Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lukas Brügger

Testing Firewall Policies using HOL-TestGen

Semester Thesis
ETH Zürich
June 29, 2006

Supervisors: Achim Brucker, Diana von Bidder, Burkhart Wolff
Professor: David Basin

Abstract

To make sure that a firewall really implements a given security policy, a systematic test method is needed. Such a test method can check if a configured firewall implementation conforms to a given security policy. To construct these tests in an effective way, a test set should be constructed as automatic as possible.

In this thesis, we developed a realistic model of both a stateless and a stateful firewall in HOL-TestGen (based on Isabelle/HOL), to construct test data for some typical policies.

Zusammenfassung

Um sicherzustellen, dass eine Firewall eine gegebene Sicherheitsrichtlinie korrekt implementiert, ist eine systematische Testmethode notwendig. Eine solche Testmethode kontrolliert, ob eine konfigurierte Firewallimplementierung einer Sicherheitspolicy entspricht. Um diese Tests effizient durchführen zu können, sollten die Testfälle möglichst automatisch erzeugt werden.

In dieser Arbeit haben wir ein realistisches Modell sowohl einer zustandslosen als auch einer zustandsbehafteten Firewall in HOL-TestGen (basierend auf Isabelle/HOL) entwickelt, um Testdaten für einige typische Sicherheitsrichtlinien zu erstellen.

Contents

1. Introduction

Today, unrestricted access to the Internet is a security risk. Firewalls are a widely used tool for controlling the access to and from computers in a (sub)network. Firewalls filter undesired TCP/IP packets in the data-flow going to and from a network. But which traffic is undesired? This varies from application to application and is usually described in a (network) security policy. This policy should be implemented by a firewall, which means that it has to be configured in a way such that it implements this policy correctly. This is normally done manually and hence highly error-prone. Furthermore, a firewall is a complex piece of software that needs testing. To ensure that a concrete firewall really implements a given security policy, a systematic test method is desirable. To construct systematic test cases in an effective and adequate way, a test set should be constructed semi-automatically.

In this thesis, the interactive test environment HOL-TestGen [2, 3, 4], based on the interactive theorem proving environment Isabelle/HOL [7], is used to achieve this goal.

1.1. Higher-Order Logic

Higher-order logic (HOL) is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, since e.g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification. HOL has been applied in various areas ranging from pure mathematics over cryptographic protocols to industrial hardware verification.

1.2. Isabelle/HOL

Isabelle is a generic theorem prover where new object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic, Zermelo-Fränkel set theory and HOL, which was chosen as framework for HOL-TestGen.

Isabelle/HOL is usually coined as proof assistant but can also be used as symbolic computation environment. Implementations on Isabelle/HOL can reuse existing powerful deduction mechanisms such as higher-order resolution and rewriting, and the overall environment provides a large collection of components ranging from documentation generators and code-generators to (generic) decision procedures for datatypes and Presburger

Arithmetic. Moreover, Isabelle/HOL already provides many useful theories like sets, lists, maps and orderings.

1.3. HOL-TestGen

HOL-TestGen is built on top of Isabelle/HOL and allows for interactive development of test cases, refinement to concrete test data, and generation of test scripts that can be used for test execution and test result verification. Its design rationale is different from most other symbolic testing tools which are designed to be fully automatic. In HOL-TestGen, the development of tests is an interactive activity, where the form of test specifications, the abstraction levels used in a test, the solution of generated logical constraints and the parameters of the test data selection must be experimented with and adopted up to the point where the generated tests are sufficiently good with respect to an underlying test adequacy criteria. As a particular feature, the automated deduction-based process can log the underlying test hypotheses made during the test; provided that the test hypotheses are valid for the program and provided the program passes the test successfully, the program is guaranteed to be correct with respect to the test specification.

1.4. Goals

In this paper we create a model of a firewall and its security policies. Our goal is that this model is realistic while omitting all the unnecessary details, meaning that only those things which matter for our purpose should be modelled.

Furthermore it should be quite easy for an administrator to specify a given security policy without having to be an expert in Isabelle.

Finally, the model should work well with HOL-TestGen such that it creates sensible test data.

1.5. Outline

The paper is structured as follows. In chapter 2 we present the necessary preliminaries of networks, firewalls and firewall testing. In chapter 3 we model the stateless firewall, in chapter 4 the stateful one. Some test data generation examples are provided in chapter 5. In chapter 6 we draw the conclusions and propose future work.

2. About Firewall Testing

2.1. A Short Introduction to TCP/IP

Data is sent over the Internet in units called *packets* using the *Internet Protocol Suite*. This suite can be seen as a set of layers, with each of the layers solving a set of problems concerning the transmission of data and providing a well-defined service to the upper layer protocols [13]. Starting from the bottom, these layers are:

1. Link layer: used to pass packets from one host to the other.

2. Network layer: getting packets from source to destination.

3. Transport layer: connect applications together and provide reliability.

4. Application layer: communication from one application to the other.

Application layer protocols include HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), and VoIP (Voice over IP). The most important protocol on the transport layer is IPv4 (Internet Protocol version 4), on the network layer it is TCP (Transmission Control Protocol). Each layer adds a protocol *header* at the beginning of a packet. For a firewall, the headers of layer 2 (TCP) and 3 (IPv4) are important. These headers are shown in Figure 2.1. Most of the header fields are of no interest for our purposes, as they are not used by a firewall as input for its decision about allowing or denying a packet. What we need is only the *addresses* and the *port numbers*.

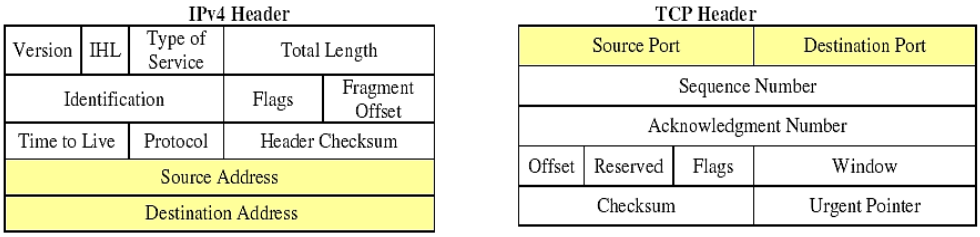

Figure 2.1.: The headers for IPv4 and TCP

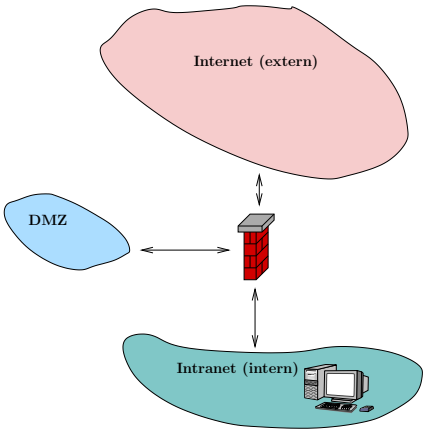

Figure 2.2.: A typical firewall scenario

2.2. Firewalls

Generally speaking, *firewalls* are devices or systems that control the flow of network traffic between subnetworks.

A typical scenario as it could be found in a small company is the one of Figure 2.2. There are three subnetworks consisting of:

1. the "Internet" (outside)
2. the "Intranet" (inside)
3. the "Demilitarized Zone" (DMZ) (for servers that speak to both sides)

There are several types of firewall platforms currently available. They mainly differ in which TCP/IP layers they take into account and whether they employ some kind of state. In this thesis we are interested in the type of firewall usually called *packet filters*, which are "routing devices that include access control functionality for system addresses and communication sessions"[12]. A packet filter receives a packet and makes a decision about allowing or denying this packet. The decision is based on a ruleset called a *policy*. The information that the firewall uses to perform the analysis is usually the source address of the packet, the destination address, the type of traffic, and some characteristics of the TCP session, such as port numbers.

An informal policy defines which application layer protocols are allowed from and to which subnetworks. This policy can be displayed in a table as the one in Figure 2.3. This informal description has to be translated into a more formal one, as it is displayed in Figure 2.4. This table has to be read as follows: when the firewall receives a packet, it looks at the topmost line and checks if the packet that it received corresponds to the entry. If it does, it applies the corresponding action, i.e., it either accepts the packet or denies it. If the packet does not match, it takes the next line and so on. The last line in this table says that all packets for which no rule applies should be denied.

$\longrightarrow$	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all but smtp
DMZ	$\emptyset$	-	smtp
Internet	$\emptyset$	http,smtp	-

Figure 2.3.: An Informal Policy

A firewall which works this way is usually called a *stateless packet filter* as it does not employ any kind of state. Another category of firewall systems are the stateful inspection firewalls which incorporate added awareness of the transport layer, as they track the state of the TCP/IP connection. We do not differentiate between these two kinds as our high-level approach doesn't need to take the state of the connection into account.

However, we provide a model for a third variant of firewalls, which are stateful on the application layer. This can be used for modelling protocols like FTP, VoIP, or even filtering proxies.

2.3. Firewall Testing

A firewall is a very security sensitive system. In case of an error, an attacker might enter the internal network. Thorough testing of the firewall is therefore needed. In principle, there are two fundamental things to distinguish while testing for policy compliance: testing the implementation of the firewall itself and testing of the correct configuration of the policy. The first kind is usually done using penetration testing, i.e. performing known attacks on a firewall. This is normally done directly by the vendor of the firewall and does not take any specific policy into account.

The second category is testing if the firewall really employs a given security policy. Even the most cautious policy is of no use if there's one little mistake in the concrete configuration of the firewall. For these reasons, it is recommended to audit and verify the configuration at least quarterly [12].

Source Address	**Destination Address**	**Protocol**	**Action**
Intranet	DMZ	smtp, imap	*accept*
Intranet	Internet	not smtp	*accept*
Intranet	Internet	smtp	*deny*
DMZ	Intranet	any	*deny*
DMZ	Internet	smtp	*allow*
Internet	Intranet	any	*deny*
Internet	DMZ	http, smtp	*accept*
any	any	any	*deny*

Figure 2.4.: A formal Policy

There are two main methodologies for comparing policies and firewall configurations. The first one is to obtain hardcopies of the configuration and compare these against the expected configuration based upon the defined policy. This works similar to code inspection in software engineering, but is very error prone, especially when the policy grows too large.

The second methodology is the one for which we want our model to be used for. It is actual in-place configuration testing. Packets are sent to the firewall and its reaction to it is compared to the expected reaction. With this approach, the correct implementation of the configured firewall is tested. However, as it is infeasible to test this exhaustively, the choice of test packets is crucial. Pure random testing doesn't seem to produce good results [5]. We therefore believe that the creation of test cases should be performed in a more systematic way.

2.4. Related Work

Most firewall testing techniques today are restricted to pure implementation testing independent of the actual policy and to vulnerability testing.

Vigna [11] proposes a methodology based on a formal model of networks, that allows the test engineer to model the network environment of the firewall system, to prove formally that the topology of the network provides protection against attacks, and to build test cases to verify the correct configuration. The main similarity to our approach is that Vigna also believes that the test cases should be constructed systematically, based upon the actual policy.

In [5] the need for systematically producing the test data is further justified. The authors claim that exhaustive testing would take years even when reducing the address space and that random testing would let pass too many faulty firewalls. They propose a segmentation of the policy in order to get reasonable test cases.

In [10], it is shown how a network security policy can be formally specified and how this specification can be used to automatically generate test cases. This approach goes into the same direction as ours, but employs a more low-level view of a network, i.e. it also models the state of the TCP connection.

HOL-TestGen is originally geared towards unit-tests of software [2]. Here we want to show that it can also be used for sequence testing thanks to the rich underlying data-structures of HOL which allow for a temporal description of a security policy as a set of admissible communication traces that a firewall is accepting.

3. The Stateless Model

We now present the model of a stateless and a stateful firewall. Subsequently we show how to produce concrete test data within this framework. These chapters were produced directly with the document generation tool of Isabelle which means that everything we present here is formally checked.

Isabelle groups its definitions and proofs in modules which are called *theories*. As we want our framework to be easy extendible, we make a lot of use of this modularisation. The dependency graph of all the theories is shown in the appendix in Figure B.1.

3.1. Firewall Basics

theory *FWBasics*
imports *Main*
begin

A firewall operates on a network and observes the packets sent over it. We hence start with modelling these two things in this theory, trying to stay as generic as possible to make the model easy extendible.

3.1.1. Packets

Of all the fields of a real TCP/IP packet, we only want to model those which are necessary for the firewall to take its decision. Recalling the ruleset of Figure 2.4, we see that we need the source and destination address and the application layer protocol. Ports, if needed, are modelled as part of the address. Additionally we include an ID and a content for later extensions.

