Simple Firewall

Cornelius Diekmann, Julius Michaelis, Max Haslbeck

February 23, 2021

Abstract

We present a simple model of a firewall. The firewall can accept or drop a packet and can match on interfaces, IP addresses, protocol, and ports. It was designed to feature nice mathematical properties: The type of match expressions was carefully crafted such that the conjunction of two match expressions is only one match expression.

This model is too simplistic to mirror all aspects of the real world. In the upcoming entry "Iptables Semantics", we will translate the Linux firewall iptables to this model.

For a fixed service (e.g. ssh, http), this entry provides an algorithm to compute an overview of the firewall's filtering behavior. The algorithm computes minimal service matrices, i.e. graphs which partition the complete IPv4 and IPv6 address space and visualize the allowed accesses between partitions.

For a detailed description, see [1].

Contents

1 Enum toString Functions

theory *Lib-Enum-toString*
imports *Main IP-Addresses.Lib-List-toString*
begin

fun *bool-toString* :: *bool* ⇒ *string* **where**
bool-toString True = *''True''* |
bool-toString False = *''False''*

1.1 Enum set to string

fun *enum-set-get-one* :: *'a list* ⇒ *'a set* ⇒ *'a option* **where**
enum-set-get-one [] *S* = *None* |
enum-set-get-one (*s#ss*) *S* = (*if s* ∈ *S then Some s else enum-set-get-one ss S*)

lemma *enum-set-get-one-empty*: *enum-set-get-one ss* {} = *None*
⟨*proof*⟩

lemma *enum-set-get-one-None*: *S* ⊆ *set ss* ⟹ *enum-set-get-one ss S* = *None* ⟷ *S* = {}
⟨*proof*⟩

lemma *enum-set-get-one-Some*: *S* ⊆ *set ss* ⟹ *enum-set-get-one ss S* = *Some x* ⟹ *x* ∈ *S*
⟨*proof*⟩
corollary *enum-set-get-one-enum-Some*: *enum-set-get-one enum-class.enum S* = *Some x* ⟹ *x* ∈ *S*
⟨*proof*⟩

lemma *enum-set-get-one-Ex-Some*: *S* ⊆ *set ss* ⟹ *S* ≠ {} ⟹ ∃ *x*. *enum-set-get-one ss S* = *Some x*
⟨*proof*⟩
corollary *enum-set-get-one-enum-Ex-Some*:
S ≠ {} ⟹ ∃ *x*. *enum-set-get-one enum-class.enum S* = *Some x*
⟨*proof*⟩

function *enum-set-to-list* :: (*'a*::*enum*) *set* ⇒ *'a list* **where**
enum-set-to-list S = (*if S* = {} *then* [] *else*
case enum-set-get-one Enum.enum S of None ⇒ []
| *Some a* ⇒ *a#enum-set-to-list* (*S* − {*a*}))
⟨*proof*⟩

termination *enum-set-to-list*
⟨*proof*⟩

lemma *enum-set-to-list-simps*: *enum-set-to-list S* =
(*case enum-set-get-one* (*Enum.enum*) *S of None* ⇒ []
| *Some a* ⇒ *a#enum-set-to-list* (*S* − {*a*}))
⟨*proof*⟩
declare *enum-set-to-list.simps*[*simp del*]

lemma *enum-set-to-list*: *set* (*enum-set-to-list A*) = *A*
⟨*proof*⟩

lemma *list-toString bool-toString* (*enum-set-to-list* {*True*, *False*}) = ''[*False*, *True*]''
⟨*proof*⟩

end
theory *L4-Protocol*
imports ../*Common*/*Lib-Enum-toString HOL*−*Library.Word*
begin

2 Transport Layer Protocols

type-synonym *primitive-protocol* = *8 word*

definition *ICMP* ≡ *1* :: *8 word*
definition *TCP* ≡ *6* :: *8 word*
definition *UDP* ≡ *17* :: *8 word*
context begin
qualified definition *SCTP* ≡ *132* :: *8 word*
qualified definition *IGMP* ≡ *2* :: *8 word*
qualified definition *GRE* ≡ *47* :: *8 word*
qualified definition *ESP* ≡ *50* :: *8 word*
qualified definition *AH* ≡ *51* :: *8 word*
qualified definition *IPv6ICMP* ≡ *58* :: *8 word*
end

datatype *protocol* = *ProtoAny* | *Proto primitive-protocol*

fun *match-proto* :: *protocol* ⇒ *primitive-protocol* ⇒ *bool* **where**
match-proto ProtoAny - ⟷ *True* |
match-proto (*Proto* (*p*)) *p-p* ⟷ *p-p* = *p*

fun *simple-proto-conjunct* :: *protocol* ⇒ *protocol* ⇒ *protocol option* **where**
simple-proto-conjunct ProtoAny proto = *Some proto* |
simple-proto-conjunct proto ProtoAny = *Some proto* |
simple-proto-conjunct (*Proto p1*) (*Proto p2*) = (*if p1* = *p2 then Some* (*Proto p1*) *else None*)
lemma *simple-proto-conjunct-asimp*[*simp*]: *simple-proto-conjunct proto ProtoAny* = *Some proto*
⟨*proof*⟩

lemma *simple-proto-conjunct-correct*: *match-proto p1 pkt* ∧ *match-proto p2 pkt* ⟷
(*case simple-proto-conjunct p1 p2 of None* ⇒ *False* | *Some proto* ⇒ *match-proto proto pkt*)
⟨*proof*⟩

lemma *simple-proto-conjunct-Some*: *simple-proto-conjunct p1 p2* = *Some proto* ⟹

match-proto proto pkt ⟷ *match-proto p1 pkt* ∧ *match-proto p2 pkt*
⟨*proof*⟩
lemma *simple-proto-conjunct-None*: *simple-proto-conjunct p1 p2* = *None* ⟹
¬ (*match-proto p1 pkt* ∧ *match-proto p2 pkt*)
⟨*proof*⟩

lemma *conjunctProtoD*:
simple-proto-conjunct a (*Proto b*) = *Some x* ⟹ *x* = *Proto b* ∧ (*a* = *ProtoAny* ∨ *a* = *Proto b*)
⟨*proof*⟩
lemma *conjunctProtoD2*:
simple-proto-conjunct (*Proto b*) *a* = *Some x* ⟹ *x* = *Proto b* ∧ (*a* = *ProtoAny* ∨ *a* = *Proto b*)
⟨*proof*⟩

Originally, there was a *nat* in the protocol definition, allowing infinitely many protocols This was intended behavior. We want to prevent things such as *TCP* ≠ *UDP*. So be careful with what you prove...

lemma *primitive-protocol-Ex-neq*: *p* = *Proto pi* ⟹ ∃ *p'*. *p'* ≠ *pi* **for** *pi*
⟨*proof*⟩
lemma *protocol-Ex-neq*: ∃ *p'*. *Proto p'* ≠ *p*
⟨*proof*⟩

3 TCP flags

datatype *tcp-flag* = *TCP-SYN* | *TCP-ACK* | *TCP-FIN* | *TCP-RST* | *TCP-URG* | *TCP-PSH*

lemma *UNIV-tcp-flag*: *UNIV* = {*TCP-SYN*, *TCP-ACK*, *TCP-FIN*, *TCP-RST*, *TCP-URG*, *TCP-PSH*} ⟨*proof*⟩
instance *tcp-flag* :: *finite*
⟨*proof*⟩
instantiation *tcp-flag* :: *enum*
begin
definition *enum-tcp-flag* = [*TCP-SYN*, *TCP-ACK*, *TCP-FIN*, *TCP-RST*, *TCP-URG*, *TCP-PSH*]

definition *enum-all-tcp-flag P* ⟷ *P TCP-SYN* ∧ *P TCP-ACK* ∧ *P TCP-FIN* ∧ *P TCP-RST* ∧ *P TCP-URG* ∧ *P TCP-PSH*

definition *enum-ex-tcp-flag P* ⟷ *P TCP-SYN* ∨ *P TCP-ACK* ∨ *P TCP-FIN* ∨ *P TCP-RST* ∨ *P TCP-URG* ∨ *P TCP-PSH*
instance ⟨*proof*⟩
end

3.1 TCP Flags to String

fun *tcp-flag-toString* :: *tcp-flag* ⇒ *string* **where**

tcp-flag-toString TCP-SYN = ''*TCP-SYN*'' |
tcp-flag-toString TCP-ACK = ''*TCP-ACK*'' |
tcp-flag-toString TCP-FIN = ''*TCP-FIN*'' |
tcp-flag-toString TCP-RST = ''*TCP-RST*'' |
tcp-flag-toString TCP-URG = ''*TCP-URG*'' |
tcp-flag-toString TCP-PSH = ''*TCP-PSH*''

definition *ipt-tcp-flags-toString* :: *tcp-flag set* ⇒ *char list* **where**
ipt-tcp-flags-toString flags ≡ *list-toString tcp-flag-toString* (*enum-set-to-list flags*)

lemma *ipt-tcp-flags-toString* {*TCP-SYN*,*TCP-SYN*,*TCP-ACK*} = ''[*TCP-SYN*, *TCP-ACK*]'' ⟨*proof*⟩

end

4 Simple Packet

theory *Simple-Packet*
imports *Primitives/L4-Protocol*
begin

Packet constants are prefixed with *p*

'*i word* is an IP address of variable length. 32bit for IPv4, 128bit for IPv6

A simple packet with IP addresses and layer four ports. Also has the following phantom fields: Input and Output network interfaces

record (**overloaded**) '*i simple-packet* = *p-iiface* :: *string*
p-oiface :: *string*
p-src :: '*i*::*len word*
p-dst :: '*i*::*len word*
p-proto :: *primitive-protocol*
p-sport :: *16 word*
p-dport :: *16 word*
p-tcp-flags :: *tcp-flag set*
p-payload :: *string*

value [*nbe*] (|
p-iiface = ''*eth1*'', *p-oiface* = '''',
p-src = *0*, *p-dst* = *0*,
p-proto = *TCP*, *p-sport* = *0*, *p-dport* = *0*,
p-tcp-flags = {*TCP-SYN*},
p-payload = ''*arbitrary payload*''
|)

We suggest to use ('*i*, '*pkt-ext*) *simple-packet-scheme* instead of '*i simple-packet* because of its extensibility which naturally models any payload

definition *simple-packet-unext* :: ('*i*::*len*, '*a*) *simple-packet-scheme* ⇒ '*i simple-packet* **where**
simple-packet-unext p ≡
(|*p-iiface* = *p-iiface p*, *p-oiface* = *p-oiface p*, *p-src* = *p-src p*, *p-dst* = *p-dst p*, *p-proto* = *p-proto p*,
p-sport = *p-sport p*, *p-dport* = *p-dport p*, *p-tcp-flags* = *p-tcp-flags p*,
p-payload = *p-payload p*|)

An extended simple packet with MAC addresses and VLAN header

record (**overloaded**) '*i simple-packet-ext* = '*i*::*len simple-packet* +
p-l2type :: *16 word*
p-l2src :: *48 word*
p-l2dst :: *48 word*
p-vlanid :: *16 word*
p-vlanprio :: *16 word*

end

5 The state of a firewall, abstracted only to the packet filtering outcome

theory *Firewall-Common-Decision-State*
imports *Main*
begin

datatype *final-decision* = *FinalAllow* | *FinalDeny*

The state during packet processing. If undecided, there are some remaining rules to process. If decided, there is an action which applies to the packet

datatype *state* = *Undecided* | *Decision final-decision*

end

6 Network Interfaces

theory *Iface*
imports *HOL−Library.Char-ord*
begin

Network interfaces, e.g. `eth0`, `wlan1`, ...

iptables supports wildcard matching, e.g. `eth+` will match `eth`, `eth1`, `ethFOO`, ... The character '+' is only a wildcard if it appears at the end.

datatype *iface* = *Iface* (*iface-sel*: *string*) — no negation supported, but wildcards

Just a normal lexicographical ordering on the interface strings. Used only for optimizing code. WARNING: not a semantic ordering.

instantiation *iface* :: *linorder*
begin
  function (*sequential*) *less-eq-iface* :: *iface* ⇒ *iface* ⇒ *bool* **where**
    (*Iface* []) ≤ (*Iface* -) ⟷ *True* |
    (*Iface* -) ≤ (*Iface* []) ⟷ *False* |
    (*Iface* (*a*#*as*)) ≤ (*Iface* (*b*#*bs*)) ⟷ (*if a* = *b then Iface as* ≤ *Iface bs else a* ≤ *b*)
   ⟨*proof*⟩
  termination *less-eq* :: *iface* ⇒ - ⇒ *bool*
    ⟨*proof*⟩

  lemma *Iface-less-eq-empty*: *Iface x* ≤ *Iface* [] ⟹ *x* = []
    ⟨*proof*⟩
  lemma *less-eq-empty*: *Iface* [] ≤ *q*
    ⟨*proof*⟩
  lemma *iface-cons-less-eq-i*:
    Iface (*b* # *bs*) ≤ *i* ⟹ ∃ *q qs*. *i*=*Iface* (*q*#*qs*) ∧ (*b* < *q* ∨ (*Iface bs*) ≤ (*Iface qs*))
    ⟨*proof*⟩

  function (*sequential*) *less-iface* :: *iface* ⇒ *iface* ⇒ *bool* **where**
    (*Iface* []) < (*Iface* []) ⟷ *False* |
    (*Iface* []) < (*Iface* -) ⟷ *True* |
    (*Iface* -) < (*Iface* []) ⟷ *False* |
    (*Iface* (*a*#*as*)) < (*Iface* (*b*#*bs*)) ⟷ (*if a* = *b then Iface as* < *Iface bs else a* < *b*)
   ⟨*proof*⟩
  termination *less* :: *iface* ⇒ - ⇒ *bool*
    ⟨*proof*⟩
instance
  ⟨*proof*⟩
end

definition *ifaceAny* :: *iface* **where**
  ifaceAny ≡ *Iface* ''+''

If the interface name ends in a "+", then any interface which begins with this name will match. (man iptables)

Here is how iptables handles this wildcard on my system. A packet for the loopback interface `lo` is matched by the following expressions

- lo
- lo+
- l+

- +

It is not matched by the following expressions

- lo++
- lo+++
- lo1+
- lo1

By the way: `Warning: weird characters in interface ` ' ('/' and ' ' are not allowed by the kernel).` However, happy snowman and shell colors are fine.

context
begin

6.1 Helpers for the interface name (*string*)

argument 1: interface as in firewall rule - Wildcard support argument 2: interface a packet came from - No wildcard support

fun *internal-iface-name-match* :: *string* ⇒ *string* ⇒ *bool* **where**
internal-iface-name-match [] [] ⟷ *True* |
internal-iface-name-match (*i*#*is*) [] ⟷ (*i* = *CHR* ''+'' ∧ *is* = []) |
internal-iface-name-match [] (-#-) ⟷ *False* |
internal-iface-name-match (*i*#*is*) (*p-i*#*p-is*) ⟷ (*if* (*i* = *CHR* ''+'' ∧ *is* = []) *then* *True* *else* (
(*p-i* = *i*) ∧ *internal-iface-name-match* *is* *p-is*
))

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

fun *iface-name-is-wildcard* :: *string* ⇒ *bool* **where**
iface-name-is-wildcard [] ⟷ *False* |
iface-name-is-wildcard [*s*] ⟷ *s* = *CHR* ''+'' |
iface-name-is-wildcard (-#*ss*) ⟷ *iface-name-is-wildcard* *ss*
private lemma *iface-name-is-wildcard-alt*: *iface-name-is-wildcard* *eth* ⟷ *eth* ≠ [] ∧ *last* *eth* = *CHR* ''+''
⟨*proof*⟩ **lemma** *iface-name-is-wildcard-alt'*: *iface-name-is-wildcard* *eth* ⟷ *eth* ≠ [] ∧ *hd* (*rev* *eth*) = *CHR* ''+''
⟨*proof*⟩ **lemma** *iface-name-is-wildcard-fst*: *iface-name-is-wildcard* (*i* # *is*) ⟹ *is* ≠ [] ⟹ *iface-name-is-wildcard* *is*
⟨*proof*⟩ **fun** *internal-iface-name-to-set* :: *string* ⇒ *string set* **where**
internal-iface-name-to-set *i* = (*if* ¬ *iface-name-is-wildcard* *i*
then
{*i*}
else

{(*butlast i*)@*cs* | *cs. True*})

private lemma {(*butlast i*)@*cs* | *cs. True*} = (λ*s.* (*butlast i*)@*s*) ‘ (*UNIV*::*string set*) ⟨*proof*⟩ **lemma** *internal-iface-name-to-set*: *internal-iface-name-match i p-iface* ⟷ *p-iface* ∈ *internal-iface-name-to-set i*

⟨*proof*⟩ **lemma** *internal-iface-name-to-set2*: *internal-iface-name-to-set ifce* = {*i. internal-iface-name-match ifce i*}

⟨*proof*⟩ **lemma** *internal-iface-name-match-refl*: *internal-iface-name-match i i*

