Firewall Conformance Testing*

Diana Senn, David Basin, Germano Caronni

ETH Zürich, 8092 Zürich, Switzerland
{dsenn,basin}@inf.ethz.ch, gec@acm.org

Abstract. Firewalls are widely used to protect networks from unauthorised access. To ensure that they implement an organisation's security policy correctly, they need to be tested. We present an approach that addresses this problem. Namely, we show how an organisation's network security policy can be formally specified in a high-level way, and how this specification can be used to automatically generate test cases to test a deployed system. In contrast to other firewall testing methodologies, such as penetration testing, our approach tests conformance to a specified policy. Our test cases are organisation-specific — i.e. they depend on the security requirements and on the network topology of an organisation — and can uncover errors both in the firewall products themselves and in their configuration.

1 Introduction

Firewalls are a common and widely deployed technology to control access to networked systems. Although they are sometimes viewed as an "appliance" that can be used out of the box, considerable work is required in practice to configure them so that they implement an organisation's network security policy. To ensure that this is done properly and that the employed firewalls then behave as expected, the entire setup must be tested. This is particularly important in high-security environments like banking or in military settings.

In this paper we present an approach to specification-based firewall testing, where an organisation's network security policy comprises the specification. This can also be called firewall conformance testing, as it tests if the firewalls conform to the network security policy. Our motivation to follow this path comes from the fact that there is a wide range of security needs and network topologies, and a firewall testing procedure should be tailored to both of these. Note that such testing says nothing about the appropriateness of the security policy itself. For this, a separate analysis of the security policy is needed.

Our approach is based on the following ideas: First, we propose a formal language for specifying network security policies. Second, we show how to automatically generate test cases from formal policies. A *test case* consists of test input, also called *test data*, and the expected test output. Our test cases consist

* This work was partially supported by armasuisse. It represents the views of the authors.

of a series of network packets (test data) and a statement per packet whether we expect this packet to reach its destination or not. We are testing firewalls and use the term *firewall implementation* to denote everything that is delivered by the firewall manufacturer, and the term *firewall configuration* (or *firewall rule set*) to denote its configuration by the customer. By executing the generated test cases directly on the real network (as opposed to simulation), we can find errors both in the firewall configuration and in the firewall implementation. These tests can be done just before deploying a network, or after configuration updates. Note that we do not explicitly search for all possible bugs in the firewall implementation as is sometimes done in penetration testing. Rather, our method succeeds in finding all policy related errors.

The contributions of this paper are a language for the formal specification of network security policies, the novel combination of different methods for generating abstract test cases, and an algorithm for generating concrete test cases from policies. As firewalls can be very complex, we have made some simplifying assumptions in this paper: We assume that all firewalls are stateful packet filters[1], and we do not test for problems with timing or sequence numbers. Future work will aim at eliminating these simplifications and at carrying out large-scale case studies.

This paper is organised as follows: We give a comparison with related work in Section 2. Then we present our formal policy specification language in Section 3, and the process of test case generation in Section 4. The entire process is then illustrated on an example in Section 5. We conclude and report on future work in Section 6.

2 Related Work

Most preexisting work on firewall testing covers different aspects of testing by hand [Hae97,Sch96]. Constructing these tests relies on human experts who mainly focus on detecting known vulnerabilities, for example forwarding external packets that claim to come from an internal source. Most of this testing falls under the heading of *penetration testing*.

Over the last few years, a new approach to firewall testing was taken by [BMNW99,JW01], which can be called *specification-based firewall testing*. The general idea comes from specification-based software testing: The specification is used to generate test cases, against which the system implementation is tested. In the case of specification-based firewall testing, the system is the firewall and

[1] A *packet filter* can filter traffic only at OSI Layer 4 (TCP and UDP), whereas an *application level firewall* can interpret and filter higher-level protocols. A *stateful packet filter* can forward (changed or unchanged), drop, or reject a packet based on its source IP address, the packet's source port, its destination IP address, its destination port, its TCP flags, and the state of the connection the packet belongs to.

the specification is a security policy. The difference to penetration testing is substantial: Whereas penetration testing tends to always use the same test cases, in specification-based testing they depend on the policy and are designed explicitly to test conformance with the policy.

