.

5. Source Files
Makefile — Makefile
main.c — main program
tpparser.l — lexer grammar
tpparser.y — parser grammar
lpcap.c — packet capturing routines
lnet.c — packet crafting and injection routines
log.c — log routines
util.c — helper routines
symboltable.c — a symbol table for the test packets file variables
semanticchecker.c — a semantic-checker for the test packets file
main.h — main definitions & TIMEOUT definition
tpparser.h — parse definitions
lpcap.h — packet capture definitions
lnet.h — packet crafting definitions
log.h — log definitions
symboltable.h — symbol table definitions
semanticchecker.h — semantic-checker definitions
E TCP End- and Midpoint Automata in CSP

We also modelled the end- versus midpoint automaton problem in *Concurrent Sequential Processes (CSP)*. But as the model checker FDR (*Failure Divergence Refinement*) was not able to check a realistic number of packets, we had to find another approach.

E.1 A short introduction to CSP and FDR

CSP consists mainly of *processes*, *functions* and *channels*. Processes, whose names by convention are written in all upper-case, are the driving forces. They consist of a series of *events*, which bring a process from one *state* to another. The process P: a → b → P for example first communicates an a, then it communicates a b and then it “starts over”. The communication takes place over channels: One process can “write” something on a channel, and another process can “read” it afterwards. We speak of *input* and *output* of channels. Input is denoted with a question mark – for example in?a means input of a on channel in – whereas output is denoted with an exclamation mark.

Different processes can be *composed* to a new process. This composition is done through different operators (see figure 11). These operators define how the communication (the fundamental means of interaction between agents) between the different processes has to take place. Or said in other words, it defines if and how the different processes must *synchronise* their events. Synchronising events means that each component must be willing to participate in a given event before the whole can make the transition.

<table>
<thead>
<tr>
<th>CSP notation</th>
<th>fdr2 notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>P □ Q</td>
<td>P □ Q</td>
<td>external choice (behaves like either P or Q: System – available input – decides)</td>
</tr>
<tr>
<td>P \ Q</td>
<td>P \ Q</td>
<td>internal choice (behaves like either P or Q: fdr decides)</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P \ Q</td>
<td>p \ q</td>
<td>hiding (P with events X internalized, it is not visible and not controllable by the environment ⇒ abstraction mechanism)</td>
</tr>
<tr>
<td>P \ Q</td>
<td>p[ [a ← b] ]</td>
<td>b in P is replaced by a</td>
</tr>
<tr>
<td>P ⊆ Q</td>
<td>p ⊆ τ q</td>
<td>Q is a traces refinement of P (see below)</td>
</tr>
</tbody>
</table>

Figure 11: CSP Operators

FDR directly supports three models, from which we will make use of the *traces model*. The FDR-Manual ([http://www.fscl.com/documentation/fdr2/html/fdr2manual_4.html#SEC4](http://www.fscl.com/documentation/fdr2/html/fdr2manual_4.html#SEC4)) describes this model as follows: “The traces model: a process P is represented by the set of finite sequences of communications it can perform. The set of P’s (finite) traces is given by traces(P).”

In the traces model, we make use of the *traces refinement* which is described in the FDR Manual ([http://www.fscl.com/documentation/fdr2/html/fdr2manual_5.html#SEC5](http://www.fscl.com/documentation/fdr2/html/fdr2manual_5.html#SEC5)) as follows: “The coarsest commonly used relationship is based on the sequences of events which a process can perform (the traces of the process). A process Q is a traces refinement of another, P, if all the possible sequences of communications which Q can do are also possible for P. This relationship is written P ⊆r Q. If we consider P to be a specification which determines possible safe states of a system, then we can think of P ⊆r Q as saying that Q is a safe implementation: no wrong events will be allowed.

P ⊆r Q ∋ traces(Q) ⊆ traces(P)

Traces refinement does not allow us to say anything about what will actually happen, however. The process STOP, which never performs any events, is a refinement of any process in this framework, and satisfies any safety
E.2 TCP Automaton for Endpoints

E.2.1 Specification [SC81]

A model of the TCP specification on page 23 of the TCP-RFC [SC81], plus retransmission, looks as follows.

{-
  TCP specification for Endpoints (as in the TCP RFC)
  Diana Senn, 2005
  $Id: tcp.csp 29550 2006-08-18 13:57:07Z dsenn $
-}

datatype TCPFlags = syn | ack | fin
channel in, out: TCPFlags

CLOSED = (LISTEN
  [ ]
  out!syn -> SYNSENT
  )
  [ ]
  in?ack -> CLOSED  — retransmission

LISTEN = in?syn -> out!syn -> out!ack -> SYNRcvd
          [ ]
          out!syn -> SYNSENT

          [ ]
          out!syn -> SYNSENT  — retransmission

SYNRcvd = in?ack -> ESTAB
          [ ]
          out!fin -> FIN_WAIT_1
          [ ]
          in?syn -> out!syn -> out!ack -> SYNRcvd  — retransmission

ESTAB = out!fin -> FIN_WAIT_1
        [ ]
        in?fin -> out!ack -> CLOSE_WAIT
        [ ]
        in?ack -> ESTAB  — retransmission