The ID is an integer.

types *id* = *int*

The protocol is a datatype. Here we included the most common ones which were used for our case studies. If necessary, more application layer protocols could easily be added here.

datatype *protocol* = *ftp*
| *http*
| *voip*
| *smtp*
| *imap*
| *unknown*

We do not want to give a specific representation of an address yet in order to make it possible to have different representations (e.g. IPv4 and IPv6, with or without ports). To achieve this, we use type variables known from functional programming languages like Haskell[1]. The source and the destination are both of the same type *'a adr*.

types *'a adr* = *'a*
'a src = *'a adr*
'a dest = *'a adr*

The content is also specified with a type variable.

types *'b content* = *'b*

A packet then is a five-tuple consisting of the above declared types.

types (*'a*,*'b*) *packet* = *id* × *protocol* × *'a src* × *'a dest* × *'b content*

A packet therefore has two parameters, the first being the address, the second the content. These must of course be specified before we can generate concrete test data later.

Alternatively, a packet could have been modelled as a record type. But as HOL-TestGen has yet some problems handling them, we used the simpler tuple definition.
In order to access the different parts of a packet directly, we define a couple of projectors.

consts
id :: (*'a*,*'b*) *packet* ⇒ *id*
protocol :: (*'a*,*'b*) *packet* ⇒ *protocol*
src :: (*'a*,*'b*) *packet* ⇒ *'a src*
dest :: (*'a*,*'b*) *packet* ⇒ *'a dest*
data :: (*'a*,*'b*) *packet* ⇒ *'b content*

defs
id_def: *id* ≡ *fst*
protocol_def: *protocol* ≡ *fst o snd*
src_def: *src* ≡ *fst o snd o snd*
dest_def: *dest* ≡ *fst o snd o snd o snd*
data_def: *data* ≡ *snd o snd o snd o snd*

3.1.2. Nets

A net is modelled as an axiomatic type class, which is roughly speaking an axiomatic specification of a class of types (see [7] for further information). Using type classes gives us the possibility to enable different representations of an address.

axclass *net* < *type*

We already provide some instances to the class net: function application, pairing, and the integers.

instance *fun* :: (*net*,*net*)*net* **..**

[1] Type variables are usually denoted with greek letters, whereas in Isabelle they are written as *'a*.

instance $*$ :: (*net*,*net*)*net* **..**
instance *int* :: *net* **..**

An address is usually part of a subnetwork (e.g. the Intranet in our example). We model a subnet as a set of sets of addresses to reflect the fact that very often a subnet consists of several subnets again.

types *'a subnet* = *'a*::*net set set*

The relation in_subnet ($\sqsubset$) checks if an address is in a specific subnet. It models a kind of subset relation and is defined as an infix operator.

constdefs
in_subnet :: *'a adr* $\Rightarrow$ *'a subnet* $\Rightarrow$ *bool* (**infixl** $\sqsubset$ *100*)
in_subnet a S $\equiv \exists\ s \in S.\ (a \in s)$

end

3.2. Policy

theory *Policy*
imports *FWBasics*
begin

3.2.1. Definitions

Having defined what a packet and a net is, the concept of a policy needs to be modelled. A policy is seen as a ruleset as in Table 2.4, which specifies for each packet an action, namely if it should get accepted or denied by the firewall. We model these two possibilities by a datatype *out*. Alternatively we could make a further distinction between a drop and a reject if desired.

datatype *'a out* = *accept 'a*
| *deny*

A policy is seen as a partial mapping from packet to packet out. Partial mappings are modelled with the datatype *option*. This datatype consists of two elements, being either *None* or *Some 'a*. The packets p that a firewall should accept, map to *Some accept p*, those that it should reject to *Some deny*[2], and those packets for which no rule applies map to *None*. This definition enables us to detect packets for which no action was specified in the policy. For partial mappings, Isabelle provides the operator "$\rightharpoonup$" which is shorthand for a function from *'a* to *'b option*.

types ('*a*, '*b*) *Policy* = ('*a*, '*b*) *packet* $\rightharpoonup$ (('*a*, '*b*) *packet*) *out*

A rule has the same type as a policy and usually a policy consists of several rules.

types ('*a*, '*b*) *Rule* = ('*a*, '*b*) *Policy*

[2] As the packet is dropped here, we don't need it anymore.

When combining several rules, the firewall is supposed to apply the first matching one. In our setting this means the first rule which maps the packet in question to *Some packet out*. This is exactly what happens when using the map-add operator (*rule1 ++ rule2*). The only difference is that the rules must be given in reverse order.

The constant *p_accept* is *True* if the policy accepts the packet and *False* otherwise.

constdefs
p_accept :: ('*a*, '*b*) *packet* ⇒ ('*a*, '*b*) *Policy* ⇒ *bool*
p_accept p policy ≡ *policy p* = *Some* (*accept p*)

end

3.2.2. Policy Combinators

In order to ease the specification of a concrete policy, we define some combinators. Their definition is pretty straighforward, which is why we only present their names and arguments here. The definitions can be found in the Appendix A. With these combinators, the specification of a policy gets very easy and similar to the configuration of popular firewalls.
The names of the combinators should be self-explanatory.

- Combinators which allow or deny everything:
  - allow_all
  - deny_all
- Combinators which allow or deny packets based on their source and/or destination address:
  - allow_all_from *src_net*
  - deny_all_from *src_net*
  - allow_all_to *dest_net*
  - deny_all_to *dest_net*
  - allow_all_from_to *src_net dest_net*
  - deny_all_from_to *src_net dest_net*
- Combinators which allow or deny packets based additionally on their protocol:
  - allow_protocol *prot*
  - deny_protocol *prot*
  - allow_prot_from *prot src_net*
  - deny_prot_from *prot src_net*
  - allow_prot_to *prot dest_net*

  - deny_prot_to *prot dest_net*
  - allow_prot_from_to *prot src_net dest_net*
  - deny_prot_from_to *prot src_net dest_net*

- Combinators which allow or deny everything except a protocol:
  - allow_all_but_prot *prot*
  - deny_all_but_prot *prot*
  - allow_all_but_prot_from *prot src_net*
  - deny_all_but_prot_from *prot src_net*
  - allow_all_but_prot_to *prot dest_net*
  - deny_all_but_prot_to *prot dest_net*
  - allow_all_but_prot_from_to *prot src_net dest_net*
  - deny_all_but_prot_from_to *prot src_net dest_net*
- Combinators which allow or deny everything of a set of protocols:
  - allow_prots_from_to *(prot set) src_net dest_net*
  - deny_prots_from_to *(prot set) src_net dest_net*
- Combinators which allow or deny everything except of a set of protocols:
  - allow_all_but_prots_from_to *(prot set) src_net dest_net*
  - deny_all_but_prots_from_to *(prot set) src_net dest_net*

All these combinators and the default rules are put into one single lemma called *stateless_rules* to make life easier when we need to unfold or simplify a policy consisting of several rules.

3.3. IPv4

theory *IPv4*
imports *FWBasics*
begin

3.3.1. IPv4-Addresses

We defined an address as an abstract type to hold the whole model as generic as possible. But of course, when we want to have concrete test data, an address has to be concretized. In this theory we define IPv4 as one particular example of a representation of an address.

ipv4_ip is a common IP address. It is defined as a four-tupel of integers, to make them look almost like the common writing of these addresses (`217.82.1.1` vs. `(217,82,1,1)`). Technically it would be easier to define an address as only one integer, but for the sake

of readability we chose this representation. For the same reason we didn't took integers instead of natural numbers, even though the numbers should of course never be negative. It should be modelled somehow that these numbers have to be smaller than 256. This has not been done yet but actually this doesn't make any problems so far as HOL-TestGen always picks low numbers in the random generation phase.

A port is in our view also a part of an address. This gives us the possibility to use the model for a scenario where we either do not have or do not care about the port numbers. They are also defined as integers. The full IPv4-address is then a tupel of the *ipv4_ip* and the *port*.

types
ipv4_ip = (*int* × *int* × *int* × *int*)
port = *int*
ipv4 = (*ipv4_ip* × *port*)

Analogous to the packet definition, we define projectors to access the ports of a packet directly.

constdefs
src_port :: (*ipv4*, 'b) *packet* ⇒ *port*
src_port ≡ *snd o fst o snd o snd*

dest_port :: (*ipv4*, 'b) *packet* ⇒ *port*
dest_port ≡ *snd o fst o snd o snd o snd*

As a help for later we also define constructors which take an address or an IP and return a subnet with all addresses having the same ipv4_ip.

constdefs
subnet_of :: *ipv4* ⇒ *ipv4 subnet*
subnet_of adr ≡ {{(*a*,*b*). (*a* = *fst adr*)}}

subnet_of_ip :: *ipv4_ip* ⇒ *ipv4 subnet*
subnet_of_ip ip ≡ {{(*a*,*b*). (*a* = *ip*)}}

This is everything we need to model an address. Other address representations could as easily be added. For example we could write a similar theory to model IPv6 addresses. The place for the respective theory is depicted in the session graph in the appendix on page 51.

end

3.3.2. Policy Combinators for IPv4 Networks

We also define some combinators which can be used to define policies for firewalls based on IPv4 networks. The rules resemble the former ones, but also take port numbers into account. If we don't want to use the port numbers in our specification, we can continue to use the original rules.

- Combinators which allow or deny packets based on their source and/or destination address:
  - allow_all_from_port *src_net src_port*
  - deny_all_from_port *src_net src_port*
  - allow_all_to_port *dest_net dest_port*
  - deny_all_to_port *dest_net dest_port*
  - allow_all_from_to_port *src_net dest_net dest_port*
  - deny_all_from_to_port *src_net dest_net dest_port*
  - allow_all_from_port_to *src_net src_port dest_net*
  - deny_all_from_port_to *src_net src_port dest_net*
  - allow_all_from_port_to_port *src_net src_port dest_net dest_port*
  - deny_all_from_port_to_port *src_net src_port dest_net dest_port*
- Combinators which allow or deny packets based additionally on their protocol
  - allow_prot_from_port *prot src_net src_port*
  - deny_prot_from_port *prot src_net src_port*
  - allow_prot_to_port *prot dest_net dest_port*
  - deny_prot_to_port *prot dest_net dest_port*
  - allow_prot_from_port_to *prot src_net src_port dest_net*
  - deny_prot_from_port_to *prot src_net src_port dest_net*
  - allow_prot_from_to_port *prot src_net dest_net dest_port*
  - deny_prot_from_to_port *prot src_net dest_net dest_port*
  - allow_prot_from_port_to_port *port src_net src_port dest_net dest_port*
  - deny_prot_from_port_to_port *port src_net src_port dest_net dest_port*

As before, we put all the rules into one lemma called *ipv4_rules* to ease writing later.

This is already everything we need for modelling a stateless firewall and we can go on to model the next kind of firewalls.

4. The Stateful Model

4.1. Stateful Firewalls

theory *Stateful*
imports *Policy*
begin

Unfortunately the simple system of a stateless packet filter is not enough to model all common real-world scenarios. Some protocols need further actions in order to be secured. A prominent example is the File Transfer Protocol (FTP) which is a popular means to move files across the Internet. It behaves quite differently from most other application layer protocols as it uses a two-way connection establishment which opens a dynamic port. A stateless packet filter would only have the possibility to either always open all the possible dynamic ports or not to allow that protocol at all. Neither of these options is satisfactory. In the first case, all ports above 1024 would have to be opened which introduces a big security hole in the system, in the second case users wouldn't be very happy. A firewall which tracks the state of the TCP connections on a system doesn't help here either, as the opening and closing of the ports takes place on the application layer. Therefore a firewall needs to have some knowledge of the application protocols being run and track the states of these protocols (see also [8]). We now go on to model this behaviour.

The key point of our model is the idea that a policy remains the same as before, meaning a mapping from packet to packet out. We still specify for every packet, based on its protocol, source, and destination, the expected action. The only thing that changes now is that this mapping is allowed to change over time. This indicates that our test data will not consist of single packets but rather of sequences thereof.