⟨*proof*⟩

6.2 Matching

fun *match-iface* :: *iface* ⇒ *string* ⇒ *bool* **where**
match-iface (*Iface i*) *p-iface* ⟷ *internal-iface-name-match i p-iface*

— Examples

lemma *match-iface* (*Iface* ''*lo*'') ''*lo*''
match-iface (*Iface* ''*lo+*'') ''*lo*''
match-iface (*Iface* ''*l+*'') ''*lo*''
match-iface (*Iface* ''*+*'') ''*lo*''
¬ *match-iface* (*Iface* ''*lo++*'') ''*lo*''
¬ *match-iface* (*Iface* ''*lo+++*'') ''*lo*''
¬ *match-iface* (*Iface* ''*lo1+*'') ''*lo*''
¬ *match-iface* (*Iface* ''*lo1*'') ''*lo*''
match-iface (*Iface* ''*+*'') ''*eth0*''
match-iface (*Iface* ''*+*'') ''*eth0*''
match-iface (*Iface* ''*eth+*'') ''*eth0*''
¬ *match-iface* (*Iface* ''*lo+*'') ''*eth0*''
match-iface (*Iface* ''*lo+*'') ''*loX*''
¬ *match-iface* (*Iface* '''') ''*loX*''
⟨*proof*⟩

lemma *match-ifaceAny*: *match-iface ifaceAny i*
⟨*proof*⟩

lemma *match-IfaceFalse*: ¬(∃ *IfaceFalse.* (∀ *i.* ¬ *match-iface IfaceFalse i*))
⟨*proof*⟩

lemma *match-iface-case-nowildcard*: ¬ *iface-name-is-wildcard i* ⟹ *match-iface* (*Iface i*) *p-i* ⟷ *i* = *p-i*
⟨*proof*⟩

lemma *match-iface-case-wildcard-prefix*:
iface-name-is-wildcard i ⟹ *match-iface* (*Iface i*) *p-i* ⟷ *butlast i* = *take* (*length i* − *1*) *p-i*
⟨*proof*⟩

lemma *match-iface-case-wildcard-length*: *iface-name-is-wildcard i* ⟹ *match-iface* (*Iface i*) *p-i* ⟹ *length p-i* ≥ (*length i* − *1*)
⟨*proof*⟩

corollary *match-iface-case-wildcard*:
iface-name-is-wildcard i ⟹ *match-iface* (*Iface i*) *p-i* ⟷ *butlast i* = *take* (*length i* − *1*) *p-i* ∧ *length p-i* ≥ (*length i* − *1*)
⟨*proof*⟩

lemma *match-iface-set*: *match-iface* (*Iface i*) *p-iface* ⟷ *p-iface* ∈ *internal-iface-name-to-set i*
⟨*proof*⟩ **definition** *internal-iface-name-wildcard-longest* :: *string* ⇒ *string* ⇒ *string option* **where**
internal-iface-name-wildcard-longest i1 i2 = (
if
take (*min* (*length i1* − *1*) (*length i2* − *1*)) *i1* = *take* (*min* (*length i1* − *1*) (*length i2* − *1*)) *i2*
then
Some (*if length i1* ≤ *length i2 then i2 else i1*)
else
None)
private lemma *internal-iface-name-wildcard-longest* ''*eth+*'' ''*eth3+*'' = *Some* ''*eth3+*'' ⟨*proof*⟩ **lemma** *internal-iface-name-wildcard-longest* ''*eth+*'' ''*e+*'' = *Some* ''*eth+*'' ⟨*proof*⟩ **lemma** *internal-iface-name-wildcard-longest* ''*eth+*'' ''*lo*'' = *None*
⟨*proof*⟩ **lemma** *internal-iface-name-wildcard-longest-commute*: *iface-name-is-wildcard i1* ⟹ *iface-name-is-wildcard i2* ⟹
internal-iface-name-wildcard-longest i1 i2 = *internal-iface-name-wildcard-longest i2 i1*
⟨*proof*⟩ **lemma** *internal-iface-name-wildcard-longest-refl*: *internal-iface-name-wildcard-longest i i* = *Some i*
⟨*proof*⟩ **lemma** *internal-iface-name-wildcard-longest-correct*:
iface-name-is-wildcard i1 ⟹ *iface-name-is-wildcard i2* ⟹
match-iface (*Iface i1*) *p-i* ∧ *match-iface* (*Iface i2*) *p-i* ⟷
(*case internal-iface-name-wildcard-longest i1 i2 of None* ⇒ *False* | *Some x* ⇒ *match-iface* (*Iface x*) *p-i*)
⟨*proof*⟩

fun *iface-conjunct* :: *iface* ⇒ *iface* ⇒ *iface option* **where**
iface-conjunct (*Iface i1*) (*Iface i2*) = (*case* (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) *of*
(*True*, *True*) ⇒ *map-option Iface* (*internal-iface-name-wildcard-longest i1 i2*) |
(*True*, *False*) ⇒ (*if match-iface* (*Iface i1*) *i2 then Some* (*Iface i2*) *else None*) |
(*False*, *True*) ⇒ (*if match-iface* (*Iface i2*) *i1 then Some* (*Iface i1*) *else None*) |
(*False*, *False*) ⇒ (*if i1* = *i2 then Some* (*Iface i1*) *else None*))

lemma *iface-conjunct-Some*: *iface-conjunct i1 i2* = *Some x* ⟹
match-iface x p-i ⟷ *match-iface i1 p-i* ∧ *match-iface i2 p-i*
⟨*proof*⟩
lemma *iface-conjunct-None*: *iface-conjunct i1 i2* = *None* ⟹ ¬ (*match-iface i1 p-i* ∧ *match-iface i2 p-i*)
⟨*proof*⟩
lemma *iface-conjunct*: *match-iface i1 p-i* ∧ *match-iface i2 p-i* ⟷
(*case iface-conjunct i1 i2 of None* ⇒ *False* | *Some x* ⇒ *match-iface x p-i*)

⟨*proof*⟩

lemma *match-iface-refl*: *match-iface* (*Iface x*) *x* ⟨*proof*⟩
lemma *match-iface-eqI*: **assumes** $x = Iface\ y$ **shows** *match-iface x y*
⟨*proof*⟩

lemma *iface-conjunct-ifaceAny*: *iface-conjunct ifaceAny i* = *Some i*
⟨*proof*⟩

lemma *iface-conjunct-commute*: *iface-conjunct i1 i2* = *iface-conjunct i2 i1*
⟨*proof*⟩ **definition** *internal-iface-name-subset* :: *string* ⇒ *string* ⇒ *bool* **where**
internal-iface-name-subset i1 i2 = (*case* (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) *of*
(*True*, *True*) ⇒ *length i1* ≥ *length i2* ∧ *take* ((*length i2*) − *1*) *i1* = *butlast i2* |
(*True*, *False*) ⇒ *False* |
(*False*, *True*) ⇒ *take* (*length i2* − *1*) *i1* = *butlast i2* |
(*False*, *False*) ⇒ *i1* = *i2*
)

private lemma *butlast-take-length-helper*:
fixes *x* ::*char list*
assumes *a1*: *length i2* ≤ *length i1*
assumes *a2*: *take* (*length i2* − *Suc 0*) *i1* = *butlast i2*
assumes *a3*: *butlast i1* = *take* (*length i1* − *Suc 0*) *x*
shows *butlast i2* = *take* (*length i2* − *Suc 0*) *x*
⟨*proof*⟩ **lemma** *internal-iface-name-subset*: *internal-iface-name-subset i1 i2* ⟷

{*i*. *internal-iface-name-match i1 i*} ⊆ {*i*. *internal-iface-name-match i2 i*}
⟨*proof*⟩

definition *iface-subset* :: *iface* ⇒ *iface* ⇒ *bool* **where**
iface-subset i1 i2 ⟷ *internal-iface-name-subset* (*iface-sel i1*) (*iface-sel i2*)

lemma *iface-subset*: *iface-subset i1 i2* ⟷ {*i*. *match-iface i1 i*} ⊆ {*i*. *match-iface i2 i*}
⟨*proof*⟩

definition *iface-is-wildcard* :: *iface* ⇒ *bool* **where**
iface-is-wildcard ifce ≡ *iface-name-is-wildcard* (*iface-sel ifce*)

lemma *iface-is-wildcard-ifaceAny*: *iface-is-wildcard ifaceAny*
⟨*proof*⟩

6.3 Enumerating Interfaces

private definition *all-chars* :: *char list* **where**
all-chars ≡ *Enum.enum*
private lemma *all-chars*: *set all-chars* = (*UNIV*::*char set*)
⟨*proof*⟩

we can compute this, but its horribly inefficient!

private lemma *strings-of-length-n*: *set* (*List.n-lists n all-chars*) = {*s*::*string*. *length s* = *n*}
⟨*proof*⟩

Non-wildacrd interfaces of length n

private definition *non-wildcard-ifaces* :: *nat* ⇒ *string list* **where**
non-wildcard-ifaces n ≡ *filter* (λ*i*. ¬ *iface-name-is-wildcard i*) (*List.n-lists n all-chars*)

Example: (any number higher than zero are probably too inefficient)

private lemma *non-wildcard-ifaces 0* = [''''] ⟨*proof*⟩ **lemma** *non-wildcard-ifaces*: *set* (*non-wildcard-ifaces n*) = {*s*::*string*. *length s* = *n* ∧ ¬ *iface-name-is-wildcard s*}
⟨*proof*⟩ **lemma** (⋃ *i* ∈ *set* (*non-wildcard-ifaces n*). *internal-iface-name-to-set i*) = {*s*::*string*. *length s* = *n* ∧ ¬ *iface-name-is-wildcard s*}
⟨*proof*⟩

Non-wildacrd interfaces up to length n

private fun *non-wildcard-ifaces-upto* :: *nat* ⇒ *string list* **where**
non-wildcard-ifaces-upto 0 = [[]] |
non-wildcard-ifaces-upto (*Suc n*) = (*non-wildcard-ifaces* (*Suc n*)) @ *non-wildcard-ifaces-upto n*
private lemma *non-wildcard-ifaces-upto*: *set* (*non-wildcard-ifaces-upto n*) = {*s*::*string*. *length s* ≤ *n* ∧ ¬ *iface-name-is-wildcard s*}
⟨*proof*⟩

6.4 Negating Interfaces

private lemma *inv-iface-name-set*: − (*internal-iface-name-to-set i*) = (
if iface-name-is-wildcard i
then
{*c* |*c*. *length c* < *length* (*butlast i*)} ∪ {*c* @ *cs* |*c cs*. *length c* = *length* (*butlast i*) ∧ *c* ≠ *butlast i*}
else
{*c* | *c*. *length c* < *length i*} ∪ {*c*@*cs* | *c cs*. *length c* ≥ *length i* ∧ *c* ≠ *i*}
)
⟨*proof*⟩

Negating is really not intuitive. The Interface ''*et*'' is in the negated set of ''*eth+*''. And the Interface ''*et+*'' is also in this set! This is because ''+'' is a normal interface character and not a wildcard here! In contrast, the set described by *''et+''* (with *''+''* a wildcard) is not a subset of the previously negated set.

lemma *''et''* ∈ − (*internal-iface-name-to-set ''eth+''*) ⟨*proof*⟩
lemma *''et+''* ∈ − (*internal-iface-name-to-set ''eth+''*) ⟨*proof*⟩
lemma *''+''* ∈ − (*internal-iface-name-to-set ''eth+''*) ⟨*proof*⟩
lemma ¬ {*i. match-iface* (*Iface ''et+''*) *i*} ⊆ − (*internal-iface-name-to-set ''eth+''*) ⟨*proof*⟩

Because *''+''* can appear as interface wildcard and normal interface character, we cannot take negate an *Iface i* such that we get back *iface list* which describe the negated interface.

lemma *''+''* ∈ − (*internal-iface-name-to-set ''eth+''*) ⟨*proof*⟩

fun *compress-pos-interfaces* :: *iface list* ⇒ *iface option* **where**
compress-pos-interfaces [] = *Some ifaceAny* |
compress-pos-interfaces [*i*] = *Some i* |
compress-pos-interfaces (*i1#i2#is*) = (*case iface-conjunct i1 i2 of None* ⇒ *None* | *Some i* ⇒ *compress-pos-interfaces* (*i#is*))

lemma *compress-pos-interfaces-Some*: *compress-pos-interfaces ifces* = *Some ifce* ⟹
match-iface ifce p-i ⟷ (∀ *i*∈ *set ifces. match-iface i p-i*)
⟨*proof*⟩

lemma *compress-pos-interfaces-None*: *compress-pos-interfaces ifces* = *None* ⟹
¬ (∀ *i*∈ *set ifces. match-iface i p-i*)
⟨*proof*⟩

declare *match-iface.simps*[*simp del*]
declare *iface-name-is-wildcard.simps*[*simp del*]
end

end

7 Simple Firewall Syntax

theory *SimpleFw-Syntax*
imports *IP-Addresses.Hs-Compat*
Firewall-Common-Decision-State
Primitives/Iface
Primitives/L4-Protocol
Simple-Packet

begin

For for IP addresses of arbitrary length

datatype *simple-action* = *Accept* | *Drop*

Simple match expressions do not allow negated expressions. However, Most match expressions can still be transformed into simple match expressions.

A negated IP address range can be represented as a set of non-negated IP ranges. For example $!8 = \{0..7\} \cup \{8 \;..\; ipv4max\}$. Using CIDR notation (i.e. the $a.b.c.d/n$ notation), we can represent negated IP ranges as a set of non-negated IP ranges with only fair blowup. Another handy result is that the conjunction of two IP ranges in CIDR notation is either the smaller of the two ranges or the empty set. An empty IP range cannot be represented. If one wants to represent the empty range, then the complete rule needs to be removed.

The same holds for layer 4 ports. In addition, there exists an empty port range, e.g. $(1,0)$. The conjunction of two port ranges is again just one port range.

But negation of interfaces is not supported. Since interfaces support a wildcard character, transforming a negated interface would either result in an infeasible blowup or requires knowledge about the existing interfaces (e.g. there only is eth0, eth1, wlan3, and vbox42) An empirical test shows that negated interfaces do not occur in our data sets. Negated interfaces can also be considered bad style: What is !eth0? Everything that is not eth0, experience shows that interfaces may come up randomly, in particular in combination with virtual machines, so !eth0 might not be the desired match. At the moment, if an negated interface occurs which prevents translation to a simple match, we recommend to abstract the negated interface to unknown and remove it (upper or lower closure rule set) before translating to a simple match. The same discussion holds for negated protocols.