FIN_WAIT_1 = in?ack -> FIN_WAIT_2
            [ ]
            in?fin -> out!ack -> CLOSING
            [ ]
            out!fin -> FIN_WAIT_1  — retransmission

CLOSE_WAIT = out!fin -> LAST_ACK
            [ ]
            in?fin -> out!ack -> CLOSE_WAIT  — retransmission

77
FIN\_WAIT\_2 = in?fin \rightarrow out!ack \rightarrow TIME\_WAIT

\[\]
\[\]
in?ack \rightarrow FIN\_WAIT\_2  \quad \text{— retransmission}

CLOSING = in?ack \rightarrow TIME\_WAIT

\[\]
\[\]
in?fin \rightarrow out!ack \rightarrow CLOSING  \quad \text{— retransmission}

LAST\_ACK = in?ack \rightarrow CLOSED

\[\]
\[\]
out!fin \rightarrow LAST\_ACK  \quad \text{— retransmission}

TIME\_WAIT = CLOSED

\[\]
\[\]
in?fin \rightarrow out!ack \rightarrow TIME\_WAIT  \quad \text{— retransmission}

\[\]
\[\]
in?ack \rightarrow TIME\_WAIT  \quad \text{— retransmission}

E.2.2 Incorrect Specification

{-
  incorrect TCP specification for Endpoints
  Diana Senn, 2005
  $Id: tcp\_inc.csp 28259 2006-07-10 09:08:53Z dsenn$
-}

--datatype TCPFlags = syn | ack | fin
--channel in, out: TCPFlags

INC = in?x \rightarrow INC

\[\]
\[\]
(\neg!syn \rightarrow INC

\[\]
\[\]
\neg!ack \rightarrow INC

\[\]
\[\]
\neg!fin \rightarrow INC

E.2.3 iptables (ip\_conntrack 2.1)

Figure 4 on page 26 as CSP specification:

{-
  TCP automaton of iptables (ip\_conntrack 2.1)
  reverse engineering: Diana Senn
  $Id: tcp-mid-iptables.csp 28259 2006-07-10 09:08:53Z dsenn$
-}

--datatype TCPFlags = syn | ack | fin
--channel left1, right1, left2, right2, left3, right3, left4, right4: TCPFlags
\[-\]
\[
\begin{align*}
\text{left1} & \rightarrow \text{right1} \quad \text{left2} & \rightarrow \text{right2} \\
\text{right4} & \rightarrow \text{left4} \quad \text{right3} & \rightarrow \text{left3}
\end{align*}
\]
\[
\text{Akzeptierender Zustand: NEW} \\
\text{Startzustand: NEW}
\]
\[\]
\[---\text{START} = \]
\[
\text{NEW} = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right3?syn} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right1?syn?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right3?syn?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right1?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right3?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right1?fin} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right3?fin} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right1?fin?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[
\text{right3?fin?ack} \rightarrow \text{NEW} \\
\quad \text{[]}
\]
\[\]
\[
\text{SYN_A} = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right3?syn} \rightarrow \text{left4!syn} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right1?syn?ack} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right3?syn?ack} \rightarrow \text{left4!syn!ack} \rightarrow \text{SYN_B} \\
\quad \text{[]}
\]
\[
\text{right1?ack} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right3?ack} \rightarrow \text{left4!ack} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right1?fin} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right3?fin} \rightarrow \text{left4!fin} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
\[
\text{right1?fin?ack} \rightarrow \text{SYN_A} \\
\quad \text{[]}
\]
right3?fin?ack \rightarrow left4!fin!ack \rightarrow SYN_A

\text{SYN}_B = \text{right1?syn} \rightarrow left2!syn \rightarrow \text{SYN}_B
\left[\right]
right3?syn \rightarrow left4!syn \rightarrow \text{SYN}_B
\left[\right]
right1?syn?ack \rightarrow left2!syn!ack \rightarrow \text{SYN}_B
\left[\right]
right3?syn?ack \rightarrow left4!syn!ack \rightarrow \text{SYN}_B
\left[\right]
right1?ack \rightarrow left2!ack \rightarrow \text{ESTABLISHED}
\left[\right]
right3?ack \rightarrow left4!ack \rightarrow \text{SYN}_B
\left[\right]
right1?fin \rightarrow left2!fin \rightarrow \text{SYN}_B
\left[\right]
right3?fin \rightarrow left4!fin \rightarrow \text{SYN}_B
\left[\right]
right1?fin?ack \rightarrow left2!fin!ack \rightarrow \text{SYN}_B
\left[\right]
right3?fin?ack \rightarrow left4!fin!ack \rightarrow \text{SYN}_B
\left[\right]