At first we hence need a state. It is a tupel from some memory to be refined later and the current policy.

types $('a,'b,'c)\ FWState = 'a \times ('b,'c)\ Policy$

Having a state, we need of course some state transitions. Such a transition can happen every time a new packet arrives. It is a mapping from this incoming packet and a state to a new state, again giving us the possibility to model undefined transitions.

types $('a,'b,'c)\ FWStateTransition =$
$((('b,'c)\ packet \times ('a,'b,'c\ )\ FWState) \rightharpoonup (('a,'b,'c)\ FWState)$

The memory could be modelled as a list of accepted packets. Other possibilities would be possible, however in this thesis only this one is used.

types $('b,'c)$ *history* $= ('b,'c)$ *packet list*

It is necessary to model the dynamic behaviour of our specification. This can be done using a fold operator which takes a history, a state, and a state-transition and returns a new state. With this operator we get the resulting state after applying a sequence of state transitions induced by a list of incoming packets to the initial state.

```
consts Mfold :: ('b,'c) history ⇒
                ('a,'b,'c) FWState ⇒
                ('a,'b,'c) FWStateTransition ⇒
                ('a,'b,'c) FWState
primrec
  Mfold [] S ST = S
  Mfold (a#H) S ST = (case ST (a,S) of None  ⇒  Mfold H S ST
                                     | Some Sn ⇒  Mfold H Sn ST)
```

As we don't want the simplifier to Mfold everything, we remove the corresponding lemmas from the simplifier set.

```
declare Mfold.simps [simp del]
```

The next two constants will help us later in defining the state transitions. The constant *before* is *True* if for all elements which appear before the first element for which q holds, p must hold.

```
consts before :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ bool
primrec
  before p q [] = False
  before p q (a # S) = (q a ∨ (p a ∧ (before p q S)))
```

Analogously there is an operator *not_before* which returns *True* if for all elements which appear before the first element for which q holds, p must not hold.

```
consts not_before :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ bool
primrec
  not_before p q [] = False
  not_before p q (a # S) = (q a ∨ (¬ (p a) ∧ (not_before p q S)))
```

The next two operators can be used to combine state transitions. The *orelse* takes the first transition which maps to *Some* $'a$, the *repeat* allows an infinite repetition.

```
constdefs
  orelse :: ('a ⇀ 'b) ⇒ ('a ⇀ 'b) ⇒ ('a ⇀ 'b)  (infixl orelse 100)
  (f orelse g) x ≡ case f x of None ⇒ g x
                             | Some y ⇒ Some y

consts repeat :: ('a ⇀ 'b) => ('a ⇀ 'b)
defs repeat_def: repeat f x ≡ case f x of None   ⇒ None
                                        | Some y ⇒ (case f y of None   ⇒ Some y
                                                             | Some z ⇒ Some z)

end
```

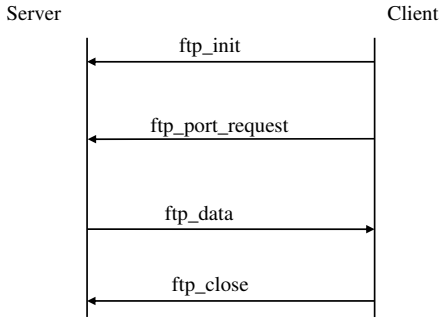

Figure 4.1.: The modelled FTP-Protocol

4.2. FTP

theory *FTP*
imports *Stateful IPv4_Combinators*
begin

The File Transfer Protocol FTP [9] is a well known example of a protocol which uses dynamic ports and is therefore a natural choice to use as an example for our model.

We model only a simplified version of the FTP protocol over IPv4 networks, still containing all messages that matter for our purposes. It consists of the following four messages (see Figure 4.1):

1. *init*: The client contacts the server indicating his wish to get some data.
2. *port_request p*: The client, usually after having received an acknowledgement of the server, indicates a port number on which he wants to receive the data.
3. *ftp_data*: The server sends the requested data over the new channel. There might be an arbitrary number of such messages, including zero.
4. *close*: The client closes the connection. The dynamic port gets closed again.

The content field of a packet therefore now consists of these four messages plus a default one.

datatype *ftp_msg* = *init*
| *port_request port*
| *ftp_data*
| *close*
| *other*

We now also make use of the ID field of a packet. It is used as session ID and we make the assumption that they are all unique.

At first, we need some predicates which check if a packet is a specific FTP message and has the correct session ID.

constdefs
is_init :: *id* ⇒ (′*a*, *ftp_msg*) *packet* ⇒ *bool*
is_init i p ≡ *id p* = *i* ∧ *data p* = *init*

constdefs
is_port_request :: *id* ⇒ *port* ⇒(′*a*, *ftp_msg*) *packet* ⇒ *bool*
is_port_request i port p ≡ *id p* = *i* ∧ *data p* = *port_request port*

constdefs
is_data :: *id* ⇒ (′*a*, *ftp_msg*) *packet* ⇒ *bool*
is_data i p ≡ *id p* = *i* ∧ *data p* = *ftp_data*

constdefs
is_close :: *id* ⇒ (′*a*, *ftp_msg*) *packet* ⇒ *bool*
is_close i p ≡ *id p* = *i* ∧ *data p* = *close*

We now have to model the respective state transitions. It is important to note that state transitions themselves allow all packets which are allowed by the policy, not only those which are allowed by the protocol. Their only task is to change the policy. As an alternative, we could have decided that they only allow packets which follow the protocol (e.g. come on the correct ports), but this should in our view rather be reflected in the policy itself.

Of course, not every message changes the policy. In such cases, we do not have to model different state transitions, one is enough. In this example, only messages 2 and 4 need special transitions. The default says that if the policy accepts the packet, it is added to the history, otherwise it is simply dropped. The policy remains the same in both cases.

consts *FTP_ST_F* :: ((*ipv4*,*ftp_msg*) *history* ,*ipv4*, *ftp_msg*) *FWStateTransition*
recdef *FTP_ST_F* {}
FTP_ST_F_1: *FTP_ST_F* ((*a*,*ftp*,*c*,*d*,*e*), (*InL*, *policy*)) =
(*if p_accept* (*a*,*ftp*,*c*,*d*,*e*) *policy then*
Some ((*a*,*ftp*,*c*,*d*,*e*)#*InL*,*policy*)
else
Some (*InL*,*policy*))

FTP_ST_F_2: *FTP_ST_F* (*x*, (*InL*,*policy*)) = *None*

The second message is the port request. If the packet is allowed by the policy, and if and only if there is an opened but not yet closed FTP-Session with the same session ID, we change our policy such that the requested port is opened. If our policy allows the packet but there is no open protocol run, we do allow the packet but do not open the requested port.

```
consts FTP_ST_2 :: ((ipv4,ftp_msg) history, ipv4, ftp_msg) FWStateTransition
recdef FTP_ST_2 {}
  FTP_ST_2_1: FTP_ST_2 ((a,ftp,c,d,port_request port_r), (InL,policy )) =
                (if p_accept (a,ftp,c,d,port_request port_r) policy then
                   (if not_before  (is_close a) (is_init a) InL then
                        Some ((a,ftp,c,d,port_request port_r)#InL, policy ++
                             (allow_prot_from_to_port ftp (subnet_of d) (subnet_of c) port_r))
                    else Some ((a,ftp,c,d,port_request port_r)#InL,policy))
                 else Some (InL,policy))

FTP_ST_2_2: FTP_ST_2 (x, (InL,policy)) = None
```

In the last message, we need to close a port which we do not know directly. It has only been specified in a preceding port_request message. Therefore a predicate is needed which checks if there is an open protocol run with an opened port

```
constdefs
  port_open :: (ipv4, ftp_msg) history ⇒ id ⇒ port ⇒ bool
  port_open L a p ≡ not_before (is_close a) (is_port_request a p) L
```

This transition is the trickiest one. We need to close the port wich has been opened but not yet closed by a packet with the same session ID. Here we use the assumption that they are supposed to be unique.

```
consts FTP_ST_4 :: ((ipv4, ftp_msg) history, ipv4, ftp_msg) FWStateTransition
recdef FTP_ST_4 {}
  FTP_ST_4_1: FTP_ST_4 ((a,ftp,c,d,close), (InL,policy)) =
                (if (p_accept (a,ftp,c,d,close) policy) then
                     (if (∃ p. port_open InL a p) then
                          Some((a,ftp,c,d,close)#InL, policy ++
deny_prot_from_to_port ftp (subnet_of d) (subnet_of c) (Eps (λ p. port_open InL a p)))
                      else Some ((a,ftp,c,d,close)#InL, policy))
                 else Some (InL,policy))

FTP_ST_4_2: FTP_ST_4 (x, (InL,policy)) = None
```

This transition introduces some kind of inconsistency. If the port that was requested was already open to start with, it gets closed here. The tester should be aware of this fact.

This transition has also some other consequences. The Hilbert epsilon operator *Eps*, also written as *SOME*, returns an arbitrary object for which the following predicate is *True* and is undefined otherwise. We use it to get the number of the port which we want to close. With the if-condition it is assured that such a port exists, but we might have problems if there are several of them. However, due to our assumption that the session IDs are unique, there won't be a problem as long as we do not open several ports in one single protocol run. This should not occur by the definition of the protocol, but if it does, which might happen if we want to test illegal protocol runs, some proof work might be needed.

The full state machine of an FTP-protocol is now the *orelse* of the single transitions.

constdefs
FTP_ST :: ((*ipv4*, *ftp_msg*) *history*, *ipv4*, *ftp_msg*) *FWStateTransition*
FTP_ST ≡ *FTP_ST_2 orelse*
FTP_ST_4 orelse
FTP_ST_F

An arbitrary repetition of protocol runs can be modelled with the operator *repeat*.

constdefs
ALL_ST :: ((*ipv4*,*ftp_msg*) *history*, *ipv4*, *ftp_msg*) *FWStateTransition*
ALL_ST ≡ *repeat FTP_ST*

Now we specify our test scenario. We could test

- one correct FTP-Protocol run,
- several runs after another,
- several runs interleaved,
- an illegal protocol run, or
- several illegal protocol runs.

We only do the the simplest case: one correct protocol run. As soon as we are modelling the more complicated ones, the state space grows rapidly and there will be problems with creating test cases within reasonable time.

There are four different states which are modelled as a datatype.

datatype *ftp_states* = *S0* | *S1* | *S2* | *S3*

The following constant is *True* for all sets which are correct FTP runs for a given source and destination address, ID, and data-port number. The empty trace is not included as it wouldn't make any sense to test this one.

consts
is_ftp :: *ftp_states* ⇒ *'a adr* ⇒ *'a adr* ⇒ *id* ⇒ *port* ⇒ (*'a*,*ftp_msg*) *history* ⇒ *bool*
primrec
is_ftp H c s i p [] = (*H*=*S3*)
is_ftp H c s i p (*x*#*InL*) = (λ (*id*,*pr*,*sr*,*de*,*co*). (((*pr* = *ftp* ∧ *id* = *i* ∧ (
(*H*=*S2* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *init* ∧ *is_ftp S3 c s i p InL*) ∨
(*H*=*S1* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *port_request p* ∧ *is_ftp S2 c s i p InL*) ∨
(*H*=*S1* ∧ *sr* = *s* ∧ *de* = (*fst c*,*p*) ∧ *co*= *ftp_data* ∧ *is_ftp S1 c s i p InL*) ∨
(*H*=*S0* ∧ *sr* = *c* ∧ *de* = *s* ∧ *co* = *close* ∧ *is_ftp S1 c s i p InL*)))))) *x*

This definition is crucial for specifying what we actually want to test. Extending it produces more test cases but increases the time necessary to create them and vice-versa. It is a good idea for the tester to experiment a little bit around here.

The following constant then returns a set of all the historys which denote such a normal behaviour FTP run, again for a given source and destination address, ID, and data-port.

constdefs
NB_ftp :: *'a src ⇒ 'a dest ⇒ id ⇒ port ⇒ ('a,ftp_msg) history set*
NB_ftp s d i p ≡ {*x*. (*is_ftp S0 s d i p x*)}
end

4.3. VoIP

theory *VOIP*
imports *Stateful IPv4_Combinators*
begin

After the FTP-Protocol which was rather simple we show the strength of the model with a more current and especially much more complicated example, namely Voice over IP (VoIP). VoIP is standardized by the ITU-T under the name H.323, which can be seen as an "umbrella standard" which aggregates standards for multimedia conferencing over packet-based networks (for a good overview of the protocol suite, see [6]). H.323 poses many problems to firewalls. These problems include (taken from [1]):

- An H.323 call is made up of many different simultaneous connections.
- Most connections are made to dynamic ports.
- The addresses and port numbers are exchanged within the data stream of the next higher connection.
- Calls can be initiated from outside the firewall.

