

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	2
1.1	Enum set to string	3
2	Transport Layer Protocols	4
3	TCP flags	6
3.1	TCP Flags to String	7
4	Simple Packet	7
5	The state of a firewall, abstracted only to the packet filtering outcome	8
6	Network Interfaces	8
6.1	Helpers for the interface name (<i>string</i>)	11
6.2	Matching	12
6.3	Enumerating Interfaces	20
6.4	Negating Interfaces	21

7	Simple Firewall Syntax	23
8	Simple Firewall Semantics	25
8.1	Simple Ports	28
8.2	Simple IPs	28
8.3	Merging Simple Matches	29
8.4	Further Properties of a Simple Firewall	31
8.5	Reality check: Validity of Simple Matches	32
9	List Product Helpers	35
10	Option to List and Option to Set	35
11	Generalize Simple Firewall	36
11.1	Semantics	36
11.2	Lemmas	36
11.3	Equality with the Simple Firewall	37
11.4	Joining two firewalls, i.e. a packet is send through both sequentially.	38
11.5	Validity	43
12	Shadowed Rules	44
12.1	Removing Shadowed Rules	44
12.1.1	Soundness	44
13	Partition a Set by a Specific Constraint	47
14	Group by Function	60
15	Helper: Pretty Printing Word Intervals which correspond to IP address Ranges	63
16	toString Functions for Primitives	63
17	Service Matrices	64
17.1	IP Address Space Partition	65
17.2	Service Matrix over an IP Address Space Partition	91
18	Simple Firewall toString Functions	98

1 Enum toString Functions

```

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

```

```

fun bool-toString :: bool  $\Rightarrow$  string where
  bool-toString True = "True" |
  bool-toString False = "False"

```

1.1 Enum set to string

```

fun enum-set-get-one :: 'a list  $\Rightarrow$  'a set  $\Rightarrow$  'a option where
  enum-set-get-one [] S = None |
  enum-set-get-one (s#ss) S = (if s  $\in$  S then Some s else enum-set-get-one ss S)

```

```

lemma enum-set-get-one-empty: enum-set-get-one ss {} = None
by(induction ss) simp-all

```

```

lemma enum-set-get-one-None: S  $\subseteq$  set ss  $\Longrightarrow$  enum-set-get-one ss S = None
 $\longleftrightarrow$  S = {}
  apply(induction ss)
  apply(simp; fail)
  apply(simp)
  apply(intro conjI)
  apply blast
by fast

```

```

lemma enum-set-get-one-Some: S  $\subseteq$  set ss  $\Longrightarrow$  enum-set-get-one ss S = Some x
 $\Longrightarrow$  x  $\in$  S
  apply(induction ss)
  apply(simp; fail)
  apply(simp split: if-split-asm)
  apply(blast)
done

```

```

corollary enum-set-get-one-enum-Some: enum-set-get-one enum-class.enum S =
Some x  $\Longrightarrow$  x  $\in$  S
using enum-set-get-one-Some[where ss=enum-class.enum, simplified enum-UNIV]
by auto

```

```

lemma enum-set-get-one-Ex-Some: S  $\subseteq$  set ss  $\Longrightarrow$  S  $\neq$  {}  $\Longrightarrow$   $\exists$  x. enum-set-get-one
ss S = Some x
  apply(induction ss)
  apply(simp; fail)
  apply(simp split: if-split-asm)
  apply(blast)
done

```

```

corollary enum-set-get-one-enum-Ex-Some:
S  $\neq$  {}  $\Longrightarrow$   $\exists$  x. enum-set-get-one enum-class.enum S = Some x
using enum-set-get-one-Ex-Some[where ss=enum-class.enum, simplified enum-UNIV]
by auto

```

```

function enum-set-to-list :: ('a::enum) set  $\Rightarrow$  'a list where
  enum-set-to-list S = (if S = {} then [] else
  case enum-set-get-one Enum.enum S of None  $\Rightarrow$  []

```

```

| Some a ⇒ a#enum-set-to-list (S - {a}))
by(pat-completeness) auto

termination enum-set-to-list
  apply(relation measure (λ(S). card S))
  apply(simp-all add: card-gt-0-iff)
  apply(drule enum-set-get-one-enum-Some)
  apply(subgoal-tac finite S)
  prefer 2
  apply force
  apply (meson card-Diff1-less)
  done

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}))
  by(simp add: enum-set-get-one-empty)
declare enum-set-to-list.simps[simp del]

lemma enum-set-to-list: set (enum-set-to-list A) = A
  apply(induction A rule: enum-set-to-list.induct)
  apply(case-tac S = {})
  apply(simp add: enum-set-to-list.simps; fail)
  apply(simp)
  apply(subst enum-set-to-list-simps)
  apply(simp)
  apply(drule enum-set-get-one-enum-Ex-Some)
  apply(clarify)
  apply(simp)
  apply(drule enum-set-get-one-enum-Some)
  by blast

lemma list-toString bool-toString (enum-set-to-list {True, False}) = "[False, True]"
by eval

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 $SCTP \equiv 132 :: 8 \text{ word}$
qualified definition $IGMP \equiv 2 :: 8 \text{ word}$
qualified definition $GRE \equiv 47 :: 8 \text{ word}$
qualified definition $ESP \equiv 50 :: 8 \text{ word}$
qualified definition $AH \equiv 51 :: 8 \text{ word}$
qualified definition $IPv6ICMP \equiv 58 :: 8 \text{ word}$
end

datatype $protocol = ProtoAny \mid Proto \text{ primitive-protocol}$

fun $match\text{-}proto :: protocol \Rightarrow primitive\text{-}protocol \Rightarrow bool$ **where**
 $match\text{-}proto ProtoAny - \longleftrightarrow True \mid$
 $match\text{-}proto (Proto (p)) p\text{-}p \longleftrightarrow p\text{-}p = p$

fun $simple\text{-}proto\text{-}conjunct :: protocol \Rightarrow protocol \Rightarrow protocol \text{ option}$ **where**
 $simple\text{-}proto\text{-}conjunct ProtoAny proto = Some proto \mid$
 $simple\text{-}proto\text{-}conjunct proto ProtoAny = Some proto \mid$
 $simple\text{-}proto\text{-}conjunct (Proto p1) (Proto p2) = (if p1 = p2 then Some (Proto p1) else None)$

lemma $simple\text{-}proto\text{-}conjunct\text{-}asimp[simp]: simple\text{-}proto\text{-}conjunct proto ProtoAny = Some proto$
by(cases proto) simp-all

lemma $simple\text{-}proto\text{-}conjunct\text{-}correct: match\text{-}proto p1 pkt \wedge match\text{-}proto p2 pkt \longleftrightarrow$
 $(case simple\text{-}proto\text{-}conjunct p1 p2 of None \Rightarrow False \mid Some proto \Rightarrow match\text{-}proto proto pkt)$
apply(cases p1)
apply(simp-all)
apply(cases p2)
apply(simp-all)
done

lemma $simple\text{-}proto\text{-}conjunct\text{-}Some: simple\text{-}proto\text{-}conjunct p1 p2 = Some proto \implies$

$match\text{-}proto proto pkt \longleftrightarrow match\text{-}proto p1 pkt \wedge match\text{-}proto p2 pkt$
using $simple\text{-}proto\text{-}conjunct\text{-}correct$ **by** simp

lemma $simple\text{-}proto\text{-}conjunct\text{-}None: simple\text{-}proto\text{-}conjunct p1 p2 = None \implies$
 $\neg (match\text{-}proto p1 pkt \wedge match\text{-}proto p2 pkt)$
using $simple\text{-}proto\text{-}conjunct\text{-}correct$ **by** simp

lemma $conjunctProtoD:$

$simple\text{-}proto\text{-}conjunct a (Proto b) = Some x \implies x = Proto b \wedge (a = ProtoAny \vee a = Proto b)$

by(cases a) (simp-all split: if-splits)

lemma $conjunctProtoD2:$

$simple\text{-}proto\text{-}conjunct (Proto b) a = Some x \implies x = Proto b \wedge (a = ProtoAny \vee a = Proto b)$

by(cases a) (simp-all split: if-splits)

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

lemma primitive-protocol-Ex-neg: $p = Proto\ pi \implies \exists p'. p' \neq pi$ **for** pi

proof

show $pi + 1 \neq pi$ **by** *simp*

qed

lemma protocol-Ex-neg: $\exists p'. Proto\ p' \neq p$

by(cases p) (simp-all add: primitive-protocol-Ex-neg)

3 TCP flags

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

lemma UNIV-tcp-flag: $UNIV = \{TCP-SYN, TCP-ACK, TCP-FIN, TCP-RST, TCP-URG, TCP-PSH\}$ **using** tcp-flag.exhaust **by** *auto*

instance tcp-flag :: *finite*

proof

from UNIV-tcp-flag **show** *finite* (UNIV:: tcp-flag set) **using** *finite.simps* **by** *auto*

qed

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 $P \longleftrightarrow P\ TCP-SYN \wedge P\ TCP-ACK \wedge P\ TCP-FIN \wedge P\ TCP-RST \wedge P\ TCP-URG \wedge P\ TCP-PSH$

definition enum-ex-tcp-flag $P \longleftrightarrow P\ TCP-SYN \vee P\ TCP-ACK \vee P\ TCP-FIN \vee P\ TCP-RST \vee P\ TCP-URG \vee P\ TCP-PSH$

instance **proof**

show $UNIV = set\ (enum-class.enum :: tcp-flag\ list)$

by(simp add: UNIV-tcp-flag enum-tcp-flag-def)

next

show *distinct* (enum-class.enum :: tcp-flag list)

by(simp add: enum-tcp-flag-def)

next

show $\bigwedge P. (enum-class.enum-all :: (tcp-flag \Rightarrow bool) \Rightarrow bool)\ P = Ball\ UNIV\ P$

by(simp add: UNIV-tcp-flag enum-all-tcp-flag-def)

next

show $\bigwedge P. (enum-class.enum-ex :: (tcp-flag \Rightarrow bool) \Rightarrow bool)\ P = Bex\ UNIV\ P$

by(simp add: UNIV-tcp-flag enum-ex-tcp-flag-def)

qed

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]" **by** eval

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* \Rightarrow *'i simple-packet* **where**

```

  simple-packet-unext p  $\equiv$ 
    (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`, `ethF00`, ... 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* \Rightarrow *iface* \Rightarrow *bool* **where**

(*Iface* []) \leq (*Iface* -) \longleftrightarrow *True* |

(*Iface* -) \leq (*Iface* []) \longleftrightarrow *False* |

(*Iface* (*a#as*)) \leq (*Iface* (*b#bs*)) \longleftrightarrow (if *a* = *b* then *Iface as* \leq *Iface bs* else *a* \leq *b*)

by(*pat-completeness*) *auto*

termination *less-eq* :: *iface* \Rightarrow - \Rightarrow *bool*

apply(*relation measure* ($\lambda is. size (iface-sel (fst is)) + size (iface-sel (snd is))$))

apply(*rule wf-measure, unfold in-measure comp-def*)

apply(*simp*)

done

lemma *Iface-less-eq-empty*: *Iface x* \leq *Iface []* \implies *x* = []

by(*induction Iface x Iface [] rule: less-eq-iface.induct*) *auto*

lemma *less-eq-empty*: *Iface []* \leq *q*

by(*induction Iface [] q rule: less-eq-iface.induct*) *auto*

lemma *iface-cons-less-eq-i*:

Iface (b # bs) \leq *i* \implies \exists *q qs. i = Iface (q#qs) \wedge (b < q \vee (Iface bs) \leq (Iface qs))*

apply(*induction Iface (b # bs) i rule: less-eq-iface.induct*)

apply(*simp-all split: if-split-asm*)

apply(*clarify*)

apply(*simp*)

done

function (*sequential*) *less-iface* :: *iface* \Rightarrow *iface* \Rightarrow *bool* **where**

(*Iface* []) < (*Iface* []) \longleftrightarrow *False* |

(*Iface* []) < (*Iface* -) \longleftrightarrow *True* |

(*Iface* -) < (*Iface* []) \longleftrightarrow *False* |

(*Iface* (*a#as*)) < (*Iface* (*b#bs*)) \longleftrightarrow (if *a* = *b* then *Iface as* < *Iface bs* else *a* < *b*)

by(*pat-completeness*) *auto*

termination *less* :: *iface* \Rightarrow - \Rightarrow *bool*

apply(*relation measure* ($\lambda is. size (iface-sel (fst is)) + size (iface-sel (snd is))$))

apply(*rule wf-measure, unfold in-measure comp-def*)

apply(*simp*)

done

instance

proof

fix *n m* :: *iface*

show *n* < *m* \longleftrightarrow *n* \leq *m* \wedge \neg *m* \leq *n*

```

    proof(induction rule: less-iface.induct)
    case 4 thus ?case by simp fastforce
    qed(simp+)
next
fix n :: iface have n = m  $\implies$  n  $\leq$  m for m
  by(induction n m rule: less-eq-iface.induct) simp+
  thus n  $\leq$  n by simp
next
fix n m :: iface
show n  $\leq$  m  $\implies$  m  $\leq$  n  $\implies$  n = m
  proof(induction n m rule: less-eq-iface.induct)
  case 1 thus ?case using Iface-less-eq-empty by blast
  next
  case 3 thus ?case by (simp split: if-split-asm)
  qed(simp)+
next
fix n m q :: iface show n  $\leq$  m  $\implies$  m  $\leq$  q  $\implies$  n  $\leq$  q
  proof(induction n q arbitrary: m rule: less-eq-iface.induct)
  case 1 thus ?case by simp
  next
  case 2 thus ?case
    apply simp
    apply (drule iface-cons-less-eq-i)
    apply (elim exE conjE disjE)
    apply (simp; fail)
    by fastforce
  next
  case 3 thus ?case
    apply simp
    apply (frule iface-cons-less-eq-i)
    by (auto split: if-split-asm)
  qed
next
fix n m :: iface show n  $\leq$  m  $\vee$  m  $\leq$  n
  apply(induction n m rule: less-eq-iface.induct)
  apply(simp-all)
  by fastforce
qed
end

```

definition *ifaceAny* :: *iface* **where**
ifaceAny \equiv *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
))
```

```
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(induction eth rule: iface-name-is-wildcard.induct)
qed(simp-all)
private lemma iface-name-is-wildcard-alt': iface-name-is-wildcard eth <-> eth
≠ [] ^ hd (rev eth) = CHR "+"
unfolding iface-name-is-wildcard-alt using hd-rev by fastforce
```

```

private lemma iface-name-is-wildcard-fst: iface-name-is-wildcard (i # is)  $\implies$ 
is  $\neq$  []  $\implies$  iface-name-is-wildcard is
  by(simp add: iface-name-is-wildcard-alt)

private fun internal-iface-name-to-set :: string  $\Rightarrow$  string set where
  internal-iface-name-to-set i = (if  $\neg$  iface-name-is-wildcard i
    then
      {i}
    else
      {(butlast i)@cs | cs. True})
private lemma {(butlast i)@cs | cs. True} = ( $\lambda$ s. (butlast i)@s) ‘(UNIV::string set)
by fastforce
private lemma internal-iface-name-to-set: internal-iface-name-match i p-iface
 $\longleftrightarrow$  p-iface  $\in$  internal-iface-name-to-set i
proof(induction i p-iface rule: internal-iface-name-match.induct)
case 4 thus ?case
  apply(simp)
  apply(safe)
  apply(simp-all add: iface-name-is-wildcard-fst)
  apply (metis (full-types) iface-name-is-wildcard.simps(3) list.exhaust)
  by (metis append-butlast-last-id)
qed(simp-all)
private lemma internal-iface-name-to-set2: internal-iface-name-to-set iface =
{i. internal-iface-name-match iface i}
by (simp add: internal-iface-name-to-set)

```

```

private lemma internal-iface-name-match-refl: internal-iface-name-match i i
proof –
{ fix i j
  have i=j  $\implies$  internal-iface-name-match i j
  by(induction i j rule: internal-iface-name-match.induct)(simp-all)
} thus ?thesis by simp
qed

```

6.2 Matching

```

fun match-iface :: iface  $\Rightarrow$  string  $\Rightarrow$  bool where
  match-iface (Iface i) p-iface  $\longleftrightarrow$  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"
   $\neg$  match-iface (Iface "lo++") "lo"
   $\neg$  match-iface (Iface "lo+++") "lo"
   $\neg$  match-iface (Iface "lo1+") "lo"
   $\neg$  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"

```

lemma *match-ifaceAny*: *match-iface ifaceAny i*

by(*cases i, simp-all add: ifaceAny-def*)

lemma *match-IfaceFalse*: $\neg(\exists \text{IfaceFalse}. (\forall i. \neg \text{match-iface IfaceFalse } i))$

apply(*simp*)

apply(*intro allI, rename-tac IfaceFalse*)

apply(*case-tac IfaceFalse, rename-tac name*)

apply(*rule-tac x=name in exI*)

by(*simp add: internal-iface-name-match-refl*)

— *match-iface* explained by the individual cases

lemma *match-iface-case-nowildcard*: $\neg \text{iface-name-is-wildcard } i \implies \text{match-iface (Iface } i) \text{ } p\text{-}i \iff i = p\text{-}i$

proof(*induction i p-i rule: internal-iface-name-match.induct*)

qed(*auto simp add: iface-name-is-wildcard-alt split: if-split-asm*)

lemma *match-iface-case-wildcard-prefix*:

$\text{iface-name-is-wildcard } i \implies \text{match-iface (Iface } i) \text{ } p\text{-}i \iff \text{butlast } i = \text{take (length } i - 1) \text{ } p\text{-}i$

apply(*induction i p-i rule: internal-iface-name-match.induct*)

apply(*simp; fail*)

apply(*simp add: iface-name-is-wildcard-alt split: if-split-asm; fail*)

apply(*simp; fail*)

apply(*simp*)

apply(*intro conjI*)

apply(*simp add: iface-name-is-wildcard-alt split: if-split-asm; fail*)

apply(*simp add: iface-name-is-wildcard-fst*)

by (*metis One-nat-def length-0-conv list.sel(1) list.sel(3) take-Cons'*)

lemma *match-iface-case-wildcard-length*: $\text{iface-name-is-wildcard } i \implies \text{match-iface (Iface } i) \text{ } p\text{-}i \implies \text{length } p\text{-}i \geq (\text{length } i - 1)$

proof(*induction i p-i rule: internal-iface-name-match.induct*)

qed(*simp-all add: iface-name-is-wildcard-alt split: if-split-asm*)

corollary *match-iface-case-wildcard*:

$\text{iface-name-is-wildcard } i \implies \text{match-iface (Iface } i) \text{ } p\text{-}i \iff \text{butlast } i = \text{take (length } i - 1) \text{ } p\text{-}i \wedge \text{length } p\text{-}i \geq (\text{length } i - 1)$

using *match-iface-case-wildcard-length match-iface-case-wildcard-prefix* **by** *blast*

lemma *match-iface-set*: $\text{match-iface (Iface } i) \text{ } p\text{-}iface \iff p\text{-}iface \in \text{internal-iface-name-to-set } i$

using *internal-iface-name-to-set* **by** *simp*

```

private 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+" by eval
private lemma internal-iface-name-wildcard-longest "eth+" "e+" = Some
"eth+" by eval
private lemma internal-iface-name-wildcard-longest "eth+" "lo" = None by
eval

private 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
apply(simp add: internal-iface-name-wildcard-longest-def)
apply(safe)
apply(simp-all add: iface-name-is-wildcard-alt)
apply (metis One-nat-def append-butlast-last-id butlast-conv-take)
by (metis min.commute)+
private lemma internal-iface-name-wildcard-longest-refl: internal-iface-name-wildcard-longest
i i = Some i
by(simp add: internal-iface-name-wildcard-longest-def)

private 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 -
assume assm1: iface-name-is-wildcard i1
and assm2: iface-name-is-wildcard i2
{ assume assm3: internal-iface-name-wildcard-longest i1 i2 = None
have ¬ (internal-iface-name-match i1 p-i ∧ internal-iface-name-match i2
p-i)
proof -
from match-iface-case-wildcard-prefix[OF assm1] have 1:
internal-iface-name-match i1 p-i = (take (length i1 - 1) i1 = take (length
i1 - 1) p-i) by(simp add: butlast-conv-take)
from match-iface-case-wildcard-prefix[OF assm2] have 2:
internal-iface-name-match i2 p-i = (take (length i2 - 1) i2 = take (length
i2 - 1) p-i) by(simp add: butlast-conv-take)

```

```

    from assm3 have 3: take (min (length i1 - 1) (length i2 - 1)) i1 ≠ take
(min (length i1 - 1) (length i2 - 1)) i2
    by(simp add: internal-iface-name-wildcard-longest-def split: if-split-asm)
    from 3 show ?thesis using 1 2 min.commute take-take by metis
qed
} note internal-iface-name-wildcard-longest-correct-None=this

{ fix X
  assume assm3: internal-iface-name-wildcard-longest i1 i2 = Some X
  have (internal-iface-name-match i1 p-i ∧ internal-iface-name-match i2 p-i)
↔ internal-iface-name-match X p-i
  proof -
    from assm3 have assm3': take (min (length i1 - 1) (length i2 - 1)) i1 =
take (min (length i1 - 1) (length i2 - 1)) i2
    unfolding internal-iface-name-wildcard-longest-def by(simp split:
if-split-asm)

    { fix i1 i2
      assume iw1: iface-name-is-wildcard i1 and iw2: iface-name-is-wildcard
i2 and len: length i1 ≤ length i2 and
        take-i1i2: take (length i1 - 1) i1 = take (length i1 - 1) i2
      from len have len': length i1 - 1 ≤ length i2 - 1 by fastforce
      { fix x::string
        from len' have take (length i1 - 1) x = take (length i1 - 1) (take
(length i2 - 1) x) by(simp add: min-def)
      } note takei1=this

