HOL-TestGenFW

Achim D. Brucker Lukas Brügger Burkhart Wolff

March 12, 2010

Contents

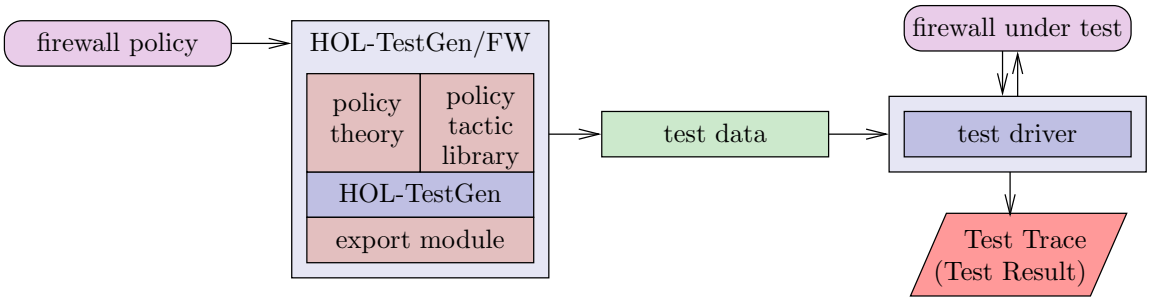

Figure 1: The HOL-TestGen/FWarchitecture.

1 Introduction

As HOL-TestGenis built on the framework of Isabelle with a general plug-in mechanism, HOL-TestGencan be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FWwhich extends HOL-TestGenby:

1. a theory (or library) formalizing networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [4].

HOL-TestGen/FWis part of the HOL-TestGendistribution. It is located in the directory `examples/hol-testgen-fw`; see [3, 2] for more details.

Figure 1 shows the overall architecture of HOL-TestGen/FW.

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see [2] and the examples provided in the dirctory `examples/hol-testgen-fw` for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

With item 2 we refer to domain-specific processing encapsulated the general HOL-TestGentest-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimized version of the procedure, now tuned for stateless firewall policies. Moreover, there are new control parameters for the simplification.

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i.e., a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [4]. This completes our toolchain which, thus,

supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

2 Preliminaries

theory
FWTesting
imports
PacketFilter/PacketFilter
FWCompilation/FWCompilationProof
StatefulFW/StatefulFW
Testing
begin

This is a formalisation in Isabelle/HOL of firewall policies and corresponding networks and packets. It first contains the formalisation of stateless packet filters as described in [2], followed by a verified policy normalisation technique (described in [1]), and a formalisation of stateful protocols described in [3].

The following statement adjusts the pre-normalization step of the test case generation algorithm. This turns out to be more efficient for the specific case of firewall policies.

setup⟪ *map-testgen-params*(*TestGen.pre-normalizeTNF-tac-update* (
fn ctxt =>
fn clasimp =>
(*TestGen.ALLCASES* (*asm-full-simp-tac* (*simpset-of*
(*ThyInfo.get-theory Int*))))))
⟫

Next, the Isar command *prepare-fw-spec-tac* is specified. It can be used to turn test specifications of the form: "$C\ x \Longrightarrow FUT\ x\ =\ policy\ x$" into the desired form for test case generation.

ML ⟪
fun prepare-fw-spec-tac ctxt =
(*TRY*((*res-inst-tac ctxt* [((*x*,*0*),*x*)] *spec 1*) *THEN*
(*resolve-tac* [*allI*] *1*) *THEN*
(*split-all-tac 1*) *THEN*
(*TRY* (*resolve-tac* [*impI*] *1*))));
⟫

method-setup *prepare-fw-spec* =
⟪

Scan.succeed (fn ctxt => SIMPLE-METHOD
(prepare-fw-spec-tac ctxt))⟩⟩ Prepares the firewall test theorem

end

3 Packets and Networks

theory *NetworkCore*
imports *Main*
begin

In networks based e.g. on TCP/IP, a message from A to B is encapsulated in *packets*, which contain the content of the message and routing information. The routing information mainly contains its source and its destination address.
In the case of stateless packet filters, a firewall bases its decision upon this routing information and, in the stateful case, on the content. Thus, we model a packet as a four-tuple of the mentioned elements, together with an id field.

The ID is just an integer:

types *id* = *int*

To enable different representations of addresses (e.g. IPv4 and IPv6, with or without ports), we model them as an unconstrained type class and directly provide several instances:

axclass *adr* < *type*
types *'α src* = *'α*::*adr*
'α dest = *'α*::*adr*

instance *int* ::*adr* **..**
instance *nat* ::*adr* **..**
instance *fun* :: (*adr*,*adr*) *adr* **..**
instance ∗ :: (*adr*,*adr*) *adr* **..**

The content is also specified with an unconstrained generic type:

types *'β content* = *'β*

For applications were the concrete representation of the content field does not matter (usually the case for stateless packet filters), we provide a default type which can be used in those cases:

datatype *DummyContent* = *data*

A packet is thus:

types $('\alpha,'\beta)$ *packet* = *id* × $('\alpha$::*adr*) *src* × $('\alpha$::*adr*) *dest* × $'\beta$ *content*

Please note that protocols (e.g. http) are not modelled explicitly. In the case of stateless packet filters, they are only visible by the destination port, which will be modelled as part of the address. Additionally, stateful firewalls will often determine the protocol by the content of a packet which is thus kept as a generic type.

Port numbers (which are part of an address) are also modelled in a generic way. The integers and the naturals are typical representations of port numbers.

axclass *port* < *type*
instance *int* ::*port* **..**
instance *nat* :: *port* **..**

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. For the sake of simplicity, we do not allow to have a different address representation format for the source and the destination of a packet respectively.

In order to access the different parts of a packet directly, we define a couple of projectors:

definition *id* :: $('\alpha,'\beta)$ *packet* ⇒ *id*
where *id* ≡ *fst*

definition *src* :: $('\alpha,'\beta)$ *packet* ⇒ $('\alpha$::*adr*) *src*
where *src* ≡ *fst* ∘ *snd*

definition *dest* :: $('\alpha,'\beta)$ *packet* ⇒ $('\alpha$::*adr*) *dest*
where *dest* ≡ *fst* ∘ *snd* ∘ *snd*

definition *content* :: $('\alpha,'\beta)$ *packet* ⇒ $'\beta$ *content*
where *content* ≡ *snd* ∘ *snd* ∘ *snd*

The following two constants give the source and destination port number of a packet. Address representations using port numbers need to provide a definition for these types.

consts *src-port* :: $('\alpha,'\beta)$ *packet* ⇒ $'\gamma$::*port*
consts *dest-port* :: $('\alpha,'\beta)$ *packet* ⇒ $'\gamma$::*port*

A subnetwork (or simply a network) is a set of sets of addresses.

types $'\alpha$ *net* = $'\alpha$::*adr set set*

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

definition
in-subnet :: *'α::adr ⇒ 'α net ⇒ bool* (**infixl** ⊏ *100*) **where**
in-subnet a S ≡ ∃ *s* ∈ *S*. *a* ∈ *s*

The following lemmas will be useful later.

lemma *in-subnet*:
(((*a*), *e*) ⊏ {{((*x1*),*y*). *P x1 y*}}) = (*P a e*)
by (*simp add*: *in-subnet-def*)

lemma *src-in-subnet*:
((*src*(*q*,((*a*),*e*),*r*,*t*)) ⊏ {{((*x1*),*y*). *P x1 y*}}) = (*P a e*)
by (*simp add*: *in-subnet-def in-subnet src-def*)

lemma *dest-in-subnet*:
((*dest* (*q*,*r*,((*a*),*e*),*t*)) ⊏ {{((*x1*),*y*). *P x1 y*}}) = (*P a e*)
by (*simp add*: *in-subnet-def in-subnet dest-def*)

Address models should provide a definition for the following constant, returning a network consisting of the input address only.

consts *subnet-of* :: *'α::adr ⇒ 'α net*

end

4 Address Representations

theory
NetworkModels
imports

DatatypeAddress
DatatypePort

IntegerAddress
IntegerPort

IPv4

begin

One can think of many different possible address representations. In this distribution, we include 5 different versions:

- DatatypeAddress: Three explicitly named addresses, which build up a network consisting of three disjunct subnetworks. I.e., there are no

overlaps and there is no way to distinguish between individual hosts within a network.