\text{ESTABLISHED} = \text{right1?syn} \rightarrow left2!syn \rightarrow \text{ESTABLISHED}
\left[\right]
right3?syn \rightarrow left4!syn \rightarrow \text{ESTABLISHED}
\left[\right]
right1?syn?ack \rightarrow left2!syn!ack \rightarrow \text{ESTABLISHED}
\left[\right]
right3?syn?ack \rightarrow left4!syn!ack \rightarrow \text{ESTABLISHED}
\left[\right]
right1?ack \rightarrow left2!ack \rightarrow \text{ESTABLISHED}
\left[\right]
right3?ack \rightarrow left4!ack \rightarrow \text{ESTABLISHED}
\left[\right]
right1?fin \rightarrow left2!fin \rightarrow \text{FIN1}_A
\left[\right]
right3?fin \rightarrow left4!fin \rightarrow \text{FIN1}_B
\left[\right]
right1?fin?ack \rightarrow left2!fin!ack \rightarrow \text{FIN1}_A
\left[\right]
right3?fin?ack \rightarrow left4!fin!ack \rightarrow \text{FIN1}_B
\left[\right]

\text{FIN1}_A = \text{right1?syn} \rightarrow left2!syn \rightarrow \text{FIN1}_A
\left[\right]
right3?syn \rightarrow left4!syn \rightarrow \text{FIN1}_A
\left[\right]
right1?syn?ack \rightarrow left2!syn!ack \rightarrow \text{FIN1}_A
\left[\right]
right3?syn?ack → left4!syn!ack → FIN1_A
[ ]
right1?ack → left2!ack → FIN1_A
[ ]
right3?ack → left4!ack → FIN2_A
[ ]
right1?fin → left2!fin → FIN1_A
[ ]
right3?fin → left4!fin → FIN1_A
[ ]
right1?fin?ack → left2!fin!ack → FIN1_A
[ ]
right3?fin?ack → left4!fin!ack → CLOSE_A

FIN2_A = right1?syn → left2!syn → FIN2_A
[ ]
right3?syn → left4!syn → FIN2_A
[ ]
right1?syn?ack → left2!syn!ack → FIN2_A
[ ]
right3?syn?ack → left4!syn!ack → FIN2_A
[ ]
right1?ack → left2!ack → FIN2_A
[ ]
right3?ack → left4!ack → FIN2_A
[ ]
right1?fin → left2!fin → FIN2_A
[ ]
right3?fin → left4!fin → CLOSE_A
[ ]
right1?fin?ack → left2!fin!ack → FIN2_A
[ ]
right3?fin?ack → left4!fin!ack → CLOSE_A

CLOSE_A = right1?syn → left2!syn → CLOSE_A
[ ]
right3?syn → left4!syn → CLOSE_A
[ ]
right1?syn?ack → left2!syn!ack → CLOSE_A
[ ]
right3?syn?ack → left4!syn!ack → CLOSE_A
[ ]
right1?ack → left2!ack → NEW
[ ]
right3?ack → left4!ack → CLOSE_A
[ ]
right1?fin → left2!fin → CLOSE_A
[ ]
right3?fin → left4!fin → CLOSE_A
[]
right1?fin?ack -> left2!fin!ack -> CLOSE_A
[]
right3?fin?ack -> left4!fin!ack -> CLOSE_A

FIN1_B = right1?syn -> left2!syn -> FIN1_B
[]
right3?syn -> left4!syn -> FIN1_B
[]
right1?syn?ack -> left2!syn!ack -> FIN1_B
[]
right3?syn?ack -> left4!syn!ack -> FIN1_B
[]
right1?ack -> left2!ack -> FIN2_B
[]
right3?ack -> left4!ack -> FIN1_B
[]
right1?fin -> left2!fin -> FIN1_B
[]
right3?fin -> left4!fin -> FIN1_B
[]
right1?fin?ack -> left2!fin!ack -> CLOSE_B
[]
right3?fin?ack -> left4!fin!ack -> FIN1_B

FIN2_B = right1?syn -> left2!syn -> FIN2_B
[]
right3?syn -> left4!syn -> FIN2_B
[]
right1?syn?ack -> left2!syn!ack -> FIN2_B
[]
right3?syn?ack -> left4!syn!ack -> FIN2_B
[]
right1?ack -> left2!ack -> FIN2_B
[]
right3?ack -> left4!ack -> FIN2_B
[]
right1?fin -> left2!fin -> CLOSE_B
[]
right3?fin -> left4!fin -> FIN2_B
[]
right1?fin?ack -> left2!fin!ack -> CLOSE_B
[]
right3?fin?ack -> left4!fin!ack -> FIN2_B

CLOSE_B = right1?syn -> left2!syn -> CLOSE_B
[]
right3?syn -> left4!syn -> CLOSE_B
E.2.4 Checkpoint

Figure 15 on page 27 as CSP specification:

\{-
      TCP automation of Checkpoint R55W
revert engineering: Diana Senn
$Id: tcp-mid-Checkpoint.csp 28259 2006-07-10 09:08:53Z dsenn $
-\}

--datatype TCPFlags = syn | ack | fin
--channel left1 , right1 , left2 , right2 , left3 , right3 , left4 , right4 : TCPFlags

\{-
      left1 --NET1-- right1       left2 --NET2-- right2
      A
      FW
      right4 <--NET4-- left4      right3 <--NET3-- left3
      B