      { fix m::nat and n::nat and a::string and b c
        have m ≤ n ⇒ take n a = take n b ⇒ take m a = take m c ⇒
take m c = take m b by (metis min-absorb1 take-take)
      } note takesmaller=this

      from match-iface-case-wildcard-prefix[OF iw1, simplified] have 1:
        internal-iface-name-match i1 p-i ↔ take (length i1 - 1) i1 = take
(length i1 - 1) p-i by(simp add: butlast-conv-take)
      also have ... ↔ take (length i1 - 1) (take (length i2 - 1) i1) = take
(length i1 - 1) (take (length i2 - 1) p-i) using takei1 by simp
      finally have internal-iface-name-match i1 p-i = (take (length i1 - 1)
(take (length i2 - 1) i1) = take (length i1 - 1) (take (length i2 - 1) p-i)) .
      from match-iface-case-wildcard-prefix[OF iw2, simplified] have 2:
        internal-iface-name-match i2 p-i ↔ take (length i2 - 1) i2 = take
(length i2 - 1) p-i by(simp add: butlast-conv-take)

      have internal-iface-name-match i2 p-i ⇒ internal-iface-name-match i1
p-i

      unfolding 1 2
      apply(rule takesmaller[of (length i1 - 1) (length i2 - 1) i2 p-i])
      using len' apply (simp; fail)
      apply (simp; fail)

```

```

    using take-i1i2 by simp
  } note longer-iface-imp-shorter=this

show ?thesis
proof(cases length i1 ≤ length i2)
case True
with assm3 have X = i2 unfolding internal-iface-name-wildcard-longest-def
by(simp split: if-split-asm)
  from True assm3' have take-i1i2: take (length i1 - 1) i1 = take (length
i1 - 1) i2 by linarith
  from longer-iface-imp-shorter[OF assm1 assm2 True take-i1i2] ⟨X = i2⟩
  show (internal-iface-name-match i1 p-i ∧ internal-iface-name-match i2
p-i) ⟷ internal-iface-name-match X p-i by fastforce
  next
  case False
  with assm3 have X = i1 unfolding internal-iface-name-wildcard-longest-def
by(simp split: if-split-asm)
  from False assm3' have take-i1i2: take (length i2 - 1) i2 = take (length
i2 - 1) i1 by (metis min-def min-diff)
  from longer-iface-imp-shorter[OF assm2 assm1 - take-i1i2] False ⟨X =
i1⟩
  show (internal-iface-name-match i1 p-i ∧ internal-iface-name-match i2
p-i) ⟷ internal-iface-name-match X p-i by auto
  qed
  qed
} note internal-iface-name-wildcard-longest-correct-Some=this

from internal-iface-name-wildcard-longest-correct-None internal-iface-name-wildcard-longest-correct-Some
show ?thesis
  by(simp split:option.split)
  qed

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
  apply(cases i1, cases i2, rename-tac i1name i2name)
  apply(simp)
  apply(case-tac iface-name-is-wildcard i1name)

```



```

    apply(case-tac [!] iface-name-is-wildcard i2name)
    apply(simp-all)
    using internal-iface-name-wildcard-longest-correct apply auto[1]
    apply (metis match-iface.simps match-iface-case-nowildcard option.distinct(1)
option.sel)
    apply (metis match-iface.simps match-iface-case-nowildcard option.distinct(1)
option.sel)
    by (metis match-iface.simps option.distinct(1) option.inject)
    lemma iface-conjunct-None: iface-conjunct i1 i2 = None  $\implies$   $\neg$  (match-iface
i1 p-i  $\wedge$  match-iface i2 p-i)
    apply(case-tac i1, cases i2, rename-tac i1name i2name)
    apply(simp split: bool.split-asm if-split-asm)
    using internal-iface-name-wildcard-longest-correct apply fastforce
    apply (metis match-iface.simps match-iface-case-nowildcard)+
    done
    lemma iface-conjunct: match-iface i1 p-i  $\wedge$  match-iface i2 p-i  $\longleftrightarrow$ 
(case iface-conjunct i1 i2 of None  $\implies$  False | Some x  $\implies$  match-iface x p-i)
    apply(simp split: option.split)
    by(blast dest: iface-conjunct-Some iface-conjunct-None)

lemma match-iface-refl: match-iface (Iface x) x by (simp add: internal-iface-name-match-refl)
lemma match-iface-eqI: assumes x = Iface y shows match-iface x y
unfolding assms using match-iface-refl .

lemma iface-conjunct-ifaceAny: iface-conjunct ifaceAny i = Some i
    apply(simp add: ifaceAny-def)
    apply(case-tac i, rename-tac iname)
    apply(simp)
    apply(case-tac iface-name-is-wildcard iname)
    apply(simp add: internal-iface-name-wildcard-longest-def iface-name-is-wildcard-alt
Suc-leI; fail)
    apply(simp)
    using internal-iface-name-match.elims(3) by fastforce

lemma iface-conjunct-commute: iface-conjunct i1 i2 = iface-conjunct i2 i1
    apply(induction i1 i2 rule: iface-conjunct.induct)
    apply(rename-tac i1 i2, simp)
    apply(case-tac iface-name-is-wildcard i1)
    apply(case-tac [!] iface-name-is-wildcard i2)
    apply(simp-all)
    by (simp add: internal-iface-name-wildcard-longest-commute)

private definition internal-iface-name-subset :: string  $\implies$  string  $\implies$  bool where
internal-iface-name-subset i1 i2 = (case (iface-name-is-wildcard i1, iface-name-is-wildcard
i2) of
(True, True)  $\implies$  length i1  $\geq$  length i2  $\wedge$  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 -
  have f4: List.gen-length 0 i2 ≤ List.gen-length 0 i1
    using a1 by (simp add: length-code)
  have f5: ∧ cs. List.gen-length 0 (cs::char list) - Suc 0 = List.gen-length 0 (tl
cs)
    by (metis (no-types) One-nat-def length-code length-tl)
  obtain nn :: (nat ⇒ nat) ⇒ nat where
    ∧ f. ¬ f (nn f) ≤ f (Suc (nn f)) ∨ f (List.gen-length 0 i2) ≤ f (List.gen-length
0 i1)
    using f4 by (meson lift-Suc-mono-le)
  hence ¬ nn (λn. n - Suc 0) - Suc 0 ≤ nn (λn. n - Suc 0) ∨ List.gen-length
0 (tl i2) ≤ List.gen-length 0 (tl i1)
    using f5 by (metis (lifting) diff-Suc-Suc diff-zero)
  hence f6: min (List.gen-length 0 (tl i2)) (List.gen-length 0 (tl i1)) =
List.gen-length 0 (tl i2)
    using diff-le-self min.absorb1 by blast
  { assume take (List.gen-length 0 (tl i2)) i1 ≠ take (List.gen-length 0 (tl i2))
x
    have List.gen-length 0 (tl i2) = 0 ∨ take (List.gen-length 0 (tl i2)) i1 =
take (List.gen-length 0 (tl i2)) x
      using f6 f5 a3 by (metis (lifting) One-nat-def butlast-conv-take length-code
take-take)
    hence take (List.gen-length 0 (tl i2)) i1 = take (List.gen-length 0 (tl i2)) x
      by force }
  thus butlast i2 = take (length i2 - Suc 0) x
    using f5 a2 by (metis (full-types) length-code)
qed

```

private 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}
unfolding internal-iface-name-subset-def
proof(cases iface-name-is-wildcard i1, case-tac [!] iface-name-is-wildcard i2,
simp-all)
  assume a1: iface-name-is-wildcard i1
  assume a2: iface-name-is-wildcard i2
  show (length i2 ≤ length i1 ∧ take (length i2 - Suc 0) i1 = butlast i2) ⟷
    ({i. internal-iface-name-match i1 i} ⊆ {i. internal-iface-name-match i2 i}

```

```

i}) (is ?l  $\longleftrightarrow$  ?r)
  proof(rule iffI)
    assume ?l with a1 a2 show ?r
      apply(clarify, rename-tac x)
      apply(drule-tac p-i=x in match-iface-case-wildcard-prefix)+
      apply(simp)
      using butlast-take-length-helper by blast
    next
      assume ?r hence r': internal-iface-name-to-set i1  $\subseteq$  internal-iface-name-to-set
i2
      apply -
      apply(subst(asm) internal-iface-name-to-set2[symmetric])+
      by assumption
      have hlp1:  $\bigwedge i1 i2. \{x. \exists cs. x = i1 @ cs\} \subseteq \{x. \exists cs. x = i2 @ cs\} \implies$ 
length i2  $\leq$  length i1
      apply(simp add: Set.Collect-mono-iff)
      by force
      have hlp2:  $\bigwedge i1 i2. \{x. \exists cs. x = i1 @ cs\} \subseteq \{x. \exists cs. x = i2 @ cs\} \implies$ 
take (length i2) i1 = i2
      apply(simp add: Set.Collect-mono-iff)
      by force
      from r' a1 a2 show ?l
      apply(simp add: internal-iface-name-to-set)
      apply(safe)
      apply(drule hlp1)
      apply(simp)
      apply (metis One-nat-def Suc-pred diff-Suc-eq-diff-pred diff-is-0-eq
iface-name-is-wildcard.simps(1) length-greater-0-conv)
      apply(drule hlp2)
      apply(simp)
      by (metis One-nat-def butlast-conv-take length-butlast length-take
take-take)
      qed
    next
      show iface-name-is-wildcard i1  $\implies$   $\neg$  iface-name-is-wildcard i2  $\implies$ 
       $\neg$  Collect (internal-iface-name-match i1)  $\subseteq$  Collect (internal-iface-name-match
i2)
      using internal-iface-name-match-refl match-iface-case-nowildcard by fastforce
      next
      show  $\neg$  iface-name-is-wildcard i1  $\implies$   $\neg$  iface-name-is-wildcard i2  $\implies$ 
      (take (length i2 - Suc 0) i1 = butlast i2)  $\longleftrightarrow$  ( $\{i. \text{internal-iface-name-match}$ 
i1 i $\} \subseteq \{i. \text{internal-iface-name-match}$  i2 i $\}$ )
      using match-iface-case-nowildcard match-iface-case-wildcard-prefix by force
      next
      show  $\neg$  iface-name-is-wildcard i1  $\implies$   $\neg$  iface-name-is-wildcard i2  $\implies$ 
      (i1 = i2)  $\longleftrightarrow$  ( $\{i. \text{internal-iface-name-match}$  i1 i $\} \subseteq \{i. \text{inter-$ 
nal-iface-name-match i2 i $\}$ )
      using match-iface-case-nowildcard by force
      qed

```

definition *iface-subset* :: *iface* \Rightarrow *iface* \Rightarrow *bool* **where**
iface-subset *i1* *i2* \longleftrightarrow *internal-iface-name-subset* (*iface-sel* *i1*) (*iface-sel* *i2*)

lemma *iface-subset*: *iface-subset* *i1* *i2* \longleftrightarrow $\{i. \text{match-iface } i1\} \subseteq \{i. \text{match-iface } i2\}$

unfolding *iface-subset-def*
apply(*cases* *i1*, *cases* *i2*)
by(*simp* *add*: *internal-iface-name-subset*)

definition *iface-is-wildcard* :: *iface* \Rightarrow *bool* **where**
iface-is-wildcard *ifce* \equiv *iface-name-is-wildcard* (*iface-sel* *ifce*)

lemma *iface-is-wildcard-ifaceAny*: *iface-is-wildcard* *ifaceAny*
by(*simp* *add*: *iface-is-wildcard-def* *ifaceAny-def*)

6.3 Enumerating Interfaces

private definition *all-chars* :: *char* *list* **where**

all-chars \equiv *Enum.enum*

private lemma *all-chars*: *set* *all-chars* = (*UNIV*::*char* *set*)

by(*simp* *add*: *all-chars-def* *enum-UNIV*)

we can compute this, but its horribly inefficient!

private lemma *strings-of-length-n*: *set* (*List.n-lists* *n* *all-chars*) = $\{s::\text{string}.$
length *s* = *n* $\}$

apply(*induction* *n*)
apply(*simp*; *fail*)
apply(*simp* *add*: *all-chars*)
apply(*safe*)
apply(*simp*; *fail*)
apply(*simp*)
apply(*rename-tac* *n* *x*)
apply(*rule-tac* *x=drop* 1 *x* **in** *exI*)
apply(*simp*)
apply(*case-tac* *x*)
apply(*simp-all*)
done

Non-wildcard interfaces of length *n*

private definition *non-wildcard-ifaces* :: *nat* \Rightarrow *string* *list* **where**

non-wildcard-ifaces *n* \equiv *filter* ($\lambda i. \neg$ *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 = [""'] **by** *eval*

private lemma *non-wildcard-ifaces*: $set (non-wildcard-ifaces\ n) = \{s::string.\ length\ s = n \wedge \neg\ iface-name-is-wildcard\ s\}$
using *strings-of-length-n non-wildcard-ifaces-def* **by** *auto*

private lemma $(\bigcup\ i \in\ set\ (non-wildcard-ifaces\ n).\ internal-iface-name-to-set\ i) = \{s::string.\ length\ s = n \wedge \neg\ iface-name-is-wildcard\ s\}$
by (*simp add: non-wildcard-ifaces*)

Non-wildcard interfaces up to length n

private fun *non-wildcard-ifaces-upto* :: $nat \Rightarrow string\ list$ **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*: $set (non-wildcard-ifaces-upto\ n) = \{s::string.\ length\ s \leq n \wedge \neg\ iface-name-is-wildcard\ s\}$
apply (*induction n*)
apply *fastforce*
using *non-wildcard-ifaces* **by** *fastforce*

6.4 Negating Interfaces

private lemma *inv-iface-name-set*: $-\ (internal-iface-name-to-set\ i) = (\text{if } iface-name-is-wildcard\ i \text{ then } \{c\ |c.\ length\ c < length\ (butlast\ i)\} \cup \{c\ @\ cs\ |c\ cs.\ length\ c = length\ (butlast\ i) \wedge c \neq butlast\ i\} \text{ else } \{c\ |c.\ length\ c < length\ i\} \cup \{c\ @\ cs\ |c\ cs.\ length\ c \geq length\ i \wedge c \neq i\})$
proof –
{ **fix** $i::string$
have *inv-i-wildcard*: $-\ \{i\ @\ cs\ |cs.\ True\} = \{c\ |c.\ length\ c < length\ i\} \cup \{c\ @\ cs\ |c\ cs.\ length\ c = length\ i \wedge c \neq i\}$
apply (*rule Set.equalityI*)
prefer 2
apply (*safe*)[1]
apply (*simp;fail*)
apply (*simp;fail*)
apply (*simp*)
apply (*rule Compl-anti-mono*[**where** $B=\{i\ @\ cs\ |cs.\ True\}$ **and** $A=-\ (\{c\ |c.\ length\ c < length\ i\} \cup \{c\ @\ cs\ |c\ cs.\ length\ c = length\ i \wedge c \neq i\})$, *simplified*])
apply (*safe*)
apply (*simp*)
apply (*case-tac (length i) = length x*)
apply (*erule-tac x=x in allE, simp*)
apply (*erule-tac x=take (length i) x in allE*)
apply (*simp add: min-def*)
by (*metis append-take-drop-id*)
}

```

    } note inv-i-wildcard=this
    { fix i::string
      have inv-i-nowildcard:  $\neg \{i::string\} = \{c \mid c.\text{length } c < \text{length } i\} \cup \{c@cs \mid c \text{ cs. length } c \geq \text{length } i \wedge c \neq i\}$ 
      proof -
        have x:  $\{c \mid c.\text{length } c = \text{length } i \wedge c \neq i\} \cup \{c \mid c.\text{length } c > \text{length } i\}$ 
        =  $\{c@cs \mid c \text{ cs. length } c \geq \text{length } i \wedge c \neq i\}$ 
        apply(safe)
        apply force+
        done
        have  $\neg \{i::string\} = \{c \mid c . c \neq i\}$ 
        by(safe, simp)
        also have  $\dots = \{c \mid c.\text{length } c < \text{length } i\} \cup \{c \mid c.\text{length } c = \text{length } i \wedge c \neq i\} \cup \{c \mid c.\text{length } c > \text{length } i\}$ 
        by(auto)
        finally show ?thesis using x by auto
      qed
    } note inv-i-nowildcard=this
show ?thesis
proof(cases iface-name-is-wildcard i)
case True with inv-i-wildcard show ?thesis by force
next
case False with inv-i-nowildcard show ?thesis by force
qed
qed

```

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+")` **by**(*simp*)

lemma `"et+" ∈ - (internal-iface-name-to-set "eth+")` **by**(*simp*)

lemma `"+" ∈ - (internal-iface-name-to-set "eth+")` **by**(*simp*)

lemma $\neg \{i.\text{match-iface } (Iface \text{ "et+"}) i\} \subseteq - (\text{internal-iface-name-to-set "eth+"})$ **by** *force*

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+")` **by**(*simp*)

```

fun compress-pos-interfaces :: iface list ⇒ iface option where
  compress-pos-interfaces [] = Some ifaceAny |
  compress-pos-interfaces [i] = Some i |

```

$compress\text{-}pos\text{-}interfaces\ (i1\#i2\#is) = (case\ iface\ conjunct\ i1\ i2\ of\ None\ \Rightarrow\ None\ |\ Some\ i\ \Rightarrow\ compress\text{-}pos\text{-}interfaces\ (i\#is))$

lemma *compress-pos-interfaces-Some*: $compress\text{-}pos\text{-}interfaces\ ifces = Some\ ifce \Rightarrow$

$match\text{-}iface\ ifce\ p\text{-}i \longleftrightarrow (\forall\ i \in set\ ifces.\ match\text{-}iface\ i\ p\text{-}i)$

proof(*induction ifces rule: compress-pos-interfaces.induct*)

case 1 thus *?case by (simp add: match-ifaceAny)*

next

case 2 thus *?case by simp*

next

case (3 i1 i2) thus *?case*

apply(*simp*)

apply(*case-tac iface-conjunct i1 i2*)

apply(*simp; fail*)

apply(*simp*)

using *iface-conjunct-Some by presburger*

qed

lemma *compress-pos-interfaces-None*: $compress\text{-}pos\text{-}interfaces\ ifces = None \Rightarrow$

$\neg (\forall\ i \in set\ ifces.\ match\text{-}iface\ i\ p\text{-}i)$

proof(*induction ifces rule: compress-pos-interfaces.induct*)

case 1 thus *?case by (simp add: match-ifaceAny)*

next

case 2 thus *?case by simp*

next

case (3 i1 i2) thus *?case*

apply(*cases iface-conjunct i1 i2, simp-all*)

apply (*blast dest: iface-conjunct-None*)

by (*blast dest: iface-conjunct-Some*)

qed

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 $!s = \{0..7\} \cup \{8 .. ipv4max\}$. 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 wild-card 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 a 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  $\Rightarrow$  'a simple-match  $\times$  simple-action
where
  simple-rule-dtor r  $\equiv$  (case r of SimpleRule m a  $\Rightarrow$  (m,a))

```

```

lemma simple-rule-dtor-ids:
  uncurry SimpleRule  $\circ$  simple-rule-dtor = id
  simple-rule-dtor  $\circ$  uncurry SimpleRule = id
unfolding simple-rule-dtor-def comp-def fun-eq-iff
by(simp-all split: simple-rule.splits)

```

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  $\times$  nat)  $\Rightarrow$  'i::len word  $\Rightarrow$  bool where
  simple-match-ip (base, len) p-ip  $\longleftrightarrow$  p-ip  $\in$  ipset-from-cidr base len

```

```

lemma wordinterval-to-set-ipcitr-tuple-to-wordinterval-simple-match-ip-set:
  wordinterval-to-set (ipcitr-tuple-to-wordinterval ip) = {d. simple-match-ip ip d}

```

```

proof —
  { fix s and d :: 'a::len word  $\times$  nat
    from wordinterval-to-set-ipcitr-tuple-to-wordinterval have
      s  $\in$  wordinterval-to-set (ipcitr-tuple-to-wordinterval d)  $\longleftrightarrow$  simple-match-ip
d s
    by(cases d) auto
  } thus ?thesis by blast
qed

```

— by the way, the words do not wrap around

```

lemma {(253::8 word) .. 8} = {} by simp

```

```

fun simple-match-port :: (16 word  $\times$  16 word)  $\Rightarrow$  16 word  $\Rightarrow$  bool where
  simple-match-port (s,e) p-p  $\longleftrightarrow$  p-p  $\in$  {s..e}

```

```

fun simple-matches :: 'i::len simple-match  $\Rightarrow$  ('i, 'a) simple-packet-scheme  $\Rightarrow$ 
bool where
  simple-matches m p  $\longleftrightarrow$ 
    (match-iface (iiface m) (p-iiface p))  $\wedge$ 
    (match-iface (oiface m) (p-oiface p))  $\wedge$ 
    (simple-match-ip (src m) (p-src p))  $\wedge$ 
    (simple-match-ip (dst m) (p-dst p))  $\wedge$ 
    (match-proto (proto m) (p-proto p))  $\wedge$ 
    (simple-match-port (sports m) (p-sport p))  $\wedge$ 
    (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  $\Rightarrow$  ('i, 'a) simple-packet-scheme  $\Rightarrow$  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  $\Rightarrow$  Decision FinalAllow | Drop  $\Rightarrow$  Decision Fi-
nalDeny) else simple-fw-alt rs p)

```

lemma simple-fw-alt: simple-fw r p = simple-fw-alt r p **by**(induction rule: sim-
ple-fw.induct) simp-all

```

definition simple-match-any :: 'i::len simple-match where
  simple-match-any  $\equiv$  (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 –

have *: (65535::16 word) = max-word

by simp

show ?thesis

by (simp add: simple-match-any-def ipset-from-cidr-0 match-ifaceAny *)

qed

we specify only one empty port range

```

definition simple-match-none :: 'i::len simple-match where

```

```

  simple-match-none  $\equiv$ 

```

```

    (iiface=ifaceAny, oiface=ifaceAny, src=(1,0), dst=(0,0),

```

```

    proto=ProtoAny, sports=(1,0), dports=(0,65535) )

```

lemma simple-match-none: \neg simple-matches simple-match-none p

proof –

show ?thesis **by**(simp add: simple-match-none-def)

qed