- DatatypePort: An address is a pair, with the first element being the same as above, and the second being a port number modelled as an Integer[1].
- IntegerAddress: An address in an Integer.
- IntegerPort: An address is a pair of an Integer and a port (which is again an Integer).
- IPv4: An address is a pair. The first element is a four-tuple of Integers, modelling an IPv4 address, the second element is an Integer denoting the port number.

The respective theories of the networks are relatively small. It suffices to provide the respective types, a couple of lemmas, and - if required - a definition for the source and destination ports of a packet.

end

4.1 Datatype Addresses

theory *DatatypeAddress*
imports *NetworkCore*
begin

A theory describing a network consisting of three subnetworks. Hosts within a network are not distinguished.

datatype *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

definition
dmz::*DatatypeAddress net* **where**
dmz ≡ {{*dmz-adr*}}
definition
intranet::*DatatypeAddress net* **where**
intranet ≡ {{*intranet-adr*}}
definition
internet::*DatatypeAddress net* **where**
internet ≡ {{*internet-adr*}}

end

[1]For technical reasons, we always use Integers instead of Naturals. As a consequence, the test specifications have to be adjusted to eliminate negative numbers.

4.2 Datatype Addresses with Ports

theory *DatatypePort*
imports *NetworkCore*
begin

A theory describing a network consisting of three subnetworks, including port numbers modelled as Integers. Hosts within a network are not distinguished.

datatype *DatatypeAddress* = *dmz-adr* | *intranet-adr* | *internet-adr*

types
port = *int*
DatatypePort = (*DatatypeAddress* × *port*)

instance *DatatypeAddress* :: *adr* **..**

definition
dmz::*DatatypePort net* **where**
dmz ≡ {{(*a*,*b*). *a* = *dmz-adr*}}
definition
intranet::*DatatypePort net* **where**
intranet ≡ {{(*a*,*b*). *a* = *intranet-adr*}}
definition
internet::*DatatypePort net* **where**
internet ≡ {{(*a*,*b*). *a* = *internet-adr*}}

defs (**overloaded**)
src-port-def: *src-port* (*x*::(*DatatypePort*,'*β*) *packet*) ≡ (*snd o fst o snd*) *x*
dest-port-def: *dest-port* (*x*::(*DatatypePort*,'*β*) *packet*) ≡ (*snd o fst o snd o snd*) *x*
subnet-of-def: *subnet-of* (*x*::*DatatypePort*) ≡ {{(*a*,*b*). *a* = *fst x*}}

lemma *src-port* : *src-port* ((*a*,*x*,*d*,*e*)::(*DatatypePort*,'*β*) *packet*) = *snd x*
by (*simp add*: *src-port-def in-subnet*)

lemma *dest-port* : *dest-port* ((*a*,*d*,*x*,*e*)::(*DatatypePort*,'*β*) *packet*) = *snd x*
by (*simp add*: *dest-port-def in-subnet*)

lemmas *DatatypePortLemmas* = *src-port dest-port src-port-def dest-port-def*
end

4.3 Integer Addresses

theory *IntegerAddress*
imports *NetworkCore*
begin

A theory where addresses are modelled as Integers.

types
IntegerAddress = *int*

end

4.4 Integer Addresses with Ports

theory *IntegerPort*
imports *NetworkCore*
begin

A theory describing addresses which are modelled as a pair of Integers - the first being the host address, the second the port number.

types
address = *int*
port = *int*
IntegerPort = *address* × *port*

defs (**overloaded**)
src-port-def: *src-port* (*x*::(*IntegerPort*,'*β*) *packet*) ≡ (*snd o fst o snd*) *x*
dest-port-def: *dest-port* (*x*::(*IntegerPort*,'*β*) *packet*) ≡ (*snd o fst o snd o snd*) *x*
subnet-of-def: *subnet-of* (*x*::(*IntegerPort*)) ≡ {{(*a*,*b*). *a* = *fst x*}}

lemma *src-port*: *src-port* (*a*,*x*::*IntegerPort*,*d*,*e*) = *snd x*
by (*simp add*: *src-port-def in-subnet*)

lemma *dest-port*: *dest-port* (*a*,*d*,*x*::*IntegerPort*,*e*) = *snd x*
by (*simp add*: *dest-port-def in-subnet*)

lemmas *IntegerPortLemmas* = *src-port dest-port src-port-def dest-port-def*

end

4.5 IPv4 Addresses

theory *IPv4*
imports *NetworkCore*
begin

A theory describing IPv4 addresses with ports. The host address is a four-tuple of Integers, the port number is a single Integer.

types
ipv4-ip = (*int* × *int* × *int* × *int*)
port = *int*
ipv4 = (*ipv4-ip* × *port*)

defs (**overloaded**)
src-port-def: *src-port* (*x*::(*ipv4*,'*b*) *packet*) ≡ (*snd o fst o snd*) *x*
defs (**overloaded**)
dest-port-def:*dest-port* (*x*::(*ipv4*,'*b*) *packet*) ≡ (*snd o fst o snd o snd*) *x*
defs (**overloaded**)
subnet-of-def: *subnet-of* (*x*::*ipv4*) ≡ {{(*a*,*b*). *a* = *fst x*}}

definition *subnet-of-ip* :: *ipv4-ip* ⇒ *ipv4 net*
where *subnet-of-ip ip* ≡ {{(*a*,*b*). (*a* = *ip*)}}

lemma *src-port*: *src-port* (*a*,(*x*::*ipv4*),*d*,*e*) = *snd x*
by (*simp add*: *src-port-def in-subnet*)

lemma *dest-port*: *dest-port* (*a*,*d*,(*x*::*ipv4*),*e*) = *snd x*
by (*simp add*: *dest-port-def in-subnet*)

lemmas *IPv4Lemmas* = *src-port dest-port src-port-def dest-port-def*

end

5 Policies

5.1 Policy Core

theory *PolicyCore*
imports *NetworkCore*
begin

Next, we define the concept of a policy. From an abstract point of view,

a policy is a partial mapping of packets to decisions. Thus, we model the decision as a datatype.

datatype $'\alpha$ *out* = *accept* $'\alpha$ | *deny* $'\alpha$

A policy is seen as a partial mapping from packet to packet out.

types ($'\alpha$, $'\beta$) *Policy* = ($'\alpha$, $'\beta$) *packet* $\rightharpoonup$ (($'\alpha$, $'\beta$) *packet*) *out*

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.

definition
p-accept :: ($'\alpha$, $'\beta$) *packet* $\Rightarrow$ ($'\alpha$, $'\beta$) *Policy* $\Rightarrow$ *bool* **where**
p-accept p policy $\equiv$ *policy p* = *Some* (*accept p*)

end

5.2 Policy Combinators

theory *PolicyCombinators*
imports
PolicyCore
begin

In order to ease the specification of a concrete policy, we define some combinators. Using these combinators, the specification of a policy gets very easy, and can be done similarly as in tools like IPTables.