These specification-based approaches share some similarities with ours. Wool et al. [BMNW99,BMNW03,MWZ00,Woo01], for example, gather the configurations of the network and the firewalls, and then simulate the network under test, i.e. they start with the firewall rules instead of with the policy. In their simulation, tests can then be executed easily and without doing harm. There are a number of tools (Firmato [BMNW99,BMNW03], Fang [MWZ00], LFA [Woo01]) implementing this approach. The disadvantages of this approach are twofold: First it relies on the correctness of the firewall implementations and second it needs to interpret the different firewall rule languages (of the different vendors). Thus it tests a model, which will never model reality perfectly. We treat firewalls as black boxes and therefore can test firewalls without having to formalise and provide a semantics for their rule language. Additionally, by carrying out tests on the real network, we can find errors in not only the firewall configuration, but also in the firewall implementation. The advantage of testing a model, as Wool et al. do, is that there is no interaction with a running system and that the tests themselves can do no harm.

An approach similar to specification-based firewall testing is to generate the firewall rules from a formal policy [Gut97,BCG+01]. What firewall rule generation has in common with specification-based firewall testing is that both need a formal specification. Guttman [Gut97], for example, takes an approach similar to formalising policies as we do, but includes all the low-level details. He then models the network as a bipartite graph and computes the individual rules for each firewall from the global policy by completing a graph traversal of this model. His aim is different from ours: he generates the firewall rules from the policy, whereas we assume that there are firewall rules whose correctness we would like to check. What our work has in common is that we also need a formal policy. However, his policies are specified in a way that they easily become too detailed and therefore are subject to policy errors. In contrast, in our approach, we separate the low-level details from the policy, thereby making policies easier to understand and policy writing less error-prone.

3 A Formal Network Security Policy

A network security policy formalises what kind of traffic is allowed between different zones. A *zone* is a part of a network that is separated from the rest of the network by means of one or more firewalls. In what follows, we will use the term *policy* to denote any part of a security policy, be it formal or informal, and the term *formal policy* for our formalisation of the *network security policy.* Note that in our work, we shall assume that users do not play a role in policies.

Figure 1 illustrates a network with three zones: the public Internet, the demilitarised zone (DMZ), and the private Intranet of a company. In this example,

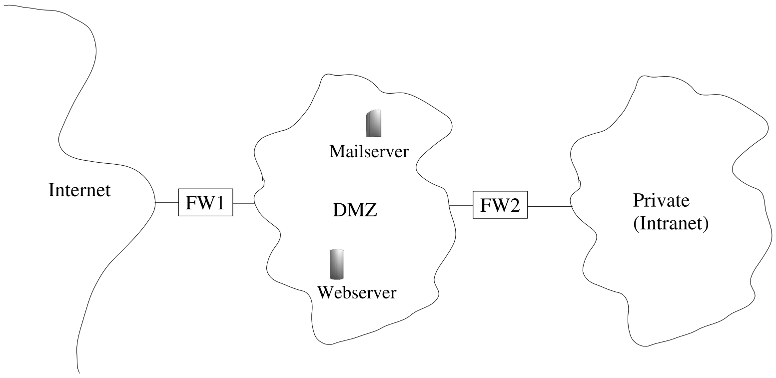

Fig. 1. A sample graphical network layout

our network also has two servers, a Mailserver and a Webserver, standing in the DMZ. A policy for this network defines what traffic is allowed to flow between these different zones. This can be direction dependent (for example, there may be different restrictions on what connections are allowed to be initiated from the private zone to the Internet than the other way) and thus we need two rules per pair of zones. We assume that all clients in a zone are equivalent, in that differences in their IP addresses have no effect on the firewalls' behaviours[2]. In contrast, we do distinguish between servers, as we would like to be able to state different policies for different servers. Thus, in the example given, instead of stating policies for traffic flowing to and from the DMZ, we state individual policies for the two servers. Policies for traffic within zones need not be specified, since compliance with these policies cannot be enforced by firewalls.

```
@ Connections to Private
DMZ → Private:              ACCEPT securetraffic
Internet → Private:         DENY *

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

@ Connections to the Internet
Private → Internet:         ACCEPT *
DMZ → Internet:             DENY *
```