```

fun empty-match :: 'i::len simple-match  $\Rightarrow$  bool where
  empty-match ( $\{iiface=-, oiface=-, src=-, dst=-, proto=-,$ 
    sports=(sps1, sps2), dports=(dps1, dps2)  $\}$ )  $\longleftrightarrow$  (sps1 > sps2)  $\vee$ 
(dps1 > dps2)

lemma empty-match: empty-match m  $\longleftrightarrow$  ( $\forall$  (p::('i::len, 'a) simple-packet-scheme).
 $\neg$  simple-matches m p)
proof
  assume empty-match m
  thus  $\forall$  p.  $\neg$  simple-matches m p by(cases m) fastforce
next
  assume assm:  $\forall$  (p::('i::len, 'a) simple-packet-scheme).  $\neg$  simple-matches m p
  obtain iif oif sip dip protocol sps1 sps2 dps1 dps2 where m:
    m = ( $\{iiface = iif, oiface = oif, src = sip, dst = dip, proto = protocol, sports$ 
= (sps1, sps2), dports = (dps1, dps2) $\}$ )
    by(cases m) force

  show empty-match m
  proof(simp add: m)
    let ?x= $\lambda$ p. dps1  $\leq$  p-dport p  $\longrightarrow$  p-sport p  $\leq$  sps2  $\longrightarrow$  sps1  $\leq$  p-sport p
 $\longrightarrow$ 
      match-protocol protocol (p-protocol p)  $\longrightarrow$  simple-match-ip dip (p-dst p)  $\longrightarrow$ 
simple-match-ip sip (p-src p)  $\longrightarrow$ 
      match-iface oif (p-oiface p)  $\longrightarrow$  match-iface iif (p-iiface p)  $\longrightarrow$   $\neg$ 
p-dport p  $\leq$  dps2
    from assm have nomatch:  $\forall$  (p::('i::len, 'a) simple-packet-scheme). ?x p
by(simp add: m)
    { fix ips::'i::len word  $\times$  nat
      have a  $\in$  ipset-from-cidr a n for a::'i::len word and n
        using ipset-from-cidr-lowest by auto
      hence simple-match-ip ips (fst ips) by(cases ips) simp
    } note ips=this
    have proto: match-protocol protocol (case protocol of ProtoAny  $\Rightarrow$  TCP |
Proto p  $\Rightarrow$  p)
      by(simp split: protocol.split)
    { fix ifce
      have match-iface ifce (iface-sel ifce)
      by(cases ifce) (simp add: match-iface-refl)
    } note ifaces=this
    { fix p::('i, 'a) simple-packet-scheme
      from nomatch have ?x p by blast
    } note pkt1=this
    obtain p::('i, 'a) simple-packet-scheme where [simp]:
p-iiface p = iface-sel iif
p-oiface p = iface-sel oif
p-src p = fst sip
p-dst p = fst dip
p-protocol p = (case protocol of ProtoAny  $\Rightarrow$  TCP | Proto p  $\Rightarrow$  p)

```

```

    p-sport p = sps1
    p-dport p = dps1
  by (meson simple-packet.select-convs)
    note pkt=pkt1[of p, simplified]
    from pkt ips proto ifaces have sps1 ≤ sps2 → ¬ dps1 ≤ dps2 by blast
    thus sps2 < sps1 ∨ dps2 < dps1 by fastforce
  qed
qed

```

lemma *nomatch*: $\neg \text{simple-matches } m \ p \implies \text{simple-fw } (\text{SimpleRule } m \ a \ \# \ rs) \ p$
 $= \text{simple-fw } rs \ p$
 by(*cases a, simp-all del: simple-matches.simps*)

8.1 Simple Ports

fun *simpl-ports-conjunct* :: $(16 \text{ word} \times 16 \text{ word}) \Rightarrow (16 \text{ word} \times 16 \text{ word}) \Rightarrow (16 \text{ word} \times 16 \text{ word})$ **where**
simpl-ports-conjunct (p1s, p1e) (p2s, p2e) = (max p1s p2s, min p1e p2e)

lemma $\{(p1s:: 16 \text{ word}) .. p1e\} \cap \{p2s .. p2e\} = \{\text{max } p1s \ p2s .. \text{min } p1e \ p2e\}$
 by(*simp*)

lemma *simple-ports-conjunct-correct*:
simple-match-port p1 pkt \wedge *simple-match-port* p2 pkt \longleftrightarrow *simple-match-port*
 (*simpl-ports-conjunct* p1 p2) pkt
 apply(*cases p1, cases p2, simp*)
 by *blast*

lemma *simple-match-port-code*[code] : *simple-match-port* (s,e) p-p = (s ≤ p-p \wedge
 p-p ≤ e) by *simp*

lemma *simple-match-port-UNIV*: $\{p. \text{simple-match-port } (s,e) \ p\} = \text{UNIV} \longleftrightarrow$
 (s = 0 \wedge e = max-word)
 apply(*simp*)
 apply(*rule*)
 apply(*case-tac s = 0*)
 using *antisym-conv* apply *blast*
 using *word-le-0-iff* apply *blast*
 using *word-zero-le* by *blast*

8.2 Simple IPs

lemma *simple-match-ip-conjunct*:
 fixes ip1 :: 'i::len word \times nat
 shows *simple-match-ip* ip1 p-ip \wedge *simple-match-ip* ip2 p-ip \longleftrightarrow
 (case *ipcidr-conjunct* ip1 ip2 of None \implies False | Some ipx \implies *simple-match-ip*
 ipx p-ip)
proof –
 {

```

fix b1 m1 b2 m2
have simple-match-ip (b1, m1) p-ip  $\wedge$  simple-match-ip (b2, m2) p-ip  $\longleftrightarrow$ 
  p-ip  $\in$  ipset-from-cidr b1 m1  $\cap$  ipset-from-cidr b2 m2
by simp
also have ...  $\longleftrightarrow$  p-ip  $\in$  (case ipcidr-conjunct (b1, m1) (b2, m2) of None  $\Rightarrow$ 
  {} | Some (bx, mx)  $\Rightarrow$  ipset-from-cidr bx mx)
  using ipcidr-conjunct-correct by blast
also have ...  $\longleftrightarrow$  (case ipcidr-conjunct (b1, m1) (b2, m2) of None  $\Rightarrow$  False |
  Some ipx  $\Rightarrow$  simple-match-ip ipx p-ip)
  by(simp split: option.split)
finally have simple-match-ip (b1, m1) p-ip  $\wedge$  simple-match-ip (b2, m2) p-ip
 $\longleftrightarrow$ 
  (case ipcidr-conjunct (b1, m1) (b2, m2) of None  $\Rightarrow$  False | Some ipx  $\Rightarrow$ 
  simple-match-ip ipx p-ip) .
} thus ?thesis by(cases ip1, cases ip2, simp)
qed

declare simple-matches.simps[simp del]

```

8.3 Merging Simple Matches

'i simple-match \wedge 'i simple-match

fun simple-match-and :: 'i::len simple-match \Rightarrow 'i simple-match \Rightarrow '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  $\Rightarrow$  None | Some sip  $\Rightarrow$ 
  (case ipcidr-conjunct dip1 dip2 of None  $\Rightarrow$  None | Some dip  $\Rightarrow$ 
  (case iface-conjunct iif1 iif2 of None  $\Rightarrow$  None | Some iif  $\Rightarrow$ 
  (case iface-conjunct oif1 oif2 of None  $\Rightarrow$  None | Some oif  $\Rightarrow$ 
  (case simple-ports-conjunct p1 p2 of None  $\Rightarrow$  None | Some p  $\Rightarrow$ 
  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 \wedge simple-matches m2 p
 \longleftrightarrow
(case simple-match-and m1 m2 of None \Rightarrow False | Some m \Rightarrow simple-matches
m p)

proof –

obtain iif1 oif1 sip1 dip1 p1 sps1 dps1 **where** m1:

m1 = (iiface=iif1, oiface=oif1, src=sip1, dst=dip1, proto=p1, sports=sps1,
dports=dps1) **by**(cases m1, blast)

obtain iif2 oif2 sip2 dip2 p2 sps2 dps2 **where** m2:

m2 = (iiface=iif2, oiface=oif2, src=sip2, dst=dip2, proto=p2, sports=sps2,
dports=dps2) **by**(cases m2, blast)

```

have sip-None: ipcidr-conjunct sip1 sip2 = None  $\implies$   $\neg$  simple-match-ip sip1
(p-src p)  $\vee$   $\neg$  simple-match-ip sip2 (p-src p)
using simple-match-ip-conjunct[of sip1 p-src p sip2] by simp
have dip-None: ipcidr-conjunct dip1 dip2 = None  $\implies$   $\neg$  simple-match-ip dip1
(p-dst p)  $\vee$   $\neg$  simple-match-ip dip2 (p-dst p)
using simple-match-ip-conjunct[of dip1 p-dst p dip2] by simp
have sip-Some:  $\bigwedge ip.$  ipcidr-conjunct sip1 sip2 = Some ip  $\implies$ 
simple-match-ip ip (p-src p)  $\longleftrightarrow$  simple-match-ip sip1 (p-src p)  $\wedge$  sim-
ple-match-ip sip2 (p-src p)
using simple-match-ip-conjunct[of sip1 p-src p sip2] by simp
have dip-Some:  $\bigwedge ip.$  ipcidr-conjunct dip1 dip2 = Some ip  $\implies$ 
simple-match-ip ip (p-dst p)  $\longleftrightarrow$  simple-match-ip dip1 (p-dst p)  $\wedge$  sim-
ple-match-ip dip2 (p-dst p)
using simple-match-ip-conjunct[of dip1 p-dst p dip2] by simp

have iiface-None: iface-conjunct iif1 iif2 = None  $\implies$   $\neg$  match-iface iif1
(p-iiface p)  $\vee$   $\neg$  match-iface iif2 (p-iiface p)
using iface-conjunct[of iif1 (p-iiface p) iif2] by simp
have oiface-None: iface-conjunct oif1 oif2 = None  $\implies$   $\neg$  match-iface oif1
(p-oiface p)  $\vee$   $\neg$  match-iface oif2 (p-oiface p)
using iface-conjunct[of oif1 (p-oiface p) oif2] by simp
have iiface-Some:  $\bigwedge iface.$  iface-conjunct iif1 iif2 = Some iface  $\implies$ 
match-iface iface (p-iiface p)  $\longleftrightarrow$  match-iface iif1 (p-iiface p)  $\wedge$  match-iface
iif2 (p-iiface p)
using iface-conjunct[of iif1 (p-iiface p) iif2] by simp
have oiface-Some:  $\bigwedge iface.$  iface-conjunct oif1 oif2 = Some iface  $\implies$ 
match-iface iface (p-oiface p)  $\longleftrightarrow$  match-iface oif1 (p-oiface p)  $\wedge$  match-iface
oif2 (p-oiface p)
using iface-conjunct[of oif1 (p-oiface p) oif2] by simp

have proto-None: simple-proto-conjunct p1 p2 = None  $\implies$   $\neg$  match-proto p1
(p-proto p)  $\vee$   $\neg$  match-proto p2 (p-proto p)
using simple-proto-conjunct-correct[of p1 (p-proto p) p2] by simp
have proto-Some:  $\bigwedge proto.$  simple-proto-conjunct p1 p2 = Some proto  $\implies$ 
match-proto proto (p-proto p)  $\longleftrightarrow$  match-proto p1 (p-proto p)  $\wedge$  match-proto
p2 (p-proto p)
using simple-proto-conjunct-correct[of p1 (p-proto p) p2] by simp

have case-Some:  $\bigwedge m.$  Some m = simple-match-and m1 m2  $\implies$ 
(simple-matches m1 p  $\wedge$  simple-matches m2 p)  $\longleftrightarrow$  simple-matches m p
apply(simp add: m1 m2 simple-matches.simps split: option.split-asm)
using simple-ports-conjunct-correct by(blast dest: sip-Some dip-Some iiface-Some
oiface-Some proto-Some)
have case-None: simple-match-and m1 m2 = None  $\implies$   $\neg$  (simple-matches
m1 p  $\wedge$  simple-matches m2 p)
apply(simp add: m1 m2 simple-matches.simps split: option.split-asm)
apply(blast dest: sip-None dip-None iiface-None oiface-None proto-None)+
done
from case-Some case-None show ?thesis by(cases simple-match-and m1 m2)

```

simp-all
qed

lemma *simple-match-and-SomeD*: *simple-match-and* *m1* *m2* = *Some m* \implies
simple-matches *m* *p* \iff (*simple-matches* *m1* *p* \wedge *simple-matches* *m2* *p*)
by(*simp add: simple-match-and-correct*)

lemma *simple-match-and-NoneD*: *simple-match-and* *m1* *m2* = *None* \implies
 \neg (*simple-matches* *m1* *p* \wedge *simple-matches* *m2* *p*)
by(*simp add: simple-match-and-correct*)

lemma *simple-matches-andD*: *simple-matches* *m1* *p* \implies *simple-matches* *m2* *p*
 \implies
 $\exists m$. *simple-match-and* *m1* *m2* = *Some m* \wedge *simple-matches* *m* *p*
by (*meson option.exhaust-sel simple-match-and-NoneD simple-match-and-SomeD*)

8.4 Further Properties of a Simple Firewall

fun *has-default-policy* :: '*i*::len *simple-rule list* \Rightarrow *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* \implies
simple-fw *rs* *p* = *Decision FinalAllow* \vee *simple-fw* *rs* *p* = *Decision FinalDeny*
proof(*induction* *rs* *rule: has-default-policy.induct*)

case 1 **thus** ?*case* **by** (*simp*)

next

case (2 *m a*) **thus** ?*case* **by**(*cases* *a*) (*simp-all add: simple-match-any*)

next

case (3 *r1 r2 rs*)

from 3 **obtain** *a m* **where** *r1* = *SimpleRule m a* **by** (*cases* *r1*) *simp*
with 3 **show** ?*case* **by** (*cases* *a*) *simp-all*

qed

lemma *has-default-policy-fst*: *has-default-policy* *rs* \implies *has-default-policy* (*r*#*rs*)
apply(*cases* *r*, *rename-tac m a, simp*)
apply(*cases* *rs*)
by(*simp-all*)

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* \Rightarrow '*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*
apply(*induction* *rs* *p* *rule: simple-fw.induct*)
by(*simp add: simple-match-any*)**+**

```

lemma simple-fw-not-matches-removeAll:  $\neg$  simple-matches m p  $\implies$ 
  simple-fw (removeAll (SimpleRule m a) rs) p = simple-fw rs p
apply(induction rs p rule: simple-fw.induct)
  apply(simp)
  apply(simp-all)
  apply blast+
done

```

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*:

```

 $\llbracket$ valid-prefix-fw p1; valid-prefix-fw p2; ipcidr-conjunct p1 p2 = Some p $\rrbracket \implies$ 
valid-prefix-fw p

```

unfolding *valid-prefix-fw-def*

by(*cases* *p*; *cases* *p1*; *cases* *p2*) (*simp* *add*: *ipcidr-conjunct.simps* *split*: *if-splits*)

definition *simple-match-valid* :: ('*i*::*len*, '*a*) *simple-match-scheme* \implies *bool* **where**
simple-match-valid *m* \equiv

```

({p. simple-match-port (sports m) p)  $\neq$  UNIV  $\vee$  {p. simple-match-port (dports
m) p}  $\neq$  UNIV  $\implies$ 

```

```

  proto m  $\in$  Proto '{TCP, UDP, L4-Protocol.SCTP}'  $\wedge$ 

```

```

  valid-prefix-fw (src m)  $\wedge$  valid-prefix-fw (dst m)

```

lemma *simple-match-valid-alt*[*code-unfold*]: *simple-match-valid* = (λ *m*.

```

  (let c = ( $\lambda$ (s,e). (s  $\neq$  0  $\vee$  e  $\neq$  max-word)) in (
```

```

    if c (sports m)  $\vee$  c (dports m) then proto m = Proto TCP  $\vee$  proto m = Proto
UDP  $\vee$  proto m = Proto L4-Protocol.SCTP else True))  $\wedge$ 

```

```

  valid-prefix-fw (src m)  $\wedge$  valid-prefix-fw (dst m)

```

proof –

```

  have simple-match-valid-alt-hlp1: {p. simple-match-port x p}  $\neq$  UNIV  $\iff$ 
(case x of (s,e)  $\implies$  s  $\neq$  0  $\vee$  e  $\neq$  max-word)

```

for *x* **using** *simple-match-port-UNIV* **by** *auto*

```

  have simple-match-valid-alt-hlp2: {p. simple-match-port x p}  $\neq$  {}  $\iff$  (case x
of (s,e)  $\implies$  s  $\leq$  e) for x by auto

```

show *?thesis*

unfolding *fun-eq-iff*

unfolding *simple-match-valid-def* *Let-def*

unfolding *simple-match-valid-alt-hlp1* *simple-match-valid-alt-hlp2*

```

  apply(clarify, rename-tac m, case-tac sports m; case-tac dports m; case-tac
proto m)

```



```

    by auto
qed

```

Example:

```

context
begin
  private definition example-simple-match1 ≡
    (iface = 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-iface = "", 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 by eval

  private 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 by eval
  private lemma  $\neg$  simple-match-valid example-simple-match2 by eval
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 –
  have simpl-ports-conjunct-not-UNIV:
    Collect (simple-match-port x) ≠ UNIV ⇒
      x = simpl-ports-conjunct p1 p2 ⇒
        Collect (simple-match-port p1) ≠ UNIV ∨ Collect (simple-match-port p2) ≠
        UNIV
  for x p1 p2 by (metis Collect-cong mem-Collect-eq simple-ports-conjunct-correct)

  have valid-prefix-fw (src m1) valid-prefix-fw (src m2) valid-prefix-fw (dst m1)
valid-prefix-fw (dst m2)
  using mv unfolding simple-match-valid-alt by simp-all
  moreover have ipcidr-conjunct (src m1) (src m2) = Some (src m)
    ipcidr-conjunct (dst m1) (dst m2) = Some (dst m)
  using mj by (cases m1; cases m2; cases m; simp split: option.splits)+
  ultimately have pv: valid-prefix-fw (src m) valid-prefix-fw (dst m)
  using ipcidr-conjunct-valid by blast+

```

```

define nmu where nmu ps  $\longleftrightarrow \{p. \text{simple-match-port } ps\} \neq UNIV$  for ps
have simpl-ports-conjunct (sports m1) (sports m2) = (sports m) (is ?ph1 sports)
  using mj by(cases m1; cases m2; cases m; simp split: option.splits)
hence sp: nmu (sports m)  $\longrightarrow nmu (sports m1) \vee nmu (sports m2)$ 
  (is ?ph2 sports)
unfolding nmu-def using simpl-ports-conjunct-not-UNIV by metis

  have ?ph1 dports using mj by(cases m1; cases m2; cases m; simp split: option.splits)
hence dp: ?ph2 dports unfolding nmu-def using simpl-ports-conjunct-not-UNIV
by metis

define php where php mr  $\longleftrightarrow proto\ mr \in Proto\ \{\text{TCP, UDP, L4-Protocol.SCTP}\}$ 
  for mr :: 'i simple-match
have pcj: simple-proto-conjunct (proto m1) (proto m2) = Some (proto m)
  using mj by(cases m1; cases m2; cases m; simp split: option.splits)
hence p: php m1  $\implies php\ m$ 
  php m2  $\implies php\ m$ 
using conjunctProtoD conjunctProtoD2 pcj unfolding php-def by auto

  have  $\bigwedge mx. \text{simple-match-valid } mx \implies nmu (sports\ mx) \vee nmu (dports\ mx)$ 
 $\implies php\ mx$ 
unfolding nmu-def php-def simple-match-valid-def by blast
hence mps: nmu (sports m1)  $\implies php\ m1$  nmu (dports m1)  $\implies php\ m1$ 
  nmu (sports m2)  $\implies php\ m2$  nmu (dports m2)  $\implies php\ m2$  using mv
by blast+

  have pa: nmu (sports m) \vee nmu (dports m)  $\longrightarrow php\ m$ 

  using sp dp mps p by fast

show ?thesis
unfolding simple-match-valid-def
using pv pa[unfolded nmu-def php-def] by blast
qed

```

definition *simple-fw-valid* $\equiv list-all (simple-match-valid \circ match-sel)$

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-prot = 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-prot = prot,
   p-sport = sport, p-dport = dport,
   p-tcp-flags = tcp-flags2, p-payload = payload2, ... = aux)

  by(simp add: simple-matches.simps)
end

```

9 List Product Helpers

```

theory List-Product-More
imports Main
begin

```

```

lemma List-product-concat-map: List.product xs ys = concat (map (λx. map (λy.
(x,y)) ys) xs)
  by(induction xs) (simp)+

```

```

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

```

```

lemma all-pairs-list-product: all-pairs xs = List.product xs xs
  by(simp add: all-pairs-def List-product-concat-map)

```

```

lemma all-pairs: ∀ (x,y) ∈ (set xs × set xs). (x,y) ∈ set (all-pairs xs)
  by(simp add: all-pairs-list-product)

```

```

lemma all-pairs-set: set (all-pairs xs) = set xs × set xs
  by(simp add: all-pairs-list-product)

```

```

end

```

10 Option to List and Option to Set

```

theory Option-Helpers
imports Main
begin

```

Those are just syntactic helpers.