definition
allow-all :: ($'\alpha$, $'\beta$) *Policy* **where**
allow-all p $\equiv$ *Some* (*accept p*)

definition
deny-all :: ($'\alpha$,$'\beta$) *Policy* **where**
deny-all p $\equiv$ *Some* (*deny p*)

definition
allow-all-from :: ($'\alpha$::*adr*) *net* $\Rightarrow$ ($'\alpha$,$'\beta$) *Policy* **where**
allow-all-from src-net $\equiv$ *allow-all* |` {*pa*. *src pa* $\sqsubset$ *src-net*}

definition
deny-all-from :: ('α::*adr*) *net* ⇒ ('α,'β) *Policy* **where**
deny-all-from src-net ≡ *deny-all* |‘ {*pa*. *src pa* ⊏ *src-net*}

definition
allow-all-to :: ('α::*adr*) *net* ⇒ ('α,'β) *Policy* **where**
allow-all-to dest-net ≡ *allow-all* |‘ {*pa*. *dest pa* ⊏ *dest-net*}

definition
deny-all-to :: ('α::*adr*) *net* ⇒ ('α,'β) *Policy* **where**
deny-all-to dest-net ≡ *deny-all* |‘ {*pa*. *dest pa* ⊏ *dest-net*}

definition
allow-all-from-to :: ('α::*adr*) *net* ⇒ ('α::*adr*) *net* ⇒ ('α,'β) *Policy* **where**
allow-all-from-to src-net dest-net ≡ *allow-all* |‘ {*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net*}

definition
deny-all-from-to :: ('α::*adr*) *net* ⇒ ('α::*adr*) *net* ⇒ ('α,'β) *Policy* **where**
deny-all-from-to src-net dest-net ≡ *deny-all* |‘ {*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net*}

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

lemmas *PolicyCombinators* =
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
map-add-def restrict-map-def

end

5.3 Policy Combinators with Ports

theory *PortCombinators*
imports *PolicyCombinators*
begin

This theory defines policy combinators for those network models which have ports. They are provided in addition to the the ones defined in the Policy-Combinators theory.

This theory requires from the network models a definition for the two following constants:

- *src_port* :: ('α,' β)*packet* ⇒ ('γ :: *port*)

- *dest_port* :: $('\alpha,'\beta)packet \Rightarrow ('\gamma :: port)$

definition
allow-all-from-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-from-port src-net s-port $\equiv$ *allow-all-from src-net* |‘ {*pa*. *src-port pa* = *s-port*}

definition
deny-all-from-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-from-port src-net s-port $\equiv$ *deny-all-from src-net* |‘ {*pa*. *src-port pa* = *s-port*}

definition
allow-all-to-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-to-port dest-net d-port $\equiv$ *allow-all-to dest-net* |‘ {*pa*. *dest-port pa* = *d-port*}

definition
deny-all-to-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-to-port dest-net d-port $\equiv$ *deny-all-to dest-net*|‘ {*pa*. *dest-port pa* = *d-port*}

definition
allow-all-from-port-to :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha$::*adr*) *net* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-from-port-to src-net s-port dest-net
$\equiv$ *allow-all-from-to src-net dest-net* |‘ {*pa*. *src-port pa* = *s-port*}

definition
deny-all-from-port-to :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha$::*adr*) *net* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-from-port-to src-net s-port dest-net
$\equiv$ *deny-all-from-to src-net dest-net*|‘ {*pa*. *src-port pa* = *s-port*}

definition
allow-all-from-port-to-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
allow-all-from-port-to-port src-net s-port dest-net d-port $\equiv$
allow-all-from-port-to src-net s-port dest-net |‘ {*pa*. *dest-port pa* = *d-port*}

definition
deny-all-from-port-to-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**
deny-all-from-port-to-port src-net s-port dest-net d-port $\equiv$
deny-all-from-port-to src-net s-port dest-net|‘ {*pa*. *dest-port pa* = *d-port*}

definition
allow-all-from-to-port :: $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha$::*adr*) *net* $\Rightarrow$ $'\gamma$::*port* $\Rightarrow$ $('\alpha, '\beta)$ *Policy* **where**

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*}

definition
deny-all-from-to-port :: (′α::*adr*) *net* ⇒ ′γ::*port* ⇒ (′α::*adr*) *net* ⇒ ′γ::*port* ⇒ (′α, ′β) *Policy* **where**
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*}

definition
allow-from-port-to :: ′γ::*port* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
allow-from-port-to port src-net dest-net ≡ *allow-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*src-port pa* = *port*)}

definition
deny-from-port-to :: ′γ::*port* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
deny-from-port-to port src-net dest-net ≡ *deny-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*src-port pa* = *port*)}

definition
allow-from-to-port :: ′γ::*port* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
allow-from-to-port port src-net dest-net ≡ *allow-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*dest-port pa* = *port*)}

definition
deny-from-to-port :: ′γ::*port* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
deny-from-to-port port src-net dest-net ≡ *deny-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*dest-port pa* = *port*)}

definition
allow-from-ports-to :: ′γ::*port set* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
allow-from-ports-to ports src-net dest-net ≡ *allow-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*src-port pa* ∈ *ports*)}

definition
allow-from-to-ports :: ′γ::*port set* ⇒ (′α::*adr*) *net* ⇒ (′α::*adr*) *net* ⇒ (′α,′β) *Policy* **where**
allow-from-to-ports ports src-net dest-net ≡ *allow-all* |‘
{*pa*. *src pa* ⊏ *src-net* ∧ *dest pa* ⊏ *dest-net* ∧ (*dest-port pa* ∈ *ports*)}

As before, we put all the rules into one lemma called PortCombinators to

ease writing later.

lemmas *PortCombinators* =
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-from-ports-to-def allow-from-to-ports-def
allow-all-from-port-to-def deny-all-from-port-to-def
allow-from-port-to-def allow-from-to-port-def deny-from-to-port-def
deny-from-port-to-def
end

5.4 Ports

theory *Ports*
imports *Main*
begin

This theory can be used if we want to specify the port numbers by names denoting their default integer values. If you want to use them, please add "Ports" to the simplifier before test data generation.

definition *http*::*int* **where** *http* ≡ *80*
lemma *http1*: *x* ≠ *80* ⟹ *x* ≠ *http*
by (*simp add*: *http-def*)

lemma *http2*: *x* ≠ *80* ⟹ *http* ≠ *x*
by (*simp add*: *http-def*)

definition *smtp*::*int* **where** *smtp* ≡ *25*
lemma *smtp1*: *x* ≠ *25* ⟹ *x* ≠ *smtp*
by (*simp add*: *smtp-def*)

lemma *smtp2*: *x* ≠ *25* ⟹ *smtp* ≠ *x*
by (*simp add*: *smtp-def*)

definition *ftp*::*int* **where** *ftp* ≡ *21*
lemma *ftp1*: *x* ≠ *21* ⟹ *x* ≠ *ftp*
by (*simp add*: *ftp-def*)

lemma *ftp2*: *x* ≠ *21* ⟹ *ftp* ≠ *x*
by (*simp add*: *ftp-def*)

And so on for all desired port numbers.

lemmas *Ports = http1 http2 ftp1 ftp2 smtp1 smtp2*

end

6 Policy Normalisation

theory
FWCompilation
imports
../PacketFilter/PacketFilter
Testing
begin

This theory contains all the definitions used for policy normalisation as described in [1]. Policy transformations are functions that map policies to policies. We decided to represent policy transformations as *syntactic rules*; this choice paves the way for expressing the entire normalisation process inside HOL by functions manipulating abstract policy syntax.