Noteworthy, simple match expressions are both expressive and support conjunction: $simple{-}match1 \wedge simple{-}match2 = simple{-}match3$

record (**overloaded**) *'i simple-match* =
iiface :: *iface* — in-interface

oiface :: *iface* — out-interface
src :: (*'i::len word* × *nat*) — source IP address
dst :: (*'i::len word* × *nat*) — destination
proto :: *protocol*
sports :: (*16 word* × *16 word*) — source-port first:last
dports :: (*16 word* × *16 word*) — destination-port first:last

context

notes [[*typedef-overloaded*]]
begin
datatype *'i simple-rule* = *SimpleRule* (*match-sel*: *'i simple-match*) (*action-sel*: *simple-action*)
end

Simple rule destructor. Removes the *'a simple-rule* type, returns a tuple with the match and action.

definition *simple-rule-dtor* :: *'a simple-rule* ⇒ *'a simple-match* × *simple-action* **where**
simple-rule-dtor r ≡ (*case r of SimpleRule m a* ⇒ (*m*,*a*))

lemma *simple-rule-dtor-ids*:
uncurry SimpleRule ∘ *simple-rule-dtor* = *id*
simple-rule-dtor ∘ *uncurry SimpleRule* = *id*
⟨*proof*⟩

end

8 Simple Firewall Semantics

theory *SimpleFw-Semantics*
imports *SimpleFw-Syntax*
IP-Addresses.IP-Address
IP-Addresses.Prefix-Match
begin

fun *simple-match-ip* :: (*'i*::*len word* × *nat*) ⇒ *'i*::*len word* ⇒ *bool* **where**
simple-match-ip (*base*, *len*) *p-ip* ⟷ *p-ip* ∈ *ipset-from-cidr base len*

lemma *wordinterval-to-set-ipcidr-tuple-to-wordinterval-simple-match-ip-set*:
wordinterval-to-set (*ipcidr-tuple-to-wordinterval ip*) = {*d*. *simple-match-ip ip d*}
⟨*proof*⟩
lemma {(*253*::*8 word*) .. *8*} = {} ⟨*proof*⟩

fun *simple-match-port* :: (*16 word* × *16 word*) ⇒ *16 word* ⇒ *bool* **where**
simple-match-port (*s*,*e*) *p-p* ⟷ *p-p* ∈ {*s*..*e*}

fun *simple-matches* :: *'i*::*len simple-match* ⇒ (*'i*, *'a*) *simple-packet-scheme* ⇒ *bool* **where**
simple-matches m p ⟷
(*match-iface* (*iiface m*) (*p-iiface p*)) ∧
(*match-iface* (*oiface m*) (*p-oiface p*)) ∧
(*simple-match-ip* (*src m*) (*p-src p*)) ∧
(*simple-match-ip* (*dst m*) (*p-dst p*)) ∧
(*match-proto* (*proto m*) (*p-proto p*)) ∧
(*simple-match-port* (*sports m*) (*p-sport p*)) ∧
(*simple-match-port* (*dports m*) (*p-dport p*))

The semantics of a simple firewall: just iterate over the rules sequentially

fun *simple-fw* :: *'i::len simple-rule list* ⇒ (*'i*, *'a*) *simple-packet-scheme* ⇒ *state*
where
simple-fw [] - = *Undecided* |
simple-fw ((*SimpleRule m Accept*)#*rs*) *p* = (*if simple-matches m p then Decision FinalAllow else simple-fw rs p*) |
simple-fw ((*SimpleRule m Drop*)#*rs*) *p* = (*if simple-matches m p then Decision FinalDeny else simple-fw rs p*)

fun *simple-fw-alt* **where**
simple-fw-alt [] - = *Undecided* |
simple-fw-alt (*r*#*rs*) *p* = (*if simple-matches* (*match-sel r*) *p then*
(*case action-sel r of Accept* ⇒ *Decision FinalAllow* | *Drop* ⇒ *Decision FinalDeny*) *else simple-fw-alt rs p*)

lemma *simple-fw-alt*: *simple-fw r p* = *simple-fw-alt r p* ⟨*proof*⟩

definition *simple-match-any* :: *'i::len simple-match* **where**
simple-match-any ≡ (|*iiface*=*ifaceAny*, *oiface*=*ifaceAny*, *src*=(*0*,*0*), *dst*=(*0*,*0*), *proto*=*ProtoAny*, *sports*=(*0*,*65535*), *dports*=(*0*,*65535*) |)
lemma *simple-match-any*: *simple-matches simple-match-any p*
⟨*proof*⟩

we specify only one empty port range

definition *simple-match-none* :: *'i::len simple-match* **where**
simple-match-none ≡
(|*iiface*=*ifaceAny*, *oiface*=*ifaceAny*, *src*=(*1*,*0*), *dst*=(*0*,*0*),
proto=*ProtoAny*, *sports*=(*1*,*0*), *dports*=(*0*,*65535*) |)
lemma *simple-match-none*: ¬ *simple-matches simple-match-none p*
⟨*proof*⟩

fun *empty-match* :: *'i::len simple-match* ⇒ *bool* **where**
empty-match (|*iiface*=-, *oiface*=-, *src*=-, *dst*=-, *proto*=-,
sports=(*sps1*, *sps2*), *dports*=(*dps1*, *dps2*) |) ⟷ (*sps1* > *sps2*) ∨ (*dps1* > *dps2*)

lemma *empty-match*: *empty-match m* ⟷ (∀ (*p*::(*'i::len*, *'a*) *simple-packet-scheme*). ¬ *simple-matches m p*)
⟨*proof*⟩

lemma *nomatch*: ¬ *simple-matches m p* ⟹ *simple-fw* (*SimpleRule m a* # *rs*) *p* = *simple-fw rs p*
⟨*proof*⟩

8.1 Simple Ports

fun *simpl-ports-conjunct* :: (*16 word* × *16 word*) ⇒ (*16 word* × *16 word*) ⇒ (*16 word* × *16 word*) **where**

simpl-ports-conjunct (*p1s*, *p1e*) (*p2s*, *p2e*) = (*max p1s p2s*, *min p1e p2e*)

lemma {(*p1s*:: *16 word*) .. *p1e*} ∩ {*p2s* .. *p2e*} = {*max p1s p2s* .. *min p1e p2e*}
⟨*proof*⟩

lemma *simple-ports-conjunct-correct*:
simple-match-port p1 pkt ∧ *simple-match-port p2 pkt* ⟷ *simple-match-port* (*simpl-ports-conjunct p1 p2*) *pkt*
⟨*proof*⟩

lemma *simple-match-port-code*[*code*] :*simple-match-port* (*s*,*e*) *p-p* = (*s* ≤ *p-p* ∧ *p-p* ≤ *e*) ⟨*proof*⟩

lemma *simple-match-port-UNIV*: {*p*. *simple-match-port* (*s*,*e*) *p*} = *UNIV* ⟷ (*s* = *0* ∧ *e* = *max-word*)
⟨*proof*⟩

8.2 Simple IPs

lemma *simple-match-ip-conjunct*:
fixes *ip1* :: ′*i*::*len word* × *nat*
shows *simple-match-ip ip1 p-ip* ∧ *simple-match-ip ip2 p-ip* ⟷
(*case ipcidr-conjunct ip1 ip2 of None* ⇒ *False* | *Some ipx* ⇒ *simple-match-ip ipx p-ip*)
⟨*proof*⟩

declare *simple-matches.simps*[*simp del*]

8.3 Merging Simple Matches

′*i simple-match* ∧ ′*i simple-match*

fun *simple-match-and* :: ′*i*::*len simple-match* ⇒ ′*i simple-match* ⇒ ′*i simple-match option* **where**
simple-match-and (|*iiface*=*iif1*, *oiface*=*oif1*, *src*=*sip1*, *dst*=*dip1*, *proto*=*p1*, *sports*=*sps1*, *dports*=*dps1* |)
(|*iiface*=*iif2*, *oiface*=*oif2*, *src*=*sip2*, *dst*=*dip2*, *proto*=*p2*, *sports*=*sps2*, *dports*=*dps2* |) =
(*case ipcidr-conjunct sip1 sip2 of None* ⇒ *None* | *Some sip* ⇒
(*case ipcidr-conjunct dip1 dip2 of None* ⇒ *None* | *Some dip* ⇒
(*case iface-conjunct iif1 iif2 of None* ⇒ *None* | *Some iif* ⇒
(*case iface-conjunct oif1 oif2 of None* ⇒ *None* | *Some oif* ⇒
(*case simple-proto-conjunct p1 p2 of None* ⇒ *None* | *Some p* ⇒
Some (|*iiface*=*iif*, *oiface*=*oif*, *src*=*sip*, *dst*=*dip*, *proto*=*p*,
sports=*simpl-ports-conjunct sps1 sps2*, *dports*=*simpl-ports-conjunct dps1 dps2* |))))))

lemma *simple-match-and-correct*: *simple-matches m1 p* ∧ *simple-matches m2 p* ⟷

(*case simple-match-and m1 m2 of None* ⇒ *False* | *Some m* ⇒ *simple-matches m p*)
⟨*proof*⟩

lemma *simple-match-and-SomeD*: *simple-match-and m1 m2* = *Some m* ⟹
simple-matches m p ⟷ (*simple-matches m1 p* ∧ *simple-matches m2 p*)
⟨*proof*⟩
lemma *simple-match-and-NoneD*: *simple-match-and m1 m2* = *None* ⟹
¬(*simple-matches m1 p* ∧ *simple-matches m2 p*)
⟨*proof*⟩
lemma *simple-matches-andD*: *simple-matches m1 p* ⟹ *simple-matches m2 p* ⟹
∃ *m*. *simple-match-and m1 m2* = *Some m* ∧ *simple-matches m p*
⟨*proof*⟩

8.4 Further Properties of a Simple Firewall

fun *has-default-policy* :: *'i::len simple-rule list* ⇒ *bool* **where**
has-default-policy [] = *False* |
has-default-policy [(*SimpleRule m* -)] = (*m* = *simple-match-any*) |
has-default-policy (-#*rs*) = *has-default-policy rs*

lemma *has-default-policy*: *has-default-policy rs* ⟹
simple-fw rs p = *Decision FinalAllow* ∨ *simple-fw rs p* = *Decision FinalDeny*
⟨*proof*⟩

lemma *has-default-policy-fst*: *has-default-policy rs* ⟹ *has-default-policy* (*r*#*rs*)
⟨*proof*⟩

We can stop after a default rule (a rule which matches anything) is observed.

fun *cut-off-after-match-any* :: *'i::len simple-rule list* ⇒ *'i simple-rule list* **where**
cut-off-after-match-any [] = [] |
cut-off-after-match-any (*SimpleRule m a* # *rs*) =
(*if m* = *simple-match-any then* [*SimpleRule m a*] *else SimpleRule m a* # *cut-off-after-match-any rs*)

lemma *cut-off-after-match-any*: *simple-fw* (*cut-off-after-match-any rs*) *p* = *simple-fw rs p*
⟨*proof*⟩

lemma *simple-fw-not-matches-removeAll*: ¬ *simple-matches m p* ⟹
simple-fw (*removeAll* (*SimpleRule m a*) *rs*) *p* = *simple-fw rs p*
⟨*proof*⟩

8.5 Reality check: Validity of Simple Matches

While it is possible to construct a *simple-fw* expression that only matches a source or destination port, such a match is not meaningful, as the presence of the port information is dependent on the protocol. Thus, a match for a port should always include the match for a protocol. Additionally, prefixes should be zero on bits beyond the prefix length.

definition *valid-prefix-fw m* = *valid-prefix* (*uncurry PrefixMatch m*)

lemma *ipcidr-conjunct-valid*:
$[\![$*valid-prefix-fw p1*; *valid-prefix-fw p2*; *ipcidr-conjunct p1 p2* = *Some p*$]\!] \Longrightarrow$ *valid-prefix-fw p*
⟨*proof*⟩

definition *simple-match-valid* :: (′*i*::*len*, ′*a*) *simple-match-scheme* ⇒ *bool* **where**
simple-match-valid m ≡
({*p*. *simple-match-port* (*sports m*) *p*} ≠ *UNIV* ∨ {*p*. *simple-match-port* (*dports m*) *p*} ≠ *UNIV* ⟶
proto m ∈ *Proto* ‘{*TCP*, *UDP*, *L4-Protocol.SCTP*}) ∧
valid-prefix-fw (*src m*) ∧ *valid-prefix-fw* (*dst m*)

lemma *simple-match-valid-alt*[*code-unfold*]: *simple-match-valid* = (λ *m*.
(*let c* = (λ(*s*,*e*). (*s* ≠ *0* ∨ *e* ≠ *max-word*)) *in* (
if c (*sports m*) ∨ *c* (*dports m*) *then proto m* = *Proto TCP* ∨ *proto m* = *Proto UDP* ∨ *proto m* = *Proto L4-Protocol.SCTP else True*)) ∧
valid-prefix-fw (*src m*) ∧ *valid-prefix-fw* (*dst m*))
⟨*proof*⟩

Example:

context
begin
private definition *example-simple-match1* ≡
(|*iiface* = *Iface* ″+″, *oiface* = *Iface* ″+″, *src* = (*0*::*32 word*, *0*), *dst* = (*0*, *0*), *proto* = *Proto TCP*, *sports* = (*0*, *1024*), *dports* = (*0*, *1024*)|)

lemma *simple-fw* [*SimpleRule example-simple-match1 Drop*]
(|*p-iiface* = ″″, *p-oiface* = ″″, *p-src* = (*1*::*32 word*), *p-dst* = *2*, *p-proto* = *TCP*, *p-sport* = *8*,
p-dport = *9*, *p-tcp-flags* = {}, *p-payload* = ″″|) =
Decision FinalDeny ⟨*proof*⟩ **definition** *example-simple-match2* ≡ *example-simple-match1*(| *proto* := *ProtoAny* |)

Thus, *example-simple-match1* is valid, but if we set its protocol match to any, it isn't anymore

private lemma *simple-match-valid example-simple-match1* ⟨*proof*⟩ **lemma** ¬ *simple-match-valid example-simple-match2* ⟨*proof*⟩
end

lemma *simple-match-and-valid*:
fixes *m1* :: ′*i*::*len simple-match*
assumes *mv*: *simple-match-valid m1 simple-match-valid m2*
assumes *mj*: *simple-match-and m1 m2* = *Some m*

shows *simple-match-valid m*
⟨*proof*⟩

definition *simple-fw-valid* ≡ *list-all* (*simple-match-valid* ∘ *match-sel*)

The simple firewall does not care about tcp flags, payload, or any other packet extensions.

lemma *simple-matches-extended-packet*:
simple-matches m
(|*p-iiface* = *iifce*,
oiface = *oifce*,
p-src = *s*, *dst* = *d*,
p-proto = *prot*,
p-sport = *sport*, *p-dport* = *dport*,
tcp-flags = *tcp-flags*, *p-payload* = *payload1*|)
⟷
simple-matches m
(|*p-iiface* = *iifce*,
oiface = *oifce*,
p-src = *s*, *p-dst* = *d*,
p-proto = *prot*,
p-sport = *sport*, *p-dport* = *dport*,
p-tcp-flags = *tcp-flags2*, *p-payload* = *payload2*, ... = *aux*|)

⟨*proof*⟩
end

9 List Product Helpers

theory *List-Product-More*
imports *Main*
begin

lemma *List-product-concat-map*: *List.product xs ys* = *concat* (*map* (λ*x*. *map* (λ*y*. (*x*,*y*)) *ys*) *xs*)
⟨*proof*⟩

definition *all-pairs* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
all-pairs xs ≡ *concat* (*map* (λ*x*. *map* (λ*y*. (*x*,*y*)) *xs*) *xs*)

lemma *all-pairs-list-product*: *all-pairs xs* = *List.product xs xs*
⟨*proof*⟩

lemma *all-pairs*: ∀ (*x*,*y*) ∈ (*set xs* × *set xs*). (*x*,*y*) ∈ *set* (*all-pairs xs*)
⟨*proof*⟩

lemma *all-pairs-set*: *set* (*all-pairs xs*) = *set xs* × *set xs*
⟨*proof*⟩

end

10 Option to List and Option to Set

theory *Option-Helpers*
imports *Main*
begin

Those are just syntactic helpers.

definition *option2set* :: *'a option* ⇒ *'a set* **where**
option2set n ≡ (*case n of None* ⇒ {} | *Some s* ⇒ {*s*})

definition *option2list* :: *'a option* ⇒ *'a list* **where**
option2list n ≡ (*case n of None* ⇒ [] | *Some s* ⇒ [*s*])

lemma *set-option2list*[*simp*]: *set* (*option2list k*) = *option2set k*
⟨*proof*⟩

lemma *option2list-simps*[*simp*]: *option2list* (*Some x*) = [*x*] *option2list* (*None*) = []
⟨*proof*⟩

lemma *option2set-None*: *option2set None* = {}
⟨*proof*⟩

lemma *option2list-map*: *option2list* (*map-option f n*) = *map f* (*option2list n*)
⟨*proof*⟩

lemma *option2set-map*: *option2set* (*map-option f n*) = *f* ‘ *option2set n*
⟨*proof*⟩

end

11 Generalize Simple Firewall

theory *Generic-SimpleFw*
imports *SimpleFw-Semantics Common/List-Product-More Common/Option-Helpers*
begin

11.1 Semantics

The semantics of the *simple-fw* is quite close to *find*. The idea of the generalized *simple-fw* semantics is that you can have anything as the resulting action, not only a *simple-action*.

definition *generalized-sfw*

:: ('i::len simple-match × 'a) list ⇒ ('i, 'pkt-ext) simple-packet-scheme ⇒ ('i simple-match × 'a) option
where
generalized-sfw l p ≡ find (λ(m,a). simple-matches m p) l

11.2 Lemmas

lemma *generalized-sfw-simps*:
generalized-sfw [] p = None
generalized-sfw (a # as) p = (if (case a of (m,-) ⇒ simple-matches m p) then Some a else generalized-sfw as p)
⟨*proof*⟩

lemma *generalized-sfw-append*:
generalized-sfw (a @ b) p = (case generalized-sfw a p of Some x ⇒ Some x | None ⇒ generalized-sfw b p)
⟨*proof*⟩

lemma *simple-generalized-undecided*:
simple-fw fw p ≠ Undecided ⟹ generalized-sfw (map simple-rule-dtor fw) p ≠ None
⟨*proof*⟩

lemma *generalized-sfwSomeD*: *generalized-sfw fw p = Some (r,d) ⟹ (r,d) ∈ set fw ∧ simple-matches r p*
⟨*proof*⟩

lemma *generalized-sfw-NoneD*: *generalized-sfw fw p = None ⟹ ∀ (a,b) ∈ set fw. ¬ simple-matches a p*
⟨*proof*⟩

lemma *generalized-fw-split*: *generalized-sfw fw p = Some r ⟹ ∃ fw1 fw3. fw = fw1 @ r # fw3 ∧ generalized-sfw fw1 p = None*
⟨*proof*⟩

lemma *generalized-sfw-filterD*:
generalized-sfw (filter f fw) p = Some (r,d) ⟹ simple-matches r p ∧ f (r,d)
⟨*proof*⟩

lemma *generalized-sfw-mapsnd*:
generalized-sfw (map (apsnd f) fw) p = map-option (apsnd f) (generalized-sfw fw p)
⟨*proof*⟩