Akzeptierender Zustand: NEW
Startzustand: NEW
-\}

--START =

NEW = right1?syn -> left2!syn -> SYN.A
[ ]
right3?syn -> NEW
[ ]
right1?syn?ack -> NEW
[ ]
right3?syn?ack -> NEW
[ ]
right1?ack -> NEW
[ ]
right3?ack -> NEW
right 1?fin?ack \rightarrow \text{NEW}
right 3?fin?ack \rightarrow \text{NEW}

\text{SYN}_A = \begin{align*}
\text{right 1?syn} & \rightarrow \text{left 2!syn} \rightarrow \text{SYN}_A \\
\text{right 3?syn} & \rightarrow \text{SYN}_A \\
\text{right 1?syn?ack} & \rightarrow \text{left 2!syn!ack} \rightarrow \text{SYN}_A \\
\text{right 3?syn?ack} & \rightarrow \text{left 4!syn!ack} \rightarrow \text{SYN}_B \\
\text{right 1?ack} & \rightarrow \text{left 2!ack} \rightarrow \text{SYN}_A \\
\text{right 3?ack} & \rightarrow \text{SYN}_A \\
\text{right 1?fin?ack} & \rightarrow \text{left 2!fin!ack} \rightarrow \text{SYN}_A \\
\text{right 3?fin?ack} & \rightarrow \text{SYN}_A
\end{align*}

\text{SYN}_B = \begin{align*}
\text{right 1?syn} & \rightarrow \text{left 2!syn} \rightarrow \text{SYN}_B \\
\text{right 3?syn} & \rightarrow \text{SYN}_B \\
\text{right 1?syn?ack} & \rightarrow \text{left 2!syn!ack} \rightarrow \text{SYN}_B \\
\text{right 3?syn?ack} & \rightarrow \text{left 4!syn!ack} \rightarrow \text{SYN}_B \\
\text{right 1?ack} & \rightarrow \text{left 2!ack} \rightarrow \text{ESTABLISHED} \\
\text{right 3?ack} & \rightarrow \text{left 4!ack} \rightarrow \text{SYN}_B \\
\text{right 1?fin?ack} & \rightarrow \text{left 2!fin!ack} \rightarrow \text{SYN}_B \\
\text{right 3?fin?ack} & \rightarrow \text{left 4!fin!ack} \rightarrow \text{SYN}_B
\end{align*}

\text{ESTABLISHED} = \begin{align*}
\text{right 1?syn} & \rightarrow \text{ESTABLISHED} \\
\text{right 3?syn} & \rightarrow \text{ESTABLISHED} \\
\text{right 1?syn?ack} & \rightarrow \text{left 2!syn!ack} \rightarrow \text{ESTABLISHED} \\
\text{right 3?syn?ack} & \rightarrow \text{left 4!syn!ack} \rightarrow \text{ESTABLISHED} \\
\text{right 1?ack} & \rightarrow \text{left 2!ack} \rightarrow \text{ESTABLISHED} \\
\text{right 3?ack} & \rightarrow \text{left 4!ack} \rightarrow \text{ESTABLISHED}
\end{align*}
```
[]
right1?fin?ack  →  left2!fin!ack  →  FIN_A
[]
right3?fin?ack  →  left4!fin!ack  →  FIN_B

FIN_A = right1?syn  →  FIN_A
[]
right3?syn  →  FIN_A
[]
right1?syn?ack  →  left2!syn!ack  →  FIN_A
[]
right3?syn?ack  →  left4!syn!ack  →  FIN_A
[]
right1?ack  →  left2!ack  →  FIN_A
[]
right3?ack  →  left4!ack  →  FIN_A
[]
right1?fin?ack  →  left2!fin!ack  →  FIN_A
[]
right3?fin?ack  →  left4!fin!ack  →  CLOSE_A

CLOSE_A = right1?syn  →  left2!syn  →  SYN_A
[]
right3?syn  →  NEW
[]
right1?syn?ack  →  NEW
[]
right3?syn?ack  →  NEW
[]
right1?ack  →  left2!ack  →  NEW
[]
right3?ack  →  left4!ack  →  CLOSE_A
[]
right1?fin?ack  →  left2!fin!ack  →  CLOSE_A
[]
right3?fin?ack  →  left4!fin!ack  →  CLOSE_A

FIN_B = right1?syn  →  FIN_B
[]
right3?syn  →  FIN_B
[]
right1?syn?ack  →  left2!syn!ack  →  FIN_B
[]
right3?syn?ack  →  left4!syn!ack  →  FIN_B
[]
right1?ack  →  left2!ack  →  FIN_B
[]
right3?ack  →  left4!ack  →  FIN_B
```

85
E.2 TCP Automaton for Endpoints

\[
\begin{align*}
\text{right1?fin?ack} & \rightarrow \text{left2!fin!ack} \rightarrow \text{CLOSE}_B \\
\text{right3?fin?ack} & \rightarrow \text{left4!fin!ack} \rightarrow \text{FIN}_B
\end{align*}
\]

\[\text{CLOSE}_B = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{SYN}_A\]

\[
\begin{align*}
\text{right3?syn} & \rightarrow \text{NEW} \\
\text{right1?syn?ack} & \rightarrow \text{NEW} \\
\text{right3?syn?ack} & \rightarrow \text{NEW} \\
\text{right1?ack} & \rightarrow \text{left2!ack} \rightarrow \text{CLOSE}_B \\
\text{right3?ack} & \rightarrow \text{NEW} \\
\text{right1?fin?ack} & \rightarrow \text{left2!fin!ack} \rightarrow \text{CLOSE}_B \\
\text{right3?fin?ack} & \rightarrow \text{left4!fin!ack} \rightarrow \text{CLOSE}_B
\end{align*}
\]