6.1 Basics

We define a very simple policy language:

datatype $('\alpha,'\beta)$ *Combinators* =
DenyAll
| *DenyAllFromTo* $'\alpha$ $'\alpha$
| *AllowPortFromTo* $'\alpha$ $'\alpha$ $'\beta$
| *Conc* $(('\alpha,'\beta)$ *Combinators*$)$ $(('\alpha,'\beta)$ *Combinators*$)$ (**infixr** $\oplus$ *80*)

And define the semantic interpretation of it. For technical reasons, we fix here the type to policies on IntegerPort addresses. However, we could easily provide definitions for other address types as well, using a generic consts for the type definition and a primrec definition for each desired address model.

fun *C* :: (*IntegerPort net*, *port*) *Combinators* $\Rightarrow$ (*IntegerPort*, *DummyContent*) *Policy*
where
C DenyAll = *deny-all*
|*C* (*DenyAllFromTo x y*) = *deny-all-from-to x y*
|*C* (*AllowPortFromTo x y p*) = *allow-from-to-port p x y*
|*C* $(x \oplus y)$ = *C x* ++ *C y*

6.2 Auxiliary definitions and functions.

This subsection defines several functions which are useful later for the combinators, invariants, and proofs.

fun *position* :: $'\alpha \Rightarrow '\alpha$ *list* $\Rightarrow$ *nat* **where**
position a [] = *0*
| (*position a* (*x*#*xs*)) = (*if a* = *x then 1 else* (*Suc* (*position a xs*)))

fun *srcNet* **where**
srcNet (*DenyAllFromTo x y*) = *x*
|*srcNet* (*AllowPortFromTo x y p*) = *x*

fun *destNet* **where**
destNet (*DenyAllFromTo x y*) = *y*
|*destNet* (*AllowPortFromTo x y p*) = *y*

fun *srcnets*::(*IntegerPort net*,*port*) *Combinators* $\Rightarrow$ (*IntegerPort net*) *list* **where**
srcnets DenyAll = []
|*srcnets* (*DenyAllFromTo x y*) = [*x*]
|*srcnets* (*AllowPortFromTo x y p*) = [*x*]
|(*srcnets* (*x* $\oplus$ *y*)) = (*srcnets x*)@(*srcnets y*)

fun *destnets*::(*IntegerPort net*,*port*) *Combinators* $\Rightarrow$ (*IntegerPort net*) *list* **where**
destnets DenyAll = []
|*destnets* (*DenyAllFromTo x y*) = [*y*]
|*destnets* (*AllowPortFromTo x y p*) = [*y*]
|(*destnets* (*x* $\oplus$ *y*)) = (*destnets x*)@(*destnets y*)

fun (*sequential*) *net-list-aux* **where**
net-list-aux [] = []
|*net-list-aux* (*DenyAll*#*xs*) = *net-list-aux xs*
|*net-list-aux* ((*DenyAllFromTo x y*)#*xs*) = *x*#*y*#(*net-list-aux xs*)
|*net-list-aux* ((*AllowPortFromTo x y p*)#*xs*) = *x*#*y*#(*net-list-aux xs*)
|*net-list-aux* ((*x*⊕*y*)#*xs*) = (*net-list-aux* [*x*])@(*net-list-aux* [*y*])@(*net-list-aux xs*)

fun *net-list* **where** *net-list p* = *remdups* (*net-list-aux p*)

definition *bothNets* **where** *bothNets x* = (*zip* (*srcnets x*) (*destnets x*))

fun (*sequential*) *normBothNets* **where**
normBothNets ((*a*,*b*)#*xs*) = (*if* ((*b*,*a*) ∈ *set xs*) ∨ (*a*,*b*) ∈ *set* (*xs*) *then* (*normBothNets xs*) *else* (*a*,*b*)#(*normBothNets xs*))
|*normBothNets x* = *x*

fun *makeSets* **where**
makeSets ((*a*,*b*)#*xs*) = ({*a*,*b*}#(*makeSets xs*))
|*makeSets* [] = []

fun *bothNet* **where**

bothNet DenyAll = {}
|*bothNet* (*DenyAllFromTo a b*) = {*a*,*b*}
|*bothNet* (*AllowPortFromTo a b p*) = {*a*,*b*}

Nets_List provides from a list of rules a list where the entries are the appearing sets of source and destination network of each rule.

definition *Nets-List* **where** *Nets-List x* = *makeSets* (*normBothNets* (*bothNets x*))

fun (*sequential*) *first-srcNet* **where**
first-srcNet (*x*⊕*y*) = *first-srcNet x*
| *first-srcNet x* = *srcNet x*

fun (*sequential*) *first-destNet* **where**
first-destNet (*x*⊕*y*) = *first-destNet x*
| *first-destNet x* = *destNet x*

fun (*sequential*) *first-bothNet* **where**
first-bothNet (*x*⊕*y*) = *first-bothNet x*
|*first-bothNet x* = *bothNet x*

fun (*sequential*) *in-list* **where**
in-list DenyAll l = *True*
|*in-list x l* = (*bothNet x* ∈ *set l*)

fun *all-in-list* **where**
all-in-list [] *l* = *True*
|*all-in-list* (*x*#*xs*) *l* = (*in-list x l* ∧ *all-in-list xs l*)

fun (*sequential*) *member* **where**
member a (*x*⊕*xs*) = ((*member a x*) ∨ (*member a xs*))
|*member a x* = (*a* = *x*)

fun *noneMT* **where**
noneMT (*x*#*xs*) = (*dom* (*C x*) ≠ {} ∧ (*noneMT xs*))
|*noneMT* [] = *True*

fun *notMTpolicy* **where**
notMTpolicy (*x*#*xs*) = (*if* (*dom* (*C x*) = {}) *then* (*notMTpolicy xs*) *else True*)
|*notMTpolicy* [] = *False*

fun *sdnets* **where**
sdnets DenyAll = {}
| *sdnets* (*DenyAllFromTo a b*) = {(*a*,*b*)}
| *sdnets* (*AllowPortFromTo a b c*) = {(*a*,*b*)}
| *sdnets* (*a* ⊕ *b*) = *sdnets a* ∪ *sdnets b*

definition *packet-Nets* **where** *packet-Nets x a b* ≡ (*src x* ⊏ *a* ∧ *dest x* ⊏ *b*) ∨ (*src x* ⊏ *b* ∧ *dest x* ⊏ *a*)

fun *matching-rule-rev* **where**
matching-rule-rev a (*x*#*xs*) = (*if a* ∈ *dom* (*C x*) *then* (*Some x*) *else* (*matching-rule-rev a xs*))
|*matching-rule-rev a* [] = *None*

Provides the first matching rule of a policy given as a list of rules.

definition *matching-rule* **where**
matching-rule a x ≡ (*matching-rule-rev a* (*rev x*))

definition *subnetsOfAdr* **where** *subnetsOfAdr a* ≡ {*x*. *a* ⊏ *x*}

definition *fst-set* **where** *fst-set s* ≡ {*a*. ∃ *b*. (*a*,*b*) ∈ *s*}

definition *snd-set* **where** *snd-set s* ≡ {*a*. ∃ *b*. (*b*,*a*) ∈ *s*}

fun *memberP* **where**
memberP r (*x*#*xs*) = (*member r x* ∨ *memberP r xs*)
|*memberP r* [] = *False*

fun *firstList* **where**
firstList (*x*#*xs*) = (*first-bothNet x*)
|*firstList* [] = {}

6.3 Invariants

If there is a DenyAll, it is at the first position

fun *wellformed-policy1*:: ((*IntegerPort net*, *port*) *Combinators*) *list* ⇒ *bool* **where**

wellformed-policy1 [] = *True*
| *wellformed-policy1* (*x*#*xs*) = (*DenyAll* ∉ (*set xs*))

There is a DenyAll at the first position

fun *wellformed-policy1-strong*:: ((*IntegerPort net*, *port*) *Combinators*) *list* ⇒ *bool* **where**
wellformed-policy1-strong [] = *False*
| *wellformed-policy1-strong* (*x*#*xs*) = (*x*=*DenyAll* ∧ (*DenyAll* ∉ (*set xs*)))