11.3 Equality with the Simple Firewall

A matching action of the simple firewall directly corresponds to a filtering decision

definition *simple-action-to-decision* :: *simple-action ⇒ state* **where**

simple-action-to-decision a ≡ *case a of Accept* ⇒ *Decision FinalAllow*
| *Drop* ⇒ *Decision FinalDeny*

The *simple-fw* and the *generalized-sfw* are equal, if the state is translated appropriately.

lemma *simple-fw-iff-generalized-fw*:
simple-fw fw p = *simple-action-to-decision a* ⟷ (∃ *r*. *generalized-sfw* (*map simple-rule-dtor fw*) *p* = *Some* (*r*,*a*))
⟨*proof*⟩

lemma *simple-fw-iff-generalized-fw-accept*:
simple-fw fw p = *Decision FinalAllow* ⟷ (∃ *r*. *generalized-sfw* (*map simple-rule-dtor fw*) *p* = *Some* (*r*, *Accept*))
⟨*proof*⟩
lemma *simple-fw-iff-generalized-fw-drop*:
simple-fw fw p = *Decision FinalDeny* ⟷ (∃ *r*. *generalized-sfw* (*map simple-rule-dtor fw*) *p* = *Some* (*r*, *Drop*))
⟨*proof*⟩

11.4 Joining two firewalls, i.e. a packet is send through both sequentially.

definition *generalized-fw-join*
:: (*'i::len simple-match* × *'a*) *list* ⇒ (*'i simple-match* × *'b*) *list* ⇒ (*'i simple-match* × *'a* × *'b*) *list*
where
generalized-fw-join l1 l2 ≡ [(*u*,(*a*,*b*)). (*m1*,*a*) ← *l1*, (*m2*,*b*) ← *l2*, *u* ← *option2list* (*simple-match-and m1 m2*)]

lemma *generalized-fw-join-1-Nil*[*simp*]: *generalized-fw-join* [] *f2* = []
⟨*proof*⟩

lemma *generalized-fw-join-2-Nil*[*simp*]: *generalized-fw-join f1* [] = []
⟨*proof*⟩

lemma *generalized-fw-join-cons-1*:
generalized-fw-join ((*am*,*ad*) # *l1*) *l2* =
[(*u*,(*ad*,*b*)). (*m2*,*b*) ← *l2*, *u* ← *option2list* (*simple-match-and am m2*)] @ *generalized-fw-join l1 l2*
⟨*proof*⟩

lemma *generalized-fw-join-1-nomatch*:
¬ *simple-matches am p* ⟹
generalized-sfw [(*u*,(*ad*,*b*)). (*m2*,*b*) ← *l2*, *u* ← *option2list* (*simple-match-and am m2*)] *p* = *None*
⟨*proof*⟩

lemma *generalized-fw-join-2-nomatch*:
¬ *simple-matches bm p* ⟹

generalized-sfw (*generalized-fw-join as* ((*bm*, *bd*) # *bs*)) *p* = *generalized-sfw* (*generalized-fw-join as bs*) *p*
⟨*proof*⟩

lemma *generalized-fw-joinI*:
⟦*generalized-sfw f1 p* = *Some* (*r1*,*d1*); *generalized-sfw f2 p* = *Some* (*r2*,*d2*)⟧ ⟹
generalized-sfw (*generalized-fw-join f1 f2*) *p* = *Some* (*the* (*simple-match-and r1 r2*), *d1*,*d2*)
⟨*proof*⟩

lemma *generalized-fw-joinD*:
generalized-sfw (*generalized-fw-join f1 f2*) *p* = *Some* (*u*, *d1*,*d2*) ⟹
∃ *r1 r2*. *generalized-sfw f1 p* = *Some* (*r1*,*d1*) ∧ *generalized-sfw f2 p* = *Some* (*r2*,*d2*) ∧ *Some u* = *simple-match-and r1 r2*
⟨*proof*⟩

We imagine two firewalls are positioned directly after each other. The first one has ruleset rs1 installed, the second one has ruleset rs2 installed. A packet needs to pass both firewalls.

theorem *simple-fw-join*:
defines *rule-translate* ≡
map (λ(*u*,*a*,*b*). *SimpleRule u* (*if a* = *Accept* ∧ *b* = *Accept then Accept else Drop*))
shows
simple-fw rs1 p = *Decision FinalAllow* ∧ *simple-fw rs2 p* = *Decision FinalAllow* ⟷
simple-fw (*rule-translate* (*generalized-fw-join* (*map simple-rule-dtor rs1*) (*map simple-rule-dtor rs2*))) *p* = *Decision FinalAllow*
⟨*proof*⟩

theorem *simple-fw-join2*:
— translates a (*match*, *action1*, *action2*) tuple of the joined generalized firewall to a *'i simple-rule list*. The two actions are translated such that you only get *Accept* if both actions are *Accept*
defines *to-simple-rule-list* ≡ *map* (*apsnd* (λ(*a*,*b*) ⇒ (*case a of Accept* ⇒ *b*
| *Drop* ⇒ *Drop*)))
shows *simple-fw rs1 p* = *Decision FinalAllow* ∧ *simple-fw rs2 p* = *Decision FinalAllow* ⟷
(∃ *m*. (*generalized-sfw* (*to-simple-rule-list*
(*generalized-fw-join* (*map simple-rule-dtor rs1*) (*map simple-rule-dtor rs2*))) *p*) = *Some* (*m*, *Accept*))
⟨*proof*⟩

lemma *generalized-fw-join-1-1*:

generalized-fw-join [(*m1*,*d1*)] *fw2* = *foldr* (λ(*m2*,*d2*). (@) (*case simple-match-and m1 m2 of None* ⇒ [] | *Some mu* ⇒ [(*mu*,*d1*,*d2*)])) *fw2* []
⟨*proof*⟩

lemma *generalized-sfw-2-join-None*:
generalized-sfw fw2 p = *None* ⟹ *generalized-sfw* (*generalized-fw-join fw1 fw2*) *p* = *None*
⟨*proof*⟩

lemma *generalized-sfw-1-join-None*:
generalized-sfw fw1 p = *None* ⟹ *generalized-sfw* (*generalized-fw-join fw1 fw2*) *p* = *None*
⟨*proof*⟩

lemma *generalized-sfw-join-set*: (*a*, *b1*, *b2*) ∈ *set* (*generalized-fw-join f1 f2*) ⟷ (∃ *a1 a2*. (*a1*, *b1*) ∈ *set f1* ∧ (*a2*, *b2*) ∈ *set f2* ∧ *simple-match-and a1 a2* = *Some a*)
⟨*proof*⟩

11.5 Validity

There's validity of matches on *generalized-sfw*, too, even on the join.

definition *gsfw-valid* :: ('*i*::*len simple-match* × '*c*) *list* ⇒ *bool* **where**
gsfw-valid ≡ *list-all* (*simple-match-valid* ∘ *fst*)

lemma *gsfw-join-valid*: *gsfw-valid f1* ⟹ *gsfw-valid f2* ⟹ *gsfw-valid* (*generalized-fw-join f1 f2*)
⟨*proof*⟩

lemma *gsfw-validI*: *simple-fw-valid fw* ⟹ *gsfw-valid* (*map simple-rule-dtor fw*)
⟨*proof*⟩

end

12 Shadowed Rules

theory *Shadowed*
imports *SimpleFw-Semantics*
begin

12.1 Removing Shadowed Rules

Testing, not executable

Assumes: *simple-ruleset*

fun *rmshadow* :: *'i::len simple-rule list* ⇒ *'i simple-packet set* ⇒ *'i simple-rule list*
where
rmshadow [] - = [] |
rmshadow ((*SimpleRule m a*)#*rs*) *P* = (*if* (∀ *p*∈*P*. ¬ *simple-matches m p*)
then
rmshadow rs P
else
(*SimpleRule m a*) # (*rmshadow rs* {*p* ∈ *P*. ¬ *simple-matches m p*}))

12.1.1 Soundness

lemma *rmshadow-sound*:
p ∈ *P* ⟹ *simple-fw* (*rmshadow rs P*) *p* = *simple-fw rs p*
⟨*proof*⟩

corollary *rmshadow*:
fixes *p* :: *'i::len simple-packet*
shows *simple-fw* (*rmshadow rs UNIV*) *p* = *simple-fw rs p*
⟨*proof*⟩

A different approach where we start with the empty set of packets and collect packets which are already "matched-away".

fun *rmshadow'* :: *'i::len simple-rule list* ⇒ *'i simple-packet set* ⇒ *'i simple-rule list*
where
rmshadow' [] - = [] |
rmshadow' ((*SimpleRule m a*)#*rs*) *P* = (*if* {*p*. *simple-matches m p*} ⊆ *P*
then
rmshadow' rs P
else
(*SimpleRule m a*) # (*rmshadow' rs* (*P* ∪ {*p*. *simple-matches m p*})))

lemma *rmshadow'-sound*:
p ∉ *P* ⟹ *simple-fw* (*rmshadow' rs P*) *p* = *simple-fw rs p*
⟨*proof*⟩

corollary
fixes *p* :: *'i::len simple-packet*
shows *simple-fw* (*rmshadow rs UNIV*) *p* = *simple-fw* (*rmshadow' rs* {}) *p*
⟨*proof*⟩

Previous algorithm is not executable because we have no code for *'i simple-packet set*. To get some code, some efficient set operations would be necessary. We either need union and subset or intersection and negation. Both subset and negation are complicated. Probably the BDDs which related work uses is really necessary.

context
begin
private type-synonym *'i simple-packet-set* = *'i simple-match list*

private definition *simple-packet-set-toSet* :: $'i$::*len simple-packet-set* ⇒ $'i$ *simple-packet set* **where**
simple-packet-set-toSet ms = {*p*. ∃ *m* ∈ *set ms*. *simple-matches m p*}

private lemma *simple-packet-set-toSet-alt*: *simple-packet-set-toSet ms* = (⋃ *m* ∈ *set ms*. {*p*. *simple-matches m p*})
⟨*proof*⟩ **definition** *simple-packet-set-union* :: $'i$::*len simple-packet-set* ⇒ $'i$ *simple-match* ⇒ $'i$ *simple-packet-set* **where**
simple-packet-set-union ps m = *m* # *ps*

private lemma *simple-packet-set-toSet* (*simple-packet-set-union ps m*) = *simple-packet-set-toSet ps* ∪ {*p*. *simple-matches m p*}
⟨*proof*⟩ **lemma** (∃ *m'* ∈ *set ms*.
{*i*. *match-iface iif i*} ⊆ {*i*. *match-iface* (*iiface m'*) *i*} ∧
{*i*. *match-iface oif i*} ⊆ {*i*. *match-iface* (*oiface m'*) *i*} ∧
{*ip*. *simple-match-ip sip ip*} ⊆ {*ip*. *simple-match-ip* (*src m'*) *ip*} ∧
{*ip*. *simple-match-ip dip ip*} ⊆ {*ip*. *simple-match-ip* (*dst m'*) *ip*} ∧
{*p*. *match-proto protocol p*} ⊆ {*p*. *match-proto* (*proto m'*) *p*} ∧
{*p*. *simple-match-port sps p*} ⊆ {*p*. *simple-match-port* (*sports m'*) *p*} ∧
{*p*. *simple-match-port dps p*} ⊆ {*p*. *simple-match-port* (*dports m'*) *p*}
)
⟹ {*p*. *simple-matches* (|*iiface*=*iif*, *oiface*=*oif*, *src*=*sip*, *dst*=*dip*, *proto*=*protocol*, *sports*=*sps*, *dports*=*dps* |) *p*} ⊆ (*simple-packet-set-toSet ms*)
⟨*proof*⟩

subset or negation ... One efficient implementation would suffice.

private lemma {*p*:: $'i$::*len simple-packet*. *simple-matches m p*} ⊆ (*simple-packet-set-toSet ms*) ⟷
{*p*:: $'i$::*len simple-packet*. *simple-matches m p*} ∩ (⋂ *m* ∈ *set ms*. {*p*. ¬ *simple-matches m p*}) = {} (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

end
end

13 Partition a Set by a Specific Constraint

theory *IP-Partition-Preliminaries*
imports *Main*
begin

Will be used for the IP address space partition of a firewall. However, this file is completely generic in terms of sets, it only imports Main.

It will be used in `../Service_Matrix.thy`. Core idea: This file partitions $'a$ *set set* by some magic condition. Later, we will show that this magic condition implies that all IPs that have been grouped by the magic condition show the same behaviour for a simple firewall.

definition *disjoint* :: *'a set set* ⇒ *bool* **where**
disjoint ts ≡ ∀ *A* ∈ *ts*. ∀ *B* ∈ *ts*. *A* ≠ *B* ⟶ *A* ∩ *B* = {} *We will call two partitioned sets* complete *iff* ⋃ *ss* = ⋃ *ts*.

The condition we use to partition a set. If this holds and A is the set of IP addresses in each rule in a firewall, then B is a partition of $\bigcup A$ where each member has the same behavior w.r.t the firewall ruleset.

A is the carrier set and B* should be a partition of $\bigcup A$ which fulfills the following condition:

definition *ipPartition* :: *'a set set* ⇒ *'a set set* ⇒ *bool* **where**
ipPartition A B ≡ ∀ *a* ∈ *A*. ∀ *b* ∈ *B*. *a* ∩ *b* = {} ∨ *b* ⊆ *a*

definition *disjoint-list* :: *'a set list* ⇒ *bool* **where**
disjoint-list ls ≡ *distinct ls* ∧ *disjoint* (*set ls*)

context begin

private fun *disjoint-list-rec* :: *'a set list* ⇒ *bool* **where**
disjoint-list-rec [] = *True* |
disjoint-list-rec (*x#xs*) = (*x* ∩ ⋃(*set xs*) = {} ∧ *disjoint-list-rec xs*)

private lemma *disjoint-equi*: *disjoint-list-rec ts* ⟹ *disjoint* (*set ts*)
⟨*proof*⟩ **lemma** *disjoint-list-disjoint-list-rec*: *disjoint-list ts* ⟹ *disjoint-list-rec ts*
⟨*proof*⟩ **definition** *addSubsetSet* :: *'a set* ⇒ *'a set set* ⇒ *'a set set* **where**
addSubsetSet s ts = *insert* (*s* − ⋃*ts*) (((∩) *s*) ' *ts*) ∪ ((λ*x*. *x* − *s*) ' *ts*)

private fun *partitioning* :: *'a set list* ⇒ *'a set set* ⇒ *'a set set* **where**
partitioning [] *ts* = *ts* |
partitioning (*s#ss*) *ts* = *partitioning ss* (*addSubsetSet s ts*)

simple examples

lemma *partitioning* [{*1*::*nat*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}] {} = {{*10*}, {*6*}, {*5*, *7*}, {}, {*3*, *4*}, {*1*, *2*}} ⟨*proof*⟩
lemma ⋃ {{*1*::*nat*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}} = ⋃ (*partitioning* [{*1*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}] {}) ⟨*proof*⟩
lemma *disjoint* (*partitioning* [{*1*::*nat*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}] {}) ⟨*proof*⟩
lemma *ipPartition* {{*1*::*nat*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}} (*partitioning* [{*1*::*nat*,*2*},{*3*,*4*},{*5*,*6*,*7*},{*6*},{*10*}] {}) ⟨*proof*⟩

lemma *ipPartition A* {} ⟨*proof*⟩

lemma *ipPartitionUnion*: *ipPartition As Cs* ∧ *ipPartition Bs Cs* ⟷ *ipPartition* (*As* ∪ *Bs*) *Cs*
⟨*proof*⟩ **lemma** *disjointAddSubset*: *disjoint ts* ⟹ *disjoint* (*addSubsetSet a ts*)
⟨*proof*⟩ **lemma** *coversallAddSubset*:⋃ (*insert a ts*) = ⋃ (*addSubsetSet a ts*)
⟨*proof*⟩ **lemma** *ipPartioningAddSubset0*: *disjoint ts* ⟹ *ipPartition ts* (*addSubsetSet a ts*)