Again we only consider a simplified VoIP scenario with the following seven messages which are grouped into four subprotocols (see Figure 4.2):

- Registration and Admission (H.225, port 1719): The caller contacts its gatekeeper with a call request. The gatekeeper either rejects or confirms the request, returning the address of the callee in the latter case.
  - Admission Request (ARQ)
  - Admission Reject (ARJ)
  - Admission Confirm (ACF) *'a*
- Call Signaling (Q.931, port 1720) The caller and the callee agree on the dynamic ports over which the call will take place.
  - Setup *port*
  - Connect *port*
- Stream (dynamic ports). The call itself. In reality, several connections are used here.

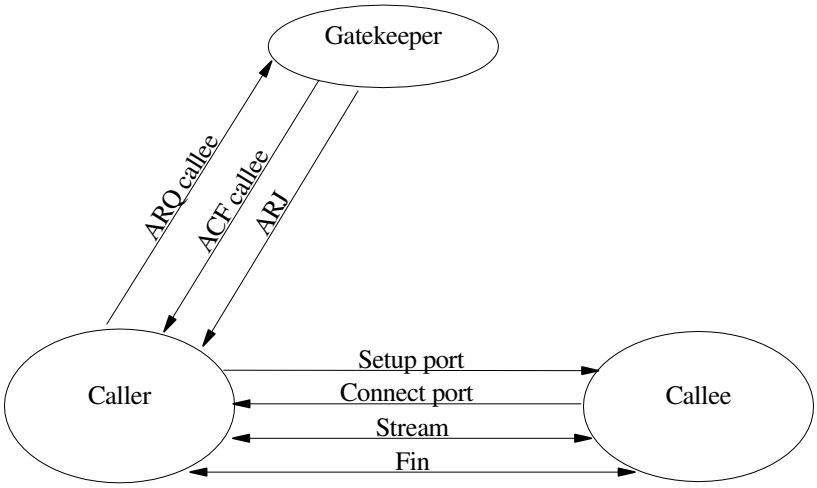

Figure 4.2.: The modelled VoIP-Protocol

- Fin (port 1720).

The two main differences to FTP are:

- In VoIP, we deal with three different entities: the caller, the callee, and the gatekeeper.
- We do not know in advance which entity will close the connection.

We model the protocol as seen from a firewall at the caller, namely we are not interested in the messages from the callee to its gatekeeper. Incoming calls are not modelled either, they would require a different set of state transitions.

The content of a packet now consists of one of the seven messages or a default one. It is parameterized with the type of the address that the gatekeeper returns.

datatype *'a voip_msg* = *ARQ*
| *ACF 'a*
| *ARJ*
| *Setup port*
| *Connect port*
| *Stream*
| *Fin*
| *other*

As before, we need operators which check if a packet contains a specific content and ID, respectively if such a packet has appeared in the trace.

constdefs
is_arq :: *id* ⇒ ('*a*, '*b voip_msg*) *packet* ⇒ *bool*
is_arq i p ≡ *id p* = *i* ∧ *data p* = *ARQ*

was_arq :: *id* ⇒ ('*a*, '*b voip_msg*) *history* ⇒ *bool*
was_arq i L ≡ ∃ *p*. (*p mem L* ∧ *is_arq i p*)

is_fin :: *id* ⇒ ('*a*, '*b voip_msg*) *packet* ⇒ *bool*
is_fin i p ≡ *id p* = *i* ∧ *data p* = *Fin*

is_connect :: *id* ⇒ *port* ⇒ ('*a*, '*b voip_msg*) *packet* ⇒ *bool*
is_connect i port p ≡ *id p* = *i* ∧ *data p* = *Connect port*

is_setup :: *id* ⇒ *port* ⇒ ('*a*, '*b voip_msg*) *packet* ⇒ *bool*
is_setup i port p ≡ *id p* = *i* ∧ *data p* = *Setup port*

We need also an operator *ports_open* to get access to the two dynamic ports.

constdefs
ports_open :: *id* ⇒ *port* × *port* ⇒ (*ipv4*, '*a voip_msg*) *history* ⇒ *bool*
ports_open i p L ≡ (*not_before* (*is_fin i*) (*is_setup i* (*fst p*)) *L*) ∧
not_before (*is_fin i*) (*is_connect i* (*snd p*)) *L*

The first state transition is for those messages which do not change the policy. In this scenario, this only happens for the Stream messages.

consts *VOIP_ST_F* :: ((*ipv4*, '*b voip_msg*) *history*, *ipv4*, '*b voip_msg*) *FWStateTransition*
recdef *VOIP_ST_F* {}
VOIP_ST_F_1: *VOIP_ST_F* ((*a*,*voip*,*c*,*d*,*e*), (*InL*, *policy*)) =
(*if p_accept* (*a*,*voip*,*c*,*d*,*e*) *policy then*
Some ((*a*,*voip*,*c*,*d*,*e*)#*InL*,*policy*)
else
Some (*InL*,*policy*))

VOIP_ST_F_2: *VOIP_ST_F* (*x*, (*InL*,*policy*)) = *None*

If our policy accepts the ARQ packet, we have to assure that we will accept the returning packet of the gatekeeper.

consts *VOIP_ST_1* :: ((*ipv4*, '*b voip_msg*) *history*, *ipv4*, '*b voip_msg*) *FWStateTransition*
recdef *VOIP_ST_1* {}
VOIP_ST_1_1: *VOIP_ST_1* ((*a*,*voip*,*c*,*d*,*ARQ*), (*InL*, *policy*)) =
(*if* (*p_accept* (*a*,*voip*,*c*,*d*,*ARQ*) *policy*)
then Some ((*a*,*voip*,*c*,*d*, *ARQ*)#*InL*, *policy* ++
allow_prot_from_to voip (*subnet_of d*) (*subnet_of c*))
else Some (*InL*,*policy*))

VOIP_ST_1_2: *VOIP_ST_1* (*x*, (*InL*,*policy*)) = *None*

And if the gatekeeper answers, no matter if it's a good or bad answer, we can close the channel again. If the answer was positive (ACF), we allow the caller to contact the callee and get contacted by him over port 1720.

```
consts VOIP_ST_2 :: ((ipv4, 'b voip_msg) history, ipv4, 'b voip_msg) FWStateTransition
recdef VOIP_ST_2 {}
  VOIP_ST_2_1: VOIP_ST_2 ((a,voip,c,d,ARJ),(InL, policy)) =
                    (if (p_accept (a,voip,c,d,ARJ) policy)
                     then (if (not_before (is_fin a) (is_arq a) InL)
                           then Some ((a,voip,c,d,ARJ)#InL,
                                policy ++ deny_prot_from_to voip (subnet_of c) (subnet_of d))
                           else Some ((a,voip,c,d,ARJ)#InL,policy))
                     else Some (InL,policy))

  VOIP_ST_2_2: VOIP_ST_2 (x, (InL,policy)) = None


consts VOIP_ST_3 :: ((ipv4,ipv4_ip voip_msg) history, ipv4, ipv4_ip voip_msg)
                      FWStateTransition
recdef VOIP_ST_3 {}
  VOIP_ST_3_1: VOIP_ST_3 ((a,voip,c,d,ACF callee), (InL, policy)) =
          (if (p_accept (a,voip,c,d,ACF callee) policy)
           then Some ((a,voip,c,d,ACF callee)#InL, policy ++
                    allow_prot_from_to_port voip (subnet_of_ip callee) (subnet_of d) 1720 ++
                    allow_prot_from_to_port voip (subnet_of d) (subnet_of_ip callee) 1720 ++
                    deny_prot_from_to voip (subnet_of d) (subnet_of c))
           else Some (InL,policy))

  VOIP_ST_3_2: VOIP_ST_3 (x, (InL,policy)) = None
```

In the Setup message, the caller specifies the port on which he wants the connection to take place so we need to open it for incoming VoIP messages.

```
consts VOIP_ST_4 :: ((ipv4, 'b voip_msg) history, ipv4, 'b voip_msg) FWStateTransition
recdef VOIP_ST_4 {}
  VOIP_ST_4_1: VOIP_ST_4 ((a,voip,c,d, Setup port), (InL, policy)) =
          (if (p_accept (a,voip,c,d,Setup port) policy)
           then Some ((a,voip,c,d,Setup port)#InL, policy ++
                    allow_prot_from_to_port voip (subnet_of d) (subnet_of c) port)
           else Some (InL,policy))

  VOIP_ST_4_2: VOIP_ST_4 (x, (InL,policy)) = None
```

The same happens after the Connect message of the callee.

```
consts VOIP_ST_5 :: ((ipv4, 'b voip_msg) history, ipv4, 'b voip_msg) FWStateTransition
recdef VOIP_ST_5 {}
  VOIP_ST_5_1: VOIP_ST_5 ((a,voip,c,d, Connect port), (InL, policy)) =
            (if (p_accept (a,voip,c,d,Connect port) policy)
             then Some ((a,voip,c,d,Connect port)#InL, policy ++
                      allow_prot_from_to_port voip (subnet_of d) (subnet_of c) port)
             else Some (InL,policy))

  VOIP_ST_5_2: VOIP_ST_5 (x, (InL,policy)) = None
```

As we do not know which entity closes the connection, we define an operator which checks if the closer is the caller.

consts
src_is_initiator :: *id* ⇒ *ipv4* ⇒ (*ipv4*,'*b voip_msg*) *history* ⇒ *bool*
primrec *src_is_initiator i a* [] = *False*
src_is_initiator i a (*p*#*S*) = (*if* ((*id p* = *i*) ∧
(∃ *port*. *data p* = *Setup port*) ∧
((*fst* (*src p*) = *fst a*))) *then*
True else src_is_initiator i a S)

In the FIN message, we have to close all the previously opened ports. This works as in the FTP close message, only a little bit more complicated.

consts *VOIP_ST_7* :: ((*ipv4*, '*b voip_msg*) *history*, *ipv4*, '*b voip_msg*) *FWStateTransition*
recdef *VOIP_ST_7* {}
VOIP_ST_7_1: *VOIP_ST_7* ((*a*,*voip*,*c*,*d*,*Fin*), (*InL*,*policy*)) =
(*if* (*p_accept* (*a*,*voip*,*c*,*d*,*Fin*) *policy*) *then*
(*if* ∃ *p1 p2*. *ports_open a* (*p1*,*p2*) *InL then* (
(*if src_is_initiator a c InL*
then (*Some* ((*a*,*voip*,*c*,*d*,*Fin*)#*InL*, *policy* ++
(*deny_prot_from_to_port voip* (*subnet_of c*) (*subnet_of d*) *1720*) ++
(*deny_prot_from_to_port voip* (*subnet_of c*) (*subnet_of d*) (*snd* (*SOME p*. *ports_open a p InL*)))++
(*deny_prot_from_to_port voip* (*subnet_of d*) (*subnet_of c*) (*fst* (*SOME p*. *ports_open a p InL*)))))
else (*Some* ((*a*,*voip*,*c*,*d*,*Fin*)#*InL*, *policy* ++
(*deny_prot_from_to_port voip* (*subnet_of c*) (*subnet_of d*) *1720*) ++
(*deny_prot_from_to_port voip* (*subnet_of c*) (*subnet_of d*) (*fst* (*SOME p*. *ports_open a p InL*))) ++
(*deny_prot_from_to_port voip* (*subnet_of d*) (*subnet_of c*) (*snd* (*SOME p*. *ports_open a p InL*)))))))
else
(*Some* ((*a*,*voip*,*c*,*d*,*Fin*)#*InL*,*policy*)))
else Some (*InL*,*policy*))

VOIP_ST_7_2: *VOIP_ST_7* (*x*,(*InL*,*policy*)) = *None*

A full VoIP protocol is the orelse of all the state transitions.

constdefs *VOIP_ST* :: ((*ipv4*, *ipv4_ip voip_msg*) *history*, *ipv4*, *ipv4_ip voip_msg*)
FWStateTransition

VOIP_ST ≡ *VOIP_ST_1 orelse*
VOIP_ST_2 orelse
VOIP_ST_3 orelse
VOIP_ST_4 orelse
VOIP_ST_5 orelse
VOIP_ST_7 orelse
VOIP_ST_F

For a full protocol run, six states are needed.

datatype *voip_states* = *S0* | *S1* | *S2* | *S3* | *S4* | *S5*

The constant *is_voip* checks if a trace corresponds to a legal VoIP protocol, given the IP-addresses of the three entities, the ID, and the two dynamic ports.