All rules appearing at the left of a DenyAllFromTo, have disjunct domains from it (except DenyAll)

fun (*sequential*) *wellformed-policy2* **where**
wellformed-policy2 [] = *True*
| *wellformed-policy2* (*DenyAll*#*xs*) = *wellformed-policy2 xs*
| *wellformed-policy2* (*x*#*xs*) = ((∀ *c a b*. *c* = *DenyAllFromTo a b* ∧ *c* ∈ *set xs* ⟶ *Map.dom* (*C x*) ∩ *Map.dom* (*C c*) = {}) ∧ *wellformed-policy2 xs*)

An allow rule is disjunct with all rules appearing at the right of it. This invariant is not necessary as it is a consequence from others, but facilitates some proofs.

fun (*sequential*) *wellformed-policy3* **where**
wellformed-policy3 [] = *True*
| *wellformed-policy3* ((*AllowPortFromTo a b p*)#*xs*) = ((∀ *r*. *r* ∈ *set xs* ⟶ *dom* (*C r*) ∩ *dom* (*C* (*AllowPortFromTo a b p*)) = {}) ∧ *wellformed-policy3 xs*)
| *wellformed-policy3* (*x*#*xs*) = *wellformed-policy3 xs*

All two networks are either disjoint or equal.

definition *netsDistinct* **where** *netsDistinct a b* ≡ ¬ (∃ *x*. *x* ⊏ *a* ∧ *x* ⊏ *b*)

definition *twoNetsDistinct* **where** *twoNetsDistinct a b c d* ≡ *netsDistinct a c* ∨ *netsDistinct b d*

definition *allNetsDistinct* **where** *allNetsDistinct p* ≡ ∀ *a b*. (*a* ≠ *b* ∧ *a* ∈ *set* (*net-list p*) ∧ *b* ∈ *set* (*net-list p*)) ⟶ *netsDistinct a b*

definition *disjSD-2* **where**
disjSD-2 x y ≡ ∀ *a b c d*. ((*a*,*b*)∈*sdnets x* ∧ (*c*,*d*) ∈*sdnets y* ⟶ (*twoNetsDistinct a b c d* ∧ *twoNetsDistinct a b d c*))

The policy is given as a list of single rules.

fun *singleCombinators* **where**
singleCombinators [] = *True*
|*singleCombinators* ((*x*⊕*y*)#*xs*) = *False*
|*singleCombinators* (*x*#*xs*) = *singleCombinators xs*

definition *onlyTwoNets* **where**
onlyTwoNets x ≡ ((∃ *a b*. (*sdnets x* = {(*a*,*b*)})) ∨ (∃ *a b*. *sdnets x* = {(*a*,*b*),(*b*,*a*)}))

Each entry of the list contains rules between two networks only.

fun *OnlyTwoNets* **where**
OnlyTwoNets (*DenyAll*#*xs*) = *OnlyTwoNets xs*
|*OnlyTwoNets* (*x*#*xs*) = (*onlyTwoNets x* ∧ *OnlyTwoNets xs*)
|*OnlyTwoNets* [] = *True*

fun *noDenyAll* **where**
noDenyAll (*x*#*xs*) = ((¬ *member DenyAll x*) ∧ *noDenyAll xs*)
|*noDenyAll* [] = *True*

fun *noDenyAll1* **where**
noDenyAll1 (*DenyAll*#*xs*) = *noDenyAll xs*
| *noDenyAll1 xs* = *noDenyAll xs*

fun *separated* **where**
separated (*x*#*xs*) = ((∀ *s*. *s* ∈ *set xs* ⟶ *disjSD-2 x s*) ∧ *separated xs*)
| *separated* [] = *True*

fun *NetsCollected* **where**
NetsCollected (*x*#*xs*) = (((*first-bothNet x* ≠ *firstList xs*) ⟶ (∀ *a*∈*set xs*. *first-bothNet x* ≠ *first-bothNet a*)) ∧ *NetsCollected* (*xs*))

| *NetsCollected* [] = *True*

fun *NetsCollected2* **where**
NetsCollected2 (*x*#*xs*) = (*xs* = [] ∨ (*first-bothNet x* ≠ *firstList xs* ∧ *NetsCollected2 xs*))
|*NetsCollected2* [] = *True*

6.4 Transformations

The following two functions transform a policy into a list of single rules and vice-versa.

fun *policy2list*::(*IntegerPort net*, *port*) *Combinators* ⇒ ((*IntegerPort net*, *port*) *Combinators*) *list* **where**
policy2list (*x* ⊕ *y*) = (*concat* [(*policy2list x*),(*policy2list y*)])
|*policy2list x* = [*x*]

fun *list2policy*::((*IntegerPort net*, *port*) *Combinators*) *list* ⇒ ((*IntegerPort net*, *port*) *Combinators*) **where**
list2policy (*x*#[]) = *x*
| *list2policy* (*x*#*y*) = *x* ⊕ (*list2policy y*)

Remove all the rules appearing before a DenyAll. There are two alternative versions.

fun *removeShadowRules1* **where**
removeShadowRules1 (*x*#*xs*) = (*if* (*DenyAll* ∈ *set xs*) *then* ((*removeShadowRules1 xs*)) *else x*#*xs*)
| *removeShadowRules1* [] = []

fun *removeShadowRules1-alternative-rev* **where**
removeShadowRules1-alternative-rev [] = []
| *removeShadowRules1-alternative-rev* (*DenyAll*#*xs*) = [*DenyAll*]
| *removeShadowRules1-alternative-rev* [*x*] = [*x*]
| *removeShadowRules1-alternative-rev* (*x*#*xs*)= *x*#(*removeShadowRules1-alternative-rev xs*)

definition *removeShadowRules1-alternative* **where** *removeShadowRules1-alternative p* = *rev* (*removeShadowRules1-alternative-rev* (*rev p*))

Remove all the rules which allow a port, but are shadowed by a deny between these subnets

fun *removeShadowRules2*::
((*IntegerPort net*, *port*) *Combinators*) *list* ⇒ ((*IntegerPort net*, *port*) *Combinators*) *list*
where
(*removeShadowRules2* ((*AllowPortFromTo x y p*)#*z*)) =
(*if* (((*DenyAllFromTo x y*) ∈ *set z*)) *then* ((*removeShadowRules2 z*)) *else* (((*AllowPortFromTo x y p*)#(*removeShadowRules2 z*))))
| *removeShadowRules2* (*x*#*y*) = *x*#(*removeShadowRules2 y*)

```
| removeShadowRules2 [] = []
```

Sorting a pocliy. We first need to define an ordering on rules. This ordering depends on the *Nets_List* of a policy.

```
fun smaller :: (IntegerPort net, port) Combinators ⇒
                        (IntegerPort net, port) Combinators ⇒
                        ((IntegerPort net) set) list ⇒ bool
where
 smaller DenyAll x l = True
| smaller x DenyAll l = False
| smaller x y l =
 ((x = y) ∨          (if (bothNet x) = (bothNet y) then
                    (case y of (DenyAllFromTo a b) ⇒ (x = DenyAllFromTo b a)
                         | - ⇒  True)
                  else
                     (position (bothNet x) l <= position (bothNet y) l)))
```

We use insertion sort for sorting a policy.

```
fun insort where
  insort a [] l = [a]
| insort a (x#xs) l = (if (smaller a x l) then a#x#xs else x#(insort a xs l))

fun sort where
  sort [] l = []
| sort (x#xs) l = insort x (sort xs l) l

fun sorted  where
sorted [] l ⟷ True |
sorted [x] l ⟷ True |
sorted (x#y#zs) l ⟷ smaller x y l ∧ sorted (y#zs) l
```

separate works on a sorted policy: it joins the rules which talk about the traffic between the same two networks.

```
fun separate where
 separate (DenyAll#x) = DenyAll#(separate x)
| separate (x#y#z) = (if (first-bothNet x = first-bothNet y)
                  then (separate ((x⊕y)#z))
                  else (x#(separate(y#z))))
|separate x = x
```

Insert the DenyAllFromTo rules, such that traffic between two networks can be tested individually

```
fun  insertDenies where
 insertDenies (x#xs) = (case x of DenyAll ⇒ (DenyAll#(insertDenies xs))
                      | - ⇒ (DenyAllFromTo (first-srcNet x) (first-destNet x) ⊕
                              (DenyAllFromTo (first-destNet x) (first-srcNet x)) ⊕
x)#
                                 (insertDenies xs))
```

| *insertDenies* [] = []

Remove duplicate rules. This is especially necessary as insertDenies might have inserted duplicate rules.