E.2.5 ISA Server

Figure 18 on page 28 as CSP specification:

\[
\begin{align*}
\{ - \\
\text{TCP automaton of ISA Server} \\
\text{reverse engineering: Diana Senn} \\
\text{Id: tcp-mid-ISA.csp 28259 2006-07-10 09:08:53Z dsenn $} \\
\}
\end{align*}
\]

\[
\text{--datatype TCPFlags = syn | ack | fin}
\]

\[
\text{--channel left1, right1, left2, right2, left3, right3, left4, right4: TCPFlags}
\]

\[
\begin{align*}
\{ - \\
\text{left1 \rightarrow NET1\rightarrow right1} \\
\text{left2 \rightarrow NET2\rightarrow right2} \\
\text{right4 \leftarrow NET4\leftarrow left4} \\
\text{right3 \leftarrow NET3\leftarrow left3}
\end{align*}
\]

\[
\text{Akzeptierender Zustand: NEW} \\
\text{Startzustand: NEW}
\]

\[
\{ - \\
\text{START =}
\]

\[
\text{NEW = right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{SYN}_A \\
\text{right3?syn} \rightarrow \text{NEW} \\
\text{right1?syn?ack} \rightarrow \text{NEW}
\]

86
right3?syn?ack -> NEW
[]
right1?ack -> NEW
[]
right3?ack -> NEW
[]
right1?fin -> NEW
[]
right3?fin -> NEW
[]
right1?fin?ack -> NEW
[]
right3?fin?ack -> NEW

SYN_A = right1?syn -> left2!syn -> SYN_A
[]
right3?syn -> SYN_A
[]
right1?syn?ack -> left2!syn!ack -> SYN_A
[]
right3?syn?ack -> left4!syn!ack -> SYN_B
[]
right1?ack -> left2!ack -> SYN_A
[]
right3?ack -> left4!ack -> SYN_A
[]
right1?fin -> left2!fin -> SYN_A
[]
right3?fin -> left4!fin -> SYN_A
[]
right1?fin?ack -> left2!fin!ack -> SYN_A
[]
right3?fin?ack -> left4!fin!ack -> SYN_A

SYN_B = right1?syn -> left2!syn -> SYN_B
[]
right3?syn -> SYN_B
[]
right1?syn?ack -> left2!syn!ack -> SYN_B
[]
right3?syn?ack -> left4!syn!ack -> SYN_B
[]
right1?ack -> left2!ack -> ESTABLISHED
[]
right3?ack -> left4!ack -> SYN_B
[]
right1?fin -> SYN_B
[]
right3?fin -> left4!fin -> SYN_B
\[\]
right1?fin?ack \rightarrow \text{left2!fin!ack} \rightarrow \text{SYN}_B
\[
right3?fin?ack \rightarrow \text{left4!fin!ack} \rightarrow \text{SYN}_B
\]

\text{ESTABLISHED} = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{ESTABLISHED}
\[
\]
right3?syn \rightarrow \text{ESTABLISHED}
\[
\]
right1?syn?ack \rightarrow \text{left2!syn!ack} \rightarrow \text{ESTABLISHED}
\[
\]
right3?syn?ack \rightarrow \text{left4!syn!ack} \rightarrow \text{ESTABLISHED}
\[
\]
right1?ack \rightarrow \text{left2!ack} \rightarrow \text{ESTABLISHED}
\[
\]
right3?ack \rightarrow \text{left4!ack} \rightarrow \text{ESTABLISHED}
\[
\]
right1?fin \rightarrow \text{left2!fin} \rightarrow \text{FIN1}_A
\[
\]
right3?fin \rightarrow \text{left4!fin} \rightarrow \text{FIN1}_B
\[
\]
right1?fin?ack \rightarrow \text{left2!fin!ack} \rightarrow \text{FIN1}_A
\[
\]
right3?fin?ack \rightarrow \text{left4!fin!ack} \rightarrow \text{FIN1}_B
\]

\text{FIN1}_A = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{FIN1}_A
\[
\]
right3?syn \rightarrow \text{FIN1}_A
\[
\]
right1?syn?ack \rightarrow \text{left2!syn!ack} \rightarrow \text{FIN1}_A
\[
\]
right3?syn?ack \rightarrow \text{left4!syn!ack} \rightarrow \text{FIN1}_A
\[
\]
right1?ack \rightarrow \text{left2!ack} \rightarrow \text{FIN1}_A
\[
\]
right3?ack \rightarrow \text{left4!ack} \rightarrow \text{FIN2}_A
\[
\]
right1?fin \rightarrow \text{left2!fin} \rightarrow \text{FIN1}_A
\[
\]
right3?fin \rightarrow \text{left4!fin} \rightarrow \text{FIN1}_A
\[
\]
right1?fin?ack \rightarrow \text{left2!fin!ack} \rightarrow \text{FIN1}_A
\[
\]
right3?fin?ack \rightarrow \text{left4!fin!ack} \rightarrow \text{FIN1}_A
\]