Fig. 2. A sample formal policy

Figure 2 gives an example of a formal policy for the given network, where lines starting with @ are comments, and where * means everything. The second

[2] This represents our *uniformity hypothesis.*

line, for example, states that only secure traffic is allowed from the DMZ zone to the private zone. But which connections are secure? This can change quickly (for example, when a new ssh weakness is found) and is therefore stated separately from the policy in what we call *keyword definitions*. The idea is that our policy is expressed at a high-level; this way it is both manageable and understandable by managers as well as security specialists. Because of this, we also use names for network zones. The low-level details (IP-addresses, etc.), which may be subject to frequent change, are stored separately in what we call a *textual network layout*. The graphical version presented in Figure 1 does not contain these low-level details.

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

Fig. 3. Keyword definitions

An example of keyword definitions is given in Figure 3. Here the security engineer has decided that SSH, SCP, HTTPS, and IMAPS are secure protocols. These protocols are application level protocols. At the TCP level — where stateful packet filters work — they are represented by their TCP port number.

DMZ: 129.132.178.192/27
Private (Intranet): 192.168.1.0/24
Internet: !DMZ, !Private

* * *

@ Name of the Firewall	Interface	Comment
FW1	eth0 (0.0.0.1)	
FW1	eth1 (129.132.178.193)	
FW2	eth0 (129.132.178.194)	Packet filter
FW2	eth1 (192.168.1.1)	Packet filter

* * *

@ Name (fac.)	IP	Service
Mailserver	129.132.178.200	smtp
Mailserver	129.132.178.200	imap
Webserver	129.132.178.197	http

Fig. 4. A sample textual network layout

An example of a textual network layout, providing the low-level details of the network shown in Figure 1, is given in Figure 4. The first part gives the IP-address-ranges for the zones in CIDR notation [FLYV93]. For example, 129.132.178.192/27 means that the first 27 bits are used to represent the network and the remaining 5 bits are used to identify hosts, which results in the 30 hosts starting at 129.132.178.193 and ending at 129.132.178.222 (note that

129.132.178.192 represents the network address, and 129.132.178.223 represents the broadcast address). The ! operator represents set complement with respect to the universe of all possible IP addresses (from 0.0.0.0 to 255.255.255.255) and thus the third line means that the Internet consists of everything other than the DMZ and the Private zone. The second part provides the IP addresses of the firewall-interfaces along with some comments. The last part lists the IP addresses of the servers together with the service they provide. The stars separate the different parts and @ again represents comments.

The grammars for formal policies, keyword definitions, and textual network layout can be found in the Appendix.

4 Test Case Generation

In this Section, we first present our method for test case generation in detail, before giving a concrete example in Section 5. Our test case generation consists of two parts. First we generate *test tuples* from the formal policy. Afterwards we generate *abstract test cases*. The idea of the abstract test cases is to test the correct stateful handling of a protocol by a firewall. For example, a stateful packet filter may be tested to determine whether it correctly handles TCP traffic. To generate the *concrete test cases*, we instantiate the abstract test cases with the test tuples.

4.1 Abstract Test Cases

We must generate a set of abstract test cases for every protocol we want to test. Once we have these test cases, we can use them for every test concerning this protocol. The generation consists of two steps: We first construct a Mealy automaton describing the protocol for which we want to generate abstract test cases. Then we generate test cases for this Mealy automaton using the well known UIO-sequences method [SD88]. We will now explain the generation in detail.