consts *is_voip* :: *voip_states* ⇒ *ipv4_ip* ⇒ *ipv4_ip* ⇒ *ipv4_ip* ⇒ *id* ⇒ *port* ⇒ *port* ⇒
(*ipv4*, *ipv4_ip voip_msg*) *history* ⇒ *bool*
primrec
is_voip H s d g i p1 p2 [] = (*H* = *S5*)
is_voip H s d g i p1 p2 (*x*#*InL*) = (λ (*id*,*pr*,*sr*,*de*,*co*). (((*pr* = *voip* ∧ *id* = *i* ∧ (
(*H* = *S1* ∧ *sr* = (*s*,*1719*) ∧ *de* = (*g*,*1719*) ∧ *co* = *ARQ* ∧ *is_voip S5 s d g i p1 p2 InL*) ∨
(*H* = *S0* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co* = *ARJ* ∧ *is_voip S1 s d g i p1 p2 InL*) ∨
(*H* = *S2* ∧ *sr* = (*g*,*1719*) ∧ *de* = (*s*,*1719*) ∧ *co* = *ACF d* ∧ *is_voip S1 s d g i p1 p2 InL*) ∨
(*H* = *S3* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Setup p1* ∧ *is_voip S2 s d g i p1 p2 InL*) ∨
(*H* = *S4* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Connect p2* ∧ *is_voip S3 s d g i p1 p2 InL*) ∨
(*H* = *S4* ∧ *sr* = (*s*,*p1*) ∧ *de* = (*d*,*p2*) ∧ *co* = *Stream* ∧ *is_voip S4 s d g i p1 p2 InL*) ∨
(*H* = *S4* ∧ *sr* = (*d*,*p2*) ∧ *de* = (*s*,*p1*) ∧ *co* = *Stream* ∧ *is_voip S4 s d g i p1 p2 InL*) ∨
(*H* = *S0* ∧ *sr* = (*d*,*1720*) ∧ *de* = (*s*,*1720*) ∧ *co* = *Fin* ∧ *is_voip S4 s d g i p1 p2 InL*) ∨
(*H* = *S0* ∧ *sr* = (*s*,*1720*) ∧ *de* = (*d*,*1720*) ∧ *co* = *Fin* ∧ *is_voip S4 s d g i p1 p2 InL*)))))) *x*

Finally, *NB_voip* returns the set of protocol traces which correspond to a correct protocol run given the three addresses, the ID, and the two dynamic ports.

constdefs
NB_voip :: *ipv4_ip* ⇒ *ipv4_ip* ⇒ *ipv4_ip* ⇒ *id* ⇒ *port* ⇒ *port* ⇒ (*ipv4*, *ipv4_ip voip_msg*) *history set*
NB_voip s d g i p1 p2≡ {*x*. (*is_voip S0 s d g i p1 p2 x*)}

end

5. Testing

After having defined the model, we show how it can be used together with HOL-TestGen to produce test data. We start with a case study of a stateless firewall and then go on to FTP and VoIP.

5.1. Test Data for a Stateless Firewall

theory *Case_study*
imports *IPv4_Combinators Testing*
begin

5.1.1. Modelling the Setting

We take the same scenario as in Chapter 2, based on IPv4 addresses. We therefore have three subnetworks. The Intranet consists of the addresses starting with 192.168, the DMZ of those starting with 192.169. All the remaining addresses are part of the Internet.

These subnetworks can be defined as follows:

constdefs
intranet::*ipv4 subnet*
intranet ≡ {{((*a*,*b*,*c*,*d*),*e*) . (*a* = *192*) ∧ (*b*=*168*)}}

dmz :: *ipv4 subnet*
dmz ≡ {{((*a*,*b*,*c*,*d*),*e*). (*a* = *192*) ∧ (*b* = *169*)}}

internet :: *ipv4 subnet*
internet ≡ {{((*a*,*b*,*c*,*d*),*e*). ¬ ((*a* = *192*) ∧ ((*b* = *168*) ∨ (*b* = *169*)))}}

Having the policy displayed as a ruleset (see Figure 5.1), the specification is pretty simple when using the provided combinators. However, an important point arises here: the correspondence between a protocol and the port number. If we want to allow only HTTP, does this mean that we allow all traffic on port 80 or any traffic declared as HTTP? The answer needs to be given individually, you could think of settings with either way being better. Therefore, the model shouldn't make a restriction here and give to the tester both possibilities. In this example we make a mixture of both to show the feasibility of the two possibilities.

constdefs
Intranet_DMZ :: (*ipv4*,'*b*) *Rule*

Source Address	Destination Address	Protocol	Action
Intranet	DMZ	smtp, imap	*accept*
Intranet	Internet	not smtp	*accept*
Intranet	Internet	smtp	*deny*
DMZ	Intranet	any	*deny*
DMZ	Internet	smtp	*allow*
Internet	Intranet	any	*deny*
Internet	DMZ	http, smtp	*accept*
any	any	any	*deny*

Figure 5.1.: The Policy

Intranet_DMZ ≡ (*allow_prot_from_to smtp intranet dmz*) ++
(*allow_prot_from_to imap intranet dmz*)

Intranet_Internet :: (*ipv4*,'*b*) *Rule*
Intranet_Internet ≡ *allow_all_but_prot_from_to smtp intranet internet*

DMZ_Intranet :: (*ipv4*,'*b*) *Rule*
DMZ_Intranet ≡ *deny_all_from_to dmz intranet*

DMZ_Internet :: (*ipv4*,'*b*) *Rule*
DMZ_Internet ≡ *allow_prot_from_to smtp dmz internet*

Internet_Intranet :: (*ipv4*,'*b*) *Rule*
Internet_Intranet ≡ *deny_all_from_to internet intranet*

Internet_DMZ :: (*ipv4*,'*b*) *Rule*
Internet_DMZ ≡ (*allow_prot_from_to_port http internet dmz 80*) ++
(*allow_prot_from_to_port smtp internet dmz 25*)

These rules can now be combined into one single policy. Remember that they have to be given in reverse order. In this setting, the ordering of all but the *deny_all* rule is arbitrary.

constdefs
test_policy :: (*ipv4*, '*b*) *Policy*
test_policy ≡ *deny_all* ++
DMZ_Internet ++
DMZ_Intranet ++
Intranet_Internet ++
Intranet_DMZ ++
Internet_DMZ ++
Internet_Intranet

Again for convenience, we put all these definitions into one lemma.

lemmas *ruleset* = *test_policy_def deny_all_def*
DMZ_Internet_def DMZ_Intranet_def

Intranet_Internet_def Intranet_DMZ_def
Internet_DMZ_def Internet_Intranet_def

Finally, we need to specify the content. As it is of no interest to us in this example, we just define a datatype called content.

datatype *content* = *content*

5.1.2. Some Auxiliary Lemmas

Next we define a couple of lemmas which will help us later in dramatically simplifying the test generation phase. They have all very easy proofs.

lemma *aux*: ((((a,b,c,d), e)) ⊏ {{((x1,x2,x3,x4),y). P x1 x2 x3 x4 y}}) = (P a b c d e)
by (*simp add*: *in_subnet_def*)

lemma *src_in_dmz*: (*src* (q,w,((a,b,c,d),e),r,t)) ⊏ *dmz* = (a = 192 ∧ b = 169)
by (*simp add*: *dmz_def src_def aux*)

lemma *dest_in_dmz*: (*dest* (q,w,r,((a,b,c,d),e),t)) ⊏ *dmz* = (a = 192 ∧ b = 169)
by (*simp add*: *dmz_def dest_def aux*)

lemma *src_in_intranet*: (*src* (q,w,((a,b,c,d),e),r,t)) ⊏ *intranet* = (a = 192 ∧ b = 168)
by (*simp add*: *src_def intranet_def aux*)

lemma *dest_in_intranet*: (*dest* (q,w,r,((a,b,c,d),e),t)) ⊏ *intranet* = (a = 192 ∧ b = 168)
by (*simp add*: *dest_def intranet_def aux*)

lemma *src_in_internet*: (*src* (q,w,((a,b,c,d),e),r,t)) ⊏ *internet* = (a ≠ 192 ∨ (b ≠ 168 ∧ b ≠ 169))
by (*simp add*: *src_def internet_def aux*)

lemma *dest_in_internet*: (*dest* (q,w,r,((a,b,c,d),e),t)) ⊏ *internet* = (a ≠ 192 ∨ (b ≠ 168 ∧ b ≠ 169))
by (*simp add*: *dest_def internet_def aux*)

lemma *src_port*: *src_port* (a,c,x,d,e) = *snd x*
by (*simp add*: *src_port_def aux*)

lemma *dest_port*: *dest_port* (a,c,d,x,e) = *snd x*
by (*simp add*: *dest_port_def aux*)

lemma *protocol_is*: *protocol* (a,p,d,b,e) = *p*
by (*simp add*: *protocol_def aux*)

lemmas *aux_lemmas* = *aux src_in_internet dest_in_intranet src_port dest_port protocol_is*
dest_in_internet src_in_intranet src_in_dmz dest_in_dmz

5.1.3. Testing

Now we have everything we need to start the testing. At first, we have to define the test specification. The specification simply says that for each packet, the Firewall Under Test (FUT), shall behave the same way as what our policy specifies for this packet.

test_spec *FUT* (*x*::(*ipv4*, *content*) *packet*) = *test_policy x*
  apply (*rule_tac x*=*x* **in** *spec*, *rule allI*)

This step is needed to introduce a meta-all-quantor which is necessary for the following split rule.

  apply (*simp only*: *split_tupled_all*)

We first have to unfold the policy and simplify the goal with the help of our auxiliary lemmas.

  apply (*unfold ruleset ipv4_rules stateless_rules*)
  apply (*simp add*: *aux_lemmas*)

Now we have the proof state we want to have to produce the abstract test cases, we can therefore apply *gen_test_cases*.

  apply (*gen_test_cases FUT*)

This step produced the test partitioning. We now bind the test theorem to a particular named *test environment*.

store_test_thm *policy_test*

Now we want to generate concrete test data, i.e., all variables in the test cases must be instantiated with concrete values. This involves a random solver which tries to solve the constraints by randomly choosing values.

gen_test_data *policy_test*

This produces a set of 390 test data which can be accessed with *thm policy_test.test_data*. All in all this took about four hours. The time used depends heavily on the policy which is tested. Already rather small changes can lead to big differences in both the time being used and the test partitioning produced. This shows that a firewall configuration should always be tested when the policy changes.
The test data all look like the following two examples. The first one is an SMTP packet from the DMZ into the Internet which is allowed, the second one, coming from the Intranet, is denied.

- `FUT (9, smtp, ((192,169,6,1),4),((9,3,4,8),9), content) =`
  `Some (accept (9, smtp, ((192,169,6,1,),4),((9,3,4,8),9), content))`
- `FUT (4, smtp, ((192,168,4,4),3),((5,9,6,6),80), content) = Some deny`

These test data look like we would have expected. The FUT receives a packet and either accepts or denies it. This action should be the same as the expected action, which is displayed on the right hand side. They can now be employed for testing a real firewall. In the Group for Information Security at ETH, a tool called fwtest [14] is developed which

automatically examines a single firewall by running predefined test cases. It crafts, injects, captures and analyzes the test packets and logs the irregularities, taking a file with descriptions of TCP packets as inputs. Although not implemented here, a simple perl script could rewrite our output and produce an input file for fwtest.

end

5.2. Test Data for FTP

theory *FTP_Test*
imports *FTP Testing*
begin

We now model the setting for which we would like to get the test data for an FTP run. We take a simple policy which allows only FTP from the Intranet to the Internet and denies everything else. The definitions for the subnetworks are taken from the previous case study.

constdefs
intranet :: *ipv4 subnet*
intranet ≡ {{((a,b,c,d),e) . (a = 192) ∧ (b=168)}}

dmz :: *ipv4 subnet*
dmz ≡ {{((a,b,c,d),e). (a = 192) ∧ (b = 169)}}

internet :: *ipv4 subnet*
internet ≡ {{((a,b,c,d),e). ¬ ((a = 192) ∧ ((b = 168) ∨ (b = 169)))}}

constdefs
ftp_policy :: (*ipv4*,'*b*) *Rule*
ftp_policy ≡ *deny_all* ++ *allow_prot_from_to_port ftp intranet internet 21*

The next two constants check if an address is in the Intranet or in the Internet respectively.

constdefs
is_in_intranet :: *ipv4 adr* ⇒ *bool*
is_in_intranet a ≡ (*fst* (*fst a*)) = 192 ∧ ((*fst* (*snd* (*fst a*)) = 168))

is_in_internet :: *ipv4 adr* ⇒ *bool*
is_in_internet a ≡ (*fst* (*fst a*)) ≠ 192 ∨ ((*fst* (*snd* (*fst a*)) ≠ 168) ∧ (*fst* (*snd* (*fst a*)) ≠ 169))