The second function is supposed to work on a list of policies. Only rules which are duplicated within the same policy, are removed.

fun *removeDuplicates* **where**
removeDuplicates (*x*⊕*xs*) = (*if member x xs then* (*removeDuplicates xs*) *else x*⊕(*removeDuplicates xs*))
| *removeDuplicates x* = *x*

fun *removeAllDuplicates* **where**
removeAllDuplicates (*x*#*xs*) = ((*removeDuplicates* (*x*))#(*removeAllDuplicates xs*))
|*removeAllDuplicates x* = *x*

Remove rules with an empty domain - they never match any packet.

fun *removeShadowRules3* **where**
removeShadowRules3 (*x*#*xs*) = (*if* (*dom* (*C x*) = {}) *then* (*removeShadowRules3 xs*) *else* (*x*#(*removeShadowRules3 xs*)))
|*removeShadowRules3* [] = []

Insert a DenyAll at the beginning of a policy.

fun *insertDeny* **where**
insertDeny (*DenyAll*#*xs*) = *DenyAll*#*xs*
|*insertDeny xs* = *DenyAll*#*xs*

Now do everything:

definition *sort' p l* ≡ *sort l p*

definition *normalize' p* ≡ (*removeAllDuplicates o insertDenies o separate o* (*sort'* (*Nets-List p*)) *o removeShadowRules2 o remdups o removeShadowRules3 o insertDeny o removeShadowRules1 o policy2list*) *p*

definition *normalize p* ≡ *removeAllDuplicates* (*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) ((*Nets-List p*)))))

definition *normalize-manual-order p l* ≡ *removeAllDuplicates* (*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) ((*l*)))))

Of course, normalize is equal to normalize', the latter looks nicer though.

lemma *normalize* = *normalize'*
by (*rule ext*, *simp add*: *normalize-def normalize'-def sort'-def*)

The following definition helps in creating the test specification for the individual parts of a normalized policy.

definition *makeFUT* **where** *makeFUT FUT p x n* =
(*packet-Nets x* (*fst*(((*normBothNets* (*bothNets p*)))!*n*)) (*snd*(((*normBothNets* (*bothNets p*)))!*n*)) ⟶ *FUT x* = *C* ((*normalize p*)!(*n*+*1*)) *x*)

declare *C.simps* [*simp del*]

lemmas *PLemmas* = *C.simps dom-def PolicyCombinators.PolicyCombinators PortCombinators.PortCombinators src-def dest-def in-subnet-def*
IntegerPort.src-port-def IntegerPort.dest-port-def

end

theory *FWCompilationProof*
imports *FWCompilation*
begin

This theory contains the complete proofs of the normalisation procedure. In the generated PDF document, we refrain from showing the complete proofs. Rather only the final theorem is provided.

lemma *C-eq-compile-manual*:
⟦*DenyAll* ∈ *set* (*policy2list p*); *all-in-list* (*policy2list p*) *l*; *allNetsDistinct* (*policy2list p*)⟧
⟹
C (*list2policy* (*removeAllDuplicates* (*insertDenies* (*separate* (*sort* (*removeShadowRules2* (*remdups* (*removeShadowRules3* (*insertDeny* (*removeShadowRules1* (*policy2list p*)))))) *l*))))) = *C p*
apply (*subst C-eq-RAD*[*symmetric*])
apply (*rule idNMT*)
apply (*simp add*: *C-eqLemmas-id*)
apply (*rule C-eq-Until-InsertDenies*)
apply *simp-all*
done

end

7 Stateful Firewalls

7.1 Basic Constructs

theory *Stateful*
imports *../PacketFilter/PacketFilter Testing*
begin

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. We next model this behaviour.

The key point of our model is the idea that a policy remains the same as before: a mapping from packet to packet out. We still specify for every packet, based on its source and destination address, 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 $('\alpha,'\beta,'\gamma)$ *FWState* $= '\alpha \times ('\beta,'\gamma)$ *Policy*

Having a state, we need of course some state transitions. Such a transition can happen every time a new packet arrives. State transitions can be modelled using a state-exception monad.

types $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $= ('\beta,'\gamma)$ *packet* $\Rightarrow$ (*unit*, $('\alpha,'\beta,'\gamma)$ *FWState*) *MON-SE*

The memory could be modelled as a list of accepted packets.

types $('\beta,'\gamma)$ *history* $= ('\beta,'\gamma)$ *packet list*

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* :: $('\alpha \Rightarrow bool) \Rightarrow ('\alpha \Rightarrow bool) \Rightarrow '\alpha$ *list* $\Rightarrow$ *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* :: $('\alpha \Rightarrow bool) \Rightarrow ('\alpha \Rightarrow bool) \Rightarrow '\alpha\ list \Rightarrow bool$
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. It takes the first transition which maps to *Some* $'\alpha$.

definition *orelse*:: $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $\Rightarrow$ $('\alpha,'\beta,'\gamma)$ *FWStateTransition* $\Rightarrow$ $('\alpha,'\beta,'\gamma)$ *FWStateTransition* (**infixl** *orelse 100*) **where**
(f orelse g) x ≡ λ σ. (case f x σ of None ⇒ g x σ | Some y ⇒ Some y)

end

7.2 FTP Protocol

theory *FTP*
imports
Stateful
begin

The File Transfer Protocol FTP 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 IntegerPort addresses, still containing all messages that matter for our purposes. It consists of the following four messages:

1. *ftp-init*: The client contacts the server indicating his wish to get some data.
2. *ftp-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. *ftp-close*: The client closes the connection. The dynamic port gets closed again.

The content field of a packet therefore now consists of either one of those four messages or a default one.

datatype *ftp-msg* = *ftp-init*
| *ftp-port-request port*
| *ftp-data*
| *ftp-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.

definition
is-init :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-init i p ≡ *id p* = *i* ∧ *content p* = *ftp-init*

definition
is-port-request :: *id* ⇒ *port* ⇒(*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-port-request i port p ≡ *id p* = *i* ∧ *content p* = *ftp-port-request port*

definition
is-data :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-data i p ≡ *id p* = *i* ∧ *content p* = *ftp-data*

definition
is-close :: *id* ⇒ (*IntegerPort*, *ftp-msg*) *packet* ⇒ *bool* **where**
is-close i p ≡ *id p* = *i* ∧ *content p* = *ftp-close*

definition
port-open :: (*IntegerPort*, *ftp-msg*) *history* ⇒ *id* ⇒ *port* ⇒ *bool* **where**
port-open L a p ≡ *not-before* (*is-close a*) (*is-port-request a p*) *L*

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 cases, one is enough. In our 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.

fun *FTP-ST* :: ((*IntegerPort*,*ftp-msg*) *history*, *IntegerPort*, *ftp-msg*) *FWStateTransition*
where

FTP-ST (*i*,*s*,*d*,*ftp-port-request pr*) (*InL*, *policy*) = (*if p-accept* (*i*,*s*,*d*,*ftp-port-request pr*) *policy then*
(*if not-before* (*is-close i*) (*is-init i*) *InL* ∧ *dest-port* (*i*,*s*,*d*,*ftp-port-request pr*) = (*21*::*port*) *then*
Some ((),((*i*,*s*,*d*,*ftp-port-request pr*)#*InL*, *policy* ++
(*allow-from-to-port pr* (*subnet-of d*) (*subnet-of s*))))
else Some ((),((*i*,*s*,*d*,*ftp-port-request pr*)#*InL*,*policy*)))
else Some ((),(*InL*,*policy*)))

|*FTP-ST* (*i*,*s*,*d*,*ftp-close*) (*InL*,*policy*) =
(*if* (*p-accept* (*i*,*s*,*d*,*ftp-close*) *policy*) *then*
(*if* (∃ *p*. *port-open InL i p*) ∧ *dest-port* (*i*,*s*,*d*,*ftp-close*) = (*21*::*port*) *then*
Some((),((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy* ++
deny-from-to-port (*Eps* (λ *p*. *port-open InL i p*)) (*subnet-of d*) (*subnet-of s*)))
else Some ((),((*i*,*s*,*d*,*ftp-close*)#*InL*, *policy*)))
else Some ((),(*InL*,*policy*)))

|*FTP-ST p* (*InL*,*policy*) = (*if p-accept p policy then*
Some ((),(*p*#*InL*,*policy*))
else
Some ((),(*InL*,*policy*)))

The second message of th protocol 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.