Mealy Automata

Definition 1. *A Mealy automaton is a six-tuple* $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_1)$*, where* $Q = \{q_1, q_2, ..., q_{|Q|}\}$ *is a finite set of* states*,* $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_{|\Sigma|}\}$ *is a finite* input alphabet*,* $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_{|\Gamma|}\}$ *is a finite* output alphabet*,* $\delta : Q \times \Sigma \to Q$ *is the* transition function*,* $\lambda : Q \times \Sigma \to \Gamma$ *is the* output function*, and* $q_1 \in Q$ *is the* initial state*.*

In our models, the input of a transition represents the packet (we only model the parts of it essential for determining the firewall's action) reaching the firewall, and the output represents the corresponding packet leaving the firewall. A typical input has the form $x : A \to B$, where x represents packet information being sent from source A to destination B. The output packet can either be the same as the input packet, different from the input packet (i.e. changed by the firewall), or non-existent (dropped by the firewall). Thus $\Sigma = \Gamma \cup \{-\}$, where the "–" symbol represents no output.

Test Cases for Mealy Automata

The general idea behind testing a specification given as a Mealy automaton M_{spec} is to ensure that every transition of M_{spec} is correctly implemented, where the implementation is also assumed to be a Mealy automaton M_{imp}. This is achieved by testing every transition in M_{spec}, say from state s_i to state s_j, according to the following steps:

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

Step one is easy if there is a reliable reset: Just apply the reset input to go back to the initial state. The TCP protocol, which we present in the next Section, has such a reliable reset.

Steps two and three can be solved by building a *test tree* T according to the following rules and afterwards traversing all the paths [Cho78]:

Level 1: Label the root of T with the initial state of M_{spec}.

Level (k+1): Examine the 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 M_{spec_i} denote its label. If on input x, machine M_{spec} goes from state M_{spec_i} to state M_{spec_j}, then we attach a branch and a successor node to the node labelled M_{spec_i} in T. The branch and the successor node are labelled with x and M_{spec_j}, respectively.

Step 4 can be achieved by using either the *W-method* [Cho78], *UIO sequences* [SD88], or *distinguishing sequences* [Gil61,Gil62]. All these methods achieve the same fault coverage. We have chosen the UIO sequences because they generate the shortest test cases of the three methods. In brief, a UIO sequence is an input/output sequence x for a state s that distinguishes s from all other states, i.e. $\lambda(s_i, x) \neq \lambda(s, x)$, for all $s_i \neq s$.

We can now fit the pieces together to generate our abstract test cases: We take every possible path in the test tree, prepend it with the reset input, and append the UIO sequence of the end state (of the path). Thus we get a set of abstract test cases, where every abstract test case consists of a series of I/O tuples (describing input and expected output). Every abstract test case starts with the reset input to bring the machine back into its initial state, followed by a series of I/O tuples to bring the machine into some state s_i and one I/O tuple to test the transition from state s_i to state s_j (extracted from the test tree), and finally a series of I/O tuples (the UIO sequences) to verify that state s_j was reached.

4.2 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. A test tuple describes whether a connection from the source *sIP* to the destination *dIP* 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. Note that the statefulness of a connection is not modelled by these test tuples, but rather by the abstract test cases.