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

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

lemma *set-option2list[simp]*: set (*option2list* k) = *option2set* k
unfolding *option2list-def option2set-def* **by** (*simp split: option.splits*)

lemma *option2list-simps[simp]*: *option2list* (Some x) = [x] *option2list* (None) = []
unfolding *option2list-def option.simps* **by**(*fact refl*)+

lemma *option2set-None*: *option2set* None = {}
by(*simp add: option2set-def*)

lemma *option2list-map*: *option2list* (map-option f n) = map f (*option2list* n)
by(*simp add: option2list-def split: option.split*)

lemma *option2set-map*: *option2set* (map-option f n) = f ` *option2set* n
by(*simp add: option2set-def split: option.split*)

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 \times 'a) list \Rightarrow ('i, 'pkt-ext) simple-packet-scheme \Rightarrow ('i simple-match \times 'a) option
where
generalized-sfw l p \equiv find ($\lambda(m,a). \text{simple-matches } m \text{ } p$) l

11.2 Lemmas

lemma *generalized-sfw-simps*:
generalized-sfw [] p = None
generalized-sfw (a # as) p = (if (case a of (m,-) \Rightarrow simple-matches m p) then Some a else *generalized-sfw* as p)
unfolding *generalized-sfw-def* **by** *simp-all*

lemma *generalized-sfw-append*:

*generalized-sfw (a @ b) p = (case generalized-sfw a p of Some x \Rightarrow Some x
| None \Rightarrow generalized-sfw b p)*
by(*induction a*) (*simp-all add: generalized-sfw-simps*)

lemma *simple-generalized-undecided*:

*simple-fw fw p \neq Undecided \Longrightarrow generalized-sfw (map simple-rule-dtor fw) p \neq
None*
by(*induction fw*)
(*clarsimp simp add: generalized-sfw-def simple-fw-alt simple-rule-dtor-def
split: prod.splits if-splits simple-action.splits simple-rule.splits*)**+**

lemma *generalized-sfwSomeD*: *generalized-sfw fw p = Some (r,d) \Longrightarrow (r,d) \in
set fw \wedge simple-matches r p*

unfolding *generalized-sfw-def*
by(*induction fw*) (*simp split: if-split-asm*)**+**

lemma *generalized-sfw-NoneD*: *generalized-sfw fw p = None \Longrightarrow $\forall (a,b) \in$ set
fw. \neg simple-matches a p*

by(*induction fw*) (*clarsimp simp add: generalized-sfw-simps split: if-splits*)**+**

lemma *generalized-fw-split*: *generalized-sfw fw p = Some r \Longrightarrow \exists fw1 fw3. fw =
fw1 @ r # fw3 \wedge generalized-sfw fw1 p = None*

apply(*induction fw rule: rev-induct*)
apply(*simp add: generalized-sfw-simps generalized-sfw-append split: option.splits;fail*)
apply(*clarsimp simp add: generalized-sfw-simps generalized-sfw-append split:
option.splits if-splits*)
apply *blast***+**
done

lemma *generalized-sfw-filterD*:

generalized-sfw (filter f fw) p = Some (r,d) \Longrightarrow simple-matches r p \wedge f (r,d)
by(*induction fw*) (*simp-all add: generalized-sfw-simps split: if-splits*)

lemma *generalized-sfw-mapsnd*:

*generalized-sfw (map (apsnd f) fw) p = map-option (apsnd f) (generalized-sfw
fw p)*
by(*induction fw*) (*simp add: generalized-sfw-simps split: prod.splits*)**+**

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 \Rightarrow state* **where**
*simple-action-to-decision a \equiv case a of Accept \Rightarrow Decision FinalAllow
| Drop \Rightarrow Decision FinalDeny*

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

lemma *simple-fw-iff-generalized-fw*:
 $simple-fw\ fw\ p = simple-action-to-decision\ a \longleftrightarrow (\exists r. generalized-sfw\ (map\ simple-rule-dtor\ fw)\ p = Some\ (r,a))$
by(*induction fw*)
(clarsimp simp add: generalized-sfw-simps simple-rule-dtor-def simple-fw-alt simple-action-to-decision-def split: simple-rule.splits if-splits simple-action.splits)+

lemma *simple-fw-iff-generalized-fw-accept*:
 $simple-fw\ fw\ p = Decision\ FinalAllow \longleftrightarrow (\exists r. generalized-sfw\ (map\ simple-rule-dtor\ fw)\ p = Some\ (r,\ Accept))$
by(*fact simple-fw-iff-generalized-fw[where a = simple-action.Accept, unfolded simple-action-to-decision-def simple-action.simps]*)

lemma *simple-fw-iff-generalized-fw-drop*:
 $simple-fw\ fw\ p = Decision\ FinalDeny \longleftrightarrow (\exists r. generalized-sfw\ (map\ simple-rule-dtor\ fw)\ p = Some\ (r,\ Drop))$
by(*fact simple-fw-iff-generalized-fw[where a = simple-action.Drop, unfolded simple-action-to-decision-def simple-action.simps]*)

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

definition *generalized-fw-join*
 $:: ('i::len\ simple-match \times 'a)\ list \Rightarrow ('i\ simple-match \times 'b)\ list \Rightarrow ('i\ simple-match \times 'a \times 'b)\ list$
where
 $generalized-fw-join\ l1\ l2 \equiv [(u,(a,b)). (m1,a) \leftarrow l1, (m2,b) \leftarrow l2, u \leftarrow option2list\ (simple-match-and\ m1\ m2)]$

lemma *generalized-fw-join-1-Nil[simp]*: $generalized-fw-join\ []\ f2 = []$
unfolding *generalized-fw-join-def* **by**(*induction f2*) *simp+*

lemma *generalized-fw-join-2-Nil[simp]*: $generalized-fw-join\ f1\ [] = []$
unfolding *generalized-fw-join-def* **by**(*induction f1*) *simp+*

lemma *generalized-fw-join-cons-1*:
 $generalized-fw-join\ ((am,ad) \# l1)\ l2 = [(u,(ad,b)). (m2,b) \leftarrow l2, u \leftarrow option2list\ (simple-match-and\ am\ m2)]\ @\ generalized-fw-join\ l1\ l2$
unfolding *generalized-fw-join-def* **by**(*simp*)

lemma *generalized-fw-join-1-nomatch*:
 $\neg\ simple-matches\ am\ p \Longrightarrow generalized-sfw\ [(u,(ad,b)). (m2,b) \leftarrow l2, u \leftarrow option2list\ (simple-match-and\ am\ m2)]\ p = None$
by(*induction l2*)
(clarsimp simp add: generalized-sfw-simps generalized-sfw-append option2list-def

simple-match-and-SomeD

split: prod.splits option.splits)+

lemma *generalized-fw-join-2-nomatch:*
 \neg *simple-matches* *bm p* \implies
generalized-sfw (*generalized-fw-join as* ((*bm*, *bd*) # *bs*)) *p* = *generalized-sfw*
(*generalized-fw-join as* *bs*) *p*
proof(*induction as*)
case (*Cons a as*)
note *mIH* = *Cons.IH[OF Cons.premis]*
obtain *am ad* **where** *a[simp]: a = (am, ad)* **by**(*cases a*)
have *: *generalized-sfw* (*concat* (*map* ($\lambda(m2, b).$ *map* ($\lambda u.$ (*u, ad, b*)) (*option2list*
(*simple-match-and am m2*))) ((*bm, bd*) # *bs*))) *p* =
generalized-sfw (*concat* (*map* ($\lambda(m2, b).$ *map* ($\lambda u.$ (*u, ad, b*)) (*option2list*
(*simple-match-and am m2*))) *bs*)) *p*
unfolding *list.map prod.simps*
apply(*cases simple-match-and am bm*)
apply(*simp add: option2list-def; fail*)
apply(*frule simple-match-and-SomeD[of - - - p]*)
apply(*subst option2list-def*)
apply(*unfold concat.simps*)
apply(*simp add: generalized-sfw-simps Cons.premis*)
done
show ?*case*
unfolding *a*
unfolding *generalized-fw-join-cons-1*
unfolding *generalized-sfw-append*
unfolding *mIH*
unfolding *
..
qed(*simp add: generalized-fw-join-def*)

lemma *generalized-fw-joinI:*
 \llbracket *generalized-sfw f1 p = Some (r1,d1); generalized-sfw f2 p = Some (r2,d2) $\rrbracket \implies$
generalized-sfw (*generalized-fw-join f1 f2*) *p* = *Some* (*the* (*simple-match-and*
r1 r2), *d1,d2*)
proof(*induction f1*)
case (*Cons a as*)
obtain *am ad* **where** *a[simp]: a = Pair am ad* **by**(*cases a*)
show ?*case* **proof**(*cases simple-matches am p*)
case *True*
hence *dra: d1 = ad r1 = am* **using** *Cons.premis* **by**(*simp-all add: general-*
ized-sfw-simps)
from *Cons.premis(2)* **show** ?*thesis* **unfolding** *a dra*
proof(*induction f2*)
case (*Cons b bs*)
obtain *bm bd* **where** *b[simp]: b = Pair bm bd* **by**(*cases b*)
thus ?*case*
proof(*cases simple-matches bm p*)*

```

case True
  hence drb:  $d2 = bd$   $r2 = bm$  using Cons.prems by(simp-all add:
generalized-sfw-simps)
  from True (simple-matches am p) obtain ruc where sma[simp]:
    simple-match-and am bm = Some ruc simple-matches ruc p
  using simple-match-and-correct[of am p bm]
  by(simp split: option.splits)
  show ?thesis unfolding b
  by(simp add: generalized-fw-join-def option2list-def generalized-sfw-simps
drb)
next
case False
with Cons.prems have bd:
  generalized-sfw (b # bs) p = generalized-sfw bs p
  generalized-sfw (b # bs) p = Some (r2, d2)
by(simp-all add: generalized-sfw-simps)
note mIH = Cons.IH[OF bd(2)[unfolded bd(1)]]
show ?thesis
  unfolding mIH[symmetric] b
  using generalized-fw-join-2-nomatch[OF False, of (am, ad) # as bd bs]
  .
qed
qed(simp add: generalized-sfw-simps generalized-fw-join-def)

next
case False
with Cons.prems have generalized-sfw (a # as) p = generalized-sfw as p
  by(simp add: generalized-sfw-simps)
with Cons.prems have generalized-sfw as p = Some (r1, d1) by simp
note mIH = Cons.IH[OF this Cons.prem(2)]
show ?thesis
  unfolding mIH[symmetric] a
  unfolding generalized-fw-join-cons-1
  unfolding generalized-sfw-append
  unfolding generalized-fw-join-1-nomatch[OF False, of ad f2]
  by simp
qed
qed(simp add: generalized-fw-join-def generalized-sfw-simps;fail)

```

```

lemma generalized-fw-joinD:
  generalized-sfw (generalized-fw-join f1 f2) p = Some (u, d1,d2) ==>
   $\exists r1 r2. \text{generalized-sfw } f1 \text{ } p = \text{Some } (r1,d1) \wedge \text{generalized-sfw } f2 \text{ } p = \text{Some}$ 
  ( $r2,d2$ )  $\wedge \text{Some } u = \text{simple-match-and } r1 \text{ } r2$ 
proof(induction f1)
case (Cons a as)
obtain am ad where a[simp]:  $a = \text{Pair } am \text{ } ad$  by(cases a)

```



```

show ?case proof(cases simple-matches am p, rule exI)
  case True
  show  $\exists r2. \text{generalized-sfw } (a \# as) p = \text{Some } (am, d1) \wedge \text{generalized-sfw } f2$ 
   $p = \text{Some } (r2, d2) \wedge \text{Some } u = \text{simple-match-and } am r2$ 
  using Cons.prem.s
  proof(induction f2)
  case (Cons b bs)
  obtain bm bd where b[simp]: b = Pair bm bd by(cases b)
  show ?case
  proof(cases simple-matches bm p, rule exI)
  case True
  with (simple-matches am p) obtain u'
  where sma: simple-match-and am bm = Some u'  $\wedge$  simple-matches u' p
  using simple-match-and-correct[of am p bm] by(simp split: option.splits)
  show  $\text{generalized-sfw } (a \# as) p = \text{Some } (am, d1) \wedge \text{generalized-sfw } (b \#$ 
  bs)  $p = \text{Some } (bm, d2) \wedge \text{Some } u = \text{simple-match-and } am bm$ 
  using Cons.prem.s True (simple-matches am p)
  by(simp add: generalized-fw-join-def generalized-sfw-append sma general-
  ized-sfw-simps)
  next
  case False
  have  $\text{generalized-sfw } (\text{generalized-fw-join } (a \# as) bs) p = \text{Some } (u, d1,$ 
  d2)
  using Cons.prem.s unfolding b unfolding generalized-fw-join-2-nomatch[OF
  False] .
  note Cons.IH[OF this]
  moreover have  $\text{generalized-sfw } (b \# bs) p = \text{generalized-sfw } bs p$  using
  False by(simp add: generalized-sfw-simps)
  ultimately show ?thesis by presburger
  qed
  qed(simp add: generalized-sfw-simps)
next
  case False
  with Cons.prem.s have  $\text{generalized-sfw } (\text{generalized-fw-join } as f2) p = \text{Some}$ 
  (u, d1, d2) by(simp add: generalized-fw-join-cons-1 generalized-sfw-append gener-
  alized-fw-join-1-nomatch)
  note Cons.IH[OF this]
  moreover have  $\text{generalized-sfw } (a \# as) p = \text{generalized-sfw } as p$  using False
  by(simp add: generalized-sfw-simps)
  ultimately show ?thesis by presburger
  qed
  qed(simp add: generalized-fw-join-def generalized-sfw-simps)

```

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  $\equiv$ 
  map  $(\lambda(u,a,b). \text{SimpleRule } u \text{ (if } a = \text{Accept} \wedge b = \text{Accept} \text{ then } \text{Accept} \text{ else$ 

```

```

Drop))
  shows
    simple-fw rs1 p = Decision FinalAllow  $\wedge$  simple-fw rs2 p = Decision FinalAllow
 $\longleftrightarrow$ 
    simple-fw (rule-translate (generalized-fw-join (map simple-rule-dtor rs1) (map
simple-rule-dtor rs2))) p = Decision FinalAllow
  proof -
    have hlp1:
      simple-rule-dtor  $\circ$  ( $\lambda(u, a, b). \text{SimpleRule } u \text{ (if } a = \text{Accept} \wedge b = \text{Accept} \text{ then}$ 
Accept else Drop)) =
      apsnd ( $\lambda(a, b). \text{if } a = \text{Accept} \wedge b = \text{Accept} \text{ then Accept else Drop}$ )
    unfolding fun-eq-iff comp-def by(simp add: simple-rule-dtor-def)
    show ?thesis
    unfolding simple-fw-iff-generalized-fw-accept
    apply(rule)
    apply(clarify)
    apply(drule (1) generalized-fw-joinI)
    apply(simp add: hlp1 rule-translate-def generalized-sfw-mapsnd ;fail)
    apply(clarsimp simp add: hlp1 generalized-sfw-mapsnd rule-translate-def)
    apply(drule generalized-fw-joinD)
    apply(clarsimp split: if-splits)
  done
qed

```

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* \equiv map (apsnd ($\lambda(a,b) \Rightarrow$ (case a of Accept \Rightarrow b
| Drop \Rightarrow Drop)))

shows simple-fw rs1 p = Decision FinalAllow \wedge simple-fw rs2 p = Decision FinalAllow \longleftrightarrow

($\exists m. (\text{generalized-sfw } (\text{to-simple-rule-list}$
(generalized-fw-join (map simple-rule-dtor rs1) (map simple-rule-dtor
rs2))) p = Some (m, Accept))

unfolding simple-fw-iff-generalized-fw-accept

apply(rule)

apply(clarify)

apply(drule (1) generalized-fw-joinI)

apply(clarsimp simp add: to-simple-rule-list-def generalized-sfw-mapsnd; fail)

apply(clarsimp simp add: to-simple-rule-list-def generalized-sfw-mapsnd)

apply(drule generalized-fw-joinD)

apply(clarsimp split: if-splits simple-action.splits)

done

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

generalized-fw-join [(m1,d1)] fw2 = foldr ($\lambda(m2,d2). (@)$) (case simple-match-and

```

m1 m2 of None  $\Rightarrow$  [] | Some mu  $\Rightarrow$  [(mu,d1,d2))] fw2 []
proof -
  have concat-map-foldr: concat (map ( $\lambda x. f x$ ) l) = foldr ( $\lambda x. (@) (f x)$ ) l [] for
f :: 'x  $\Rightarrow$  'y list and l
  by(induction l) simp-all
  show ?thesis
  apply(simp add: generalized-fw-join-cons-1 option2list-def)
  apply(simp add: concat-map-foldr)
  apply(unfold list.map prod.case-distrib option.case-distrib)
  by simp
qed

lemma generalized-sfw-2-join-None:
  generalized-sfw fw2 p = None  $\implies$  generalized-sfw (generalized-fw-join fw1 fw2)
p = None
  by(induction fw2) (simp-all add: generalized-sfw-simps generalized-sfw-append
generalized-fw-join-2-nomatch split: if-splits option.splits prod.splits)

lemma generalized-sfw-1-join-None:
  generalized-sfw fw1 p = None  $\implies$  generalized-sfw (generalized-fw-join fw1 fw2)
p = None
  by(induction fw1) (simp-all add: generalized-sfw-simps generalized-fw-join-cons-1
generalized-sfw-append generalized-fw-join-1-nomatch split: if-splits option.splits prod.splits)

lemma generalized-sfw-join-set: (a, b1, b2)  $\in$  set (generalized-fw-join f1 f2)  $\longleftrightarrow$ 
( $\exists a1 a2. (a1, b1) \in$  set f1  $\wedge (a2, b2) \in$  set f2  $\wedge$  simple-match-and a1 a2 =
Some a)
  unfolding generalized-fw-join-def
  apply(rule iffI)
  subgoal unfolding generalized-fw-join-def by(clarsimp simp: option2set-def
split: option.splits) blast
  by(clarsimp simp: option2set-def split: option.splits) fastforce

```

11.5 Validity

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

definition gsfw-valid :: ('i::len simple-match \times 'c) list \Rightarrow bool **where**
gsfw-valid \equiv list-all (simple-match-valid \circ fst)

lemma gsfw-join-valid: gsfw-valid f1 \implies gsfw-valid f2 \implies gsfw-valid (generalized-fw-join f1 f2)

```

unfolding gsfw-valid-def
apply(induction f1)
apply(simp;fail)
apply(simp)
apply(rename-tac a f1)
apply(case-tac a)
apply(simp add: generalized-fw-join-cons-1)

```

```

apply(clarify)
apply(thin-tac list-all - f1)
apply(thin-tac list-all - (generalized-fw-join - -))
apply(induction f2)
apply(simp;fail)
apply(simp)
apply(clarsimp simp add: option2list-def list-all-iff)
using simple-match-and-valid apply metis
done

```

```

lemma gsfw-validI: simple-fw-valid fw  $\implies$  gsfw-valid (map simple-rule-dtor fw)
  unfolding gsfw-valid-def simple-fw-valid-def
  by(clarsimp simp add: simple-rule-dtor-def list-all-iff split: simple-rule.splits)
fastforce

```

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  $\Rightarrow$  'i simple-packet set  $\Rightarrow$  'i simple-rule list
where
  rmshadow [] - = [] |
  rmshadow ((SimpleRule m a)#rs) P = (if ( $\forall p \in P. \neg$  simple-matches m p)
    then
      rmshadow rs P
    else
      (SimpleRule m a) # (rmshadow rs {p  $\in$  P.  $\neg$  simple-matches m p}))

```

12.1.1 Soundness

```

lemma rmshadow-sound:
  p  $\in$  P  $\implies$  simple-fw (rmshadow rs P) p = simple-fw rs p
proof(induction rs arbitrary: P)
case Nil thus ?case by simp
next
case (Cons r rs)
  from Cons.IH Cons.prem have IH1: simple-fw (rmshadow rs P) p = simple-fw
rs p by (simp)
  let ?P'={p  $\in$  P.  $\neg$  simple-matches (match-sel r) p}

```

```

from Cons.IH Cons.premis have IH2:  $\bigwedge m. p \in ?P' \implies \text{simple-fw } (\text{rmshadow}$ 
rs ?P')  $p = \text{simple-fw } rs \ p$  by simp
from Cons.premis show ?case
  apply(cases r, rename-tac m a)
  apply(simp)
  apply(case-tac  $\forall p \in P. \neg \text{simple-matches } m \ p$ )
  apply(simp add: IH1 nomatch)
  apply(case-tac  $p \in ?P'$ )
  apply(frule IH2)
  apply(simp add: nomatch IH1)
  apply(simp)
  apply(case-tac a)
  apply(simp-all)
by fast+
qed

```

corollary *rmshadow*:

```

fixes p :: 'i::len simple-packet
shows simple-fw (rmshadow rs UNIV) p = simple-fw rs p
using rmshadow-sound[of p] by simp

```

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 \Rightarrow 'i simple-packet set \Rightarrow 'i simple-rule list **where**

```

  rmshadow' [] - = [] |
  rmshadow' ((SimpleRule m a)#rs) P = (if {p. simple-matches m p}  $\subseteq$  P
    then
      rmshadow' rs P
    else
      (SimpleRule m a) # (rmshadow' rs (P  $\cup$  {p. simple-matches m p})))

```

lemma *rmshadow'-sound*:

```

   $p \notin P \implies \text{simple-fw } (\text{rmshadow}' rs P) \ p = \text{simple-fw } rs \ p$ 
proof(induction rs arbitrary: P)
case Nil thus ?case by simp
next
case (Cons r rs)
  from Cons.IH Cons.premis have IH1:  $\text{simple-fw } (\text{rmshadow}' rs P) \ p = \text{simple-fw}$ 
rs p by (simp)
  let ?P'={p. simple-matches (match-sel r) p}
  from Cons.IH Cons.premis have IH2:  $\bigwedge m. p \notin (\text{Collect } (\text{simple-matches } m))$ 
 $\implies \text{simple-fw } (\text{rmshadow}' rs (P \cup \text{Collect } (\text{simple-matches } m))) \ p = \text{simple-fw } rs$ 
p by simp
  have nomatch-m:  $\bigwedge m. p \notin P \implies \{p. \text{simple-matches } m \ p\} \subseteq P \implies \neg$ 
simple-matches m p by blast
  from Cons.premis show ?case
    apply(cases r, rename-tac m a)
    apply(simp)

```

```

apply(case-tac {p. simple-matches m p}  $\subseteq P$ )
apply(simp add: IH1)
apply(drule nomatch-m)
apply(assumption)
apply(simp add: nomatch)
apply(simp)
apply(case-tac a)
apply(simp-all)
apply(simp-all add: IH2)
done
qed

```

corollary

```

fixes p :: 'i::len simple-packet
shows simple-fw (rmshadow rs UNIV) p = simple-fw (rmshadow' rs {}) p
using rmshadow'-sound[of p] rmshadow-sound[of p] by simp

```

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  $\Rightarrow$  'i simple-packet set where

```

```

simple-packet-set-toSet ms = {p.  $\exists m \in set ms. simple-matches m p$ }

```

```

private lemma simple-packet-set-toSet-alt: simple-packet-set-toSet ms = ( $\bigcup m \in set ms. \{p. simple-matches m p\}$ )

```

```

unfolding simple-packet-set-toSet-def by blast

```

```

private definition simple-packet-set-union :: 'i::len simple-packet-set  $\Rightarrow$  'i simple-match  $\Rightarrow$  '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  $\cup$  {p. simple-matches m p}

```

```

unfolding simple-packet-set-toSet-def simple-packet-set-union-def by simp blast

```

```

private lemma ( $\exists m' \in set ms.$ 

```

```

{i. match-iface iif i}  $\subseteq$  {i. match-iface (iiface m') i}  $\wedge$ 

```

```

{i. match-iface oif i}  $\subseteq$  {i. match-iface (oiface m') i}  $\wedge$ 

```

```

{ip. simple-match-ip sip ip}  $\subseteq$  {ip. simple-match-ip (src m') ip}  $\wedge$ 

```

```

{ip. simple-match-ip dip ip}  $\subseteq$  {ip. simple-match-ip (dst m') ip}  $\wedge$ 

```