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. 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. 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.

Now we specify our test scenario in more detail. 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 here: one correct protocol run.

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.

consts
is-ftp :: ftp-states ⇒ IntegerPort ⇒ IntegerPort ⇒ id ⇒ port ⇒ (IntegerPort,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,sr,de,co). (((id = i ∧ (
(H=S2 ∧ sr = c ∧ de = s ∧ co = ftp-init ∧ is-ftp S3 c s i p InL) ∨
(H=S1 ∧ sr = c ∧ de = s ∧ co = ftp-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 = ftp-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.

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.

definition

NB-ftp :: *IntegerPort src* ⇒ *IntegerPort dest* ⇒ *id* ⇒ *port* ⇒ (*IntegerPort*,*ftp-msg*) *history set* **where**
NB-ftp s d i p ≡ {*x*. (*is-ftp S0 s d i p x*)}

Contrary to the case of a stateless packet filter, a lot of the proof work will only be done during the test *data* generation. This means that we need to add the required lemmas to the simplifier set, such that they will be used. The following additional lemmas are neccessary when we use the IntegerPort address representation. They should be added to the simplifier set just before test data generation.

lemma *subnetOf-lemma*: (*a*::*int*) ≠ (*c*::*int*) ⟹ ∀ *x*∈*subnet-of* (*a*, *b*::*port*). (*c*, *d*) ∉ *x*
apply (*rule ballI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma2*: ∀ *x*∈*subnet-of* (*a*::*int*, *b*::*port*). (*a*, *b*) ∈ *x*
apply (*rule ballI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma3*: (∃ *x*. *x* ∈ *subnet-of* (*a*::*int*, *b*::*port*))
apply (*rule exI*)
apply (*simp add*: *IntegerPort.subnet-of-def*)
done

lemma *subnetOf-lemma4*: ∃ *x*∈*subnet-of* (*a*::*int*, *b*::*port*). (*a*, *c*::*port*) ∈ *x*
apply (*rule bexI*)
apply (*simp-all add*: *IntegerPort.subnet-of-def*)
done

lemma *port-open-lemma*: ¬ (*Ex* (*port-open* [] (*x*::*port*)))
apply (*simp add*: *port-open-def*)
done

end

8 Examples

8.1 Stateless Example

theory
SimpleDMZIntegerDocument
imports

FWTesting
begin

This is a typical example for a small stateless packet filter. There are three subnetworks, with either none or some protocols allowed between then.
We use IntegerPort as the address model.

constdefs
intranet::*IntegerPort net*
intranet ≡ {{(*a*,*b*) . *a* = *3*}}

dmz :: *IntegerPort net*
dmz ≡ {{(*a*,*b*). *a* = *7*}}

internet :: *IntegerPort net*
internet ≡ {{(*a*,*b*). ¬ (*a*=*3* ∨ *a* =*7*)}}

constdefs
Intranet-DMZ-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Intranet-DMZ-Port ≡ *allow-all-from-port-to intranet ftp dmz*

Intranet-Internet-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Intranet-Internet-Port ≡ *allow-all-from-port-to intranet http internet*

Internet-DMZ-Port :: (*IntegerPort*,*DummyContent*) *Policy*
Internet-DMZ-Port ≡ *allow-all-from-port-to internet smtp dmz*

The policy:

definition *policy* :: (*IntegerPort*, *DummyContent*) *Policy* **where**
policy ≡ *deny-all* ++
Intranet-Internet-Port ++
Intranet-DMZ-Port ++
Internet-DMZ-Port

lemmas *PolicyLemmas* = *dmz-def internet-def intranet-def*
Intranet-Internet-Port-def Intranet-DMZ-Port-def
Internet-DMZ-Port-def policy-def
src-def dest-def in-subnet-def
IntegerPortLemmas
content-def

Only create test cases crossing network boundaries.

definition *not-in-same-net* :: (*IntegerPort*,*DummyContent*) *packet* ⇒ *bool* **where**
not-in-same-net x ≡ (*src x* ⊏ *internet* ⟶ ¬ *dest x* ⊏ *internet*) ∧
(*src x* ⊏ *intranet* ⟶ ¬ *dest x* ⊏ *intranet*) ∧
(*src x* ⊏ *dmz* ⟶ ¬ *dest x* ⊏ *dmz*)

declare *Ports* [*simp add*]

The test specification:

test-spec *not-in-same-net x* ⟶ *FUT x = policy x*
apply (*prepare-fw-spec*)
apply (*simp add*: *not-in-same-net-def PolicyLemmas PortCombinators PolicyCombinators*)
apply (*gen-test-cases FUT*)
apply (*simp-all add*: *PolicyLemmas*)
store-test-thm *PolicyTest*

testgen-params[*iterations=100*]

gen-test-data *PolicyTest*

The set of generated test data is:
FUT (−*10*, (*7*, *http*), (−*4*, −*8*), *data*) = *Some* (*deny* (−*10*, (*7*, *http*), (−*4*, −*8*), *data*))
FUT (*8*, (*7*, *ftp*), (*1*, *0*), *data*) = *Some* (*deny* (*8*, (*7*, *ftp*), (*1*, *0*), *data*))
FUT (−*7*, (*7*, *smtp*), (−*2*, −*7*), *data*) = *Some* (*deny* (−*7*, (*7*, *smtp*), (−*2*, −*7*), *data*))
FUT (*9*, (*7*, *0*), (*2*, −*5*), *data*) = *Some* (*deny* (*9*, (*7*, *0*), (*2*, −*5*), *data*))
FUT (−*2*, (*3*, *http*), (*7*, −*1*), *data*) = *Some* (*deny* (−*2*, (*3*, *http*), (*7*, −*1*), *data*))
FUT (−*8*, (*3*, *ftp*), (*7*, −*4*), *data*) = *Some* (*accept* (−*8*, (*3*, *ftp*), (*7*, −*4*), *data*))
FUT (*5*, (*3*, *smtp*), (*7*, *2*), *data*) = *Some* (*deny* (*5*, (*3*, *smtp*), (*7*, *2*), *data*))
FUT (−*9*, (*3*, *2*), (*7*, *0*), *data*) = *Some* (*deny* (−*9*, (*3*, *2*), (*7*, *0*), *data*))
FUT (−*3*, (*3*, *http*), (*6*, −*9*), *data*) = *Some* (*accept* (−*3*, (*3*, *http*), (*6*, −*9*), *data*))
FUT (*2*, (*3*, *ftp*), (−*8*, −*6*), *data*) = *Some* (*deny* (*2*, (*3*, *ftp*), (−*8*, −*6*), *data*))
FUT (*0*, (*3*, *smtp*), (*8*, *6*), *data*) = *Some* (*deny* (*0*, (*3*, *smtp*), (*8*, *6*), *data*))
FUT (−*8*, (*3*, −*6*), (*4*, −*8*), *data*) = *Some* (*deny* (−*8*, (*3*, −*6*), (*4*, −*8*), *data*))
FUT (*1*, (*7*, *http*), (*1*, −*10*), *data*) = *Some* (*deny* (*1*, (*7*, *http*), (*1*, −*10*), *data*))
FUT (*2*, (*7*, *ftp*), (*6*, *3*), *data*) = *Some* (*deny* (*2*, (*7*, *ftp*), (*6*, *3*), *data*))