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

```
source → 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 `source` and `destination` are replaced with the corresponding IP ranges.

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 (grey for ACCEPT, black for DROP). An example of this is given in Section 5, Figure 7.

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.

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 7 for an example). In the graph, we coloured all the areas where we have an explicit policy statement (either in grey or black). This means that for all the uncoloured areas there exists no explicit policy statement. Note that a part of the uncoloured area is not testable since, as we stated earlier, policies for traffic within zones cannot be enforced by firewalls. But the rest of the uncoloured areas can be partitioned into rectangles and then test tuples can be chosen, analogous to the procedure given above, where the expectation is set to DROP.

The resulting test tuples are then used to instantiate the abstract test cases. How this is done is explained in the next Subsection.

4.3 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 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.

In this paper, we only consider the testing of stateful packet filters. This means that we only need abstract test cases for TCP and UDP, but not for every possible (application-level) protocol. 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.

As described above, our abstract test cases are generated from Mealy automata using the UIO sequences method. The resulting unoptimised test sequences have length $O(mn^2)$ per automaton, where m denotes the number of transitions and n denotes the number of states of the automaton (Theorem 3 of [SD88]). As test sequences can be optimised, i.e. subsequences completely contained in others can be eliminated, the above complexity bound represents the worst case.

The work needed for generating test tuples is the following: If we have a policy containing r rules, and at most p protocols per keyword, we get $O(rp)$ test tuples. The generation of the abstract test cases needs only be done once per protocol, i.e. this is a one-time cost. The generation of the test tuples and the instantiation of concrete test cases based on them has to be done once per policy. As we use each test tuple to instantiate at most $O(mn^2)$ abstract test cases, in the worst case we generate $O(rpmn^2)$ concrete test cases.

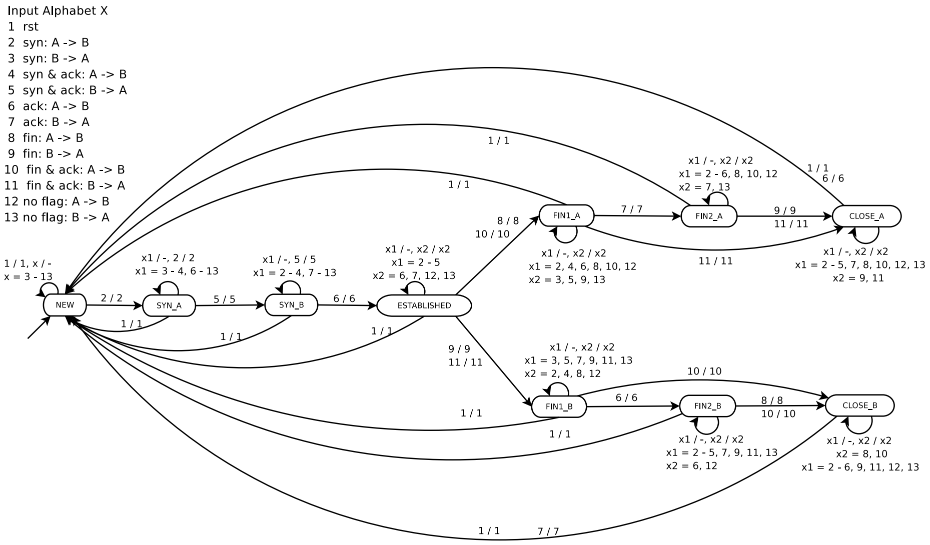

Fig. 5. Automaton for TCP

When testing Mealy automata, we can distinguish between two types of errors: *operation errors*, which are errors in the output function, and *transfer errors*, which are errors in the next state function. If the implementation automaton has the same number of states as the specification automaton, then we can detect all errors of both kinds, and our abstract test cases are reliable and valid in the sense described by [GG75]. If there are extra states in the implementation, the UIO sequences method we use may however miss errors.

If our uniformity hypothesis holds, i.e. the firewall reacts in the same way to all clients within a zone, then our test tuples represent all possible connections (between every possible source and destination). Hence instantiating the abstract test cases with these test tuples, the resulting concrete test cases are reliable and valid.