```

{p. match-proto protocol p}  $\subseteq$  {p. match-proto (proto m') p}  $\wedge$ 

```

```

    {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 (|iface=iif, oiface=oif, src=sip, dst=dip, proto=protocol,
sports=sps, dports=dps |) p} ⊆ (simple-packet-set-toSet ms)
  unfolding simple-packet-set-toSet-def simple-packet-set-union-def
  apply (simp add: simple-matches.simps)
  apply (simp add: Set.Collect-mono-iff)
  apply clarify
  apply meson
  done

```

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 -
    have ?l ⟷ {p. simple-matches m p} - (simple-packet-set-toSet ms) = {}
  by blast
    also have ... ⟷ {p. simple-matches m p} - (∪ m ∈ set ms. {p: 'i::len
simple-packet. simple-matches m p}) = {}
    using simple-packet-set-toSet-alt by blast
    also have ... ⟷ ?r by blast
    finally show ?thesis .
  qed

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 \Rightarrow 'a set set \Rightarrow bool **where**
ipPartition $A B \equiv \forall a \in A. \forall b \in B. a \cap b = \{\} \vee b \subseteq a$

definition *disjoint-list* :: 'a set list \Rightarrow bool **where**
disjoint-list $ls \equiv \text{distinct } ls \wedge \text{disjoint } (\text{set } ls)$

context begin

private fun *disjoint-list-rec* :: 'a set list \Rightarrow bool **where**
disjoint-list-rec [] = True |
disjoint-list-rec (x#xs) = (x \cap \bigcup (set xs) = $\{\}$) \wedge *disjoint-list-rec* xs)

private lemma *disjoint-equi*: *disjoint-list-rec* ts \Longrightarrow *disjoint* (set ts)
apply(*induction* ts)
apply(*simp-all* add: *disjoint-def*)
by *fast*

private lemma *disjoint-list-disjoint-list-rec*: *disjoint-list* ts \Longrightarrow *disjoint-list-rec* ts
apply(*induction* ts)
apply(*simp-all* add: *disjoint-list-def* *disjoint-def*)
by *fast*

private definition *addSubsetSet* :: 'a set \Rightarrow 'a set set \Rightarrow 'a set set **where**
addSubsetSet s ts = insert (s - \bigcup ts) (((\cap) s) ' ts) \cup (($\lambda x. x - s$) ' ts)

private fun *partitioning* :: 'a set list \Rightarrow 'a set set \Rightarrow '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}} **by** *eval*

lemma \bigcup [{1::nat,2},{3,4},{5,6,7},{6},{10}] = \bigcup (*partitioning* [{1,2},{3,4},{5,6,7},{6},{10}] {}) **by** *eval*

lemma *disjoint* (*partitioning* [{1::nat,2},{3,4},{5,6,7},{6},{10}] {}) **by** *eval*

lemma *ipPartition* [{1::nat,2},{3,4},{5,6,7},{6},{10}] (*partitioning* [{1::nat,2},{3,4},{5,6,7},{6},{10}] {}) **by** *eval*

lemma *ipPartition* A {} **by**(*simp* add: *ipPartition-def*)

lemma *ipPartitionUnion*: *ipPartition* As Cs \wedge *ipPartition* Bs Cs \longleftrightarrow *ipPartition* (As \cup Bs) Cs

unfolding *ipPartition-def* **by** *fast*


```

private lemma disjointAddSubset:  $\text{disjoint } ts \implies \text{disjoint } (\text{addSubsetSet } a \ ts)$ 
  by (auto simp add: disjoint-def addSubsetSet-def)

private lemma coversallAddSubset:  $\bigcup (\text{insert } a \ ts) = \bigcup (\text{addSubsetSet } a \ ts)$ 
  by (auto simp add: addSubsetSet-def)

private lemma ipPartitioningAddSubset0:  $\text{disjoint } ts \implies \text{ipPartition } ts (\text{addSubsetSet } a \ ts)$ 
  apply (simp add: addSubsetSet-def ipPartition-def)
  apply (safe)
  apply blast
  apply (simp-all add: disjoint-def)
  apply blast+
  done

private lemma ipPartitioningAddSubset1:  $\text{disjoint } ts \implies \text{ipPartition } (\text{insert } a \ ts) (\text{addSubsetSet } a \ ts)$ 
  apply (simp add: addSubsetSet-def ipPartition-def)
  apply (safe)
  apply blast
  apply (simp-all add: disjoint-def)
  apply blast+
  done

private lemma addSubsetSetI:
   $s - \bigcup ts \in \text{addSubsetSet } s \ ts$ 
   $t \in ts \implies s \cap t \in \text{addSubsetSet } s \ ts$ 
   $t \in ts \implies t - s \in \text{addSubsetSet } s \ ts$ 
  unfolding addSubsetSet-def by blast+

private lemma addSubsetSetE:
  assumes  $A \in \text{addSubsetSet } s \ ts$ 
  obtains  $A = s - \bigcup ts \mid T \text{ where } T \in ts \ A = s \cap T \mid T \text{ where } T \in ts \ A = T - s$ 
  using assms unfolding addSubsetSet-def by blast

private lemma Union-addSubsetSet:  $\bigcup (\text{addSubsetSet } b \ As) = b \cup \bigcup As$ 
  unfolding addSubsetSet-def by auto

private lemma addSubsetSetCom:  $\text{addSubsetSet } a (\text{addSubsetSet } b \ As) = \text{addSubsetSet } b (\text{addSubsetSet } a \ As)$ 
  proof –
  {
    fix  $A \ a \ b$  assume  $A \in \text{addSubsetSet } a (\text{addSubsetSet } b \ As)$ 
    hence  $A \in \text{addSubsetSet } b (\text{addSubsetSet } a \ As)$ 
    apply (rule addSubsetSetE)
    proof (goal-cases)
      case 1
      assume  $A = a - \bigcup (\text{addSubsetSet } b \ As)$ 

```

```

    hence  $A = (a - \bigcup As) - b$  by (auto simp add: Union-addSubsetSet)
    thus ?thesis by (auto intro: addSubsetSetI)
  next
    case (2 T)
    have  $A = b \cap (a - \bigcup As) \vee (\exists S \in As. A = b \cap (a \cap S)) \vee (\exists S \in As. A = (a \cap S) - b)$ 
    by (rule addSubsetSetE[OF 2(1)]) (auto simp: 2(2))
    thus ?thesis by (blast intro: addSubsetSetI)
  next
    case (3 T)
    have  $A = b - \bigcup (addSubsetSet a As) \vee (\exists S \in As. A = b \cap (S - a)) \vee (\exists S \in As. A = (S - a) - b)$ 
    by (rule addSubsetSetE[OF 3(1)]) (auto simp: 3(2) Union-addSubsetSet)
    thus ?thesis by (blast intro: addSubsetSetI)
  qed
}
thus ?thesis by blast
qed

```

```

private lemma ipPartitioningAddSubset2: ipPartition {a} (addSubsetSet a ts)
  apply(simp add: addSubsetSet-def ipPartition-def)
  by blast

```

```

private lemma disjointPartitioning-helper : disjoint As  $\implies$  disjoint (partitioning ss As)
proof(induction ss arbitrary: As)
  case Nil thus ?case by(simp)
  next
  case (Cons s ss)
  from Cons.prem1 disjointAddSubset have d: disjoint (addSubsetSet s As) by fast
  from Cons.IH d have disjoint (partitioning ss (addSubsetSet s As)) .
  thus ?case by simp
qed

```

```

private lemma disjointPartitioning: disjoint (partitioning ss {})
proof -
  have disjoint {} by(simp add: disjoint-def)
  from this disjointPartitioning-helper show ?thesis by fast
qed

```

```

private lemma coversallPartitioning:  $\bigcup (set ts) = \bigcup (partitioning ts \{\})$ 
proof -
  have  $\bigcup (set ts \cup As) = \bigcup (partitioning ts As)$  for As
  apply(induction ts arbitrary: As)
  apply(simp-all)
  by (metis Union-addSubsetSet sup-left-commute)
  thus ?thesis by (metis sup-bot.right-neutral)
qed

```

```

private lemma  $\bigcup A s = \bigcup B s \implies ipPartition A s \implies ipPartition A s$ 
(addSubsetSet a Bs)
  by(auto simp add: ipPartition-def addSubsetSet-def)

private lemma ipPartitionSingleSet:  $ipPartition \{t\} (addSubsetSet t Bs)$ 
 $\implies ipPartition \{t\} (partitioning ts (addSubsetSet t Bs))$ 
  apply(induction ts arbitrary: Bs t)
  apply(simp-all)
  by (metis addSubsetSetCom ipPartitioningAddSubset2)

private lemma ipPartitioning-helper:  $disjoint A s \implies ipPartition (set ts) (partitioning$ 
ts As)
  proof(induction ts arbitrary: As)
  case Nil thus ?case by(simp add: ipPartition-def)
  next
  case (Cons t ts)
  from Cons.prems ipPartitioningAddSubset0 have d:  $ipPartition A s (addSubsetSet$ 
t As) by blast
  from Cons.prems Cons.IH d disjointAddSubset ipPartitioningAddSubset1
  have e:  $ipPartition (set ts) (partitioning ts (addSubsetSet t As))$  by blast
  from ipPartitioningAddSubset2 Cons.prems
  have  $ipPartition \{t\} (addSubsetSet t As)$  by blast
  from this Cons.prems ipPartitionSingleSet
  have f:  $ipPartition \{t\} (partitioning ts (addSubsetSet t As))$  by fast
  have  $set (t \# ts) = insert t (set ts)$  by auto
  from ipPartitionUnion have  $\bigwedge A s B s C s. ipPartition A s C s \implies ipPartition$ 
B s C s \implies ipPartition (A s \cup B s) C s by fast
  with this e f
  have  $ipPartition (set (t \# ts)) (partitioning ts (addSubsetSet t As))$  by
fastforce
  thus ?case by simp
  qed

private lemma ipPartitioning:  $ipPartition (set ts) (partitioning ts \{\})$ 
  proof –
  have  $disjoint \{\}$  by(simp add: disjoint-def)
  from this ipPartitioning-helper show ?thesis by fast
  qed

private lemma inter-dif-help-lemma:  $A \cap B = \{\} \implies B - S = B - (S - A)$ 
  by blast

private lemma disjoint-list-lem:  $disjoint-list ls \implies \forall s \in set(ls). \forall t \in set(ls). s$ 
 $\neq t \implies s \cap t = \{\}$ 
  proof(induction ls)
  qed(simp-all add: disjoint-list-def disjoint-def)

```

```

private lemma disjoint-list-empty: disjoint-list []
  by (simp add: disjoint-list-def disjoint-def)

private lemma disjoint-sublist: disjoint-list (t#ts)  $\implies$  disjoint-list ts
  proof(induction ts arbitrary: t)
  qed(simp-all add: disjoint-list-empty disjoint-list-def disjoint-def)

private fun intersection-list :: 'a set  $\implies$  'a set list  $\implies$  '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  $\implies$  'a set list  $\implies$  '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  $\implies$  a  $\in$  A  $\implies$  b  $\subseteq$  a  $\implies$  disjoint
((A - {a})  $\cup$  {b})
  apply(simp add: disjoint-def)
  by blast

private lemma disjoint-intersection: disjoint A  $\implies$  a  $\in$  A  $\implies$  disjoint ({a  $\cap$ 
b}  $\cup$  (A - {a}))
  apply(simp add: disjoint-def)
  by(blast)

private lemma intList-equi: disjoint-list-rec ts  $\implies$  intersection-list s ts = inter-
section-list-opt s ts
  proof(induction ts)
  case Nil thus ?case by simp
  next
  case (Cons t ts)
  from Cons.prem have intersection-list-opt s ts = intersection-list-opt (s - t)
ts
    proof(induction ts arbitrary: s t)
    case Nil thus ?case by simp
    next
    case Cons
    have  $\forall t \in \text{set } ts. u \cap t = \{\}$   $\implies$  intersection-list-opt s ts = intersection-list-opt
(s - u) ts
      for u
      apply(induction ts arbitrary: s u)
      apply(simp-all)
    by (metis Diff-Int-distrib2 Diff-empty Diff-eq Un-Diff-Int sup-bot.right-neutral)
    with Cons show ?case
      apply(simp)
      by (metis Diff-Int-distrib2 Diff-empty Un-empty inf-sup-distrib1)
    qed
  with Cons show ?case by simp

```

qed

```
private fun difference-list :: 'a set  $\Rightarrow$  'a set list  $\Rightarrow$  'a set list where
  difference-list - [] = [] |
  difference-list s (t#ts) = (t - s)#(difference-list s ts)
```

```
private fun difference-list-opt :: 'a set  $\Rightarrow$  'a set list  $\Rightarrow$  '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  $\implies$  difference-list s ts = differ-
ence-list-opt s ts
```

```
proof(induction ts arbitrary: s)
```

```
case Nil thus ?case by simp
```

```
next
```

```
case (Cons t ts)
```

```
  have difference-list-opt-lem0:  $\forall t \in \text{set}(ts). u \cap t = \{\} \implies$ 
```

```
    difference-list-opt s ts = difference-list-opt (s - u) ts
```

```
  for u proof(induction ts arbitrary: s u)
```

```
  case Cons thus ?case
```

```
    apply(simp-all add: inter-dif-help-lemma)
```

```
    by (metis Diff-Int-distrib2 Diff-eq Un-Diff-Int sup-bot.right-neutral)
```

```
  qed(simp)
```

```
  have disjoint-list-rec (t # ts)  $\implies$  difference-list-opt s ts = difference-list-opt
(s - t) ts
```

```
  proof(induction ts arbitrary: s t)
```

```
  case Cons thus ?case
```

```
    apply(simp-all add: difference-list-opt-lem0)
```

```
    by (metis Un-empty inf-sup-distrib1 inter-dif-help-lemma)
```

```
  qed(simp)
```

```
  with Cons show ?case by simp
```

```
qed
```

```
private fun partList0 :: 'a set  $\Rightarrow$  'a set list  $\Rightarrow$  'a set list where
```

```
  partList0 s [] = [] |
```

```
  partList0 s (t#ts) = (s  $\cap$  t)#((t - s)#(partList0 s ts))
```

```
private lemma partList0-set-equi: set(partList0 s ts) = ((( $\cap$ ) s) ‘ (set ts))  $\cup$ 
(( $\lambda x. x - s$ ) ‘ (set ts))
```

```
  by(induction ts arbitrary: s) auto
```

```
private lemma partList-sub-equi0: set(partList0 s ts) =
```

```
  set(difference-list s ts)  $\cup$  set(intersection-list s ts)
```

```
  by(induction ts arbitrary: s) simp+
```

```
private fun partList1 :: 'a set  $\Rightarrow$  'a set list  $\Rightarrow$  'a set list where
```

```
  partList1 s [] = [] |
```

```
  partList1 s (t#ts) = (s  $\cap$  t)#((t - s)#(partList1 (s - t) ts))
```

```

private lemma partList-sub-equi: set(partList1 s ts) =
    set(difference-list-opt s ts)  $\cup$  set(intersection-list-opt s ts)
by(induction ts arbitrary: s) (simp-all)

private lemma partList0-partList1-equi: disjoint-list-rec ts  $\implies$  set (partList0 s
ts) = set (partList1 s ts)
by (simp add: partList-sub-equi partList-sub-equi0 intList-equi difList-equi)

private fun partList2 :: 'a set  $\implies$  'a set list  $\implies$  'a set list where
    partList2 s [] = [] |
    partList2 s (t#ts) = (if s  $\cap$  t = {} then (t#(partList2 (s - t) ts))
        else (s  $\cap$  t)#((t - s)#(partList2 (s - t) ts)))

private lemma partList2-empty: partList2 {} ts = ts
by(induction ts) (simp-all)

private lemma partList1-partList2-equi: set(partList1 s ts) - {{}} = set(partList2
s ts) - {{}}
by(induction ts arbitrary: s) (auto)

private fun partList3 :: 'a set  $\implies$  'a set list  $\implies$  'a set list where
    partList3 s [] = [] |
    partList3 s (t#ts) = (if s = {} then (t#ts) else
        (if s  $\cap$  t = {} then (t#(partList3 (s - t) ts))
            else
                (if t - s = {} then (t#(partList3 (s - t) ts))
                    else (t  $\cap$  s)#((t - s)#(partList3 (s - t) ts))))))

private lemma partList2-partList3-equi: set(partList2 s ts) - {{}} = set(partList3
s ts) - {{}}
apply(induction ts arbitrary: s)
apply(simp; fail)
apply(simp add: partList2-empty)
by blast

fun partList4 :: 'a set  $\implies$  'a set list  $\implies$  'a set list where
    partList4 s [] = [] |
    partList4 s (t#ts) = (if s = {} then (t#ts) else
        (if s  $\cap$  t = {} then (t#(partList4 s ts))
            else
                (if t - s = {} then (t#(partList4 (s - t) ts))
                    else (t  $\cap$  s)#((t - s)#(partList4 (s - t) ts))))))

private lemma partList4: partList4 s ts = partList3 s ts
apply(induction ts arbitrary: s)
apply(simp; fail)
apply (simp add: Diff-triv)
done

```

```

private lemma partList0-addSubsetSet-equi:  $s \subseteq \bigcup (\text{set } ts) \implies$ 
      addSubsetSet  $s$  (set  $ts$ ) -  $\{\{\}\}$  = set(partList0  $s$   $ts$ )
-  $\{\{\}\}$ 
  by(simp add: addSubsetSet-def partList0-set-equi)

private fun partitioning-nontail :: 'a set list  $\Rightarrow$  'a set set  $\Rightarrow$  '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  $ts$ ) = parti-
tioning ss (addSubsetSet  $a$   $ts$ )
  apply(induction ss arbitrary:  $a$   $ts$ )
  apply(simp; fail)
  apply(simp add: addSubsetSetCom)
done

private lemma partitioning-nottail-equi: partitioning-nontail ss  $ts$  = partitioning
ss  $ts$ 
  apply(induction ss arbitrary:  $ts$ )
  apply(simp; fail)
  apply(simp add: addSubsetSetCom partitioningCom)
done

fun partitioning1 :: 'a set list  $\Rightarrow$  'a set list  $\Rightarrow$  'a set list where
  partitioning1 []  $ts$  =  $ts$  |
  partitioning1 (s#ss)  $ts$  = partList4  $s$  (partitioning1 ss  $ts$ )

lemma partList4-empty:  $\{\}$   $\notin$  set  $ts \implies \{\}$   $\notin$  set (partList4  $s$   $ts$ )
  apply(induction  $ts$  arbitrary:  $s$ )
  apply(simp; fail)
  by auto

lemma partitioning1-empty0:  $\{\}$   $\notin$  set  $ts \implies \{\}$   $\notin$  set (partitioning1 ss  $ts$ )
  apply(induction ss arbitrary:  $ts$ )
  apply(simp; fail)
  apply(simp add: partList4-empty)
done

lemma partitioning1-empty1:  $\{\}$   $\notin$  set  $ts \implies$ 
      set(partitioning1 ss  $ts$ ) -  $\{\{\}\}$  = set(partitioning1 ss  $ts$ )
  by(simp add: partitioning1-empty0)

lemma partList4-subset:  $a \subseteq \bigcup (\text{set } ts) \implies a \subseteq \bigcup (\text{set } (\text{partList4 } b \text{ } ts))$ 
  apply(simp add: partList4)
  apply(induction  $ts$  arbitrary:  $a$   $b$ )
  apply(simp; fail)
  apply(simp)
  by fast

```

private lemma $a \neq \{\}$ \implies $\text{disjoint-list-rec } (a \# ts) \longleftrightarrow \text{disjoint-list-rec } ts \wedge a \cap \bigcup (\text{set } ts) = \{\}$ **by** *auto*

lemma $\text{partList4-complete0}$: $s \subseteq \bigcup (\text{set } ts) \implies \bigcup (\text{set } (\text{partList4 } s \ ts)) = \bigcup (\text{set } ts)$

unfolding partList4
proof(*induction ts arbitrary: s*)
 case Nil thus *?case* **by**(*simp*)
 next
 case Cons thus *?case* **by** (*simp add: Diff-subset-conv Un-Diff-Int inf-sup-aci*(7)
sup commute)
qed

private lemma $\text{partList4-disjoint}$: $s \subseteq \bigcup (\text{set } ts) \implies \text{disjoint-list-rec } ts \implies \text{disjoint-list-rec } (\text{partList4 } s \ ts)$

apply(*induction ts arbitrary: s*)
apply(*simp; fail*)
apply(*simp add: Diff-subset-conv*)
apply(*rule conjI*)
apply (*metis Diff-subset-conv Int-absorb1 Int-lower2 Un-absorb1 partList4-complete0*)
apply(*safe*)
 using $\text{partList4-complete0}$ **apply** (*metis Diff-subset-conv Diff-triv IntI UnionI*)
 apply (*metis Diff-subset-conv Diff-triv*)
 using $\text{partList4-complete0}$ **by** (*metis Diff-subset-conv IntI UnionI*)+

lemma $\text{union-set-partList4}$: $\bigcup (\text{set } (\text{partList4 } s \ ts)) = \bigcup (\text{set } ts)$
by (*induction ts arbitrary: s, auto*)

private lemma $\text{partList4-distinct-hlp}$: **assumes** $a \neq \{\}$ $a \notin \text{set } ts$ *disjoint (insert a (set ts))*

shows $a \notin \text{set } (\text{partList4 } s \ ts)$

proof –
 from *assms* **have** $\neg a \subseteq \bigcup (\text{set } ts)$ **unfolding** disjoint-def **by** *fastforce*
 hence $\neg a \subseteq \bigcup (\text{set } (\text{partList4 } s \ ts))$ **using** $\text{union-set-partList4}$ **by** *metis*
 thus *?thesis* **by** *blast*

qed

private lemma $\text{partList4-distinct}$: $\{\} \notin \text{set } ts \implies \text{disjoint-list } ts \implies \text{distinct } (\text{partList4 } s \ ts)$

proof(*induction ts arbitrary: s*)
 case Nil thus *?case* **by** *simp*
 next
 case(*Cons t ts*)
 have $x1: \bigwedge x \ xa \ xb \ xc.$
 $t \notin \text{set } ts \implies$
 $\text{disjoint } (\text{insert } t \ (\text{set } ts)) \implies$
 $xa \in t \implies$