$FUT\ (1,\ (7,\ smtp),\ (-7,\ -2),\ data) = Some\ (deny\ (1,\ (7,\ smtp),\ (-7,\ -2),\ data))$
$FUT\ (8,\ (7,\ 0),\ (-4,\ 7),\ data) = Some\ (deny\ (8,\ (7,\ 0),\ (-4,\ 7),\ data))$
$FUT\ (-5,\ (-9,\ http),\ (3,\ 10),\ data) = Some\ (deny\ (-5,\ (-9,\ http),\ (3,\ 10),\ data))$
$FUT\ (-10,\ (-3,\ ftp),\ (3,\ 3),\ data) = Some\ (deny\ (-10,\ (-3,\ ftp),\ (3,\ 3),\ data))$
$FUT\ (0,\ (-9,\ smtp),\ (3,\ -1),\ data) = Some\ (deny\ (0,\ (-9,\ smtp),\ (3,\ -1),\ data))$
$FUT\ (-2,\ (-3,\ -4),\ (3,\ 7),\ data) = Some\ (deny\ (-2,\ (-3,\ -4),\ (3,\ 7),\ data))$
$FUT\ (4,\ (7,\ http),\ (3,\ -1),\ data) = Some\ (deny\ (4,\ (7,\ http),\ (3,\ -1),\ data))$
$FUT\ (-3,\ (7,\ ftp),\ (3,\ -9),\ data) = Some\ (deny\ (-3,\ (7,\ ftp),\ (3,\ -9),\ data))$
$FUT\ (-5,\ (7,\ smtp),\ (3,\ -8),\ data) = Some\ (deny\ (-5,\ (7,\ smtp),\ (3,\ -8),\ data))$
$FUT\ (-4,\ (7,\ -8),\ (3,\ -6),\ data) = Some\ (deny\ (-4,\ (7,\ -8),\ (3,\ -6),\ data))$
$FUT\ (-10,\ (8,\ http),\ (3,\ -3),\ data) = Some\ (deny\ (-10,\ (8,\ http),\ (3,\ -3),\ data))$
$FUT\ (-10,\ (-2,\ ftp),\ (3,\ -9),\ data) = Some\ (deny\ (-10,\ (-2,\ ftp),\ (3,\ -9),\ data))$
$FUT\ (3,\ (6,\ smtp),\ (3,\ 4),\ data) = Some\ (deny\ (3,\ (6,\ smtp),\ (3,\ 4),\ data))$
$FUT\ (4,\ (4,\ 5),\ (3,\ -1),\ data) = Some\ (deny\ (4,\ (4,\ 5),\ (3,\ -1),\ data))$
$FUT\ (9,\ (-6,\ http),\ (7,\ 8),\ data) = Some\ (deny\ (9,\ (-6,\ http),\ (7,\ 8),\ data))$
$FUT\ (6,\ (-10,\ ftp),\ (7,\ -5),\ data) = Some\ (deny\ (6,\ (-10,\ ftp),\ (7,\ -5),\ data))$
$FUT\ (-8,\ (-2,\ smtp),\ (7,\ -4),\ data) = Some\ (accept\ (-8,\ (-2,\ smtp),\ (7,\ -4),\ data))$
$FUT\ (-1,\ (-3,\ -3),\ (7,\ 10),\ data) = Some\ (deny\ (-1,\ (-3,\ -3),\ (7,\ 10),\ data))$

end

8.2 FTP Example

theory *FTPTestDocument*
imports
FWTesting
begin

In this theory we generate the test data for correct runs of the FTP protocol. As usual, we start with defininining the networks and the policy. We use a rather simple policy which allows only FTP connections starting from the intranet going to the internet and denies everything else.

constdefs
intranet :: *IntegerPort net*
intranet ≡ {{(*a*,*e*) . *a* = *3*}}

internet :: *IntegerPort net*
internet ≡ {{(*a*,*c*). *a* > *3*}}

constdefs
ftp-policy :: (*IntegerPort*,*ftp-msg*) *Policy*
ftp-policy ≡ *deny-all* ++ *allow-from-to-port* (*21*::*port*) *intranet internet*

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

constdefs
is-in-intranet :: *IntegerPort* ⇒ *bool*
is-in-intranet a ≡ (*fst a*) = *3*

is-in-internet :: *IntegerPort* ⇒ *bool*
is-in-internet a ≡ (*fst a*) > *3*

The next definition is our starting state: an empty trace and the just defined policy.

constdefs
σ-0-ftp :: (*IntegerPort*, *ftp-msg*) *history* ×
(*IntegerPort*, *ftp-msg*) *Policy*
σ-0-ftp ≡ ([],*ftp-policy*)

Next we state the conditions we have on our trace: a normal behaviour FTP run from the intranet to some server in the internet on port 21.

constdefs *accept-ftp* :: (*IntegerPort*, *ftp-msg*) *history* ⇒ *bool*
accept-ftp t ≡ ∃ *c s i p*. *t* ∈ *NB-ftp c s i p* ∧ *is-in-intranet c* ∧ *is-in-internet s* ∧ (*snd s*) = *21*

fun *packet-with-id* **where**
packet-with-id [] *i* = []
|*packet-with-id* (*x*#*xs*) *i* = (*if id x* = *i then* (*x*#(*packet-with-id xs i*)) *else* (*packet-with-id xs i*))

The depth of the test case generation corresponds to the maximal length of generated traces. 4 is the minimum to get a full FTP protocol run.

testgen-params [*depth=4*]

The test specification:

test-spec *accept-ftp* (*rev t*) ⟶
(*σ-0-ftp* ⊨ (*os* ← *mbind t FTP-ST*; (λ *σ*. *Some* (*FUT* (*rev t*) = *σ*, *σ*))))
apply(*simp add*: *accept-ftp-def σ-0-ftp-def*)
apply (*rule impI*)+
apply (*unfold NB-ftp-def is-in-internet-def is-in-intranet-def*)
apply *simp*
apply (*gen-test-cases FUT split*: *HOL.split-if-asm*)
apply (*simp-all*)
store-test-thm *ftp-test*

We need to add all required lemmas to the simplifier set, such that they can be used during test data generation.

lemmas *ST-simps* = *Let-def valid-def unit-SE-def bind-SE-def orelse-def*
in-subnet-def src-def dest-def IntegerPort.dest-port-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 content-def
PolicyCombinators PortCombinators is-in-intranet-def
is-in-internet-def intranet-def internet-def exI subnetOf-lemma
subnetOf-lemma2 subnetOf-lemma3 subnetOf-lemma4 port-open-lemma
ftp-policy-def

declare *ST-simps* [*simp*]

gen-test-data ftp-test

declare *ST-simps* [*simp del*]

The generated test data look as follows (with the unfolded policy rewritten):

- FUT [(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)] = ([(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)],policy)
- FUT [(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6),(1, (3, 7), (9, 21), *ftp_init*)] = ([(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6), (1, (3, 7), (9, 21), *ftp_init*)],policy)

end

References

[1] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In A. Cavalli and S. Ghosh, editors, *International Conference on Software Testing (ICST10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[2] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki and T. Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

[3] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[4] D. von Bidder. *Specification-based Firewall Testing*. Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17172. Diana von Bidder's maiden name is Diana Senn.