The search depth of HOL-TestGen denotes the maximal length of a history. Its default value is 3. As a full FTP protocol run needs at least 4 packets, we have to adjust it with the following command:

testgen_params [*depth=4*]

The test specification looks differently now. It says that for all traces which are correct protocol runs, and for which some constraints hold, our firewall under test behaves the

same way as the Mfold of the reversed trace, applied on an initial state with the empty trace and the initial policy under the FTP state transitions. The Mfold returns a state which is a list of the packets which got accepted and the new policy. In testing we are only interested in the first part of it.

test_spec $t \in NB\_ftp\ c\ s\ i\ p \wedge is\_in\_intranet\ c \wedge is\_in\_internet\ s \wedge (snd\ s) = 21 \implies$
$FUT\ t = Mfold\ (rev\ t)\ ([], ftp\_policy)\ FTP\_ST$

This time we do not unfold the policy, but only the constraints of the trace.

apply (*unfold NB_ftp_def is_in_internet_def is_in_intranet_def*)

This already specifies our desired test partitioning, so we can apply *gen_test_cases* and store our theorem.

apply (*gen_test_cases FUT*)
store_test_thm *ftp_test*

Now we can create the concrete test data. When the values of the fields of a packet are guessed, we need to Mfold it, and for that, the policy needs too be unfolded and also the definitions used in the state transitions need to be added to the simplifier.

lemmas *ST_simps* =
Let_def Mfold.simps ALL_ST_def FTP_ST_def repeat_def orelse_def in_subnet_def
src_def dest_def dest_port_def protocol_def subnet_of_def id_def port_open_def
is_init_def is_data_def is_port_request_def is_close_def p_accept_def
data_def exI stateless_rules ipv4_rules ftp_policy_def is_in_intranet_def
is_in_internet_def intranet_def internet_def

declare *ST_simps* [*simp*]

We are now ready to produce the test data. This time, as more calculation is involved, this phase takes longer than in the stateless case. Together, the two phases took about seven minutes. The time which is used mainly depends upon the search depth of HOL-TestGen.

gen_test_data *ftp_test*

We get four different test data. They all look similar to the following example, after having resimplified the new policy:

```
FUT [(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), close),
(6, ftp, ((8, 5, 10, 6), 21), ((192, 168, 5, 1), 4), ftp_data),
(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), port_request 4),
(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), init)] =
([(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), close),
(6, ftp, ((8, 5, 10, 6), 21), ((192, 168, 5, 1), 4), ftp_data),
(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), port_request 4),
(6, ftp, ((192, 168, 5, 1), 7), ((8, 5, 10, 6), 21), init)],
ftp_policy ++
allow_prot_from_to_port ftp (subnet_of ((8, 5, 10, 6), 21)) (subnet_of ((192, 168,
5, 1), 4)) 4 ++
```

```
deny_prot_from_to_port ftp (subnet_of ((8, 5, 10, 6), 21)) (subnet_of ((192, 168, 5,
1), 4)) 4)
```

How do this test data have to be interpreted? On the left hand side we always have a FUT on a list. This list denotes the packets which should be sent to the firewall. The first part of the right hand side is the expected reaction, being a list of all the packets which were allowed by the firewall. The second part of the right hand side is the new policy unfolded. It is usually not used when testing and can therefore normally be ignored. If it is desired to reuse it, it can of course be simplified drastically.

end

5.3. Test Data for VoIP

theory *VOIP_Test*
imports *VOIP Testing*
begin

We can now produce test data for the VoIP protocol. We need three subnetworks, one being the caller, one the gatekeeper, and one the internet, which is where the callee is.

constdefs
caller :: *ipv4 subnet*
caller ≡ {{((*a*,*b*,*c*,*d*),*e*) . (*a*=*192*) ∧ (*b*=*168*) ∧ (*c*=*1*) ∧ (*d*=*1*)}}

gatekeeper ::*ipv4 subnet*
gatekeeper ≡ {{((*a*,*b*,*c*,*d*),*e*). *a*=*217* ∧ *b* = *145* ∧ *c* = *22* ∧ *d* = *2*}}

internet :: *ipv4 subnet*
internet ≡ {{(*a*,*b*). *a* ≠ (*192*,*168*,*1*,*1*) ∧ *a* ≠ (*217*,*145*,*22*,*2*)}}

The policy forbids everything but VoIP on port 1719 from the caller to the gatekeeper.

constdefs
voip_policy ::(*ipv4*, *'a voip_msg*) *Policy*
voip_policy ≡ *deny_all* ++ *allow_prot_from_to_port voip caller gatekeeper 1719*

The next two constants check if an IP-address is in a respective subnetwork.

constdefs
is_in_caller :: *ipv4_ip adr* ⇒ *bool*
is_in_caller a ≡ (*a* = (*192*,*168*,*1*,*1*))

is_in_gatekeeper :: *ipv4_ip adr* ⇒ *bool*
is_in_gatekeeper a ≡ (*a* = (*217*,*145*,*22*,*2*))

The testgen depth has to be increased to at least six, if we want to have a full protocol run.

testgen_params [*depth*=*6*]

We proceed the same way as when we were testing an FTP protocol. We start with the test specification:

test_spec $t \in NB\_voip\ a\ b\ c\ x\ y\ z \wedge is\_in\_gatekeeper\ c \wedge is\_in\_caller\ a \Longrightarrow$
$FUT\ t = Mfold\ (rev\ t)\ ([],voip\_policy)\quad VOIP\_ST$

We go on unfolding the constraints, applying *gen_test_cases* and storing the test theorem.

apply (*unfold NB_voip_def is_in_gatekeeper_def is_in_caller_def*)
apply (*gen_test_cases FUT*)

store_test_thm *voip_test*

And again, we need to add a couple of lemmas to the simplifier before the test data generation phase.

lemmas *ST_simps* =
Mfold.simps VOIP_ST_def orelse_def voip_policy_def subnet_of_def
in_subnet_def protocol_def id_def p_accept_def ports_open_def is_connect_def
was_arq_def is_arq_def is_fin_def is_setup_def
src_port_def src_def dest_def dest_port_def stateless_rules ipv4_rules
caller_def gatekeeper_def data_def internet_def subnet_of_ip_def

declare *ST_simps* [*simp*]

gen_test_data *voip_test*

Due to the bigger search depth which is necessary to model the whole protocol, this scenario takes much longer than FTP and produces seven test data. They also look similar to the ones of the FTP scenario. We provide one example, but without the unfolded policy:

```
FUT [(6, voip, ((32, 4, 32, 4), 1), ((192, 168, 1, 1), 0), Fin),
(6, voip, ((32, 4, 32, 4), 1), ((192, 168, 1, 1), 0), Stream),
(6, voip, ((32, 4, 32, 4), 1720), ((192, 168, 1, 1), 1720), Connect 1),
(6, voip, ((192, 168, 1, 1), 1720), ((32, 4, 32, 4), 1720), Setup 0),
(6, voip, ((217, 145, 22, 2), 1719), ((192, 168, 1, 1), 1719), ACF (32, 4, 32, 4)),
(6, voip, ((192, 168, 1, 1), 1719), ((217, 145, 22, 2), 1719), ARQ)] =
([(6, voip, ((32, 4, 32, 4), 1), ((192, 168, 1, 1), 0), Fin),
(6, voip, ((32, 4, 32, 4), 1), ((192, 168, 1, 1), 19), Stream),
(6, voip, ((32, 4, 32, 4), 1720), ((192, 168, 1, 1), 1720), Connect 1),
(6, voip, ((192, 168, 1, 1), 1720), ((32, 4, 32, 4), 1720), Setup 0),
(6, voip, ((217, 145, 22, 2), 1719), ((192, 168, 1, 1), 1719), ACF (32, 4, 32, 4)),
(6, voip, ((192, 168, 1, 1), 1719), ((217, 145, 22, 2), 1719), ARQ)]
```

We see that our model produces nice test data for VoIP as well. Other protocols could be added in a similar way.

end

6. Conclusion

6.1. Results

We have presented a formal model of both a stateless and a stateful firewall in HOL. One of our goals in the beginning was to hold this model as simple as possible while being very close to the real world. Especially when looking at the stateless case, we can say that his goal is achieved. It's possible to specify a policy in a very easy and natural way, the correspondence between such a specification and a ruleset as in Figure 2.4 is quite trivial.

The key points of the model of a stateless packet filter were taken along to the stateful case. Clearly, that model is somewhat more complicated, as the different state transitions have to be modelled. However, once all required protocols are modelled, a specific policy can be specified almost as easily as before.

Another goal was to hold the model as generic and extendible as possible. This was achieved as well. Our model provides a very good basis for further extensions. Other address representations, contents, or protocols could easily be added. We could even model the state transition behaviour differently without changing much in the other parts of the model.

We wanted the model to be used for creating test data with HOL-TestGen. We showed how to do this by modeling a realistic scenario of a stateless firewall between three subnetworks with a simple policy. For this setting, HOL-TestGen produced 390 test cases. These test cases look sensible. The test partitioning seems to make sense and the test data can be used for testing a real configuration, as we get the specified reaction to packets with a source and destination address and the protocol - exactly those things typically indicated in a policy.

We then created test data for correct runs of FTP and VoIP. Although these test data look differently now, they should also be usable for fwtest. Instead of individual packets and the desired reaction, we now get lists of incoming packets together with the lists of the corresponding packets which got accepted. The packets themselves look as before.

We hence have all the tools necessary for being able to produce a realistic test plan. It is very situation specific how such a test plan will look like, the model merely provides the necessary tools.

An example test plan might look like this:

1. Produce stateless test data
2. Perform testing with this test data

3. Produce test data for a stateful protocol (could be one correct run, several correct runs, incorrect runs etc.)
4. Perform testing with this test data
5. After each testing of the stateful protocol, perform testing with the stateless test data from step 1 to check that the firewall behaves as before protocol execution
6. Go back to 3

Naturally, the question arises if this couldn't get automated such that HOL-TestGen produces big traces which follow this procedure. However, this is not feasible for the moment. The state explosion is much too big, so some handwork is still needed (and might even be sensible for fine-tuning the testing).

6.2. Future Work

The model is as generic as possible, making it very easy to extend. Possible extensions include modelling different representations of IPv4, IPv6, extended versions of FTP and VoIP and other stateful protocols.

The model of the stateless firewall should be checked with a real policy to see if the policy specification tools are reasonable. Then the test data should be rewritten, used as input to fwtest, and checked on a real configured firewall implementation.

The test case generation in our examples already took quite long. This seems to indicate that we will hit limits by state-space explosions of the approach when extended to large policies except that new abstractions, test techniques or algorithms were developed in HOL-TestGen.

A. Combinators

Here we provide the definitions of the combinators.
theory *Policy_Combinators*
imports *Policy*
begin

Combinators which allow or deny everything

Allow every packet:

constdefs
allow_all :: ('*a*, '*b*) *Rule*
allow_all p ≡ *Some* (*accept p*)

Deny every packet:

constdefs
deny_all :: ('*a*,'*b*) *Rule*
deny_all p ≡ *Some* (*deny*)

Combinators which allow or deny packets based on their source and/or destination address.