```

     $xb \in s \implies$ 
     $xb \in t \implies$ 
     $xb \notin \{\} \implies$ 
     $xc \in s \implies$ 
     $xc \notin \{\} \implies$ 
     $t \cap s \in \text{set } (\text{partList4 } (s - t) \text{ } ts) \implies$ 
     $\neg t \cap s \subseteq \bigcup (\text{set } (\text{partList4 } (s - t) \text{ } ts))$ 
  by(simp add: union-set-partList4 disjoint-def, force)
  have  $x2: \bigwedge x xa xb xc.$ 
     $t \notin \text{set } ts \implies$ 
     $\text{disjoint } (\text{insert } t \text{ } (\text{set } ts)) \implies$ 
     $x \in t \implies$ 
     $xa \in t \implies$ 
     $xa \notin s \implies$ 
     $xb \in s \implies$ 
     $xc \in s \implies$ 
     $\neg t - s \subseteq \bigcup (\text{set } (\text{partList4 } (s - t) \text{ } ts))$ 
  by(simp add: union-set-partList4 disjoint-def, force)
  from Cons have IH: distinct (partList4 s ts) for s
    using disjoint-sublist list.set-intros(2) by auto
  from Cons.prem(1,2) IH show ?case
  unfolding disjoint-list-def
  apply(simp)
  apply(safe)
    apply(metis partList4-distinct-hlp)
    apply(metis partList4-distinct-hlp)
    apply(metis partList4-distinct-hlp)
  apply blast
  using x1 apply blast
  using x2 by blast
  qed

```

lemma *partList4-disjoint-list*: **assumes** $s \subseteq \bigcup (\text{set } ts)$ *disjoint-list* $ts \{\} \notin \text{set } ts$
shows *disjoint-list* (*partList4* *s* *ts*)
unfolding *disjoint-list-def*
proof
 from *assms(2,3)* **show** *distinct* (*partList4* *s* *ts*)
 using *partList4-distinct disjoint-list-def* by *auto*
 show *disjoint* (*set* (*partList4* *s* *ts*))
proof –
 have *disjoint-list-disjoint-list-rec: disjoint-list* *ts* \implies *disjoint-list-rec* *ts*
proof(*induction* *ts*)
 case *Cons* **thus** ?*case* by(*auto simp add: disjoint-list-def disjoint-def*)
qed(*simp*)
 with *partList4-disjoint disjoint-equi assms(1,2)* **show** ?*thesis* by *blast*
qed
qed

lemma *partitioning1-subset*: $a \subseteq \bigcup (\text{set } ts) \implies a \subseteq \bigcup (\text{set } (\text{partitioning1 } ss))$

```

ts))
  apply(induction ss arbitrary: ts a)
  apply(simp)
  apply(simp add: partList4-subset)
done

lemma partitioning1-disjoint-list:  $\{\} \notin (\text{set } ts) \implies \bigcup (\text{set } ss) \subseteq \bigcup (\text{set } ts) \implies$ 
  disjoint-list ts  $\implies$  disjoint-list (partitioning1 ss ts)

proof(induction ss)
case Nil thus ?case by simp
next
case (Cons t ts) thus ?case
  apply(clarsimp)
  apply(rule partList4-disjoint-list)
  using partitioning1-subset apply(metis)
  apply(blast)
  using partitioning1-empty0 apply(metis)
done
qed

private lemma partitioning1-disjoint:  $\bigcup (\text{set } ss) \subseteq \bigcup (\text{set } ts) \implies$ 
  disjoint-list-rec ts  $\implies$  disjoint-list-rec (partitioning1 ss ts)

proof(induction ss arbitrary: ts)
qed(simp-all add: partList4-disjoint partitioning1-subset)

private lemma partitioning-equi:  $\{\} \notin \text{set } ts \implies$  disjoint-list-rec ts  $\implies$   $\bigcup (\text{set } ss) \subseteq \bigcup (\text{set } ts) \implies$ 
  set(partitioning1 ss ts) = partitioning-nontail ss (set ts) -  $\{\{\}\}$ 

proof(induction ss)
case Nil thus ?case by simp
next
case (Cons s ss)
  have addSubsetSet-empty: addSubsetSet s (ts -  $\{\{\}\}$ ) -  $\{\{\}\}$  = addSubsetSet
s ts -  $\{\{\}\}$ 
  for s and ts::'a set set
  unfolding addSubsetSet-def by blast
  have r: disjoint-list-rec ts  $\implies$   $s \subseteq \bigcup (\text{set } ts) \implies$ 
  addSubsetSet s (set ts) -  $\{\{\}\}$  = set (partList4 s
ts) -  $\{\{\}\}$ 
  for ts::'a set list
  unfolding partList4
  by(simp add: partList0-addSubsetSet-equi partList0-partList1-equi partList1-partList2-equi
partList2-partList3-equi)
  have 1: disjoint-list-rec (partitioning1 ss ts)
  using partitioning1-disjoint Cons.prem by auto
  from Cons.prem have 2:  $s \subseteq \bigcup (\text{set } (\text{partitioning1 } ss \text{ ts}))$ 
  by (meson Sup-upper dual-order.trans list.set-intros(1) partitioning1-subset)
  from Cons have IH: set (partitioning1 ss ts) = partitioning-nontail ss (set
ts) -  $\{\{\}\}$  by auto

```

```

with r[OF 1 2] show ?case by (simp add: partList4-empty addSubsetSet-empty)
qed

lemma ipPartitioning-helper-opt: {} ∉ set ts ⇒ disjoint-list ts ⇒ ⋃ (set ss)
⊆ ⋃ (set ts)
    ⇒ ipPartition (set ss) (set (partitioning1 ss ts))
  apply (drule disjoint-list-disjoint-list-rec)
  apply (simp add: partitioning-equi partitioning-nottail-equi)
  by (meson Diff-subset disjoint-equi ipPartition-def ipPartitioning-helper-subsetCE)

lemma complete-helper: {} ∉ set ts ⇒ ⋃ (set ss) ⊆ ⋃ (set ts) ⇒
  ⋃ (set ts) = ⋃ (set (partitioning1 ss ts))
  apply (induction ss arbitrary: ts)
  apply (simp-all)
  by (metis partList4-complete0)

lemma partitioning1 [{1::nat},{2},{3}] [{1},{2},{3}] = [{1}, {2}, {3}] by
eval

lemma partitioning-foldr: partitioning X B = foldr addSubsetSet X B
  apply (induction X)
  apply (simp; fail)
  apply (simp)
  by (metis partitioningCom)

lemma ipPartition (set X) (foldr addSubsetSet X {})
  apply (subst partitioning-foldr[symmetric])
  using ipPartitioning by auto

lemma ⋃ (set X) = ⋃ (foldr addSubsetSet X {})
  apply (subst partitioning-foldr[symmetric])
  by (simp add: coversallPartitioning)

lemma partitioning1 X B = foldr partList4 X B
  by (induction X)(simp-all)

lemma ipPartition (set X) (set (partitioning1 X [UNIV]))
  apply (rule ipPartitioning-helper-opt)
  by (simp-all add: disjoint-list-def disjoint-def)

lemma (⋃ (set (partitioning1 X [UNIV]))) = UNIV
  apply (subgoal-tac UNIV = ⋃ (set (partitioning1 X [UNIV])))
  prefer 2
  apply (rule complete-helper[where ts=[UNIV], simplified])
  apply (simp)
  done

end

```

end

14 Group by Function

```
theory GroupF
imports Main
begin
```

Grouping elements of a list according to a function.

```
fun groupF :: ('a ⇒ 'b) ⇒ 'a list ⇒ '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 ⇒ bool) ⇒ 'a ⇒ ('a list × 'a list) ⇒ ('a list ×
'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 ⇒ bool) ⇒ 'a list ⇒ ('a list × '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 -
```

```
  {fix ts-accu fs-accu
```

```
    have foldr (select-p-tuple f) as (ts-accu, fs-accu) =
      (filter f as @ ts-accu, filter (λx. ¬f x) as @ fs-accu)
```

```
    by(induction as arbitrary: ts-accu fs-accu) simp-all
```

```
  } thus ?thesis unfolding partition-tailrec-def by simp
```

```
qed
```

```
private lemma
```

```
  groupF f (x#xs) = (let (ts, fs) = partition-tailrec (λy. f x = f y) xs in
(x#ts)#(groupF f fs))
```

```
  by(simp add: partition-tailrec)
```

```
private function groupF-code :: ('a ⇒ 'b) ⇒ 'a list ⇒ '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))
```

```

by(pat-completeness) auto

private termination groupF-code
  apply(relation measure (λ(f,as). length (filter (λx. (λy. f x = f y) x) as)))
  apply(simp; fail)
  apply(simp add: partition-tailrec)
  using le-imp-less-Suc length-filter-le by blast

lemma groupF-code[code]: groupF f as = groupF-code f as
  by(induction f as rule: groupF-code.induct) (simp-all add: partition-tailrec)

export-code groupF checking SML
end

lemma groupF-concat-set: set (concat (groupF f xs)) = set xs
  proof(induction f xs rule: groupF.induct)
  case 2 thus ?case by (simp) blast
  qed(simp)

lemma groupF-Union-set: (⋃ x ∈ set (groupF f xs). set x) = set xs
  proof(induction f xs rule: groupF.induct)
  case 2 thus ?case by (simp) blast
  qed(simp)

lemma groupF-set: ∀ X ∈ set (groupF f xs). ∀ x ∈ set X. x ∈ set xs
  using groupF-concat-set by fastforce

lemma groupF-equality:
  defines same f A ≡ ∀ a1 ∈ set A. ∀ a2 ∈ set A. f a1 = f a2
  shows ∀ A ∈ set (groupF f xs). same f A
  proof(induction f xs rule: groupF.induct)
  case 1 thus ?case by simp
  next
  case (2 f x xs)
  have groupF-fst:
    groupF f (x # xs) = (x # [y←xs . f x = f y]) # groupF f [y←xs . f x ≠ f
y] by force
  have step: ∀ A ∈ set [x # [y←xs . f x = f y]]. same f A unfolding same-def
  by fastforce
  with 2 show ?case unfolding groupF-fst by fastforce
  qed

lemma groupF-inequality: A ∈ set (groupF f xs) ⇒ B ∈ set (groupF f xs) ⇒ A
≠ B ⇒
  ∀ a ∈ set A. ∀ b ∈ set B. f a ≠ f b
  proof(induction f xs rule: groupF.induct)
  case 1 thus ?case by simp
  next
  case 2 thus ?case

```

```

apply -
apply(subst (asm) groupF.simps)+
using groupF-set by fastforce
qed

```

```

lemma groupF-cong: fixes xs::'a list and f1::'a ⇒ 'b and f2::'a ⇒ 'c
assumes ∀ x ∈ set xs. ∀ y ∈ set xs. (f1 x = f1 y ↔ f2 x = f2 y)
shows groupF f1 xs = groupF f2 xs
using assms proof(induction f1 xs rule: groupF.induct)
  case (2 f x xs) thus ?case using filter-cong[of xs xs λy. f x = f y λy. f2 x =
f2 y]
  filter-cong[of xs xs λy. f x ≠ f y λy. f2 x ≠ f2 y] by
auto
qed (simp)

```

```

lemma groupF-empty: groupF f xs ≠ [] ↔ xs ≠ []
by(induction f xs rule: groupF.induct) auto
lemma groupF-empty-elem: x ∈ set (groupF f xs) ⇒ x ≠ []
by(induction f xs rule: groupF.induct) auto

```

```

lemma groupF-distinct: distinct xs ⇒ distinct (concat (groupF f xs))
by (induction f xs rule: groupF.induct) (auto simp add: groupF-Union-set)

```

It is possible to use `map (map fst) (groupF snd (map (λx. (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 (λx. (x, f
x)) xs))
proof(induction f xs rule: groupF.induct)
  case (1 f) thus ?case by simp
  next
  case (2 f x xs)
    have g1: [y←xs . f x = f y] = map fst [y←map (λx. (x, f x)) xs . f x = snd y]
    proof(induction xs arbitrary: f x)
      case Cons thus ?case by fastforce
    qed(simp)
    have g2: (map (λx. (x, f x)) [y←xs . f x ≠ f y]) = [y←map (λx. (x, f x)) xs . f
x ≠ snd y]
    proof(induction xs)
      case Cons thus ?case by fastforce
    qed(simp)
  from 2 g1 g2 show ?case by simp
qed
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  $\Rightarrow$  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  $\Rightarrow$  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  $\times$  nat)  $\Rightarrow$  string where
  ipv4-cidr-toString ip-n = (case ip-n of (base, n)  $\Rightarrow$  (ipv4addr-toString base
  @""/"@ string-of-nat n))
lemma ipv4-cidr-toString (ipv4addr-of-dotdecimal (192,168,0,1), 22) = "192.168.0.1/22"
by eval

definition ipv6-cidr-toString :: (ipv6addr  $\times$  nat)  $\Rightarrow$  string where
  ipv6-cidr-toString ip-n = (case ip-n of (base, n)  $\Rightarrow$  (ipv6addr-toString base
  @""/"@ string-of-nat n))
lemma ipv6-cidr-toString (42540766411282592856906245548098208122, 64) = "2001:db8::8:800:200c:417a/64"
by eval

definition primitive-protocol-toString :: primitive-protocol  $\Rightarrow$  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 "+") = "" by eval
lemma iface-toString "in: " (Iface "eth0") = "in: eth0" by eval

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) = "" by eval
lemma ports-toString "spt: " (1024,2048) = "spt: 1024:2048" by eval
lemma ports-toString "spt: " (1024,1024) = "spt: 1024" by eval

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 \Rightarrow 'i word \times nat) \Rightarrow 'i simple-rule list \Rightarrow ('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*

by(*induction rs rule: extract-IPSets-generic0.induct*) (*simp-all*)

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)] **by** *eval*

fun *extract-src-dst-ips*

:: 'i::len simple-rule list \Rightarrow ('i word \times nat) list \Rightarrow ('i word \times 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(*induction rs arbitrary: acc*)

case (*Cons r rs*) **thus** ?*case* **by**(*cases r, simp*)

qed(*simp*)

definition *extract-IPSets*

:: 'i::len simple-rule list \Rightarrow ('i wordinterval) list **where**

extract-IPSets rs \equiv 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) \cup set (extract-IPSets-generic0 dst rs)

```

proof –
  { fix acc
    have ipcidr-tuple-to-wordinterval ‘ set (extract-src-dst-ips rs acc) =
      ipcidr-tuple-to-wordinterval ‘ set acc ∪ set (extract-IPSets-generic0 src
rs) ∪
      set (extract-IPSets-generic0 dst rs)
    proof(induction rs arbitrary: acc)
    case (Cons r rs) thus ?case
      apply(cases r)
      apply(simp)
      by fast
      qed(simp)
  } thus ?thesis unfolding extract-IPSets-def by (simp-all add: extract-IPSets-def
mergesort-remdups-correct)
qed

```

lemma (*a::nat*) *div 2 + a mod 2 ≤ a* **by** *fastforce*

lemma *merge-length: length (merge l1 l2) ≤ length l1 + length l2*
by(*induction l1 l2 rule: merge.induct*) *auto*

lemma *merge-list-length: length (merge-list as ls) ≤ length (concat (as @ ls))*
proof(*induction as ls rule: merge-list.induct*)
case (*5 l1 l2 acc2 ls*)
have *length (merge l2 acc2) ≤ length l2 + length acc2* **using** *merge-length by*
blast
with *5 show ?case by simp*
qed(*simp-all*)

lemma *mergesort-remdups-length: length (mergesort-remdups as) ≤ length as*
unfolding *mergesort-remdups-def*
proof –
have *concat ([] @ (map (λx. [x]) as)) = as* **by** *simp*
with *merge-list-length show length (merge-list [] (map (λx. [x]) as)) ≤ length as*
by *metis*
qed

lemma *extract-IPSets-length: length (extract-IPSets rs) ≤ 2 * length rs*
apply(*simp add: extract-IPSets-def*)
using *extract-src-dst-ips-length mergesort-remdups-length by (metis add.right-neutral*
list.size(3))

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(induction rs)
case (Cons r rs) thus ?case
  apply(cases r, simp)
  using wordinterval-to-set-ipcidr-tuple-to-wordinterval by fastforce
qed(simp)

lemma src-ipPart-motivation:
fixes rs
defines  $X \equiv (\lambda(\text{base}, \text{len}). \text{ipset-from-cidr } \text{base } \text{len}) \text{ ` src ` match-sel ` set } rs$ 
assumes  $\forall A \in X. B \subseteq A \vee B \cap A = \{\}$  and  $s1 \in B$  and  $s2 \in B$ 
shows  $\text{simple-fw } rs \ (p(\text{p-src}:=s1)) = \text{simple-fw } rs \ (p(\text{p-src}:=s2))$ 
proof -
  have  $\forall A \in (\lambda(\text{base}, \text{len}). \text{ipset-from-cidr } \text{base } \text{len}) \text{ ` src ` match-sel ` set } rs. B \subseteq$ 
 $A \vee B \cap A = \{\} \implies ?thesis$ 
  proof(induction rs)
    case Nil thus ?case by simp
  next
    case (Cons r rs)
      { fix m
        from  $\langle s1 \in B \rangle \langle s2 \in B \rangle$  have
           $B \subseteq (\text{case } \text{src } m \text{ of } (x, xa) \Rightarrow \text{ipset-from-cidr } x \text{ } xa) \vee B \cap (\text{case } \text{src } m \text{ of } (x, xa)$ 
 $\Rightarrow \text{ipset-from-cidr } x \text{ } xa) = \{\} \implies$ 
           $\text{simple-matches } m \ (p(\text{p-src} := s1)) \longleftrightarrow \text{simple-matches } m \ (p(\text{p-src} :=$ 
 $s2))$ 
          apply(cases m)
          apply(rename-tac iface oiface srca dst proto sports dports)
          apply(case-tac srca)
          apply(simp add: simple-matches.simps)
          by blast
        } note helper=this
      from Cons[simplified] show ?case
      apply(cases r, rename-tac m a)
      apply(simp)
      apply(case-tac a)
      using helper apply force+
      done
    qed
  with assms show ?thesis by blast
qed

```

```

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  $\text{simple-fw } rs \ (p(\text{p-src}:=s1)) = \text{simple-fw } rs \ (p(\text{p-src}:=s2))$ 
proof -
  from src-ipPart-motivation[OF - assms(3) assms(4)]

```

```

have  $\forall A \in (\lambda(\text{base}, \text{len}). \text{ipset-from-cidr } \text{base } \text{len}) \text{ ' src ' match-sel ' set } rs. B \subseteq$ 
 $A \vee B \cap A = \{\}$   $\implies$ 
   $\text{simple-fw } rs \text{ (p(p-src:=s1))} = \text{simple-fw } rs \text{ (p(p-src:=s2))}$  by fast
thus ?thesis using assms(1) assms(2)
  unfolding ipPartition-def
  by (metis (full-types) Int-commute extract-equi0)
qed

```

lemma *dst-ipPart*:

```

assumes ipPartition (set (map wordinterval-to-set (extract-IPSets-generic0 dst
rs))) A

```

```

   $B \in A \text{ s1} \in B \text{ s2} \in B$ 

```

```

shows  $\text{simple-fw } rs \text{ (p(p-dst:=s1))} = \text{simple-fw } rs \text{ (p(p-dst:=s2))}$ 

```

proof –

```

have  $\forall A \in (\lambda(\text{base}, \text{len}). \text{ipset-from-cidr } \text{base } \text{len}) \text{ ' dst ' match-sel ' set } rs. B \subseteq$ 
 $A \vee B \cap A = \{\}$   $\implies$ 

```

```

   $\text{simple-fw } rs \text{ (p(p-dst:=s1))} = \text{simple-fw } rs \text{ (p(p-dst:=s2))}$ 

```

proof(*induction* *rs*)

```

  case Nil thus ?case by simp

```

next

```

  case (Cons r rs)

```

```

  { fix m

```

```

    from  $\langle s1 \in B \rangle \langle s2 \in B \rangle$  have

```

```

     $B \subseteq (\text{case } \text{dst } m \text{ of } (x, xa) \Rightarrow \text{ipset-from-cidr } x \text{ xa}) \vee B \cap (\text{case } \text{dst } m \text{ of}$ 
 $(x, xa)$ 

```

```

       $\Rightarrow \text{ipset-from-cidr } x \text{ xa}) = \{\}$   $\implies$ 

```

```

     $\text{simple-matches } m \text{ (p(p-dst := s1))} \longleftrightarrow \text{simple-matches } m \text{ (p(p-dst := s2))}$ 

```

```

    apply(cases m)

```

```

    apply(rename-tac iface oiface src dsta proto sports dports)

```

```

    apply(case-tac dsta)

```

```

    apply(simp add: simple-matches.simps)

```

```

    by blast

```

```

  } note helper=this

```

```

from Cons show ?case

```

```

  apply(simp)

```

```

  apply(case-tac r, rename-tac m a)

```

```

  apply(simp)

```

```

  apply(case-tac a)

```

```

  using helper apply force+

```

```

  done

```

qed

```

thus ?thesis using assms(1) assms(2)

```

```

unfolding ipPartition-def

```

```

by (metis (full-types) Int-commute extract-equi0)

```

qed

definition *wordinterval-list-to-set* :: 'a::len wordinterval list \Rightarrow 'a::len word set
where

wordinterval-list-to-set ws = \bigcup (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(*induction* xs)

qed(*simp-all* add: *wordinterval-compress wordinterval-list-to-set-def*)

fun *partIps* :: 'a::len wordinterval \Rightarrow 'a::len wordinterval list

\Rightarrow '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] **by** *eval*

lemma *partIps-length*: length (*partIps* s ts) \leq (length ts) * 2

proof(*induction* ts *arbitrary*: s)

case *Cons* **thus** ?*case*

apply(*simp*)

using *le-Suc-eq* **by** *blast*

qed(*simp*)

fun *partitioningIps* :: 'a::len wordinterval list \Rightarrow 'a::len wordinterval list \Rightarrow