\text{FIN2}_A = \text{right1?syn} \rightarrow \text{left2!syn} \rightarrow \text{FIN2}_A
\[
\]
right3?syn \rightarrow \text{FIN2}_A
\]

88
[]
right1?syn?ack → left2!syn!ack → FIN2\_A
[]
right3?syn?ack → left4!syn!ack → FIN2\_A
[]
right1?ack → left2!ack → FIN2\_A
[]
right3?ack → left4!ack → FIN2\_A
[]
right1?fin → left2!fin → FIN2\_A
[]
right3?fin → left4!fin → CLOSE\_A
[]
right1?fin?ack → left2!fin!ack → FIN2\_A
[]
right3?fin?ack → left4!fin!ack → CLOSE\_A

CLOSE\_A = right1?syn → left2!syn → CLOSE\_A
[]
right3?syn → CLOSE\_A
[]
right1?syn?ack → left2!syn!ack → CLOSE\_A
[]
right3?syn?ack → left4!syn!ack → CLOSE\_A
[]
right1?ack → left2!ack → NEW
[]
right3?ack → left4!ack → CLOSE\_A
[]
right1?fin → left2!fin → CLOSE\_A
[]
right3?fin → left4!fin → CLOSE\_A
[]
right1?fin?ack → left2!fin!ack → CLOSE\_A
[]
right3?fin?ack → left4!fin!ack → CLOSE\_A

FIN1\_B = right1?syn → left2!syn → FIN1\_B
[]
right3?syn → FIN1\_B
[]
right1?syn?ack → left2!syn!ack → FIN1\_B
[]
right3?syn?ack → left4!syn!ack → FIN1\_B
[]
right1?ack → left2!ack → FIN2\_B
[]
right3?ack → left4!ack → FIN1\_B
[]
right1?fin -> left2!fin -> FIN1_B
[]
right3?fin -> left4!fin -> FIN1_B
[]
right1?fin?ack -> left2!fin!ack -> FIN1_B
[]
right3?fin?ack -> left4!fin!ack -> FIN1_B

FIN2_B = right1?syn -> left2!syn -> FIN2_B
[]
right3?syn -> FIN2_B
[]
right1?syn?ack -> left2!syn!ack -> FIN2_B
[]
right3?syn?ack -> left4!syn!ack -> FIN2_B
[]
right1?ack -> left2!ack -> FIN2_B
[]
right3?ack -> left4!ack -> FIN2_B
[]
right1?fin -> left2!fin -> CLOSE_B
[]
right3?fin -> left4!fin -> FIN2_B
[]
right1?fin?ack -> left2!fin!ack -> CLOSE_B
[]
right3?fin?ack -> left4!fin!ack -> FIN2_B

CLOSE_B = right1?syn -> left2!syn -> CLOSE_B
[]
right3?syn -> CLOSE_B
[]
right1?syn?ack -> left2!syn!ack -> CLOSE_B
[]
right3?syn?ack -> left4!syn!ack -> CLOSE_B
[]
right1?ack -> left2!ack -> CLOSE_B
[]
right3?ack -> left4!ack -> NEW
[]
right1?fin -> left2!fin -> CLOSE_B
[]
right3?fin -> left4!fin -> CLOSE_B
[]
right1?fin?ack -> left2!fin!ack -> CLOSE_B
[]
right3?fin?ack -> left4!fin!ack -> CLOSE_B
E.3 TCP Automaton for Midpoints

E.3.1 CSP Model of the Problem

We start by modelling the ideal communication between two partners Alice (A) and Bob (B). In an ideal world, Bob directly sees all output from Alice as his input and vice versa. This results in the following CSP-Code, where `left1` and `left3` are output channels – of A and B respectively – and `right4` and `right2` are input channels.

```
{-
    ideal communication between A and B: specification

    left1 ----> right2
        A               B
    right4 <---- left3
-
} VSYSTEM6 = (A [||left1,right4||] (B[|right2<left1||left3<right4|])
```

The next step is to think about how a communication between A and B looks like if it takes part in a network with a firewall. This setting is shown in figure 42. If we take a more precise look at the two networks in this figure, we see that in every of these networks, there is communication in two ways. Our setting therefore looks as follows (where A and B are two agents correctly following the protocol):

```
{-
    real world communication between A and B: specification

    left1 --NET1---> right1   left2 --NET2---> right2
        A             FW        B
    right4 <--NET4-- left4    right3 <--NET3-- left3
-
} RSYSTEM = (A [||left1,right4||] (NET1 ||| NET4)
   [||right1,left4||] (FW)
   [||left2,right3||] (NET2 ||| NET3)
   [||right2,left3||] B))) \ {||right1,left4,left2,right3||}
```

Now we face the questions: What is the Firewall doing and what are the networks doing. Let us first look at the networks. A network gets packets as input. These packets can be...