Allow every packet coming from a specific subnet

constdefs
allow_all_from :: '*a*::*net subnet* ⇒ ('*a*,'*b*) *Rule*
allow_all_from src_net ≡ *allow_all* |‘ {*pa*. *src pa* ⊏ *src_net*}

Deny every packet coming from a specific subnet

constdefs
deny_all_from :: '*a*:: *net subnet* ⇒ ('*a*,'*b*) *Rule*
deny_all_from src_net ≡ *deny_all* |‘ {*pa*. *src pa* ⊏ *src_net*}

Allow every packet going to a specific subnet

constdefs
allow_all_to :: '*a*::*net subnet* ⇒ ('*a*,'*b*) *Rule*
allow_all_to dest_net ≡ *allow_all* |‘ {*pa*. *dest pa* ⊏ *dest_net*}

Deny every packet going to a specific subnet

constdefs
deny_all_to :: '*a*::*net subnet* ⇒ ('*a*,'*b*) *Rule*
deny_all_to dest_net ≡ *deny_all* |‘ {*pa*. *dest pa* ⊏ *dest_net*}

Allow all packets coming from and going to specific subnets

constdefs
allow_all_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_all_from_to src_net dest_net ≡ *allow_all* |‘ {*pa. src pa* ⊏ *src_net* ∧ *dest pa* ⊏ *dest_net*}

Deny all packets coming from and going to specific subnets

constdefs
deny_all_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_all_from_to src_net dest_net ≡ *deny_all* |‘ {*pa. src pa* ⊏ *src_net* ∧ *dest pa* ⊏ *dest_net*}

Combinators which allow or deny packets based additionally on their protocol

Allow all packets of a specific protocol

constdefs
allow_protocol :: *protocol* ⇒ *('a,'b) Rule*
allow_protocol prot ≡ *allow_all* |‘ {*pa. protocol pa* = *prot*}

Deny all packets of a specific protocol

constdefs
deny_protocol :: *protocol* ⇒ *('a,'b) Rule*
deny_protocol prot ≡ *deny_all* |‘ {*pa. protocol pa* = *prot*}

Allow all packets of a specific protocol coming from a specific subnet

constdefs
allow_prot_from :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_prot_from prot src_net ≡ *allow_all* |‘ {*pa. src pa* ⊏ *src_net* ∧ *protocol pa* = *prot*}

Deny all packets of a specific protocol coming from a specific subnet

constdefs
deny_prot_from :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_prot_from prot src_net ≡ *deny_all* |‘ {*pa. src pa* ⊏ *src_net* ∧ *protocol pa* = *prot*}

Allow all packets of a specific protocol going to a specific subnet

constdefs
allow_prot_to :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_prot_to prot dest_net ≡ *allow_all* |‘ {*pa. dest pa* ⊏ *dest_net* ∧ *protocol pa* = *prot*}

Deny all packets of a specific protocol going to a specific subnet

constdefs *deny_prot_to* :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_prot_to prot dest_net ≡ *deny_all* |‘ {*pa. dest pa* ⊏ *dest_net* ∧ *protocol pa* = *prot*}

Allow all packets of a specific protocol coming from and going to specific subnets

constdefs
allow_prot_from_to :: *protocol* ⇒ *'a::net subnet* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_prot_from_to prot src_net dest_net ≡ *allow_all* |‘
{*pa. src pa* ⊏ *src_net* ∧ *dest pa* ⊏ *dest_net* ∧ *protocol pa* = *prot*}

Deny all packets of a specific protocol coming from and going to specific subnets

constdefs
deny_prot_from_to :: *protocol* ⇒ *'a::net subnet* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_prot_from_to prot src_net dest_net ≡ *deny_all|‘*
{*pa. src pa* ⊏ *src_net* ∧ *dest pa* ⊏ *dest_net* ∧ *protocol pa* = *prot*}

Combinators which allow or deny everything except a protocol

Allow everything except a specific protocol:

constdefs
allow_all_but_prot :: *protocol* ⇒ *('a,'b) Rule*
allow_all_but_prot prot ≡ *allow_protocol prot |‘* {*pa. protocol pa* ≠ *prot*} ++
deny_protocol prot |‘{*pa. protocol pa* = *prot*}

Deny everything except a specific protocol:

constdefs
deny_all_but_prot :: *protocol* ⇒ *('a,'b) Rule*
deny_all_but_prot prot ≡ *deny_protocol prot |‘* {*pa. protocol pa* ≠ *prot*} ++
allow_protocol prot |‘ {*pa. protocol pa* = *prot*}

Allow every packet coming from a specific subnet, except those of a specific protocol:

constdefs
allow_all_but_prot_from :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_all_but_prot_from prot src_net ≡ *allow_all_from src_net |‘* {*pa. protocol pa* ≠ *prot*} ++
deny_all_from src_net |‘ {*pa. protocol pa* = *prot*}

Deny every packet coming from a specific subnet, except those of a specific protocol:

constdefs
deny_all_but_prot_from :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_all_but_prot_from prot src_net ≡ *deny_all_from src_net |‘* {*pa. protocol pa* ≠ *prot*} ++
allow_all_from src_net |‘ {*pa. protocol pa* = *prot*}

Allow every packet going to a specific subnet, except those of a specific protocol:

constdefs
allow_all_but_prot_to :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_all_but_prot_to prot dest_net ≡ *allow_all_to dest_net |‘* {*pa. protocol pa* ≠ *prot*} ++
deny_all_to dest_net |‘ {*pa. protocol pa* = *prot*}

Deny every packet going to a specific subnet, except those of a specific protocol:

constdefs
deny_all_but_prot_to :: *protocol* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
deny_all_but_prot_to prot dest_net ≡
deny_all_to dest_net |‘ {*pa. protocol pa* ≠ *prot*} ++
allow_all_to dest_net |‘ {*pa. protocol pa* = *prot*}

Allow every packet coming from and going to a specific subnet, except those of a specific protocol:

constdefs
allow_all_but_prot_from_to :: *protocol* ⇒ *'a::net subnet* ⇒ *'a::net subnet* ⇒ *('a,'b) Rule*
allow_all_but_prot_from_to prot src_net dest_net ≡
allow_all_from_to src_net dest_net |‘ {pa. protocol pa ≠ prot} ++
deny_all_from_to src_net dest_net |‘ {pa. protocol pa = prot}

Deny every packet coming from and going to a specific subnet, except those of a specific protocol:

constdefs
deny_all_but_prot_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *protocol* ⇒ *('a,'b) Rule*
deny_all_but_prot_from_to src_net dest_net prot ≡
deny_all_from_to src_net dest_net |‘ {pa. protocol pa ≠ prot} ++
allow_all_from_to src_net dest_net |‘ {pa. protocol pa = prot}

Combinators which allow or deny everything of a set of protocols

Allow all packets of a specific set of protocols coming from and going to specific subnets

constdefs
allow_prots_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *protocol set* ⇒ *('a,'b) Rule*
allow_prots_from_to src_net dest_net prots ≡ *allow_all |‘*
{pa. src pa ⊏ src_net ∧ dest pa ⊏ dest_net ∧ protocol pa ∈ prots}

Deny all packets of a specific set of protocols coming from and going to specific subnets

constdefs
deny_prots_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *protocol set* ⇒ *('a,'b) Rule*
deny_prots_from_to src_net dest_net prots ≡ *deny_all |‘*
{pa. src pa ⊏ src_net ∧ dest pa ⊏ dest_net ∧ protocol pa ∈ prots}

Combinators which allow or deny everything except of a set of protocols

Allow every packet coming from and going to a specific subnet, except those of a specific set of protocols

constdefs
allow_all_but_prots_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *protocol set* ⇒ *('a,'b) Rule*
allow_all_but_prots_from_to src_net dest_net prots ≡
allow_all_from_to src_net dest_net |‘ {pa. protocol pa ∉ prots} ++
deny_all_from_to src_net dest_net |‘ {pa. protocol pa ∈ prots}

Deny every packet coming from and going to a specific subnet, except those of a specific set of protocols

constdefs
deny_all_but_prots_from_to :: *'a::net subnet* ⇒ *'a::net subnet* ⇒ *protocol set* ⇒ *('a,'b) Rule*
deny_all_but_prots_from_to src_net dest_net prots ≡
deny_all_from_to src_net dest_net |‘ {pa. protocol pa ∉ prots} ++
allow_all_from_to src_net dest_net |‘ {pa. protocol pa ∈ prots}

All these combinators and the default rules are put into one single lemma called *stateless_rules* to make life easier when we need to unfold a policy consisting of several rules.

lemmas *stateless_rules* =
allow_all_def deny_all_def allow_all_from_def deny_all_from_def
allow_all_to_def deny_all_to_def allow_all_from_to_def deny_all_from_to_def
allow_protocol_def deny_protocol_def allow_prot_from_def deny_prot_from_def
allow_prot_to_def deny_prot_to_def allow_prot_from_to_def deny_prot_from_to_def
allow_all_but_prot_def deny_all_but_prot_def allow_all_but_prot_from_def
deny_all_but_prot_from_def allow_all_but_prot_to_def deny_all_but_prot_to_def
allow_all_but_prot_from_to_def deny_all_but_prot_from_to_def
allow_prots_from_to_def deny_prots_from_to_def
allow_all_but_prots_from_to_def deny_all_but_prots_from_to_def
map_add_def restrict_map_def

end
theory *IPv4_Combinators*
imports *Policy_Combinators IPv4*
begin

In this theory, we defines the combinators which can be used to define policies for stateless firewalls based on IPv4 networks. The rules are generally the ones from stateless.thy, but also take port numbers into account. If we don't want to use the port numbers in our specification, we can continue to use the original rules.

Combinators which allow or deny packets based on their source and or destination address.

Allow every packet coming from a specific subnet on a specific port

constdefs
allow_all_from_port :: *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_all_from_port src_net s_port ≡ *allow_all_from src_net* |‘ {*pa. src_port pa* = *s_port*}

Deny every packet coming from a specific subnet on specific port

constdefs
deny_all_from_port :: *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
deny_all_from_port src_net s_port ≡ *deny_all_from src_net* |‘ {*pa. src_port pa* = *s_port*}

Allow every packet going to a specific subnet on a specific port

constdefs
allow_all_to_port :: *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_all_to_port dest_net d_port ≡ *allow_all_to dest_net* |‘ {*pa. dest_port pa* = *d_port*}

Deny every packet going to a specific subnet on a specific port

constdefs
deny_all_to_port :: *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
deny_all_to_port dest_net d_port ≡ *deny_all_to dest_net*|‘ {*pa. dest_port pa* = *d_port*}

Allow all packets coming from a specific port on a specific subnet to a specific subnet

constdefs

allow_all_from_port_to :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ (*ipv4*, *'b*) *Rule*
allow_all_from_port_to src_net s_port dest_net ≡ *allow_all_from_to src_net dest_net* |‘ {*pa. src_port pa* = *s_port*}

Deny every packet coming from a specific port on a specific subnet to a specific subnet

constdefs
deny_all_from_port_to :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ (*ipv4*, *'b*) *Rule*
deny_all_from_port_to src_net s_port dest_net ≡ *deny_all_from_to src_net dest_net*|‘ {*pa. src_port pa* = *s_port*}

Allow all packets coming from a specific port on a specific subnet to a port on a specific subnet

constdefs
allow_all_from_port_to_port :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_all_from_port_to_port src_net s_port dest_net d_port ≡ *allow_all_from_port_to src_net s_port dest_net* |‘ {*pa. dest_port pa* = *d_port*}

Deny every packet coming from a specific port on a specific subnet to a port on a specific subnet

constdefs
deny_all_from_port_to_port :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
deny_all_from_port_to_port src_net s_port dest_net d_port ≡ *deny_all_from_port_to src_net s_port dest_net*|‘ {*pa. dest_port pa* = *d_port*}

Allow all packets coming from and going to specific subnets on specific ports

constdefs
allow_all_from_to_port :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_all_from_to_port src_net s_port dest_net d_port ≡ *allow_all_from_to src_net dest_net*|‘
{*pa. src_port pa* = *s_port* ∧ *dest_port pa* = *d_port*}

Deny all packets coming from and going to specific subnets on specific ports

constdefs
deny_all_from_to_port :: *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
deny_all_from_to_port src_net s_port dest_net d_port ≡ *deny_all_from_to src_net dest_net* |‘
{*pa. src_port pa* = *s_port* ∧ *dest_port pa* = *d_port*}

Combinators which allow or deny packets based additionally on their protocol

Allow all packets of a specific protocol coming from a specific subnet on a specific port

constdefs
allow_prot_from_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_prot_from_port prot src_net s_port ≡ *allow_prot_from prot src_net* |‘
{*pa. src_port pa* = *s_port*}

Deny all packets of a specific protocol coming from a specific subnet on a specific port

constdefs
deny_prot_from_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*

deny_prot_from_port prot src_net s_port ≡ *deny_prot_from prot src_net* |‘
{*pa*. *src_port pa* = *s_port*}

Allow all packets of a specific protocol going to a specific subnet on a specific port

constdefs
allow_prot_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, ′*b*) *Rule*
allow_prot_to_port prot dest_net d_port ≡ *allow_prot_to prot dest_net* |‘
{*pa*. *dest_port pa* = *d_port*}

Deny all packets of a specific protocol going to a specific subnet on a specific port

constdefs
deny_prot_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, ′*b*) *Rule*
deny_prot_to_port prot dest_net d_port ≡ *deny_prot_to prot dest_net* |‘
{*pa*. *dest_port pa* = *d_port*}

Allow all packets of a specific protocol coming from a specific subnet on a specific port and going to a specific subnet

constdefs
allow_prot_from_port_to :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ (*ipv4*, ′*b*) *Rule*
allow_prot_from_port_to pr src_net s_port dest_net ≡ *allow_prot_from_to pr src_net dest_net* |‘
{*pa*. *src_port pa* = *s_port*}