⟨*proof*⟩ **lemma** *ipPartitioningAddSubset1*: *disjoint ts* $\Longrightarrow$ *ipPartition* (*insert a ts*) (*addSubsetSet a ts*)
⟨*proof*⟩ **lemma** *addSubsetSetI*:
$s - \bigcup ts \in addSubsetSet\ s\ ts$
$t \in ts \Longrightarrow s \cap t \in addSubsetSet\ s\ ts$
$t \in ts \Longrightarrow t - s \in addSubsetSet\ s\ ts$
⟨*proof*⟩ **lemma** *addSubsetSetE*:
assumes $A \in addSubsetSet\ s\ ts$
obtains $A = s - \bigcup ts$ | T **where** $T \in ts$ $A = s \cap T$ | T **where** $T \in ts$ $A = T - s$
⟨*proof*⟩ **lemma** *Union-addSubsetSet*: $\bigcup(addSubsetSet\ b\ As) = b \cup \bigcup As$
⟨*proof*⟩ **lemma** *addSubsetSetCom*: *addSubsetSet a* (*addSubsetSet b As*) = *addSubsetSet b* (*addSubsetSet a As*)
⟨*proof*⟩ **lemma** *ipPartitioningAddSubset2*: *ipPartition* $\{a\}$ (*addSubsetSet a ts*)
⟨*proof*⟩ **lemma** *disjointPartitioning-helper* :*disjoint As* $\Longrightarrow$ *disjoint* (*partitioning ss As*)
⟨*proof*⟩ **lemma** *disjointPartitioning*: *disjoint* (*partitioning ss* $\{\}$)
⟨*proof*⟩ **lemma** *coversallPartitioning*:$\bigcup$ (*set ts*) = $\bigcup$ (*partitioning ts* $\{\}$)
⟨*proof*⟩ **lemma** $\bigcup$ *As* = $\bigcup$ *Bs* $\Longrightarrow$ *ipPartition As Bs* $\Longrightarrow$ *ipPartition As* (*addSubsetSet a Bs*)
⟨*proof*⟩ **lemma** *ipPartitionSingleSet*: *ipPartition* $\{t\}$ (*addSubsetSet t Bs*)
$\Longrightarrow$ *ipPartition* $\{t\}$ (*partitioning ts* (*addSubsetSet t Bs*))
⟨*proof*⟩ **lemma** *ipPartitioning-helper*: *disjoint As* $\Longrightarrow$ *ipPartition* (*set ts*) (*partitioning ts As*)
⟨*proof*⟩ **lemma** *ipPartitioning*: *ipPartition* (*set ts*) (*partitioning ts* $\{\}$)
⟨*proof*⟩ **lemma** *inter-dif-help-lemma*: $A \cap B = \{\} \Longrightarrow B - S = B - (S - A)$
⟨*proof*⟩ **lemma** *disjoint-list-lem*: *disjoint-list ls* $\Longrightarrow \forall s \in set(ls).\ \forall t \in set(ls).\ s \neq t \longrightarrow s \cap t = \{\}$
⟨*proof*⟩ **lemma** *disjoint-list-empty*: *disjoint-list* []
⟨*proof*⟩ **lemma** *disjoint-sublist*: *disjoint-list* (*t#ts*) $\Longrightarrow$ *disjoint-list ts*
⟨*proof*⟩ **fun** *intersection-list* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
intersection-list - [] = [] |
intersection-list s (*t#ts*) = $(s \cap t)$#(*intersection-list s ts*)

private fun *intersection-list-opt* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
intersection-list-opt - [] = [] |
intersection-list-opt s (*t#ts*) = $(s \cap t)$#(*intersection-list-opt* $(s - t)$ *ts*)

private lemma *disjoint-subset*: *disjoint A* $\Longrightarrow a \in A \Longrightarrow b \subseteq a \Longrightarrow$ *disjoint* $((A - \{a\}) \cup \{b\})$
⟨*proof*⟩ **lemma** *disjoint-intersection*: *disjoint A* $\Longrightarrow a \in A \Longrightarrow$ *disjoint* $(\{a \cap b\} \cup (A - \{a\}))$
⟨*proof*⟩ **lemma** *intList-equi*: *disjoint-list-rec ts* $\Longrightarrow$ *intersection-list s ts* = *intersection-list-opt s ts*
⟨*proof*⟩ **fun** *difference-list* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
difference-list - [] = [] |
difference-list s (*t#ts*) = $(t - s)$#(*difference-list s ts*)

private fun *difference-list-opt* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**

difference-list-opt - [] = [] |
difference-list-opt s (*t*#*ts*) = (*t* − *s*)#(*difference-list-opt* (*s* − *t*) *ts*)

private lemma *difList-equi*: *disjoint-list-rec ts* ⟹ *difference-list s ts* = *difference-list-opt s ts*
⟨*proof*⟩ **fun** *partList0* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
partList0 s [] = [] |
partList0 s (*t*#*ts*) = (*s* ∩ *t*)#((*t* − *s*)#(*partList0 s ts*))

private lemma *partList0-set-equi*: *set*(*partList0 s ts*) = (((∩) *s*) ' (*set ts*)) ∪ ((λ*x*. *x* − *s*) ' (*set ts*))
⟨*proof*⟩ **lemma** *partList-sub-equi0*: *set*(*partList0 s ts*) =
set(*difference-list s ts*) ∪ *set*(*intersection-list s ts*)
⟨*proof*⟩ **fun** *partList1* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
partList1 s [] = [] |
partList1 s (*t*#*ts*) = (*s* ∩ *t*)#((*t* − *s*)#(*partList1* (*s* − *t*) *ts*))

private lemma *partList-sub-equi*: *set*(*partList1 s ts*) =
set(*difference-list-opt s ts*) ∪ *set*(*intersection-list-opt s ts*)
⟨*proof*⟩ **lemma** *partList0-partList1-equi*: *disjoint-list-rec ts* ⟹ *set* (*partList0 s ts*) = *set* (*partList1 s ts*)
⟨*proof*⟩ **fun** *partList2* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
partList2 s [] = [] |
partList2 s (*t*#*ts*) = (*if s* ∩ *t* = {} *then* (*t*#(*partList2* (*s* − *t*) *ts*))
else (*s* ∩ *t*)#((*t* − *s*)#(*partList2* (*s* − *t*) *ts*)))

private lemma *partList2-empty*: *partList2* {} *ts* = *ts*
⟨*proof*⟩ **lemma** *partList1-partList2-equi*: *set*(*partList1 s ts*) − {{}} = *set*(*partList2 s ts*) − {{}}
⟨*proof*⟩ **fun** *partList3* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
partList3 s [] = [] |
partList3 s (*t*#*ts*) = (*if s* = {} *then* (*t*#*ts*) *else*
(*if s* ∩ *t* = {} *then* (*t*#(*partList3* (*s* − *t*) *ts*))
else
(*if t* − *s* = {} *then* (*t*#(*partList3* (*s* − *t*) *ts*))
else (*t* ∩ *s*)#((*t* − *s*)#(*partList3* (*s* − *t*) *ts*)))))

private lemma *partList2-partList3-equi*: *set*(*partList2 s ts*) − {{}} = *set*(*partList3 s ts*) − {{}}
⟨*proof*⟩

fun *partList4* :: *'a set* ⇒ *'a set list* ⇒ *'a set list* **where**
partList4 s [] = [] |
partList4 s (*t*#*ts*) = (*if s* = {} *then* (*t*#*ts*) *else*
(*if s* ∩ *t* = {} *then* (*t*#(*partList4 s ts*))
else
(*if t* − *s* = {} *then* (*t*#(*partList4* (*s* − *t*) *ts*))
else (*t* ∩ *s*)#((*t* − *s*)#(*partList4* (*s* − *t*) *ts*)))))

private lemma *partList4*: *partList4 s ts* = *partList3 s ts*
⟨*proof*⟩ **lemma** *partList0-addSubsetSet-equi*: $s \subseteq \bigcup(set\ ts) \Longrightarrow$
$addSubsetSet\ s\ (set\ ts) - \{\{\}\} = set(partList0\ s\ ts) - \{\{\}\}$
⟨*proof*⟩ **fun** *partitioning-nontail* :: *'a set list* ⇒ *'a set set* ⇒ *'a set set* **where**
partitioning-nontail [] *ts* = *ts* |
partitioning-nontail (*s#ss*) *ts* = *addSubsetSet s* (*partitioning-nontail ss ts*)

private lemma *partitioningCom*: *addSubsetSet a* (*partitioning ss ts*) = *partitioning ss* (*addSubsetSet a ts*)
⟨*proof*⟩ **lemma** *partitioning-nottail-equi*: *partitioning-nontail ss ts* = *partitioning ss ts*
⟨*proof*⟩

fun *partitioning1* :: *'a set list* ⇒ *'a set list* ⇒ *'a set list* **where**
partitioning1 [] *ts* = *ts* |
partitioning1 (*s#ss*) *ts* = *partList4 s* (*partitioning1 ss ts*)

lemma *partList4-empty*: $\{\} \notin set\ ts \Longrightarrow \{\} \notin set\ (partList4\ s\ ts)$
⟨*proof*⟩

lemma *partitioning1-empty0*: $\{\} \notin set\ ts \Longrightarrow \{\} \notin set\ (partitioning1\ ss\ ts)$
⟨*proof*⟩

lemma *partitioning1-empty1*: $\{\} \notin set\ ts \Longrightarrow$
$set(partitioning1\ ss\ ts) - \{\{\}\} = set(partitioning1\ ss\ ts)$
⟨*proof*⟩

lemma *partList4-subset*: $a \subseteq \bigcup(set\ ts) \Longrightarrow a \subseteq \bigcup(set\ (partList4\ b\ ts))$
⟨*proof*⟩ **lemma** $a \neq \{\} \Longrightarrow disjoint\text{-}list\text{-}rec\ (a\ \#\ ts) \longleftrightarrow disjoint\text{-}list\text{-}rec\ ts \land a \cap \bigcup\ (set\ ts) = \{\}$ ⟨*proof*⟩

lemma *partList4-complete0*: $s \subseteq \bigcup(set\ ts) \Longrightarrow \bigcup(set\ (partList4\ s\ ts)) = \bigcup(set\ ts)$
⟨*proof*⟩ **lemma** *partList4-disjoint*: $s \subseteq \bigcup(set\ ts) \Longrightarrow disjoint\text{-}list\text{-}rec\ ts \Longrightarrow$
$disjoint\text{-}list\text{-}rec\ (partList4\ s\ ts)$
⟨*proof*⟩

lemma *union-set-partList4*: $\bigcup(set\ (partList4\ s\ ts)) = \bigcup(set\ ts)$
⟨*proof*⟩ **lemma** *partList4-distinct-hlp*: **assumes** $a \neq \{\}$ $a \notin set\ ts$ *disjoint* (*insert a* (*set ts*))
shows $a \notin set\ (partList4\ s\ ts)$
⟨*proof*⟩ **lemma** *partList4-distinct*: $\{\} \notin set\ ts \Longrightarrow disjoint\text{-}list\ ts \Longrightarrow distinct\ (partList4\ s\ ts)$
⟨*proof*⟩

lemma *partList4-disjoint-list*: **assumes** $s \subseteq \bigcup(set\ ts)$ *disjoint-list ts* $\{\} \notin set\ ts$
shows *disjoint-list* (*partList4 s ts*)

⟨*proof*⟩

lemma *partitioning1-subset*: $a \subseteq \bigcup$ (*set ts*) $\Longrightarrow$ $a \subseteq \bigcup$ (*set* (*partitioning1 ss ts*))
⟨*proof*⟩

lemma *partitioning1-disjoint-list*: $\{\} \notin$ (*set ts*) $\Longrightarrow \bigcup$ (*set ss*) $\subseteq \bigcup$ (*set ts*) $\Longrightarrow$
disjoint-list ts $\Longrightarrow$ *disjoint-list* (*partitioning1 ss ts*)
⟨*proof*⟩ **lemma** *partitioning1-disjoint*: $\bigcup$ (*set ss*) $\subseteq \bigcup$ (*set ts*) $\Longrightarrow$
disjoint-list-rec ts $\Longrightarrow$ *disjoint-list-rec* (*partitioning1 ss ts*)
⟨*proof*⟩ **lemma** *partitioning-equi*: $\{\} \notin$ *set ts* $\Longrightarrow$ *disjoint-list-rec ts* $\Longrightarrow \bigcup$ (*set ss*) $\subseteq \bigcup$ (*set ts*) $\Longrightarrow$
set(*partitioning1 ss ts*) = *partitioning-nontail ss* (*set ts*) $-$ $\{\{\}\}$
⟨*proof*⟩

lemma *ipPartitioning-helper-opt*: $\{\} \notin$ *set ts* $\Longrightarrow$ *disjoint-list ts* $\Longrightarrow \bigcup$ (*set ss*) $\subseteq \bigcup$ (*set ts*)
$\Longrightarrow$ *ipPartition* (*set ss*) (*set* (*partitioning1 ss ts*))
⟨*proof*⟩

lemma *complete-helper*: $\{\} \notin$ *set ts* $\Longrightarrow \bigcup$ (*set ss*) $\subseteq \bigcup$ (*set ts*)$\Longrightarrow$
$\bigcup$ (*set ts*) = $\bigcup$ (*set* (*partitioning1 ss ts*))
⟨*proof*⟩

lemma *partitioning1* [{*1*::*nat*},{*2*},{}] [{*1*},{},{*2*},{*3*}] = [{*1*}, {}, {*2*}, {*3*}]
⟨*proof*⟩

lemma *partitioning-foldr*: *partitioning X B* = *foldr addSubsetSet X B*
⟨*proof*⟩

lemma *ipPartition* (*set X*) (*foldr addSubsetSet X* {})
⟨*proof*⟩

lemma $\bigcup$ (*set X*) = $\bigcup$ (*foldr addSubsetSet X* {})
⟨*proof*⟩

lemma *partitioning1 X B* = *foldr partList4 X B*
⟨*proof*⟩

lemma *ipPartition* (*set X*) (*set* (*partitioning1 X* [*UNIV*]))
⟨*proof*⟩

lemma ($\bigcup$(*set* (*partitioning1 X* [*UNIV*]))) = *UNIV*
⟨*proof*⟩

end
end

14 Group by Function

theory *GroupF*
imports *Main*
begin

Grouping elements of a list according to a function.

fun *groupF* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'a\ list\ list$ **where**
groupF f [] = [] |
groupF f (x#xs) = (x#(*filter* (λy. f x = f y) xs))#(*groupF f* (*filter* (λy. f x ≠ f y) xs))

trying a more efficient implementation of *groupF*

context
begin
private fun *select-p-tuple* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow ('a\ list \times 'a\ list) \Rightarrow ('a\ list \times 'a\ list)$
where
select-p-tuple p x (ts,fs) = (if p x then (x#ts, fs) else (ts, x#fs))

private definition *partition-tailrec* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow ('a\ list \times 'a\ list)$
where
partition-tailrec p xs = foldr (select-p-tuple p) xs ([],[])

private lemma *partition-tailrec*: *partition-tailrec f as =* (*filter f as*, *filter* (λx. ¬f x) *as*)
⟨*proof*⟩ **lemma**
groupF f (x#xs) = (let (ts, fs) = partition-tailrec (λy. f x = f y) xs in (x#ts)#(groupF f fs))
⟨*proof*⟩ **function** *groupF-code* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'a\ list\ list$ **where**
groupF-code f [] = [] |
groupF-code f (x#xs) = (*let*
(ts, fs) = *partition-tailrec* (λy. f x = f y) xs
in
(x#ts)#(*groupF-code f fs*))
⟨*proof*⟩ **termination** *groupF-code*
⟨*proof*⟩

lemma *groupF-code*[*code*]: *groupF f as = groupF-code f as*
⟨*proof*⟩

export-code *groupF* **checking** *SML*
end

lemma *groupF-concat-set*: *set (concat (groupF f xs)) = set xs*
⟨*proof*⟩

lemma *groupF-Union-set*: $(\bigcup x \in set\ (groupF\ f\ xs).\ set\ x) = set\ xs$
⟨*proof*⟩

lemma *groupF-set*: $\forall X \in set\ (groupF\ f\ xs).\ \forall x \in set\ X.\ x \in set\ xs$
⟨*proof*⟩

lemma *groupF-equality*:
defines $same\ f\ A \equiv \forall a1 \in set\ A.\ \forall a2 \in set\ A.\ f\ a1 = f\ a2$
shows $\forall A \in set\ (groupF\ f\ xs).\ same\ f\ A$
⟨*proof*⟩

lemma *groupF-nequality*: $A \in set\ (groupF\ f\ xs) \Longrightarrow B \in set\ (groupF\ f\ xs) \Longrightarrow A \neq B \Longrightarrow$
$\forall a \in set\ A.\ \forall b \in set\ B.\ f\ a \neq f\ b$
⟨*proof*⟩

lemma *groupF-cong*: **fixes** $xs{::}'a\ list$ **and** $f1{::}'a \Rightarrow 'b$ **and** $f2{::}'a \Rightarrow 'c$
assumes $\forall x \in set\ xs.\ \forall y \in set\ xs.\ (f1\ x = f1\ y \longleftrightarrow f2\ x = f2\ y)$
shows $groupF\ f1\ xs = groupF\ f2\ xs$
⟨*proof*⟩

lemma *groupF-empty*: $groupF\ f\ xs \neq [] \longleftrightarrow xs \neq []$
⟨*proof*⟩
lemma *groupF-empty-elem*: $x \in set\ (groupF\ f\ xs) \Longrightarrow x \neq []$
⟨*proof*⟩

lemma *groupF-distinct*: $distinct\ xs \Longrightarrow distinct\ (concat\ (groupF\ f\ xs))$
⟨*proof*⟩

It is possible to use $map\ (map\ fst)\ (groupF\ snd\ (map\ (\lambda x.\ (x,\ f\ x))\ P))$ instead of $groupF\ f\ P$ for the following reasons: *groupF* executes its compare function (first parameter) very often; it always tests for $f\ x = f\ y$. The function f may be really expensive. At least polyML does not share the result of f but (probably) always recomputes (part of) it. The optimization pre-computes f and tells *groupF* to use a really cheap function (*snd*) to compare. The following lemma tells that those are equal.