- output (in arbitrary order)
- lost
• duplicated

Thus we model the networks as buffers – of a maximal size – with the above properties:

```plaintext
-- Buffer
-- s: the elements of the buffer
-- N: max. number of elements in the buffer
BUFF(s, N) =
   if null(s) then
     left?x -> BUFF(<x>, N)
   else
     \[|\| x:set(s) @ (\n       BUFF(<y | y<-s, y != x>, N) -- loose an element
     \|)
     right!x -> (\n       BUFF(<y | y<-s, y != x>, N) -- output element x
     \|)  \n   \|)  \n   #s<N & (STOP |\| left?x -> BUFF(s^<x>, N)) -- input element x
```

-- TODO: larger buffers

```plaintext
NET1 = BUFF(<>, 1)[[left<-left1]][[right<-right1]]
NET2 = BUFF(<>, 1)[[left<-left2]][[right<-right2]]
NET3 = BUFF(<>, 1)[[left<-left3]][[right<-right3]]
NET4 = BUFF(<>, 1)[[left<-left4]][[right<-right4]]
```

Now let us take a look at the firewall. What is the duty of a firewall? There exist many answers to this question! Our answer is that it should allow all communications possible when the endpoints correctly follow the protocol. So if we model the endpoint protocols correctly for A and B and the above is our answer to the firewall's duty, then the following is our model:

```plaintext
COPY = from?x -> to!x -> COPY

FW1 = COPY[[from<-left1]][[to<-left2]]
FW2 = COPY[[from<-left3]][[to<-left4]]
FW = FW1 ||| FW2
```

Having a specification, we can now compare it to the different automata used by firewall vendors. We will show how this looks like at the example of TCP. To find out if these automata conform to our specification (RSYSTEM given above), we must analyse them in a setting where the endpoints can do anything – that is what happens in the real world – and see if the firewall blocks all unwanted traffic, it is if the same traces – with respect to the firewall output – are possible in the two settings. We have modelled the described real world setting as follows (where A2 and B2 are two agents who do not follow the protocol correctly):

```plaintext
-- real world with FW working as in our specification
RSYSTEM2 = (A2 [[[{left1,right4}]]] (NET1 ||| NET4)
   [/[{{right1,left4}}]] ((FWSPEC)
   [/[{{left2,right3}}]] (NET2 ||| NET3)
   [/[{{right2,left3}}]] B2))) \ {[right1,left4,left2,right3]}
```
-- real world with FW working as in major firewall products
-- (either iptables, Checkpoint or ISA Server)
RSYSTEM3 = \{\{\{\{\text{left1}, \text{right4}\}\}\}\}\} (\text{NET1} \lor \text{NET4})
        \{\{\{\text{right1}, \text{left4}\}\}\}\} (\text{FWIMPL})
        \{\{\{\text{left2}, \text{right3}\}\}\}\} (\text{NET2} \lor \text{NET3})
        \{\{\{\text{right2}, \text{left3}\}\}\}\} \}\} \}\} \}\} B2))) \{\{\{\{\text{right1}, \text{left4}, \text{left2}, \text{right3}\}\}\}\}\} \\

E.3.2 Results

In the last section we gave our model of the problem. Now we want to do some checks on this model, but
first we need to specify some details:

-- TCP specification for endpoints
include "tcp.csp"
A = CLOSED[[in<-right4]][[out<-left1]] -- correct behaviour
B = LISTEN[[in<-right2]][[out<-left3]]

include "tcp_inc.csp" -- incorrect behaviour
A2 = INC[[in<-right4]][[out<-left1]]
B2 = INC[[in<-right2]][[out<-left3]]

-- TCP specification for midpoints
include "tcp-mid-diana.csp" -- ours
FWSPEC = START

--include "tcp-mid-iptables.csp" -- iptables
--include "tcp-mid-Checkpoint.csp" -- Checkpoint R55W
include "tcp-mid-ISA.csp" -- ISA Server
FWIMPL = NEW

channel left1, left2, left3, left4: TCPFlags
channel right1, right2, right3, right4: TCPFlags
channel from, to: TCPFlags
channel right, left: TCPFlags

Now we have everything and we can start to do some checks. But before giving the CSP code for them, we first want to elaborate on what has to be checked. We want to check the following things:

- real world and ideal specification show equivalent behaviour
- the relationship between real world firewall implementations and our real world specification

{- the implementation is a refinement of the specification -} -- TODO
assert VSYSTEM6 [T= (RSYSTEM \ \{\{\text{right2}, \text{left3}\}\}\} --
assert (RSYSTEM \ \{\{\text{right2}, \text{left3}\}\}\} [T= VSYSTEM6 --

assert VSYSTEM5 [T= RSYSTEM --
assert RSYSTEM [T= VSYSTEM5 --

assert VSYSTEM6 [T= (VSYSTEM5 \ \{\{\text{right2}, \text{left3}\}\}\} -- false
{- VSYSTEM5: Performs left1.syn, _tau, _tau, right4.syn, _tau, right4.ack,
left1.ack, _tau, left1.fin, _tau, _tau, right4.ack, _tau, right4.fin, left1.ack, _tau, left1.syn

VSYSTEM6: Behaviour not relevant

\)
assert (VSYSTEM5 \ {left2, left3}) \T= VSYSTEM --- o.k.