Deny all packets of a specific protocol coming from a specific subnet on a specific port and going to a specific subnet

constdefs
deny_prot_from_port_to :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ (*ipv4*, ′*b*) *Rule*
deny_prot_from_port_to pr src_net s_port dest_net ≡ *deny_prot_from_to pr src_net dest_net* |‘
{*pa*. *src_port pa* = *s_port*}

Allow all packets of a specific protocol coming from a specific subnet and going to a specific subnet on a specific port

constdefs
allow_prot_from_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, ′*b*) *Rule*
allow_prot_from_to_port pr src_net dest_net d_port ≡ *allow_prot_from_to pr src_net dest_net* |‘
{*pa*. *dest_port pa* = *d_port*}

Deny all packets of a specific protocol coming from a specific subnet and going to a specific subnet on a specific port

constdefs
deny_prot_from_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, ′*b*) *Rule*
deny_prot_from_to_port pr src_net dest_net d_port ≡ *deny_prot_from_to pr src_net dest_net* |‘
{*pa*. *dest_port pa* = *d_port*}

Allow all packets of a specific protocol coming from a specific subnet on a specific port and going to a specific subnet on a specific port

constdefs

allow_prot_from_port_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
allow_prot_from_port_to_port pr src_net s_port dest_net d_port ≡ *allow_all_from_port_to_port src_net s_port dest_net d_port*|‘
{*pa*. *protocol pa* = *pr*}

Deny all packets of a specific protocol coming from a specific subnet and going to a specific subnet on a specific port

constdefs
deny_prot_from_port_to_port :: *protocol* ⇒ *ipv4 subnet* ⇒ *port* ⇒ *ipv4 subnet* ⇒ *port* ⇒ (*ipv4*, *'b*) *Rule*
deny_prot_from_port_to_port pr src_net s_port dest_net d_port ≡ *deny_all_from_port_to_port src_net s_port dest_net d_port*|‘
{*pa*. *protocol pa* = *pr*}

As before, we put all the rules into one lemma called *ipv4_rules* to ease writing later.

lemmas *ipv4_rules* =
allow_all_from_port_def deny_all_from_port_def
allow_all_to_port_def deny_all_to_port_def allow_all_from_to_port_def
deny_all_from_to_port_def allow_prot_from_port_def deny_prot_from_port_def
allow_prot_to_port_def deny_prot_to_port_def allow_prot_from_port_to_def
deny_prot_from_port_to_def allow_prot_from_to_port_def deny_prot_from_to_port_def

end

B. Session Graph

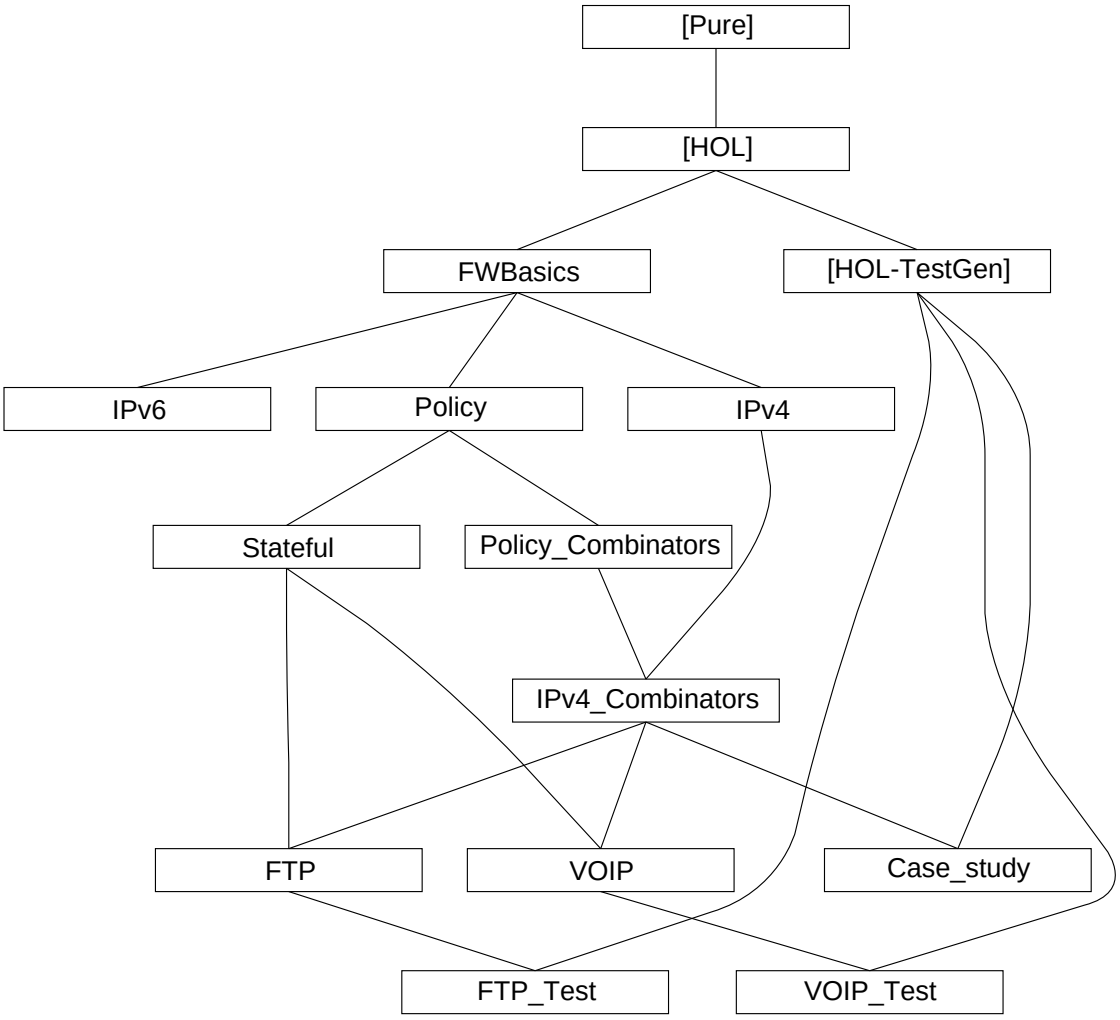

Figure B.1.: Session Graph

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester thesis for Lukas Brügger

Testing Firewalls Policies using HOL-TestGen

Supervisors: Achim Brucker, Diana Senn and Burkhart Wolff
Professor: Prof. D. Basin
Issue Date: November, 3rd 2005
Submission Date: February 2006

Introduction and Motivation

Today, unrestricted access to the Internet is a security risk. Therefore, *firewalls* are a widely used tool for controlling the access to computers in a (sub)network and services implemented on them. Firewalls[1] filter out undesired TCP/IP packets in the data-flow going to and from a network. But which traffic is undesired? This varies from company to company and should therefore be described in a *(network) security policy*. This policy should then be implemented by the firewalls, which are normally configured "by hand"; this is usually a highly error-prone activity. In order to be sure that a firewall really implements a given security policy, a systematic test method is desirable, that checks if a concrete firewall configuration conforms to a given security policy. To construct systematic tests in an effective and trustworthy way, a test set should be constructed semi-automatically.

Several approaches for the generation of test-cases are well-known: while *unit-test* oriented test generation methods essentially use pre- and postconditions of system operation specifications, *sequence-test* oriented approaches essentially use temporal specifications or automata based specifications of system behavior.

In our group, we are developing an interactive test environment environment HOL-TestGen [1, 2, 3], based on the interactive theorem proving environment Isabelle/HOL [4]. While HOL-TestGen is originally geared towards unit-tests, the rich underlying data-structures of higher-order logic (HOL) allow for a temporal description of a security policy as a set of admissible communication traces that a firewall can accept. On the basis of a realistic description of this set as recursive predicates, HOL-TestGen can generate test cases even for firewalls.

[1] In this document we use the term firewall to denote a *stateless* or *stateful packet filter*. A packet filter can filter traffic at OSI Layer 4 (TCP and UDP). It can forward (changed or unchanged), drop, or reject a packet based on its source IP address, its source port, its destination IP address, its destination port and its TCP flags. With stateful packet filters (contrary to stateless packet filters) the filtering is based on an additional criterion: the state of the connection the packet belongs to.

Moreover, a tool named fwtest [5] is developed in our group that reads in test specifications for firewall tests and then executes them physically on the network.

Assignment

During this work, the student should develop realistic models of stateless and stateful firewalls, several combinators that ease the task of specifying security policies in the sense above, and several well-chosen examples of typical policies used in practice. The models should be geared towards executability, such that they can be processed by HOL-TestGen.

Objective

The main goal of this project is to develop a new approach for testing (validating) that a firewall implements a given policy correctly.

The long-term goal is to develop a tool-supported formal software development process with built-in support for security.

Tasks

The core of this project is to use HOL-TestGen for generating abstract test data that can be used by fwtest for testing real firewalls.

The following tasks are mandatory:

- carrying out case studies in formalizing security policies
- building a formal (abstract) model of firewalls
- applying HOL-TestGen (based on the formal model) to the case studies

In addition to the mandatory tasks described above, the student chooses to extend the work in at least one of the following directions:

- extending the formal firewall model to support content based filtering (i.e., to model application level firewalls),
- extending HOL-TestGen in a way that it generates output usable for fwtest,
- generalizing combinators for temporal descriptions of security policies,
- writing scripts that automate the overall specification, test-case generation and test driving process,
- a topic suggested by the student, discussed with and acknowledged by the supervisors.

Deliverables

- At the beginning of the thesis, an agreement must be signed which allows the supervisors of this thesis, their project partners, and ETH Zürich to use and distribute the software written during the thesis.
- At the end of the first week of the thesis, a time schedule of the semester thesis must be given and discussed with the supervisors. Regular meetings are expected to be hold between the supervisor(s) and the student.
- At the end of the semster thesis, a presentation of 20 minutes must be given[2]. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain a summary written in both English and German, this assignment and the schedule. It should include an introduction, an overview of related work, and a detailed description of the work done by the student. Five copies of the final report must be delivered to the supervisors.
- Software and configuration scripts developed during the thesis must be delivered to the supervisors on a CD-ROM.

References

[1] The HOL-TestGen website, January 2005. `http://www.brucker.ch/projects/hol-testgen/index.en.html`.

[2] Achim D. Brucker and Burkhart Wolff. Interactive testing using HOL-TestGen. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Testing of Software (FATES 05)*, LNCS. Springer-Verlag, Edinburgh, 2005. to appear.

[3] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software (FATES 04)*, LNCS 3395, pages 16–32. Springer-Verlag, Linz 04, 2005.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[5] Gerry Zaugg. Firewall testing. `http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf`.

[2] The possible dates are 07.02.06 and 10.03.06.

Bibliography

[1] H.323 and firewalls. http://vtel.com/Support/galaxy/H323_Proxies_Firewalls.html.

[2] The HOL-TestGen website, January 2006. http://www.brucker.ch/projects/hol-testgen/index.en.html.

[3] Achim D. Brucker and Burkhart Wolff. Interactive testing using HOL-TestGen. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Testing of Software (FATES 05)*, LNCS 3997, pages 87–102. Springer-Verlag, Edinburgh, 2005.

[4] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Testing of Software (FATES 04)*, LNCS 3395, pages 16–32. Springer-Verlag, Linz 04, 2005.

[5] Adel El-Atawy, Khaled Ibrahim, Hazem Hamed, and Ehab Al-Shaer. Policy segmentation for intelligent firewall testing. In *The 1st Workshop on Secure Network Protocols (NPSEC 2005)*, November 2005.

[6] Paul E. Jones. H.323 protocol overview. http://www.packetizer.com/voip/h323/paperes/h323_protocol_overview.ppt.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[8] Stephen Northcutt, Lenny Zeltser, Scott Winters, Karen Fredrick, and Ronald W. Ritchey. *Inside Network Perimeter Security: The Definitive Guide to Firewalls, Virtual Private Networks (VPNs), Routers, and Intrusion Detection Systems*. Que Corporation, pub-QUE:adr, 2002.

[9] J. Postel and J. Reynolds. File transfer protocol. RFC 959 (Standard), oct 1985. Updated by RFCs 2228, 2640, 2773.

[10] Diana Senn, David A. Basin, and Germano Caronni. Firewall conformance testing. In Ferhat Khendek and Rachida Dssouli, editors, *TestCom*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2005.

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

[12] John Wack, Ken Cutler, and Jamie Pole. Guidelines on firewalls and firewall policy. Special Publication SP 800-41, National Institute of Standards and Technology (NIST), January 2002.

[13] Wkipedia. Internet protocol suite — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Internet_protocol_suite, 2006.

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