lemma *groupF-tuple*: $groupF\ f\ xs = map\ (map\ fst)\ (groupF\ snd\ (map\ (\lambda x.\ (x,\ f\ x))\ xs))$
⟨*proof*⟩
end

15 Helper: Pretty Printing Word Intervals which correspond to IP address Ranges

theory *IP-Addr-WordInterval-toString*
imports *IP-Addresses.IP-Address-toString*
begin

fun *ipv4addr-wordinterval-toString* :: *32 wordinterval* ⇒ *string* **where**
ipv4addr-wordinterval-toString (*WordInterval s e*) =
(*if s = e then ipv4addr-toString s else ''{''@ipv4addr-toString s@'' .. ''@ipv4addr-toString e@''}''*) |
ipv4addr-wordinterval-toString (*RangeUnion a b*) =
ipv4addr-wordinterval-toString a @ '' u ''@ipv4addr-wordinterval-toString b

fun *ipv6addr-wordinterval-toString* :: *128 wordinterval* ⇒ *string* **where**
ipv6addr-wordinterval-toString (*WordInterval s e*) =
(*if s = e then ipv6addr-toString s else ''{''@ipv6addr-toString s@'' .. ''@ipv6addr-toString e@''}''*) |
ipv6addr-wordinterval-toString (*RangeUnion a b*) =
ipv6addr-wordinterval-toString a @ '' u ''@ipv6addr-wordinterval-toString b

end

16 toString Functions for Primitives

theory *Primitives-toString*
imports ../*Common*/*Lib-Enum-toString*
IP-Addresses.IP-Address-toString
Iface
L4-Protocol
begin

definition *ipv4-cidr-toString* :: (*ipv4addr* × *nat*) ⇒ *string* **where**
ipv4-cidr-toString ip-n = (*case ip-n of* (*base*, *n*) ⇒ (*ipv4addr-toString base* @''/''@ *string-of-nat n*))
lemma *ipv4-cidr-toString* (*ipv4addr-of-dotdecimal* (*192,168,0,1*), *22*) = *''192.168.0.1/22''*
⟨*proof*⟩

definition *ipv6-cidr-toString* :: (*ipv6addr* × *nat*) ⇒ *string* **where**
ipv6-cidr-toString ip-n = (*case ip-n of* (*base*, *n*) ⇒ (*ipv6addr-toString base* @''/''@ *string-of-nat n*))
lemma *ipv6-cidr-toString* (*42540766411282592856906245548098208122*, *64*) = *''2001:db8::8:800:200c:417a/6*
⟨*proof*⟩

definition *primitive-protocol-toString* :: *primitive-protocol* ⇒ *string* **where**
primitive-protocol-toString protid ≡ (
if protid = TCP then ''tcp'' else
if protid = UDP then ''udp'' else
if protid = ICMP then ''icmp'' else
if protid = L4-Protocol.SCTP then ''sctp'' else
if protid = L4-Protocol.IGMP then ''igmp'' else
if protid = L4-Protocol.GRE then ''gre'' else
if protid = L4-Protocol.ESP then ''esp'' else
if protid = L4-Protocol.AH then ''ah'' else

if protid = L4-Protocol.IPv6ICMP then ''ipv6−icmp'' else ''protocolid:''@dec-string-of-word0 protid)

fun *protocol-toString* :: *protocol* ⇒ *string* **where**
protocol-toString (*ProtoAny*) = *''all''* |
protocol-toString (*Proto protid*) = *primitive-protocol-toString protid*

definition *iface-toString* :: *string* ⇒ *iface* ⇒ *string* **where**
iface-toString descr iface = (*if iface = ifaceAny then '''' else* (*case iface of* (*Iface name*) ⇒ *descr@name*))
lemma *iface-toString ''in: ''* (*Iface ''+''*) = *''''* ⟨*proof*⟩
lemma *iface-toString ''in: ''* (*Iface ''eth0''*) = *''in: eth0''* ⟨*proof*⟩

definition *port-toString* :: *16 word* ⇒ *string* **where**
port-toString p ≡ *dec-string-of-word0 p*

fun *ports-toString* :: *string* ⇒ (*16 word* × *16 word*) ⇒ *string* **where**
ports-toString descr (*s*,*e*) = (*if s = 0* ∧ *e = max-word then '''' else descr @* (*if s=e then port-toString s else port-toString s@'':''@port-toString e*))
lemma *ports-toString ''spt: ''* (*0*,*65535*) = *''''* ⟨*proof*⟩
lemma *ports-toString ''spt: ''* (*1024*,*2048*) = *''spt: 1024:2048''* ⟨*proof*⟩
lemma *ports-toString ''spt: ''* (*1024*,*1024*) = *''spt: 1024''* ⟨*proof*⟩

definition *ipv4-cidr-opt-toString* :: *string* ⇒ *ipv4addr* × *nat* ⇒ *string* **where**
ipv4-cidr-opt-toString descr ip = (*if ip* = (*0*,*0*) *then '''' else descr@ipv4-cidr-toString ip*)

definition *protocol-opt-toString* :: *string* ⇒ *protocol* ⇒ *string* **where**
protocol-opt-toString descr prot = (*if prot = ProtoAny then '''' else descr@protocol-toString prot*)

end

17 Service Matrices

theory *Service-Matrix*
imports *Common/List-Product-More*
Common/IP-Partition-Preliminaries
Common/GroupF
Common/IP-Addr-WordInterval-toString
Primitives/Primitives-toString
SimpleFw-Semantics
IP-Addresses.WordInterval-Sorted
begin

17.1 IP Address Space Partition

fun *extract-IPSets-generic0*
:: (*'i::len simple-match* ⇒ *'i word* × *nat*) ⇒ *'i simple-rule list* ⇒ (*'i wordinterval*)

list
  where
  extract-IPSets-generic0 - [] = [] |
  extract-IPSets-generic0 sel ((SimpleRule m -)#ss) = (ipcidr-tuple-to-wordinterval (sel m)) #
  (extract-IPSets-generic0 sel ss)

lemma *extract-IPSets-generic0-length*: *length (extract-IPSets-generic0 sel rs) = length rs*
  ⟨*proof*⟩

lemma *mergesort-remdups* [(*1*::*ipv4addr*, *2*::*nat*), (*8*,*0*), (*8*,*1*), (*2*,*2*), (*2*,*4*), (*1*,*2*), (*2*,*2*)] =
  [(*1*, *2*), (*2*, *2*), (*2*, *4*), (*8*, *0*), (*8*, *1*)] ⟨*proof*⟩

fun *extract-src-dst-ips*
  :: *'i*::*len simple-rule list* ⇒ (*'i word* × *nat*) *list* ⇒ (*'i word* × *nat*) *list* **where**
  extract-src-dst-ips [] *ts* = *ts* |
  extract-src-dst-ips ((*SimpleRule m* -)#*ss*) *ts* = *extract-src-dst-ips ss* (*src m* # *dst m* # *ts*)

lemma *extract-src-dst-ips-length*: *length* (*extract-src-dst-ips rs acc*) = *2*∗*length rs* + *length acc*
⟨*proof*⟩

definition *extract-IPSets*
  :: *'i*::*len simple-rule list* ⇒ (*'i wordinterval*) *list* **where**
  extract-IPSets rs ≡ *map ipcidr-tuple-to-wordinterval* (*mergesort-remdups* (*extract-src-dst-ips rs* []))
lemma *extract-IPSets*:
  set (*extract-IPSets rs*) = *set* (*extract-IPSets-generic0 src rs*) ∪ *set* (*extract-IPSets-generic0 dst rs*)
⟨*proof*⟩

lemma (*a*::*nat*) *div 2* + *a mod 2* ≤ *a* ⟨*proof*⟩

lemma *merge-length*: *length* (*merge l1 l2*) ≤ *length l1* + *length l2*
⟨*proof*⟩

lemma *merge-list-length*: *length* (*merge-list as ls*) ≤ *length* (*concat* (*as* @ *ls*))
⟨*proof*⟩

lemma *mergesort-remdups-length*: *length* (*mergesort-remdups as*) ≤ *length as*

⟨*proof*⟩

lemma *extract-IPSets-length*: *length* (*extract-IPSets rs*) ≤ *2* ∗ *length rs*
⟨*proof*⟩

lemma *extract-equi0*:
set (*map wordinterval-to-set* (*extract-IPSets-generic0 sel rs*)) =
(λ(*base*,*len*). *ipset-from-cidr base len*) ' *sel* ' *match-sel* ' *set rs*
⟨*proof*⟩

lemma *src-ipPart-motivation*:
fixes *rs*
defines $X \equiv$ (λ(*base*,*len*). *ipset-from-cidr base len*) ' *src* ' *match-sel* ' *set rs*
assumes $\forall A \in X.\ B \subseteq A \vee B \cap A = \{\}$ **and** $s1 \in B$ **and** $s2 \in B$
shows *simple-fw rs* (*p*(|*p-src*:=*s1*|)) = *simple-fw rs* (*p*(|*p-src*:=*s2*|))
⟨*proof*⟩

lemma *src-ipPart*:
assumes *ipPartition* (*set* (*map wordinterval-to-set* (*extract-IPSets-generic0 src rs*))) *A*
$B \in A$ $s1 \in B$ $s2 \in B$
shows *simple-fw rs* (*p*(|*p-src*:=*s1*|)) = *simple-fw rs* (*p*(|*p-src*:=*s2*|))
⟨*proof*⟩

lemma *dst-ipPart*:
assumes *ipPartition* (*set* (*map wordinterval-to-set* (*extract-IPSets-generic0 dst rs*))) *A*
$B \in A$ $s1 \in B$ $s2 \in B$
shows *simple-fw rs* (*p*(|*p-dst*:=*s1*|)) = *simple-fw rs* (*p*(|*p-dst*:=*s2*|))
⟨*proof*⟩

definition *wordinterval-list-to-set* :: *'a*::*len wordinterval list* ⇒ *'a*::*len word set*
where
wordinterval-list-to-set ws = ⋃(*set* (*map wordinterval-to-set ws*))

lemma *wordinterval-list-to-set-compressed*:
wordinterval-to-set (*wordinterval-compress* (*foldr wordinterval-union xs Empty-WordInterval*)) =
wordinterval-list-to-set xs

⟨proof⟩

fun *partIps* :: *'a::len wordinterval* ⇒ *'a::len wordinterval list*
⇒ *'a::len wordinterval list* **where**
partIps - [] = [] |
partIps s (t#ts) = (if wordinterval-empty s then (t#ts) else
(if wordinterval-empty (wordinterval-intersection s t)
then (t#(partIps s ts))
else
(if wordinterval-empty (wordinterval-setminus t s)
then (t#(partIps (wordinterval-setminus s t) ts))
else (wordinterval-intersection t s)#((wordinterval-setminus t s)#
(partIps (wordinterval-setminus s t) ts)))))

lemma *partIps (WordInterval (1::ipv4addr) 1) [WordInterval 0 1] = [WordInterval 1 1, WordInterval 0 0]* ⟨proof⟩

lemma *partIps-length*: *length (partIps s ts)* $\leq$ *(length ts)* $*$ *2*
⟨proof⟩

fun *partitioningIps* :: *'a::len wordinterval list* ⇒ *'a::len wordinterval list* ⇒
'a::len wordinterval list **where**
partitioningIps [] ts = ts |
partitioningIps (s#ss) ts = partIps s (partitioningIps ss ts)

lemma *partitioningIps-length*: *length (partitioningIps ss ts)* $\leq$ *(2^length ss)* $*$ *length ts*
⟨proof⟩

lemma *partIps-equi*: *map wordinterval-to-set (partIps s ts) =*
partList4 (wordinterval-to-set s) (map wordinterval-to-set ts)
⟨proof⟩

lemma *partitioningIps-equi*: *map wordinterval-to-set (partitioningIps ss ts)*
= (partitioning1 (map wordinterval-to-set ss) (map wordinterval-to-set ts))
⟨proof⟩

definition *getParts* :: *'i::len simple-rule list* ⇒ *'i wordinterval list* **where**
getParts rs = partitioningIps (extract-IPSets rs) [wordinterval-UNIV]

lemma *partitioningIps-foldr*: *partitioningIps ss ts = foldr partIps ss ts*
⟨proof⟩

lemma *getParts-foldr*: *getParts rs = foldr partIps (extract-IPSets rs) [wordinterval-UNIV]*
⟨proof⟩

lemma *getParts-length*: *length* (*getParts rs*) ≤ *2*^(*2* ∗ *length rs*)
⟨*proof*⟩

lemma *getParts-ipPartition*: *ipPartition* (*set* (*map wordinterval-to-set* (*extract-IPSets rs*)))
(*set* (*map wordinterval-to-set* (*getParts rs*)))
⟨*proof*⟩

lemma *getParts-complete*: *wordinterval-list-to-set* (*getParts rs*) = *UNIV*
⟨*proof*⟩

theorem *getParts-samefw*:
assumes *A* ∈ *set* (*map wordinterval-to-set* (*getParts rs*)) *s1* ∈ *A s2* ∈ *A*
shows *simple-fw rs* (*p*(|*p-src*:=*s1*|)) = *simple-fw rs* (*p*(|*p-src*:=*s2*|)) ∧
simple-fw rs (*p*(|*p-dst*:=*s1*|)) = *simple-fw rs* (*p*(|*p-dst*:=*s2*|))
⟨*proof*⟩

lemma *partIps-nonempty*: *ts* ≠ [] ⟹ *partIps s ts* ≠ []
⟨*proof*⟩
lemma *partitioningIps-nonempty*: *ts* ≠ [] ⟹ *partitioningIps ss ts* ≠ []
⟨*proof*⟩

lemma *getParts-nonempty*: *getParts rs* ≠ [] ⟨*proof*⟩
lemma *getParts-nonempty-elems*: ∀ *w*∈*set* (*getParts rs*). ¬ *wordinterval-empty w*
⟨*proof*⟩

fun *getOneIp* :: *'a*::*len wordinterval* ⇒ *'a*::*len word* **where**
getOneIp (*WordInterval b* -) = *b* |
getOneIp (*RangeUnion r1 r2*) = (*if wordinterval-empty r1 then getOneIp r2*
else getOneIp r1)

lemma *getOneIp-elem*: ¬ *wordinterval-empty W* ⟹ *wordinterval-element* (*getOneIp W*) *W*
⟨*proof*⟩

record *parts-connection* = *pc-iiface* :: *string*
pc-oiface :: *string*
pc-proto :: *primitive-protocol*
pc-sport :: *16 word*
pc-dport :: *16 word*

definition *same-fw-behaviour* :: ~~'pkt-ext itself ⇒~~ 'i::len word ⇒ 'i word ⇒ 'i simple-rule list ⇒ bool **where**
same-fw-behaviour ~~TYPE('pkt-ext)~~ a b rs ≡
∀ (p:: 'i::len simple-packet).
simple-fw rs (p(|p-src:=a|)) = simple-fw rs (p(|p-src:=b|)) ∧
simple-fw rs (p(|p-dst:=a|)) = simple-fw rs (p(|p-dst:=b|))

lemma *getParts-same-fw-behaviour*:
A ∈ set (map wordinterval-to-set (getParts rs)) ⟹ s1 ∈ A ⟹ s2 ∈ A ⟹
same-fw-behaviour s1 s2 rs
⟨proof⟩

definition *runFw s d c rs = simple-fw rs (|p-iiface=pc-iiface c,p-oiface=pc-oiface c,*
p-src=s,p-dst=d,
p-proto=pc-proto c,
p-sport=pc-sport c,p-dport=pc-dport c,
p-tcp-flags={TCP-SYN},
p-payload=''''|)

We use *runFw* for executable code, but in general, everything applies to generic packets

definition *runFw-scheme* :: 'i::len word ⇒ 'i word ⇒ 'b parts-connection-scheme ⇒
('i, 'a) simple-packet-scheme ⇒ 'i simple-rule list ⇒ state
where
runFw-scheme s d c p rs = simple-fw rs
(p(|p-iiface:=pc-iiface c,
p-oiface:=pc-oiface c,
p-src:=s,
p-dst:=d,
p-proto:=pc-proto c,
p-sport:=pc-sport c,
p-dport:=pc-dport c|))

lemma *runFw-scheme*: *runFw s d c rs = runFw-scheme s d c p rs*
⟨proof⟩

lemma *has-default-policy-runFw*: *has-default-policy rs ⟹ runFw s d c rs = Decision FinalAllow ∨ runFw s d c rs = Decision FinalDeny*
⟨proof⟩

definition *same-fw-behaviour-one* :: 'i::len word ⇒ 'i word ⇒ 'a parts-connection-scheme