'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(*induction* ss)

case *Nil* **thus** ?*case* **by** *simp*

next

case (*Cons* s ss)

have $\text{length } (\text{partIps } s \ (\text{partitioningIps } ss \ ts)) \leq \text{length } (\text{partitioningIps } ss \ ts) * 2$
using *partIps-length* **by** *fast*
with *Cons* **show** *?case* **by** *force*
qed

lemma *partIps-equi*: $\text{map } \text{wordinterval-to-set } (\text{partIps } s \ ts) =$
 $\text{partList}_4 \ (\text{wordinterval-to-set } s) \ (\text{map } \text{wordinterval-to-set } ts)$
proof(*induction* *ts* *arbitrary*: *s*)
qed(*simp-all*)

lemma *partitioningIps-equi*: $\text{map } \text{wordinterval-to-set } (\text{partitioningIps } ss \ ts)$
 $= (\text{partitioning1 } (\text{map } \text{wordinterval-to-set } ss) \ (\text{map } \text{wordinterval-to-set } ts))$
apply(*induction* *ss* *arbitrary*: *ts*)
apply(*simp*; *fail*)
apply(*simp* *add*: *partIps-equi*)
done

definition *getParts* :: *'i::len* *simple-rule* *list* \Rightarrow *'i* *wordinterval* *list* **where**
 $\text{getParts } rs = \text{partitioningIps } (\text{extract-IPSets } rs) \ [\text{wordinterval-UNIV}]$

lemma *partitioningIps-foldr*: $\text{partitioningIps } ss \ ts = \text{foldr } \text{partIps } ss \ ts$
by(*induction* *ss*) (*simp-all*)

lemma *getParts-foldr*: $\text{getParts } rs = \text{foldr } \text{partIps } (\text{extract-IPSets } rs) \ [\text{wordinterval-UNIV}]$
by(*simp* *add*: *getParts-def* *partitioningIps-foldr*)

lemma *getParts-length*: $\text{length } (\text{getParts } rs) \leq 2^{(2 * \text{length } rs)}$
proof –
from *partitioningIps-length* **[where** *ss* $= \text{extract-IPSets } rs$ **and** *ts* $= [\text{wordinterval-UNIV}]$ **]**
have
 $1: \text{length } (\text{partitioningIps } (\text{extract-IPSets } rs) \ [\text{wordinterval-UNIV}]) \leq 2^{\text{length } (\text{extract-IPSets } rs)}$ **by** *simp*
from *extract-IPSets-length* **have** $(2::\text{nat})^{\text{length } (\text{extract-IPSets } rs)} \leq 2^{(2 * \text{length } rs)}$ **by** *simp*
with *1* **have** $\text{length } (\text{partitioningIps } (\text{extract-IPSets } rs) \ [\text{wordinterval-UNIV}]) \leq 2^{(2 * \text{length } rs)}$ **by** *linarith*
thus *?thesis* **by**(*simp* *add*: *getParts-def*)
qed

lemma *getParts-ipPartition*: $\text{ipPartition } (\text{set } (\text{map } \text{wordinterval-to-set } (\text{extract-IPSets } rs)))$
 $(\text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts } rs)))$

proof –
have *hlp-rule*: $\{\} \notin \text{set } (\text{map } \text{wordinterval-to-set } ts) \implies \text{disjoint-list } (\text{map } \text{wordinterval-to-set } ts) \implies$
 $(\text{wordinterval-list-to-set } ss) \subseteq (\text{wordinterval-list-to-set } ts) \implies$
 $\text{ipPartition } (\text{set } (\text{map } \text{wordinterval-to-set } ss))$
 $(\text{set } (\text{map } \text{wordinterval-to-set } (\text{partitioningIps } ss \ ts)))$ **for** *ts* *ss*:*'a*

```

wordinterval list
by (metis ipPartitioning-helper-opt partitioningIps-equi wordinterval-list-to-set-def)
have disjoint-list [UNIV] by (simp add: disjoint-list-def disjoint-def)
have ipPartition (set (map wordinterval-to-set ss))
  (set (map wordinterval-to-set (partitioningIps ss [wordinterval-UNIV])))
  for ss::'a wordinterval list
apply (rule hlp-rule)
  apply (simp-all add: wordinterval-list-to-set-def ⟨disjoint-list [UNIV]⟩)
done
thus ?thesis
unfolding getParts-def by blast
qed

```

```

lemma getParts-complete: wordinterval-list-to-set (getParts rs) = UNIV
proof -
have {} ⊄ set (map wordinterval-to-set ts) ⇒
  (wordinterval-list-to-set ss) ⊆ (wordinterval-list-to-set ts) ⇒
  wordinterval-list-to-set (partitioningIps ss ts) = (wordinterval-list-to-set ts)
  for ss ts::'a wordinterval list
using complete-helper by (metis partitioningIps-equi wordinterval-list-to-set-def)
hence wordinterval-list-to-set (getParts rs) = wordinterval-list-to-set [wordinterval-UNIV]
  unfolding getParts-def by (simp add: wordinterval-list-to-set-def)
also have ... = UNIV by (simp add: wordinterval-list-to-set-def)
finally show ?thesis .
qed

```

```

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 -
let ?X=(set (map wordinterval-to-set (getParts rs)))
from getParts-ipPartition have ipPartition (set (map wordinterval-to-set (extract-IPSets
rs))) ?X .
hence ipPartition (set (map wordinterval-to-set (extract-IPSets-generic0 src rs)))
?X ∧
  ipPartition (set (map wordinterval-to-set (extract-IPSets-generic0 dst rs)))
?X
by (simp add: extract-IPSets ipPartitionUnion image-Un)
thus ?thesis using assms dst-ipPart src-ipPart by blast
qed

```

```

lemma partIps-nonempty: ts ≠ [] ⇒ partIps s ts ≠ []
  by (induction ts arbitrary: s) simp-all
lemma partitioningIps-nonempty: ts ≠ [] ⇒ partitioningIps ss ts ≠ []
proof (induction ss arbitrary: ts)
  case Nil thus ?case by simp

```

```

next
case (Cons s ss) thus ?case
  apply(cases ts)
  apply(simp; fail)
  apply(simp)
  using partIps-nonempty by blast
qed

```

lemma *getParts-nonempty*: $getParts\ rs \neq []$ **by** (*simp add: getParts-def partitioningIps-nonempty*)

lemma *getParts-nonempty-elems*: $\forall w \in set\ (getParts\ rs). \neg\ wordinterval\ empty\ w$
unfolding *getParts-def*

proof –

have *partitioning-nonempty*: $\forall t \in set\ ts. \neg\ wordinterval\ empty\ t \implies$

$\{\} \notin set\ (map\ wordinterval\ to\ set\ (partitioningIps\ ss\ ts))$

for *ts ss*: 'a *wordinterval list*

proof(*induction ss arbitrary: ts*)

case Nil thus ?*case* **by** *auto*

case Cons thus ?*case* **by** (*simp add: partIps-equi partList4-empty*)

qed

have $\forall t \in set\ [wordinterval\ UNIV]. \neg\ wordinterval\ empty\ t$ **by** (*simp*)

with *partitioning-nonempty* **have**

$\{\} \notin set\ (map\ wordinterval\ to\ set\ (partitioningIps\ (extract\ IP\ Sets\ rs)\ [wordinterval\ UNIV]))$

by *blast*

thus $\forall w \in set\ (partitioningIps\ (extract\ IP\ Sets\ rs)\ [wordinterval\ UNIV]). \neg$

wordinterval-empty w **by** *auto*

qed

fun *getOneIp* :: 'a::len *wordinterval* \Rightarrow '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*: $\neg\ wordinterval\ empty\ W \implies\ wordinterval\ element\ (getOneIp\ W)\ W$

by (*induction W simp-all*)

record *parts-connection* = *pc-iface* :: *string*

pc-oiface :: *string*

pc-proto :: *primitive-protocol*

pc-sport :: 16 *word*

pc-dport :: 16 *word*

definition *same-fw-behaviour* :: $'i::\text{len word} \Rightarrow 'i \text{ word} \Rightarrow 'i$
simple-rule list $\Rightarrow \text{bool}$ **where**
same-fw-behaviour $a b rs \equiv$
 $\forall (p:: 'i::\text{len simple-packet}).$
 $\text{simple-fw } rs (p(p\text{-src}=a)) = \text{simple-fw } rs (p(p\text{-src}=b)) \wedge$
 $\text{simple-fw } rs (p(p\text{-dst}=a)) = \text{simple-fw } rs (p(p\text{-dst}=b))$

lemma *getParts-same-fw-behaviour*:

$A \in \text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts } rs)) \implies s1 \in A \implies s2 \in A \implies$
 $\text{same-fw-behaviour } s1 s2 rs$

unfolding *same-fw-behaviour-def*

using *getParts-samefw* **by** *blast*

definition *runFw* $s d c rs = \text{simple-fw } rs (p(p\text{-iface}=pc\text{-iface } c, p\text{-oiface}=pc\text{-oiface } c,$

$p\text{-src}=s, p\text{-dst}=d,$
 $p\text{-proto}=pc\text{-proto } c,$
 $p\text{-sport}=pc\text{-sport } c, p\text{-dport}=pc\text{-dport } c,$
 $p\text{-tcp-flags}=\{TCP\text{-SYN}\},$
 $p\text{-payload}=""$)

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

definition *runFw-scheme* :: $'i::\text{len word} \Rightarrow 'i \text{ word} \Rightarrow 'b \text{ parts-connection-scheme}$
 \Rightarrow

$('i, 'a) \text{ simple-packet-scheme} \Rightarrow 'i \text{ simple-rule list} \Rightarrow \text{state}$

where

runFw-scheme $s d c p rs = \text{simple-fw } rs$
 $(p(p\text{-iface}=pc\text{-iface } c,$
 $p\text{-oiface}=pc\text{-oiface } c,$
 $p\text{-src}=s,$
 $p\text{-dst}=d,$
 $p\text{-proto}=pc\text{-proto } c,$
 $p\text{-sport}=pc\text{-sport } c,$
 $p\text{-dport}=pc\text{-dport } c))$

lemma *runFw-scheme*: $\text{runFw } s d c rs = \text{runFw-scheme } s d c p rs$

apply(*simp add*: *runFw-def runFw-scheme-def*)

apply(*case-tac* *p*)

apply(*simp*)

apply(*thin-tac* $-$, *simp*)

proof(*induction* *rs*)

case *Nil* **thus** *?case* **by**(*simp*; *fail*)

next

```

case(Cons r rs)
obtain m a where r: r = SimpleRule m a by(cases r) simp
from simple-matches-extended-packet[symmetric, of - pc-iiface c pc-oiface c
s d pc-proto c pc-sport c pc-dport c - - - {TCP-SYN}]
[]
have pext: simple-matches m
  (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-flags2, p-payload = payload2, ... = aux) =
  simple-matches m
  (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 = []) for tcp-flags2 payload2 and aux::'c
by fast
show ?case
apply(simp add: r, cases a, simp)
using Cons.IH by(simp add: pext)+
qed

```

lemma *has-default-policy-runFw*: *has-default-policy rs* \implies *runFw s d c rs* = *Decision FinalAllow* \vee *runFw s d c rs* = *Decision FinalDeny*
by(*simp add: runFw-def has-default-policy*)

definition *same-fw-behaviour-one* :: '*i*::*len word* \Rightarrow '*i* *word* \Rightarrow '*a* *parts-connection-scheme*
 \Rightarrow '*i* *simple-rule list* \Rightarrow *bool* **where**
same-fw-behaviour-one ip1 ip2 c rs \equiv
 $\forall d s. \text{runFw } ip1 d c rs = \text{runFw } ip2 d c rs \wedge \text{runFw } s ip1 c rs = \text{runFw } s ip2 c rs$

lemma *same-fw-spec*: *same-fw-behaviour ip1 ip2 rs* \implies *same-fw-behaviour-one ip1 ip2 c rs*
apply(*simp add: same-fw-behaviour-def same-fw-behaviour-one-def runFw-def*)
apply(*rule conjI*)
apply(*clarify*)
apply(*erule-tac x=(p-iiface = pc-iiface c, p-oiface = pc-oiface c, p-src = ip1,*
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="") **in** *allE*)
apply(*simp;fail*)
apply(*clarify*)
apply(*erule-tac x=(p-iiface = pc-iiface c, p-oiface = pc-oiface c, p-src = s,*
p-dst= ip1,
p-proto = pc-proto c, p-sport = pc-sport c, p-dport = pc-dport c,
p-tcp-flags = {TCP-SYN},
p-payload="") **in** *allE*)
apply(*simp*)
done

Is an equivalence relation

lemma *same-fw-behaviour-one-equi*:

same-fw-behaviour-one $x x c rs$

same-fw-behaviour-one $x y c rs = \text{same-fw-behaviour-one } y x c rs$

same-fw-behaviour-one $x y c rs \wedge \text{same-fw-behaviour-one } y z c rs \implies \text{same-fw-behaviour-one } x z c rs$

unfolding *same-fw-behaviour-one-def* **by** *metis+*

lemma *same-fw-behaviour-equi*:

same-fw-behaviour $x x rs$

same-fw-behaviour $x y rs = \text{same-fw-behaviour } y x rs$

same-fw-behaviour $x y rs \wedge \text{same-fw-behaviour } y z rs \implies \text{same-fw-behaviour } x z rs$

unfolding *same-fw-behaviour-def* **by** *auto*

lemma *runFw-sameFw-behave*:

fixes $W :: 'i::\text{len word set set}$

shows

$\forall A \in W. \forall a1 \in A. \forall a2 \in A. \text{same-fw-behaviour-one } a1 a2 c rs \implies \bigcup W = UNIV \implies$

$\forall B \in W. \exists b \in B. \text{runFw } ip1 b c rs = \text{runFw } ip2 b c rs \implies$

$\forall B \in W. \exists b \in B. \text{runFw } b ip1 c rs = \text{runFw } b ip2 c rs \implies$

same-fw-behaviour-one $ip1 ip2 c rs$

proof –

assume $a1: \forall A \in W. \forall a1 \in A. \forall a2 \in A. \text{same-fw-behaviour-one } a1 a2 c rs$

and $a2: \bigcup W = UNIV$

and $a3: \forall B \in W. \exists b \in B. \text{runFw } ip1 b c rs = \text{runFw } ip2 b c rs$

and $a4: \forall B \in W. \exists b \in B. \text{runFw } b ip1 c rs = \text{runFw } b ip2 c rs$

have *relation-lem*: $\forall D \in W. \forall d1 \in D. \forall d2 \in D. \forall s. f s d1 = f s d2 \implies \bigcup W = UNIV \implies$

$\forall B \in W. \exists b \in B. f s1 b = f s2 b \implies$

$f s1 d = f s2 d$ **for** W **and** $f::'c \Rightarrow 'b \Rightarrow 'd$ **and** $s1 d s2$

by (*metis UNIV-I Union-iff*)

from $a1$ **have** $a1': \forall A \in W. \forall a1 \in A. \forall a2 \in A. \forall s. \text{runFw } s a1 c rs = \text{runFw } s a2 c rs$

unfolding *same-fw-behaviour-one-def* **by** *fast*

from *relation-lem*[*OF* $a1' a2 a3$] **have** $s1: \bigwedge d. \text{runFw } ip1 d c rs = \text{runFw } ip2 d c rs$ **by** *simp*

from $a1$ **have** $a1'': \forall A \in W. \forall a1 \in A. \forall a2 \in A. \forall d. \text{runFw } a1 d c rs = \text{runFw } a2 d c rs$

unfolding *same-fw-behaviour-one-def* **by** *fast*

from *relation-lem*[*OF* $a1'' a2 a4$] **have** $s2: \bigwedge s. \text{runFw } s ip1 c rs = \text{runFw } s ip2 c rs$ **by** *simp*

from $s1 s2$ **show** *same-fw-behaviour-one* $ip1 ip2 c rs$

unfolding *same-fw-behaviour-one-def* **by** *fast*

qed

lemma *sameFw-behave-sets*:

$\forall w \in \text{set } A. \forall a1 \in w. \forall a2 \in w. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs \implies$
 $\forall w1 \in \text{set } A. \forall w2 \in \text{set } A. \exists a1 \in w1. \exists a2 \in w2. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$
 \implies
 $\forall w1 \in \text{set } A. \forall w2 \in \text{set } A.$
 $\forall a1 \in w1. \forall a2 \in w2. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$

proof –

assume *a1*: $\forall w \in \text{set } A. \forall a1 \in w. \forall a2 \in w. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$
and

$\forall w1 \in \text{set } A. \forall w2 \in \text{set } A. \exists a1 \in w1. \exists a2 \in w2. \text{same-fw-behaviour-one } a1 \ a2$
c rs

from *this have* $\forall w1 \in \text{set } A. \forall w2 \in \text{set } A. \exists a1 \in w1. \forall a2 \in w2. \text{same-fw-behaviour-one}$
a1 a2 c rs

using *same-fw-behaviour-one-equi(3)* **by** *metis*

from *a1 this show* $\forall w1 \in \text{set } A. \forall w2 \in \text{set } A. \forall a1 \in w1. \forall a2 \in w2. \text{same-fw-behaviour-one}$
a1 a2 c rs

using *same-fw-behaviour-one-equi(3)* **by** *metis*

qed

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* = ($\lambda wi. (\text{map } (\lambda d. \text{runFw } (\text{getOneIp } wi) \ d \ c \ rs) (\text{map}$
getOneIp W),
 $\text{map } (\lambda s. \text{runFw } s \ (\text{getOneIp } wi) \ c \ rs) (\text{map } \text{getOneIp}$
W))) in
groupF f W))

lemma *groupWIs-not-empty*: *groupWIs c rs* \neq []

proof –

have *getParts rs* \neq [] **by** (*simp add: getParts-def partitioningIps-nonempty*)

with *groupF-empty* **have** $\bigwedge f. \text{groupF } f \ (\text{getParts } rs) \neq []$ **by** *blast*

thus *?thesis* **by** (*simp add: groupWIs-def Let-def*) *blast*

qed

lemma *groupWIs-not-empty-elem*: $V \in \text{set } (\text{groupWIs } c \ rs) \implies V \neq []$

by (*simp add: groupWIs-def Let-def groupF-empty-elem*)

lemma *groupWIs-not-empty-elems*:

assumes *V*: $V \in \text{set } (\text{groupWIs } c \ rs)$ **and** *w*: $w \in \text{set } V$

shows $\neg \text{wordinterval-empty } w$

proof –

have $\forall w \in \text{set } (\text{concat } (\text{groupWIs } c \ rs)). \neg \text{wordinterval-empty } w$

apply (*subst groupWIs-def*)

```

apply(subst Let-def)+
apply(subst groupF-concat-set)
using getParts-nonempty-elems by blast
from this V w show ?thesis by auto
qed

```

```

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 –
  have ∀ A ∈ set (map wordinterval-to-set (concat (groupWIs c rs))).
    ∀ a1 ∈ A. ∀ a2 ∈ A. same-fw-behaviour-one a1 a2 c rs
  apply(subst groupWIs-def)
  apply(subst Let-def)+
  apply(subst set-map)
  apply(subst groupF-concat-set)
  using getParts-same-fw-behaviour same-fw-spec by fastforce
  from this assms show ?thesis by force
qed

```

```

lemma groupWIs-same-fw-not: A ∈ set (groupWIs c rs) ⇒ B ∈ set (groupWIs
c rs) ⇒

```

$$\begin{aligned}
& A \neq B \implies \\
& \quad \forall aw \in \text{set } (\text{map } \text{wordinterval-to-set } A). \\
& \quad \forall bw \in \text{set } (\text{map } \text{wordinterval-to-set } B). \\
& \quad \forall a \in aw. \forall b \in bw. \neg \text{same-fw-behaviour-one } a \ b \ c \ rs
\end{aligned}$$

```

proof –
  assume asm: A ∈ set (groupWIs c rs) B ∈ set (groupWIs c rs) A ≠ B
  from this have b1: ∀ aw ∈ set A. ∀ bw ∈ set B.
    (map (λd. runFw (getOneIp aw) d c rs) (map getOneIp (getParts
rs))),
    map (λs. runFw s (getOneIp aw) c rs) (map getOneIp (getParts
rs))) ≠
    (map (λd. runFw (getOneIp bw) d c rs) (map getOneIp (getParts
rs))),
    map (λs. runFw s (getOneIp bw) c rs) (map getOneIp (getParts
rs)))
  apply(simp add: groupWIs-def Let-def)
  using groupF-inequality by fastforce
  have same-behave-runFw-not:
    (map (λd. runFw x1 d c rs) W, map (λs. runFw s x1 c rs) W) ≠
    (map (λd. runFw x2 d c rs) W, map (λs. runFw s x2 c rs) W) ⇒
    ¬ same-fw-behaviour-one x1 x2 c rs for x1 x2 W
  by (simp add: same-fw-behaviour-one-def) (blast)
  have ∀ C ∈ set (groupWIs c rs). ∀ c ∈ set C. getOneIp c ∈ wordinterval-to-set c
  apply(simp add: groupWIs-def Let-def)
  using getParts-nonempty-elems groupF-set getOneIp-elem by fastforce
  from this b1 asm have

```

$\forall aw \in \text{set } (\text{map } \text{wordinterval-to-set } A). \forall bw \in \text{set } (\text{map } \text{wordinterval-to-set } B).$
 $\exists a \in aw. \exists b \in bw. (\text{map } (\lambda d. \text{runFw } a \ d \ c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs)),$
 $\text{map } (\lambda s. \text{runFw } s \ a \ c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs))) \neq$
 $(\text{map } (\lambda d. \text{runFw } b \ d \ c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs)), \text{map } (\lambda s. \text{runFw } s \ b$
 $c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs)))$
by (*simp*) (*blast*)
from *this same-behave-runFw-not asm*
have $\forall aw \in \text{set } (\text{map } \text{wordinterval-to-set } A). \forall bw \in \text{set } (\text{map } \text{wordinterval-to-set } B).$
 $\exists a \in aw. \exists b \in bw. \neg \text{same-fw-behaviour-one } a \ b \ c \ rs$ **by** *fast*
from *this groupParts-same-fw-wi0[of A c rs] groupParts-same-fw-wi0[of B c rs]*
asm
have $\forall aw \in \text{set } (\text{map } \text{wordinterval-to-set } A).$
 $\forall bw \in \text{set } (\text{map } \text{wordinterval-to-set } B).$
 $\forall a \in aw. \exists b \in bw. \neg \text{same-fw-behaviour-one } a \ b \ c \ rs$
apply(*simp*) **using** *same-fw-behaviour-one-equi(3)* **by** *blast*
from *this groupParts-same-fw-wi0[of A c rs] groupParts-same-fw-wi0[of B c rs]*
asm
show $\forall aw \in \text{set } (\text{map } \text{wordinterval-to-set } A).$
 $\forall bw \in \text{set } (\text{map } \text{wordinterval-to-set } B).$
 $\forall a \in aw. \forall b \in bw. \neg \text{same-fw-behaviour-one } a \ b \ c \ rs$
apply(*simp*) **using** *same-fw-behaviour-one-equi(3)* **by** *fast*
qed

lemma *groupParts-same-fw-wi1:*

$V \in \text{set } (\text{groupWIs } c \ rs) \implies \forall w1 \in \text{set } V. \forall w2 \in \text{set } V.$