5 An Example

Abstract Test Cases for TCP

A graphical Mealy automaton for the TCP protocol is given in Figure 5. The automaton is not a full specification: sequence numbers and acknowledgement numbers have been omitted. Also the input alphabet does not contain all possible combinations of flags. But the central parts of the protocol are specified. The respective input and output of each transition are written next to the transition and are separated by a slash. The input `fin: A` $\rightarrow$ `B`, for example, stands for a TCP packet sent from A to B, where exactly the `fin` flag is set. A and B stand for two hosts and are instantiated with concrete IP addresses later.

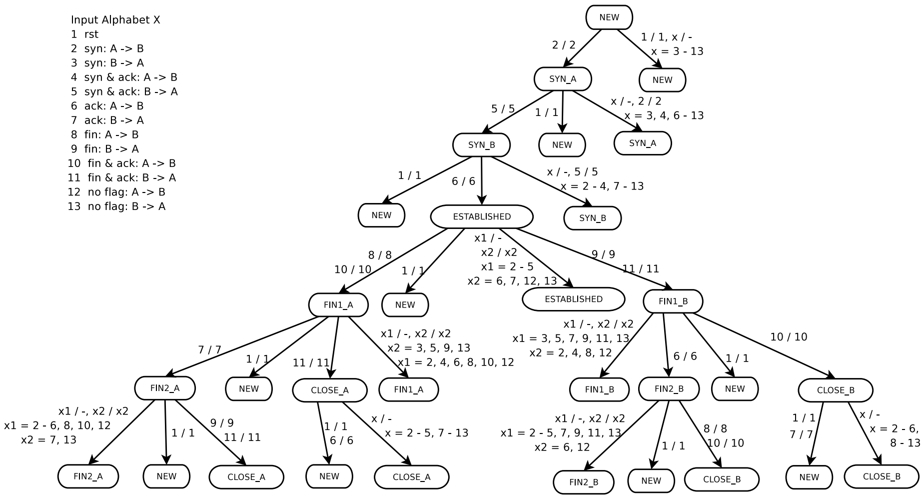

Fig. 6. Test Tree for TCP

From the Mealy automaton for TCP, we construct a test tree using the method given in Section 4.1. The test tree for TCP is given in Figure 6: NEW is the start state of the Mealy automaton and represents level 1 of the test tree. From the state NEW there are two transitions, one back to NEW and one to the state SYN_A. Thus the states NEW and SYN_A represent level 1 of the test tree. As we already had state NEW in the test tree, the test tree is continued only for state SYN_A.

Proposition 1. *The UIO sequences for TCP are:*

NEW:	*(5/-)(2/2)*
SYN_A:	*(6/-)(5/5)*
SYN_B:	*(7/-)(6/6)(9/9)(10/10)*
ESTABLISHED:	*(8/8)(11/11)*
FIN1_A:	*(7/7)(9/9)(6/6)(2/2)*
FIN2_A:	*(9/9)(6/6)(2/2)*
CLOSE_A:	*(6/6)(2/2)*
FIN1_B:	*(6/6)(8/8)(7/7)(2/2)*
FIN2_B:	*(8/8)(7/7)(2/2)*
CLOSE_B:	*(7/7)(2/2)*

As an example, consider the UIO sequence of the state NEW. On input `syn: A` $\rightarrow$ `B`, only the states NEW and SYN_A will respond with output `syn: A` $\rightarrow$ `B`; all the other states will have no output. As the state SYN_A, in contrast to state NEW, will also respond to `syn & ack: B` $\rightarrow$ `A`, we identify the state SYN_A if we send the packets `syn: A` $\rightarrow$ `B` and `syn & ack: B` $\rightarrow$ `A` and only see the second packet behind the firewall.

We will now construct two test cases according to the four step procedure given in Section 4.1. Our first test case should test the transition from state NEW to itself.

1) Bring the machine into its initial state: (1 / 1).
2) Transfer the machine into state NEW: no action is needed here.
3) Test the transition: (8 / -) is one possibility.
4) Verify that the machine is now in state SYN_A: (5 / -)(2 / 2) is the UIO sequence of state SYN_A.

Thus the resulting test case is (1 / 1)(8 / -)(5 / -)(2 / 2).

Our second test case should test the transition from state SYN_B to state ESTABLISHED.

1) Bring the machine into its initial state: (1 / 1).
2) Transfer the machine into state SYN_B: (2 / 2)(5 / 5).
3) Test the transition: (6 / 6).
4) Verify that the machine is now in state ESTABLISHED: (8 / 8)(11 / 11) is the UIO sequence of state ESTABLISHED.

Thus the resulting test case is (1 / 1)(2 / 2)(5 / 5)(6 / 6)(8 / 8)(11 / 11). Analogous to the two examples given, test cases for all the other transitions need to be constructed.

An Example of Test Tuples

We will illustrate the generation of test tuples on the example of the formal policy given in Figure 2. Apart from the policy, we need the keyword definitions given in Figure 3 and some knowledge (i.e. IP addresses) of the network under test (given in Figure 1). Let us assume that we have the information about the network under test given in Figure 4.