$\Rightarrow$ *'i simple-rule list* $\Rightarrow$ *bool* **where**
same-fw-behaviour-one ip1 ip2 c rs $\equiv$
$\forall$ *d s. runFw ip1 d c rs = runFw ip2 d c rs* $\wedge$ *runFw s ip1 c rs = runFw s ip2 c rs*

lemma *same-fw-spec*: *same-fw-behaviour ip1 ip2 rs* $\Longrightarrow$ *same-fw-behaviour-one ip1 ip2 c rs*
⟨*proof*⟩

Is an equivalence relation

lemma *same-fw-behaviour-one-equi*:
same-fw-behaviour-one x x c rs
same-fw-behaviour-one x y c rs = same-fw-behaviour-one y x c rs
same-fw-behaviour-one x y c rs $\wedge$ *same-fw-behaviour-one y z c rs* $\Longrightarrow$ *same-fw-behaviour-one x z c rs*
⟨*proof*⟩

lemma *same-fw-behaviour-equi*:
same-fw-behaviour x x rs
same-fw-behaviour x y rs = same-fw-behaviour y x rs
same-fw-behaviour x y rs $\wedge$ *same-fw-behaviour y z rs* $\Longrightarrow$ *same-fw-behaviour x z rs*
⟨*proof*⟩

lemma *runFw-sameFw-behave*:
fixes *W* :: *'i::len word set set*
shows
$\forall$ *A* $\in$ *W*. $\forall$ *a1* $\in$ *A*. $\forall$ *a2* $\in$ *A*. *same-fw-behaviour-one a1 a2 c rs* $\Longrightarrow$ $\bigcup$ *W = UNIV* $\Longrightarrow$
$\forall$ *B* $\in$ *W*. $\exists$ *b* $\in$ *B*. *runFw ip1 b c rs = runFw ip2 b c rs* $\Longrightarrow$
$\forall$ *B* $\in$ *W*. $\exists$ *b* $\in$ *B*. *runFw b ip1 c rs = runFw b ip2 c rs* $\Longrightarrow$
same-fw-behaviour-one ip1 ip2 c rs
⟨*proof*⟩

lemma *sameFw-behave-sets*:
$\forall$ *w*$\in$*set A*. $\forall$ *a1* $\in$ *w*. $\forall$ *a2* $\in$ *w*. *same-fw-behaviour-one a1 a2 c rs* $\Longrightarrow$
$\forall$ *w1*$\in$*set A*. $\forall$ *w2*$\in$*set A*. $\exists$ *a1*$\in$*w1*. $\exists$ *a2*$\in$*w2*. *same-fw-behaviour-one a1 a2 c rs* $\Longrightarrow$
$\forall$ *w1*$\in$*set A*. $\forall$ *w2*$\in$*set A*.
$\forall$ *a1*$\in$*w1*. $\forall$ *a2*$\in$*w2*. *same-fw-behaviour-one a1 a2 c rs*
⟨*proof*⟩

definition *groupWIs* :: *parts-connection* $\Rightarrow$ *'i::len simple-rule list* $\Rightarrow$ *'i wordinterval list list* **where**
groupWIs c rs = (*let W* = *getParts rs in*
(*let f* = (λ*wi*. (*map* (λ*d*. *runFw* (*getOneIp wi*) *d c rs*) (*map*

getOneIp W),
map (*λs. runFw s* (*getOneIp wi*) *c rs*) (*map getOneIp W*))) *in*
groupF f W))

lemma *groupWIs-not-empty*: *groupWIs c rs* ≠ []
⟨*proof*⟩
lemma *groupWIs-not-empty-elem*: *V* ∈ *set* (*groupWIs c rs*) ⟹ *V* ≠ []
⟨*proof*⟩
lemma *groupWIs-not-empty-elems*:
assumes *V*: *V* ∈ *set* (*groupWIs c rs*) **and** *w*: *w* ∈ *set V*
shows ¬ *wordinterval-empty w*
⟨*proof*⟩

lemma *groupParts-same-fw-wi0*:
assumes *V* ∈ *set* (*groupWIs c rs*)
shows ∀ *w* ∈ *set* (*map wordinterval-to-set V*). ∀ *a1* ∈ *w*. ∀ *a2* ∈ *w*. *same-fw-behaviour-one a1 a2 c rs*
⟨*proof*⟩

lemma *groupWIs-same-fw-not*: *A* ∈ *set* (*groupWIs c rs*) ⟹ *B* ∈ *set* (*groupWIs c rs*) ⟹
A ≠ *B* ⟹
∀ *aw* ∈ *set* (*map wordinterval-to-set A*).
∀ *bw* ∈ *set* (*map wordinterval-to-set B*).
∀ *a* ∈ *aw*. ∀ *b* ∈ *bw*. ¬ *same-fw-behaviour-one a b c rs*
⟨*proof*⟩

lemma *groupParts-same-fw-wi1*:
V ∈ *set* (*groupWIs c rs*) ⟹ ∀ *w1* ∈ *set V*. ∀ *w2* ∈ *set V*.
∀ *a1* ∈ *wordinterval-to-set w1*. ∀ *a2* ∈ *wordinterval-to-set w2*. *same-fw-behaviour-one a1 a2 c rs*
⟨*proof*⟩

lemma *groupParts-same-fw-wi2*: *V* ∈ *set* (*groupWIs c rs*) ⟹
∀ *ip1* ∈ *wordinterval-list-to-set V*.
∀ *ip2* ∈ *wordinterval-list-to-set V*.
same-fw-behaviour-one ip1 ip2 c rs
⟨*proof*⟩

lemma *groupWIs-same-fw-not2*: *A* ∈ *set* (*groupWIs c rs*) ⟹ *B* ∈ *set* (*groupWIs c rs*) ⟹

$A \neq B \Longrightarrow$
$\forall$ *ip1* $\in$ *wordinterval-list-to-set A*.
$\forall$ *ip2* $\in$ *wordinterval-list-to-set B*.
$\neg$ *same-fw-behaviour-one ip1 ip2 c rs*

⟨*proof*⟩

lemma *A* $\in$ *set* (*groupWIs c rs*) $\Longrightarrow$ *B* $\in$ *set* (*groupWIs c rs*) $\Longrightarrow$
$\exists$ *ip1* $\in$ *wordinterval-list-to-set A*.
$\exists$ *ip2* $\in$ *wordinterval-list-to-set B*. *same-fw-behaviour-one ip1 ip2 c rs*
$\Longrightarrow$ *A* = *B*

⟨*proof*⟩

lemma *groupWIs-complete*: ($\bigcup$*x*$\in$ *set* (*groupWIs c rs*). *wordinterval-list-to-set x*) = (*UNIV*::′*i*::*len word set*)
⟨*proof*⟩

definition *groupWIs1* :: ′*a parts-connection-scheme* ⇒ ′*i*::*len simple-rule list* ⇒ ′*i wordinterval list list* **where**
groupWIs1 c rs = (*let P* = *getParts rs in*
(*let W* = *map getOneIp P in*
(*let f* = (λ*wi*. (*map* (λ*d*. *runFw* (*getOneIp wi*) *d c rs*) *W*,
map (λ*s*. *runFw s* (*getOneIp wi*) *c rs*) *W*)) *in*
map (*map fst*) (*groupF snd* (*map* (λ*x*. (*x*, *f x*)) *P*)))))

lemma *groupWIs-groupWIs1-equi*: *groupWIs1 c rs* = *groupWIs c rs*
⟨*proof*⟩

definition *simple-conn-matches* :: ′*i*::*len simple-match* ⇒ *parts-connection* ⇒ *bool* **where**
simple-conn-matches m c ⟷
(*match-iface* (*iiface m*) (*pc-iiface c*)) ∧
(*match-iface* (*oiface m*) (*pc-oiface c*)) ∧
(*match-proto* (*proto m*) (*pc-proto c*)) ∧
(*simple-match-port* (*sports m*) (*pc-sport c*)) ∧
(*simple-match-port* (*dports m*) (*pc-dport c*))

lemma *simple-conn-matches-simple-match-any*: *simple-conn-matches simple-match-any c*
⟨*proof*⟩

lemma *has-default-policy-simple-conn-matches*:
has-default-policy rs $\Longrightarrow$ *has-default-policy* [*r*←*rs* . *simple-conn-matches* (*match-sel r*) *c*]
⟨*proof*⟩

lemma *filter-conn-fw-lem*:
runFw s d c (*filter* (λ*r*. *simple-conn-matches* (*match-sel r*) *c*) *rs*) = *runFw s d c rs*
⟨*proof*⟩

definition *groupWIs2* :: *parts-connection* ⇒ *'i*::*len simple-rule list* ⇒ *'i wordinterval list list* **where**
groupWIs2 c rs = (*let P* = *getParts rs in*
(*let W* = *map getOneIp P in*
(*let filterW* = (*filter* (λ*r*. *simple-conn-matches* (*match-sel r*) *c*) *rs*) *in*
(*let f* = (λ*wi*. (*map* (λ*d*. *runFw* (*getOneIp wi*) *d c filterW*) *W*,
map (λ*s*. *runFw s* (*getOneIp wi*) *c filterW*) *W*)) *in*
map (*map fst*) (*groupF snd* (*map* (λ*x*. (*x*, *f x*)) *P*))))))

lemma *groupWIs1-groupWIs2-equi*: *groupWIs2 c rs* = *groupWIs1 c rs*
⟨*proof*⟩

lemma *groupWIs-code*[*code*]: *groupWIs c rs* = *groupWIs2 c rs*
⟨*proof*⟩

fun *matching-dsts* :: *'i*::*len word* ⇒ *'i simple-rule list* ⇒ *'i wordinterval* ⇒ *'i wordinterval* **where**
matching-dsts - [] - = *Empty-WordInterval* |
matching-dsts s ((*SimpleRule m Accept*)#*rs*) *acc-dropped* =
(*if simple-match-ip* (*src m*) *s then*
wordinterval-union (*wordinterval-setminus* (*ipcidr-tuple-to-wordinterval* (*dst m*)) *acc-dropped*) (*matching-dsts s rs acc-dropped*)
else
matching-dsts s rs acc-dropped) |
matching-dsts s ((*SimpleRule m Drop*)#*rs*) *acc-dropped* =
(*if simple-match-ip* (*src m*) *s then*
matching-dsts s rs (*wordinterval-union* (*ipcidr-tuple-to-wordinterval* (*dst m*)) *acc-dropped*)
else
matching-dsts s rs acc-dropped)

lemma *matching-dsts-pull-out-accu*:
wordinterval-to-set (*matching-dsts s rs* (*wordinterval-union a1 a2*)) = *wordinterval-to-set* (*matching-dsts s rs a2*) − *wordinterval-to-set a1*
⟨*proof*⟩

fun *matching-srcs* :: *'i::len word* ⇒ *'i simple-rule list* ⇒ *'i wordinterval* ⇒ *'i wordinterval* **where**
matching-srcs - [] - = *Empty-WordInterval* |
matching-srcs d ((*SimpleRule m Accept*)#*rs*) *acc-dropped* =
(*if simple-match-ip* (*dst m*) *d then*
wordinterval-union (*wordinterval-setminus* (*ipcidr-tuple-to-wordinterval* (*src m*)) *acc-dropped*) (*matching-srcs d rs acc-dropped*)
else
matching-srcs d rs acc-dropped) |
matching-srcs d ((*SimpleRule m Drop*)#*rs*) *acc-dropped* =
(*if simple-match-ip* (*dst m*) *d then*
matching-srcs d rs (*wordinterval-union* (*ipcidr-tuple-to-wordinterval* (*src m*)) *acc-dropped*)
else
matching-srcs d rs acc-dropped)

lemma *matching-srcs-pull-out-accu*:
wordinterval-to-set (*matching-srcs d rs* (*wordinterval-union a1 a2*)) = *wordinterval-to-set* (*matching-srcs d rs a2*) − *wordinterval-to-set a1*
⟨*proof*⟩

lemma *matching-dsts*: ∀ *r* ∈ *set rs. simple-conn-matches* (*match-sel r*) *c* ⟹
wordinterval-to-set (*matching-dsts s rs Empty-WordInterval*) = {*d. runFw s d c rs* = *Decision FinalAllow*}
⟨*proof*⟩
lemma *matching-srcs*: ∀ *r* ∈ *set rs. simple-conn-matches* (*match-sel r*) *c* ⟹
wordinterval-to-set (*matching-srcs d rs Empty-WordInterval*) = {*s. runFw s d c rs* = *Decision FinalAllow*}
⟨*proof*⟩

definition *groupWIs3-default-policy* :: *parts-connection* ⇒ *'i::len simple-rule list* ⇒ *'i wordinterval list list* **where**
groupWIs3-default-policy c rs = (*let P* = *getParts rs in*
(*let W* = *map getOneIp P in*
(*let filterW* = (*filter* (λ*r. simple-conn-matches* (*match-sel r*) *c*) *rs*) *in*
(*let f* = (λ*wi. let mtch-dsts* = (*matching-dsts* (*getOneIp wi*) *filterW Empty-WordInterval*);
mtch-srcs = (*matching-srcs* (*getOneIp wi*) *filterW Empty-WordInterval*) *in*
(*map* (λ*d. wordinterval-element d mtch-dsts*) *W*,
map (λ*s. wordinterval-element s mtch-srcs*) *W*)) *in*
map (*map fst*) (*groupF snd* (*map* (λ*x.* (*x*, *f x*)) *P*))))))

lemma *groupWIs3-default-policy-groupWIs2*:
fixes *rs* :: *'i::len simple-rule list*
assumes *has-default-policy rs*
shows *groupWIs2 c rs* = *groupWIs3-default-policy c rs*
⟨*proof*⟩

definition *groupWIs3* :: *parts-connection* ⇒ *'i::len simple-rule list* ⇒ *'i wordinterval list list* **where**
groupWIs3 c rs = (*if has-default-policy rs then groupWIs3-default-policy c rs else groupWIs2 c rs*)

lemma *groupWIs3*: *groupWIs3* = *groupWIs*
⟨*proof*⟩

definition *build-ip-partition* :: *parts-connection* ⇒ *'i::len simple-rule list* ⇒ *'i wordinterval list* **where**
build-ip-partition c rs = *map*
(*λxs. wordinterval-sort* (*wordinterval-compress* (*foldr wordinterval-union xs Empty-WordInterval*)))
(*groupWIs3 c rs*)

theorem *build-ip-partition-same-fw*: *V* ∈ *set* (*build-ip-partition c rs*) ⟹
∀ *ip1*::*'i::len word* ∈ *wordinterval-to-set V*.
∀ *ip2*::*'i::len word* ∈ *wordinterval-to-set V*.
same-fw-behaviour-one ip1 ip2 c rs
⟨*proof*⟩

theorem *build-ip-partition-same-fw-min*: *A* ∈ *set* (*build-ip-partition c rs*) ⟹ *B* ∈ *set* (*build-ip-partition c rs*) ⟹
A ≠ *B* ⟹
∀ *ip1*::*'i::len word* ∈ *wordinterval-to-set A*.
∀ *ip2*::*'i::len word* ∈ *wordinterval-to-set B*.
¬ *same-fw-behaviour-one ip1 ip2 c rs*
⟨*proof*⟩

theorem *build-ip-partition-complete*: (⋃*x*∈*set* (*build-ip-partition c rs*). *wordinterval-to-set x*) = (*UNIV* :: *'i::len word set*)
⟨*proof*⟩

lemma *build-ip-partition-no-empty-elems*: *wi* ∈ *set* (*build-ip-partition c rs*) ⟹ ¬ *wordinterval-empty wi*

⟨*proof*⟩

lemma *build-ip-partition-disjoint*:
$V1 \in set\ (build\text{-}ip\text{-}partition\ c\ rs) \Longrightarrow V2 \in set\ (build\text{-}ip\text{-}partition\ c\ rs) \Longrightarrow$
$V1 \neq V2 \Longrightarrow$
$wordinterval\text{-}to\text{-}set\ V1 \cap wordinterval\text{-}to\text{-}set\ V2 = \{\}$
⟨*proof*⟩

lemma *map-wordinterval-to-set-distinct*:
assumes *distinct*: *distinct xs*
and *disjoint*: $(\forall x1 \in set\ xs.\ \forall x2 \in set\ xs.\ x1 \neq x2 \longrightarrow wordinterval\text{-}to\text{-}set\ x1 \cap wordinterval\text{-}to\text{-}set\ x2 = \{\})$
and *notempty*: $\forall x \in set\ xs.\ \neg\ wordinterval\text{-}empty\ x$
shows *distinct* (*map wordinterval-to-set xs*)
⟨*proof*⟩

lemma *map-getOneIp-distinct*: **assumes**
distinct: *distinct xs*
and *disjoint*: $(\forall x1 \in set\ xs.\ \forall x2 \in set\ xs.\ x1 \neq x2 \longrightarrow wordinterval\text{-}to\text{-}set\ x1 \cap wordinterval\text{-}to\text{-}set\ x2 = \{\})$
and *notempty*: $\forall x \in set\ xs.\ \neg\ wordinterval\text{-}empty\ x$
shows *distinct* (*map getOneIp xs*)
⟨*proof*⟩

lemma *getParts-disjoint-list*: *disjoint-list* (*map wordinterval-to-set* (*getParts rs*))
⟨*proof*⟩

lemma *build-ip-partition-distinct*: *distinct* (*map wordinterval-to-set* (*build-ip-partition c rs*))
⟨*proof*⟩

lemma *build-ip-partition-distinct′*: *distinct* (*build-ip-partition c rs*)
⟨*proof*⟩