{- a complete TCP connection initiation and termination is possible -}

KOMABLAUF = left1!syn \rightarrow right4?syn \rightarrow right4?ack \rightarrow left1!ack \rightarrow left1!fin

\rightarrow right4?ack \rightarrow right4?fin \rightarrow left1!ack \rightarrow STOP

assert (RSYSTEM \ {left2, left3}) \T= KOMABLAUF --- o.k.
assert (VSYSTEM5 \ {left2, left3}) \T= KOMABLAUF --- o.k.
assert (VSYSTEM6 \ {left2, left3}) \T= KOMABLAUF --- o.k.

{- our TCP midpoint specification is correct -}
assert (RSYSTEM \ {left1,right2,left3,right4}) --- o.k.

\T=
(RSYSTEM2 \ {left1,right2,left3,right4})
assert (RSYSTEM2 \ {left1,right2,left3,right4}) --- o.k.

\T=
(RSYSTEM \ {left1,right2,left3,right4})

{- the real world implementations allow the same connections as the
real world specification
assert (RSYSTEM \ {left1,right2,left3,right4})

\T=
(RSYSTEM3 \ {left1,right2,left3,right4})

{-
o.k. fr iptables und BUFF-Grasse 1
fr Checkpoint und BUFF-Grasse 1
o.k. fr ISA Server und BUFF-Grasse 1
-}
assert (RSYSTEM3 \ {left1,right2,left3,right4})

\T=
(RSYSTEM \ {left1,right2,left3,right4})

{-
o.k. fr iptables und BUFF-Grasse 1
fr Checkpoint und BUFF-Grasse 1
fr ISA Server und BUFF-Grasse 1
-}

\T=

-- TODO: RSYSTEM2 vs. RSYSTEM3

E.3.3 Conclusion
Already with buffer sizes of 1, FDR2 took about 12 hours to check one assertion, e.g.

assert (RSYSTEM \ {left1,right2,left3,right4})

\T=
(RSYSTEM2 \ {\{left1, right2, left3, right4\}})

When incrementing the buffer size (to 2 or 3), FDR2 either crashed or could not compute the result due to memory lack (even with 2GB of memory).

Due to this, we cannot make any statements, based on this model, for the midpoint automaton problem in a realistic setting. In a real network, thousands of packets are present at the same time. We do not have to be able to model thousands of packets for being able to make realistic statements, but at least 10 or better 100 packets per buffer are necessary, to let the complex interactions in networks – such as changed order of packets – come into play.

We therefore conclude that CSP is suitable to model the problem, but that there exists – to our knowledge – no model checker for CSP which can handle the complexity of our problem.
References


REFERENCES


[oCS] Polytechnic University Department of Computer and Information Science. Security policy.

[OGA] Xinming Ou, Sudhakar Govindavajhala, and Andrew Appel. Network security management with high-level security policies.

97
REFERENCES

[0SC81] Information Sciences Institute University of Southern California. RFC 793: Transmission control protocol, September 1981.


Index

CSP, 76
  channel, 76
  input, 76
  output, 76
  function, 76
  process, 76
  composition, 76
  state, 76
  synchronisation, 76
endpoint, 18 24
endpoint automaton, 18
FDR, 76
  traces model, 76
  traces refinement, 76
firewall, 5 8 13
  abstract rules, 8 63
  application layer firewall, 5
  configuration, 8
  filter, 13
  implementation, 6 8
  iptables, 43
  network layer firewall, 5
  packet filter, 8
    stateful, 6
    stateless, 5
  rules, 5 18
  ruleset, 6
  testing, 7

Internet, 5

keyword definitions, 9

Mealy machine, 18 61
midpoint, 23
  protocol automaton, 19
model checking, 60
NAT, 5
  network, 18
    connection, 18
    initiator, 18
    responder, 18
DMZ, 18
layout, 9 13 38
  packet
    destination, 18
    source, 18
  typical, 15
network model, 30

policy
  access policy, 5
  formal policy, 3 10 61
  informal, 8
  network policy, 9
  security policy, 13 51
protocol
  automaton, 18
  protocol specification, 10

test case, 11
  abstract test case, 18 10
  adaptive, 13 14 22
  concrete test case, 10
  preset, 13 14 22
  test tuple, 18

test data, 11
  reliable, 11
  valid, 11

test generation, 11
  DS method, 13
  UIO sequences method, 18
  UIOv method, 13
  W-Method, 12
  Wp-Method, 13

test input, 11

test output, 11

test set
  complete, 11
  successful, 11

test tree, 12


testing, 18 52
  automated testing, 21
  characterisation set, 12
  conformance testing, 11
  firewall testing, 7
  penetration testing, 55 60
  software testing, 72
  UIO sequence, 18

tool
tfittest, 41

validation, 11

verification, 11 60


99