As an example, we generate test tuples for the HTTPS protocol. HTTPS is contained in the keywords *securetraffic* and *webtraffic*. Therefore we have to build and colour a graph for all rules of the formal policy except the fourth one. The result can be seen in Figure 7. In this graph, test tuples are marked by a circle, and untestable areas are marked with a question mark. One example of such a test tuple is (129.132.178.192, 192.168.0.255, https, DENY).

An Example of a Concrete Test Case

In this example, we have only generated abstract test cases for TCP. Thus we instantiate these abstract test cases with the above test tuples, to generate concrete test cases. For this, we represent every application level protocol with its TCP port number (e.g. 443 for HTTPS).

As an example, we present the instantiation of one abstract test case with two test tuples.

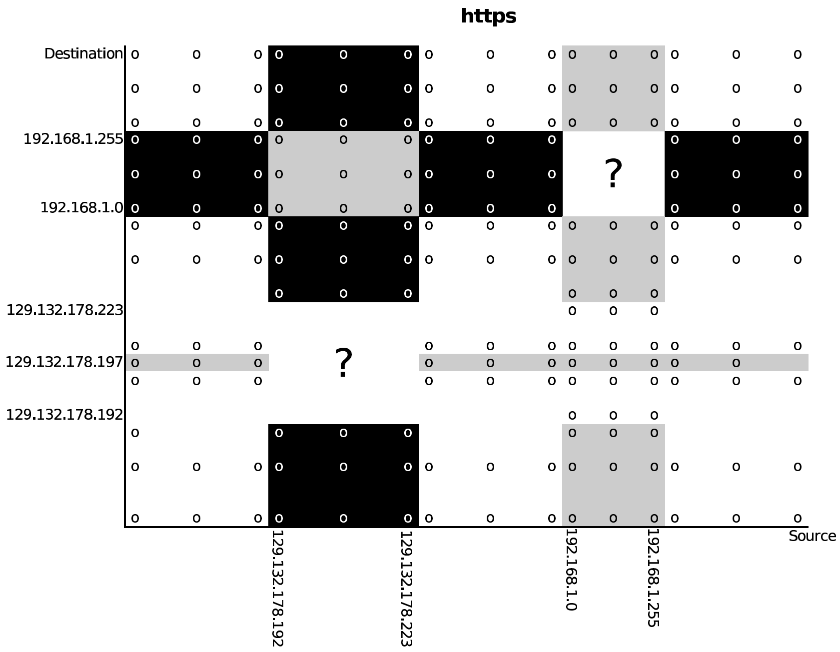

Fig. 7. Policy for https with test points

Example 1. Test Case Instantiation
an abstract test case for TCP: (1 / 1)(8 / -)(5 / -)(2 / 2).
test tuple 1: (129.132.178.192, 192.168.0.255, HTTPS, DENY)
test tuple 2: (129.132.178.192, 192.168.1.0, HTTPS, ACCEPT)

Using the first test tuple we get the concrete test case:
(rst: 129.132.178.192 → 192.168.0.255:443 / –)
(fin: 129.132.178.192 → 192.168.0.255:443 / –)
(syn & ack: 192.168.0.255:443 → 129.132.178.192 / –)
(syn: 129.132.178.192 → 192.168.0.255:443 / –)

Using the second test tuple we get:
(rst: 129.132.178.192 → 192.168.1.0:443 / rst 129.132.178.192 → 192.168.1.0:443)
(fin: 129.132.178.192 → 192.168.1.0:443 / –)
(syn & ack: 192.168.1.0:443 → 129.132.178.192 / –)
(syn: 129.132.178.192 → 192.168.1.0:443 / syn: 129.132.178.192 → 192.168.1.0:443)

Let us explain these two test cases. Recall that a test case (this holds for the abstract and the concrete test cases) is composed of a series of input and expected output packets. Each of the above test cases contains four such I/O pairs. Thus for the first concrete test case we try to initiate a https-connection from 129.132.178.192 to 192.168.0.255, where we expect the firewall to drop all these packets. That is we test that a https-connection from 129.132.178.192 to 192.168.0.255 is not allowed.

The second concrete test case belongs to a series of test cases that test if a https-connection can be initiated from 129.132.178.192 to 192.168.1.0 and if this is done correctly, i.e. they test whether the firewall handles the TCP connection correctly. This specific test case tests the start of such a connection (as explained in the last Subsection). The first packet resets the connection and should be accepted by the firewall. The second packet attempts to close the connection, but as the connection no longer exists (it was reset before), this packet should not be allowed, and therefore should be dropped by the firewall. The third packet is not the start of a new connection and thus should be dropped as well, and finally the fourth packet initiates a new connection and should be let through.

With the second concrete test case we can find different kinds of errors: 1) A bug in the firewall implementation if the `fin` or the `syn & ack` packet is let through (i.e. the stateful connection handling is incorrect), and 2) a bug in the firewall configuration if the `syn` packet is blocked.

6 Conclusion