17.2 Service Matrix over an IP Address Space Partition

definition *simple-firewall-without-interfaces* :: *′i::len simple-rule list* ⇒ *bool* **where**
$simple\text{-}firewall\text{-}without\text{-}interfaces\ rs \equiv \forall m \in match\text{-}sel\ `\ set\ rs.\ iiface\ m = ifaceAny \wedge oiface\ m = ifaceAny$

lemma *simple-fw-no-interfaces*:
assumes *no-ifaces*: *simple-firewall-without-interfaces rs*
shows *simple-fw rs p* = *simple-fw rs* (*p*⦇ *p-iiface*:= *x*, *p-oiface*:= *y*⦈)
⟨*proof*⟩

lemma *runFw-no-interfaces*:

assumes *no-ifaces*: *simple-firewall-without-interfaces rs*
shows *runFw s d c rs* = *runFw s d* (*c*(| *pc-iiface*:= *x*, *pc-oiface*:= *y*|)) *rs*
⟨*proof*⟩

lemma[*code-unfold*]: *simple-firewall-without-interfaces rs* ≡
∀ *m* ∈ *set rs*. *iiface* (*match-sel m*) = *ifaceAny* ∧ *oiface* (*match-sel m*) = *ifaceAny*
⟨*proof*⟩

definition *access-matrix*
:: *parts-connection* ⇒ *'i*::*len simple-rule list* ⇒ (*'i word* × *'i wordinterval*) *list* × (*'i word* × *'i word*) *list*
where
access-matrix c rs ≡
(*let W* = *build-ip-partition c rs*;
R = *map getOneIp W*
in
(*zip R W*, [(*s*, *d*)←*all-pairs R*. *runFw s d c rs* = *Decision FinalAllow*]))

lemma *access-matrix-nodes-defined*:
(*V*,*E*) = *access-matrix c rs* ⟹ (*s*, *d*) ∈ *set E* ⟹ *s* ∈ *dom* (*map-of V*) **and**
(*V*,*E*) = *access-matrix c rs* ⟹ (*s*, *d*) ∈ *set E* ⟹ *d* ∈ *dom* (*map-of V*)
⟨*proof*⟩

For all the entries E of the matrix, the access is allowed

lemma (*V*,*E*) = *access-matrix c rs* ⟹ (*s*, *d*) ∈ *set E* ⟹ *runFw s d c rs* = *Decision FinalAllow*
⟨*proof*⟩

However, the entries are only a representation of a whole set of IP addresses. For all IP addresses which the entries represent, the access must be allowed.

lemma *map-of-zip-map*: *map-of* (*zip* (*map f rs*) *rs*) *k* = *Some v* ⟹ *k* = *f v*
⟨*proof*⟩

lemma *access-matrix-sound*: **assumes** *matrix*: (*V*,*E*) = *access-matrix c rs* **and**
repr: (*s-repr*, *d-repr*) ∈ *set E* **and**
s-range: (*map-of V*) *s-repr* = *Some s-range* **and** *s*: *s* ∈ *wordinterval-to-set s-range* **and**
d-range: (*map-of V*) *d-repr* = *Some d-range* **and** *d*: *d* ∈ *wordinterval-to-set d-range*
shows *runFw s d c rs* = *Decision FinalAllow*
⟨*proof*⟩

lemma *distinct-map-getOneIp-obtain*: *v* ∈ *set xs* ⟹ *distinct* (*map getOneIp xs*) ⟹
∃ *s-repr*. *map-of* (*zip* (*map getOneIp xs*) *xs*) *s-repr* = *Some v*
⟨*proof*⟩

lemma *access-matrix-complete*:
  fixes *rs* :: *'i::len simple-rule list*
  assumes *matrix*: $(V,E) = $ *access-matrix c rs* **and**
    allow: *runFw s d c rs = Decision FinalAllow*
  shows $\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$
    (*map-of V*) *s-repr* = *Some s-range* $\wedge$ *s* $\in$ *wordinterval-to-set s-range* $\wedge$
    (*map-of V*) *d-repr* = *Some d-range* $\wedge$ *d* $\in$ *wordinterval-to-set d-range*
⟨*proof*⟩

theorem *access-matrix*:
  fixes *rs* :: *'i::len simple-rule list*
  assumes *matrix*: $(V,E) = $ *access-matrix c rs*
  shows ($\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$
    (*map-of V*) *s-repr* = *Some s-range* $\wedge$ *s* $\in$ *wordinterval-to-set s-range* $\wedge$
    (*map-of V*) *d-repr* = *Some d-range* $\wedge$ *d* $\in$ *wordinterval-to-set d-range*)
    $\longleftrightarrow$
    runFw s d c rs = Decision FinalAllow
⟨*proof*⟩

For a *'i simple-rule list* and a fixed *parts-connection*, we support to partition the IP address space; for IP addresses of arbitrary length (eg., IPv4, IPv6).

All members of a partition have the same access rights: $V \in$ *set* (*build-ip-partition c rs*) $\Longrightarrow \forall$ *ip1*$\in$*wordinterval-to-set V*. $\forall$ *ip2*$\in$*wordinterval-to-set V*. *same-fw-behaviour-one ip1 ip2 c rs*

Minimal: ⟦$A \in$ *set* (*build-ip-partition c rs*); $B \in$ *set* (*build-ip-partition c rs*); $A \neq B$⟧ $\Longrightarrow \forall$ *ip1*$\in$*wordinterval-to-set A*. $\forall$ *ip2*$\in$*wordinterval-to-set B*. $\neg$ *same-fw-behaviour-one ip1 ip2 c rs*

The resulting access control matrix is sound and complete:

$(V, E) = $ *access-matrix c rs* $\Longrightarrow$ ($\exists$ *s-repr d-repr s-range d-range*. (*s-repr*, *d-repr*) $\in$ *set E* $\wedge$ *map-of V s-repr* = *Some s-range* $\wedge$ *s* $\in$ *wordinterval-to-set s-range* $\wedge$ *map-of V d-repr* = *Some d-range* $\wedge$ *d* $\in$ *wordinterval-to-set d-range*) = (*runFw s d c rs = Decision FinalAllow*)

Theorem reads: For a fixed connection, you can look up IP addresses (source and destination pairs) in the matrix if and only if the firewall accepts this src,dst IP address pair for the fixed connection. Note: The matrix is actually a graph (nice visualization!), you need to look up IP addresses in the Vertices and check the access of the representants in the edges. If you want to visualize the graph (e.g. with Graphviz or tkiz): The vertices are the node description (i.e. header; *dom V* is the label for each node which will also be referenced in the edges, *ran V* is the human-readable description for each node (i.e. the full IP range it represents)), the edges are the edges. Result looks nice. Theorem also tells us that this visualization is correct.

Only defined for *simple-firewall-without-interfaces*

definition *access-matrix-pretty-ipv4*
:: *parts-connection* ⇒ *32 simple-rule list* ⇒ (*string* × *string*) *list* × (*string* × *string*) *list*
where
access-matrix-pretty-ipv4 c rs ≡
if ¬ *simple-firewall-without-interfaces rs then undefined else*
(*let* (*V*,*E*) = (*access-matrix c rs*);
formatted-nodes =
map (λ(*v-repr*, *v-range*). (*ipv4addr-toString v-repr*, *ipv4addr-wordinterval-toString v-range*)) *V*;
formatted-edges =
map (λ(*s*,*d*). (*ipv4addr-toString s*, *ipv4addr-toString d*)) *E*
in
(*formatted-nodes*, *formatted-edges*)
)

definition *access-matrix-pretty-ipv4-code*
:: *parts-connection* ⇒ *32 simple-rule list* ⇒ (*string* × *string*) *list* × (*string* × *string*) *list*
where
access-matrix-pretty-ipv4-code c rs ≡
if ¬ *simple-firewall-without-interfaces rs then undefined else*
(*let W* = *build-ip-partition c rs*;
R = *map getOneIp W*;
U = *all-pairs R*
in
(*zip* (*map ipv4addr-toString R*) (*map ipv4addr-wordinterval-toString W*),
map (λ(*x*,*y*). (*ipv4addr-toString x*, *ipv4addr-toString y*)) [(*s*, *d*)←*all-pairs R*. *runFw s d c rs* = *Decision FinalAllow*]))

lemma *access-matrix-pretty-ipv4-code*[*code*]: *access-matrix-pretty-ipv4* = *access-matrix-pretty-ipv4-code*
⟨*proof*⟩

definition *access-matrix-pretty-ipv6*
:: *parts-connection* ⇒ *128 simple-rule list* ⇒ (*string* × *string*) *list* × (*string* × *string*) *list*
where
access-matrix-pretty-ipv6 c rs ≡
if ¬ *simple-firewall-without-interfaces rs then undefined else*
(*let* (*V*,*E*) = (*access-matrix c rs*);
formatted-nodes =
map (λ(*v-repr*, *v-range*). (*ipv6addr-toString v-repr*, *ipv6addr-wordinterval-toString v-range*)) *V*;
formatted-edges =
map (λ(*s*,*d*). (*ipv6addr-toString s*, *ipv6addr-toString d*)) *E*
in
(*formatted-nodes*, *formatted-edges*)
)

definition *access-matrix-pretty-ipv6-code*
:: *parts-connection* ⇒ *128 simple-rule list* ⇒ (*string* × *string*) *list* × (*string* × *string*) *list*
where
access-matrix-pretty-ipv6-code c rs ≡
if ¬ *simple-firewall-without-interfaces rs then undefined else*
(*let W* = *build-ip-partition c rs*;
R = *map getOneIp W*;
U = *all-pairs R*
in
(*zip* (*map ipv6addr-toString R*) (*map ipv6addr-wordinterval-toString W*),
map (λ(*x*,*y*). (*ipv6addr-toString x*, *ipv6addr-toString y*)) [(*s*, *d*)←*all-pairs R*. *runFw s d c rs* = *Decision FinalAllow*]))

lemma *access-matrix-pretty-ipv6-code*[*code*]: *access-matrix-pretty-ipv6* = *access-matrix-pretty-ipv6-code*
⟨*proof*⟩

definition *parts-connection-ssh* **where**
parts-connection-ssh ≡ (|*pc-iiface*=''1'', *pc-oiface*=''1'', *pc-proto*=*TCP*, *pc-sport*=*10000*, *pc-dport*=*22*|)

definition *parts-connection-http* **where**
parts-connection-http ≡ (|*pc-iiface*=''1'', *pc-oiface*=''1'', *pc-proto*=*TCP*, *pc-sport*=*10000*, *pc-dport*=*80*|)

definition *mk-parts-connection-TCP* :: *16 word* ⇒ *16 word* ⇒ *parts-connection*
where
mk-parts-connection-TCP sport dport = (|*pc-iiface*=''1'', *pc-oiface*=''1'', *pc-proto*=*TCP*,
pc-sport=*sport*, *pc-dport*=*dport*|)

lemma *mk-parts-connection-TCP 10000 22* = *parts-connection-ssh*
mk-parts-connection-TCP 10000 80 = *parts-connection-http*
⟨*proof*⟩

value[*code*] *partitioningIps* [*WordInterval* (*0*::*ipv4addr*) *0*] [*WordInterval 0 2*, *WordInterval 0 2*]*Here is an example of a really large and complicated firewall:*
end

18 Simple Firewall toString Functions

theory *SimpleFw-toString*
imports *Primitives/Primitives-toString*
SimpleFw-Syntax
begin

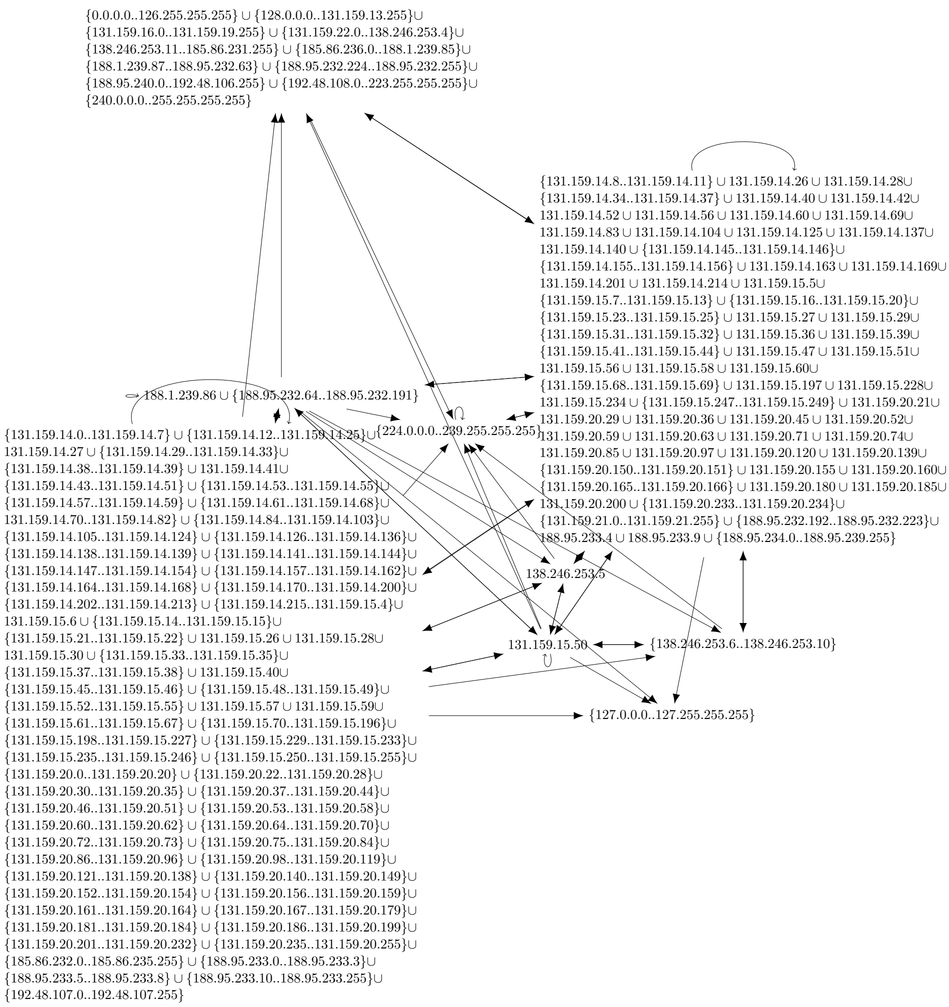

Figure 1: TUM ssh Service Matrix

```
fun simple-action-toString :: simple-action ⇒ string where
  simple-action-toString Accept = ''ACCEPT'' |
  simple-action-toString Drop = ''DROP''

fun simple-rule-ipv4-toString :: 32 simple-rule ⇒ string where
  simple-rule-ipv4-toString (SimpleRule (|iiface=iif, oiface=oif, src=sip, dst=dip,
proto=p, sports=sps, dports=dps |) a) =
      simple-action-toString a @ ''     '' @
      protocol-toString p @ ''  --  '' @
      ipv4-cidr-toString sip @ ''            '' @
      ipv4-cidr-toString dip @ '' '' @
      iface-toString ''in: '' iif @ '' '' @
      iface-toString ''out: '' oif @ '' '' @
      ports-toString ''sports: '' sps @ '' '' @
      ports-toString ''dports: '' dps

fun simple-rule-ipv6-toString :: 128 simple-rule ⇒ string where
  simple-rule-ipv6-toString
    (SimpleRule (|iiface=iif, oiface=oif, src=sip, dst=dip, proto=p, sports=sps,
dports=dps |) a) =
      simple-action-toString a @ ''     '' @
      protocol-toString p @ ''  --  '' @
      ipv6-cidr-toString sip @ ''            '' @
      ipv6-cidr-toString dip @ '' '' @
      iface-toString ''in: '' iif @ '' '' @
      iface-toString ''out: '' oif @ '' '' @
      ports-toString ''sports: '' sps @ '' '' @
      ports-toString ''dports: '' dps

fun simple-rule-iptables-save-toString :: string ⇒ 32 simple-rule ⇒ string where
 simple-rule-iptables-save-toString chain (SimpleRule (|iiface=iif, oiface=oif, src=sip,
dst=dip, proto=p, sports=sps, dports=dps |) a) =
    ''-A ''@chain@iface-toString '' -i '' iif @
               iface-toString '' -o '' oif @
               ipv4-cidr-opt-toString '' -s '' sip @
               ipv4-cidr-opt-toString '' -d '' dip @
               protocol-opt-toString '' -p '' p @
               ports-toString '' --sport '' sps @
               ports-toString '' --dport '' dps @
               '' -j '' @ simple-action-toString a


end
```

References

[1] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *IFIP Networking 2016*, Vienna, Austria, may 2016.