We have presented a new approach to test the conformance of firewalls to a given security policy. Our contributions are the following: a language for the formal specification of network security policies, the novel combination of different methods for generating abstract test cases, and an algorithm for generating concrete test cases from the policy. Overall, our method is designed to find errors both in the firewall implementation and the firewall specification.

In this paper our focus has been on the theoretical basis of our approach. We are currently implementing a prototype testing tool based on this work. We plan to use this tool to conduct case studies, to see how effective our method is in finding errors as well as to determine its robustness. An interesting scenario will be to stress test the firewalls, i.e. to run many different test cases at the same time.

To reduce the complexity of the problem, we have simplified matters by assuming that our firewalls are stateful packet filters and by not testing for problems with timing or sequence numbers. As a next step, we plan to eliminate these simplifications. In particular, we shall adapt our approach to application-level firewalls. As some application-level protocols are difficult to handle by a firewall, e.g. SIP [RSC+02] needs dynamic port opening, this problem is quite challenging. With respect to timing properties, at the moment we can only test the correct ordering of test packets over time. It would be interesting to test, for example, what happens when there is a long pause between test packets belonging to the same test case.

References

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

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

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

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

[FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. http://www.ietf.org/rfc/rfc1519.txt, September 1993.

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

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

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

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

[Hae97] Reto E. Haeni. Firewall penetration testing. Technical report, The George Washington University Cyberspace Policy Institute, 2033 K St, Suite 340N, Washington, DC, 20006, US, January 1997.

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

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

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

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

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

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

A Grammars

A.1 General

```
IP       = DDD'.'DDD'.'DDD'.'DDD .
D        = [digit] .
PROTO    = letter {letter | '-' | '+' | digit | '.' | '_'} | NUM .
NUM      = {digit} .
NAME     = letter {letter | digit} .
ACTION   = 'accept' | 'deny' .
PRE      = 'pre' .
POST     = 'post' .
COMMENT  = '@' TEXT '\n'
TEXT     = {letter | digit ...}
```

A.2 Formal Network Policy

```
POLICY   = {RULE | COMMENT}
RULE     = SOURCE '→' DEST : ACTION KEYWORDS
SOURCE   = NETWORK
DEST     = NETWORK
NETWORK  = NAME
KEYWORDS = ('∗' | NAME) {',' KEYWORDS}
```

A.3 Keyword Definitions

```
KEYWORD-DEFINITIONS = {DEFINITION | COMMENT}
DEFINITION          = NAME '=' PROTO {',' PROTO}
```

A.4 Network Layout

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