$\forall a1 \in \text{wordinterval-to-set } w1. \forall a2 \in \text{wordinterval-to-set } w2. \text{same-fw-behaviour-one}$
 $a1 \ a2 \ c \ rs$

proof –

assume *asm*: $V \in \text{set } (\text{groupWIs } c \ rs)$

from *getParts-same-fw-behaviour same-fw-spec*

have *b1*: $\forall A \in \text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts } rs)) . \forall a1 \in A. \forall a2 \in$
 $A.$

same-fw-behaviour-one a1 a2 c rs **by** *fast*

from *getParts-complete* **have** *complete*: $\bigcup (\text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts}$
 $rs))) = \text{UNIV}$

by(*simp add: wordinterval-list-to-set-def*)

from *getParts-nonempty-elems* **have** *nonempty*: $\forall w \in \text{set } (\text{getParts } rs). \neg \text{wordinter-}$
 $\text{val-empty } w$ **by** *simp*

{ **fix** $W \ x1 \ x2$

assume *a1*: $\forall A \in \text{set } (\text{map } \text{wordinterval-to-set } W). \forall a1 \in A. \forall a2 \in A.$
 $\text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$

and *a2*: *wordinterval-list-to-set W = UNIV*

and $a3: \forall w \in \text{set } W. \neg \text{wordinterval-empty } w$
and $a4: (\text{map } (\lambda d. \text{runFw } x1 \ d \ c \ rs) (\text{map } \text{getOneIp } W), \text{map } (\lambda s. \text{runFw } s \ x1 \ c \ rs) (\text{map } \text{getOneIp } W)) =$
 $(\text{map } (\lambda d. \text{runFw } x2 \ d \ c \ rs) (\text{map } \text{getOneIp } W), \text{map } (\lambda s. \text{runFw } s \ x2 \ c \ rs) (\text{map } \text{getOneIp } W))$
from $a3 \ a4 \ \text{getOneIp-elem}$
have $b1: \forall B \in \text{set } (\text{map } \text{wordinterval-to-set } W). \exists b \in B. \text{runFw } x1 \ b \ c \ rs$
 $= \text{runFw } x2 \ b \ c \ rs$
by *fastforce*
from $a3 \ a4 \ \text{getOneIp-elem}$
have $b2: \forall B \in \text{set } (\text{map } \text{wordinterval-to-set } W). \exists b \in B. \text{runFw } b \ x1 \ c \ rs$
 $= \text{runFw } b \ x2 \ c \ rs$
by *fastforce*
from $\text{runFw-sameFw-behave}[OF \ a1 - b1 \ b2] \ a2[\text{unfolded wordinterval-list-to-set-def}]$
have
 $\text{same-fw-behaviour-one } x1 \ x2 \ c \ rs$ **by** *simp*
} note *same-behave-runFw=this*

from $\text{same-behave-runFw}[OF \ b1 \ \text{getParts-complete nonempty}]$
 $\text{groupF-equality}[of \ (\lambda wi. (\text{map } (\lambda d. \text{runFw } (\text{getOneIp } wi) \ d \ c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs)),$
 $\text{map } (\lambda s. \text{runFw } s \ (\text{getOneIp } wi) \ c \ rs) (\text{map } \text{getOneIp } (\text{getParts } rs))))$
 $(\text{getParts } rs)] \ \text{asm}$
have $b2: \forall a1 \in \text{set } V. \forall a2 \in \text{set } V. \text{same-fw-behaviour-one } (\text{getOneIp } a1) (\text{getOneIp } a2) \ c \ rs$
apply $(\text{subst } (\text{asm}) \ \text{groupWIs-def})$
apply $(\text{subst } (\text{asm}) \ \text{Let-def})+$
by *fast*
from $\text{groupWIs-not-empty-elems } \text{asm}$ **have** $\forall w \in \text{set } V. \neg \text{wordinterval-empty } w$ **by** *blast*
from $\text{this } b2 \ \text{getOneIp-elem}$
have $b3: \forall w1 \in \text{set } (\text{map } \text{wordinterval-to-set } V). \forall w2 \in \text{set } (\text{map } \text{wordinterval-to-set } V).$
 $\exists ip1 \in w1. \exists ip2 \in w2.$
 $\text{same-fw-behaviour-one } ip1 \ ip2 \ c \ rs$ **by** $(\text{simp}) \ (\text{blast})$
from $\text{groupParts-same-fw-wi0 } \text{asm}$
have $\forall A \in \text{set } (\text{map } \text{wordinterval-to-set } V). \forall a1 \in A. \forall a2 \in A. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$
by *metis*
from $\text{sameFw-behave-sets}[OF \ \text{this } b3]$
show $\forall w1 \in \text{set } V. \forall w2 \in \text{set } V.$
 $\forall a1 \in \text{wordinterval-to-set } w1. \forall a2 \in \text{wordinterval-to-set } w2. \text{same-fw-behaviour-one } a1 \ a2 \ c \ rs$
by *force*
qed

lemma $\text{groupParts-same-fw-wi2}: V \in \text{set } (\text{groupWIs } c \ rs) \implies$
 $\forall ip1 \in \text{wordinterval-list-to-set } V.$

$\forall ip2 \in \text{wordinterval-list-to-set } V.$
 $\text{same-fw-behaviour-one } ip1 \ ip2 \ c \ rs$

using *groupParts-same-fw-wi0 groupParts-same-fw-wi1*
apply (*simp add: wordinterval-list-to-set-def*)
by *fast*

lemma *groupWIs-same-fw-not2*: $A \in \text{set } (\text{groupWIs } c \ rs) \implies B \in \text{set } (\text{groupWIs } c \ rs) \implies$
 $A \neq B \implies$
 $\forall ip1 \in \text{wordinterval-list-to-set } A.$
 $\forall ip2 \in \text{wordinterval-list-to-set } B.$
 $\neg \text{same-fw-behaviour-one } ip1 \ ip2 \ c \ rs$

apply(*simp add: wordinterval-list-to-set-def*)
using *groupWIs-same-fw-not* **by** *fastforce*

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

using *groupWIs-same-fw-not2* **by** *blast*

lemma *groupWIs-complete*: $(\bigcup x \in \text{set } (\text{groupWIs } c \ rs). \text{wordinterval-list-to-set } x)$
 $= (\text{UNIV}::'i::\text{len word set})$

proof –
have $(\bigcup y \in (\bigcup x \in \text{set } (\text{groupWIs } c \ rs). \text{set } x). \text{wordinterval-to-set } y) = (\text{UNIV}::'i$
 $\text{word set})$

apply(*simp add: groupWIs-def Let-def groupF-Union-set*)
using *getParts-complete wordinterval-list-to-set-def* **by** *fastforce*
thus *?thesis* **by**(*simp add: wordinterval-list-to-set-def*)

qed

definition *groupWIs1* :: $'a \text{ parts-connection-scheme} \Rightarrow 'i::\text{len simple-rule list} \Rightarrow$
 $'i \text{ wordinterval list list}$ **where**

$\text{groupWIs1 } c \ rs = (\text{let } P = \text{getParts } rs \text{ in}$
 $(\text{let } W = \text{map } \text{getOneIp } P \text{ in}$
 $(\text{let } f = (\lambda wi. (\text{map } (\lambda d. \text{runFw } (\text{getOneIp } wi) \ d \ c \ rs) \ W,$
 $\text{map } (\lambda s. \text{runFw } s \ (\text{getOneIp } wi) \ c \ rs) \ W)) \text{ in}$
 $\text{map } (\text{map } \text{fst}) \ (\text{groupF } \text{snd} \ (\text{map } (\lambda x. (x, f \ x)) \ P))))$

lemma *groupWIs-groupWIs1-equi*: $\text{groupWIs1 } c \ rs = \text{groupWIs } c \ rs$

apply(*subst groupWIs1-def*)
apply(*subst groupWIs-def*)
using *groupF-tuple* **by** *metis*

definition *simple-conn-matches* :: $'i::\text{len simple-match} \Rightarrow \text{parts-connection} \Rightarrow$

bool where

```
simple-conn-matches m c  $\longleftrightarrow$ 
  (match-iface (iiface m) (pc-iiface c))  $\wedge$ 
  (match-iface (oiface m) (pc-oiface c))  $\wedge$ 
  (match-proto (proto m) (pc-proto c))  $\wedge$ 
  (simple-match-port (sports m) (pc-sport c))  $\wedge$ 
  (simple-match-port (dports m) (pc-dport c))
```

lemma *simple-conn-matches-simple-match-any: simple-conn-matches simple-match-any*
c

```
apply (simp add: simple-conn-matches-def simple-match-any-def match-ifaceAny)
apply (subgoal-tac (65535::16 word) = max-word)
apply (simp only:)
apply simp-all
done
```

lemma *has-default-policy-simple-conn-matches:*

has-default-policy rs \implies has-default-policy [r \leftarrow rs . simple-conn-matches (match-sel r) c]

```
apply (induction rs rule: has-default-policy.induct)
  apply (simp; fail)
  apply (simp add: simple-conn-matches-simple-match-any; fail)
  apply (simp)
  apply (intro conjI)
  apply (simp split: if-split-asm; fail)
  apply (simp add: has-default-policy-fst split: if-split-asm)
done
```

lemma *filter-conn-fw-lem:*

runFw s d c (filter (λr . simple-conn-matches (match-sel r) c) rs) = runFw s d
c rs

```
apply (simp add: runFw-def simple-conn-matches-def match-sel-def)
apply (induction rs ( $\lambda p$ -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=""))
  rule: simple-fw.induct)
  apply (simp add: simple-matches.simps)+
done
```

definition *groupWIs2 :: parts-connection \Rightarrow 'i::len simple-rule list \Rightarrow 'i wordinterval list list* **where**

```
groupWIs2 c rs = (let P = getParts rs in
  (let W = map getOneIp P in
    (let filterW = (filter ( $\lambda r$ . simple-conn-matches (match-sel r)
c) rs) in
```

(let f = (λwi. (map (λd. runFw (getOneIp wi) d c filterW) W,
map (λs. runFw s (getOneIp wi) c filterW) W)) in
map (map fst) (groupF snd (map (λx. (x, f x)) P))))))

lemma groupWIs1-groupWIs2-equi: groupWIs2 c rs = groupWIs1 c rs
by(simp add: groupWIs2-def groupWIs1-def filter-conn-fw-lem)

lemma groupWIs-code[code]: groupWIs c rs = groupWIs2 c rs
using groupWIs1-groupWIs2-equi groupWIs-groupWIs1-equi **by** metis

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)) = wordinter-
val-to-set (matching-dsts s rs a2) - wordinterval-to-set a1
apply(induction s rs a2 arbitrary: a1 a2 rule: matching-dsts.induct)
apply(simp-all)
by blast+

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 (ipcldr-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)) = wordinter-
val-to-set (matching-srcs d rs a2) - wordinterval-to-set a1
apply(induction d rs a2 arbitrary: a1 a2 rule: matching-srcs.induct)
apply(simp-all)
by blast+

```

lemma *matching-dsts*: $\forall r \in \text{set } rs. \text{simple-conn-matches (match-sel } r) c \implies$

```

wordinterval-to-set (matching-dsts s rs Empty-WordInterval) = {d. runFw
s d c rs = Decision FinalAllow}
proof(induction rs)
case Nil thus ?case by (simp add: runFw-def)
next
case (Cons r rs)
obtain m a where r: r = SimpleRule m a by(cases r, blast)

```

```

from Cons.prem1 r have simple-match-ip-Accept:  $\bigwedge d. \text{simple-match-ip (src}
m) s \implies$ 
runFw s d c (SimpleRule m Accept # rs) = Decision FinalAllow  $\longleftrightarrow$ 
simple-match-ip (dst m) d  $\vee$  runFw s d c rs = Decision FinalAllow
by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)

```

```

{ fix d a
have  $\neg \text{simple-match-ip (src } m) s \implies$ 
runFw s d c (SimpleRule m a # rs) = Decision FinalAllow  $\longleftrightarrow$  runFw s d
c rs = Decision FinalAllow
apply(cases a)
by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)+
} note not-simple-match-ip=this

```

```

from Cons.prem1 r have simple-match-ip-Drop:  $\bigwedge d. \text{simple-match-ip (src } m)
s \implies$ 
runFw s d c (SimpleRule m Drop # rs) = Decision FinalAllow  $\longleftrightarrow$   $\neg$ 
simple-match-ip (dst m) d  $\wedge$  runFw s d c rs = Decision FinalAllow
by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)

```

show ?case

```

proof(cases a)
case Accept with r Cons show ?thesis
apply(simp, intro conjI impI)

```

```

apply(simp add: simple-match-ip-Accept wordinterval-to-set-ipcldr-tuple-to-wordinterval-simple-match-ip)
apply blast
apply(simp add: not-simple-match-ip; fail)

```

```

done
next
case Drop with r Cons show ?thesis
  apply(simp,intro conjI impI)
  apply(simp add: simple-match-ip-Drop matching-dsts-pull-out-accu
wordinterval-to-set-ipcldr-tuple-to-wordinterval-simple-match-ip-set)
  apply blast
  apply(simp add: not-simple-match-ip; fail)
done
qed
qed
lemma matching-srcs:  $\forall r \in \text{set } rs. \text{simple-conn-matches } (\text{match-sel } r) c \implies$ 
  wordinterval-to-set (matching-srcs d rs Empty-WordInterval) = {s. runFw
s d c rs = Decision FinalAllow}
proof(induction rs)
case Nil thus ?case by (simp add: runFw-def)
next
case (Cons r rs)
  obtain m a where r: r = SimpleRule m a by(cases r, blast)

  from Cons.prem1 r have simple-match-ip-Accept:  $\bigwedge s. \text{simple-match-ip } (\text{dst } m) d \implies$ 
    runFw s d c (SimpleRule m Accept # rs) = Decision FinalAllow  $\longleftrightarrow$ 
    simple-match-ip (src m) s  $\vee$  runFw s d c rs = Decision FinalAllow
  by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)

  { fix s a
    have  $\neg \text{simple-match-ip } (\text{dst } m) d \implies$ 
      runFw s d c (SimpleRule m a # rs) = Decision FinalAllow  $\longleftrightarrow$  runFw s d
c rs = Decision FinalAllow
    apply(cases a)
    by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)+
  } note not-simple-match-ip=this

  from Cons.prem2 r have simple-match-ip-Drop:  $\bigwedge s. \text{simple-match-ip } (\text{dst } m) d \implies$ 
    runFw s d c (SimpleRule m Drop # rs) = Decision FinalAllow  $\longleftrightarrow$ 
     $\neg \text{simple-match-ip } (\text{src } m) s \wedge \text{runFw s d c rs} = \text{Decision FinalAllow}$ 
  by(simp add: simple-conn-matches-def runFw-def simple-matches.simps)

show ?case
proof(cases a)
case Accept with r Cons show ?thesis
  apply(simp, intro conjI impI)
  apply(simp add: simple-match-ip-Accept wordinterval-to-set-ipcldr-tuple-to-wordinterval-simple-match-ip)
  apply blast
  apply(simp add: not-simple-match-ip; fail)
done
next

```

```

case Drop with r Cons show ?thesis
  apply(simp,intro conjI impI)
    apply(simp add: simple-match-ip-Drop matching-srcs-pull-out-accu
wordinterval-to-set-ipcidr-tuple-to-wordinterval-simple-match-ip-set)
      apply blast
    apply(simp add: not-simple-match-ip; fail)
  done
qed
qed

```

```

definition groupWIs3-default-policy :: parts-connection  $\Rightarrow$  'i::len simple-rule list
 $\Rightarrow$  'i wordinterval list list where
  groupWIs3-default-policy c rs = (let P = getParts rs in
    (let W = map getOneIp P in
      (let filterW = (filter ( $\lambda$ r. simple-conn-matches (match-sel r)
c) rs) in
        (let f = ( $\lambda$ wi. let mtch-dsts = (matching-dsts (getOneIp wi)
filterW Empty-WordInterval);
          mtch-srcs = (matching-srcs (getOneIp wi)
filterW Empty-WordInterval) in
            (map ( $\lambda$ d. wordinterval-element d mtch-dsts) W,
              map ( $\lambda$ s. wordinterval-element s mtch-srcs) W)) in
              map (map fst) (groupF snd (map ( $\lambda$ 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 -
  { fix filterW s d
    from matching-dsts[where c=c] have filterW = filter ( $\lambda$ r. simple-conn-matches
(match-sel r) c) rs  $\implies$ 
      wordinterval-element d (matching-dsts s filterW Empty-WordInterval)  $\longleftrightarrow$ 
runFw s d c filterW = Decision FinalAllow
    by force
    } note matching-dsts-filterW=this[simplified]

  { fix filterW s d
    from matching-srcs[where c=c] have filterW = filter ( $\lambda$ r. simple-conn-matches
(match-sel r) c) rs  $\implies$ 
      wordinterval-element s (matching-srcs d filterW Empty-WordInterval)
 $\longleftrightarrow$  runFw s d c filterW = Decision FinalAllow
    by force
    } note matching-srcs-filterW=this[simplified]

```

```

{ fix W and rs :: 'i::len simple-rule list
  assume assms': has-default-policy rs
  have groupF (λwi. (map (λd. runFw (getOneIp wi) d c rs = Decision
FinalAllow) (map getOneIp W),
              map (λs. runFw s (getOneIp wi) c rs = Decision FinalAllow)
(map getOneIp W))) W =
  groupF (λwi. (map (λd. runFw (getOneIp wi) d c rs) (map getOneIp
W),
          map (λs. runFw s (getOneIp wi) c rs) (map getOneIp W))) W

proof -
{
  fix f1::'w ⇒ 'u ⇒ 'v and f2:: 'w ⇒ 'u ⇒ 'x and x and y and g1::'w ⇒
'u ⇒ 'y and g2::'w ⇒ 'u ⇒ 'z and W::'u list
  assume 1: ∀ w ∈ set W. (f1 x) w = (f1 y) w ↔ (f2 x) w = (f2 y) w
  and 2: ∀ w ∈ set W. (g1 x) w = (g1 y) w ↔ (g2 x) w = (g2 y) w
  have
    ((map (f1 x) W, map (g1 x) W) = (map (f1 y) W, map (g1 y) W))
    ↔
    ((map (f2 x) W, map (g2 x) W) = (map (f2 y) W, map (g2 y) W))
  proof -
    from 1 have p1: (map (f1 x) W = map (f1 y) W ↔ map (f2 x) W =
map (f2 y) W) by(induction W)(simp-all)
    from 2 have p2: (map (g1 x) W = map (g1 y) W ↔ map (g2 x) W
= map (g2 y) W) by(induction W)(simp-all)
    from p1 p2 show ?thesis by fast
  qed
} note map-over-tuples-equal-helper=this

show ?thesis
apply(rule groupF-cong)
apply(intro ballI)
apply(rule map-over-tuples-equal-helper)
using has-default-policy-runFw[OF assms'] by metis+
qed
} note has-default-policy-groupF=this[simplified]

from assms show ?thesis
apply(simp add: groupWIs3-default-policy-def groupWIs-code[symmetric])
apply(subst groupF-tuple[symmetric])
apply(simp add: Let-def)
apply(simp add: matching-srcs-filterW matching-dsts-filterW)
apply(subst has-default-policy-groupF)
apply(simp add: has-default-policy-simple-conn-matches; fail)
apply(simp add: groupWIs-def Let-def filter-conn-fw-lem)
done
qed

```

```

definition groupWIs3 :: parts-connection ⇒ 'i::len simple-rule list ⇒ 'i wordinter-

```

val 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*

by (*simp add: fun-eq-iff groupWIs3-def groupWIs-code groupWIs3-default-policy-groupWIs2*)

definition *build-ip-partition :: parts-connection \Rightarrow 'i::len simple-rule list \Rightarrow 'i
wordinterval list where*

*build-ip-partition c rs = map
(λ xs. wordinterval-sort (wordinterval-compress (foldr wordinterval-union xs
Empty-WordInterval)))
(groupWIs3 c rs)*

theorem *build-ip-partition-same-fw: $V \in \text{set (build-ip-partition c rs)} \implies$*

$\forall ip1::'i::len \text{ word} \in \text{wordinterval-to-set } V.$

$\forall ip2::'i::len \text{ word} \in \text{wordinterval-to-set } V.$

same-fw-behaviour-one ip1 ip2 c rs

apply (*simp add: build-ip-partition-def groupWIs3*)

using *wordinterval-list-to-set-compressed groupParts-same-fw-wi2 wordinterval-sort*
by *blast*

theorem *build-ip-partition-same-fw-min: $A \in \text{set (build-ip-partition c rs)} \implies B$
 $\in \text{set (build-ip-partition c rs)} \implies$*

$A \neq B \implies$

$\forall ip1::'i::len \text{ word} \in \text{wordinterval-to-set } A.$

$\forall ip2::'i::len \text{ word} \in \text{wordinterval-to-set } B.$

$\neg \text{same-fw-behaviour-one ip1 ip2 c rs}$

apply (*simp add: build-ip-partition-def groupWIs3*)

using *groupWIs-same-fw-not2 wordinterval-list-to-set-compressed wordinterval-sort*
by *blast*

theorem *build-ip-partition-complete: $(\bigcup x \in \text{set (build-ip-partition c rs)}. \text{wordinter-}$
*val-to-set x) = (UNIV :: 'i::len word set)**

proof –

have *wordinterval-to-set (foldr wordinterval-union x Empty-WordInterval) =*
 $\bigcup (\text{set (map wordinterval-to-set x)})$

for *$x::'i \text{ wordinterval list}$*

by (*induction x*) *simp-all*

thus *?thesis*

apply (*simp add: build-ip-partition-def groupWIs3 wordinterval-compress wordinter-
val-sort*)

using *groupWIs-complete[simplified wordinterval-list-to-set-def]* **by** *simp*

qed

lemma *build-ip-partition-no-empty-elems*: $wi \in \text{set } (\text{build-ip-partition } c \text{ rs}) \implies \neg \text{wordinterval-empty } wi$
proof –
assume $wi \in \text{set } (\text{build-ip-partition } c \text{ rs})$
hence *assm*: $wi \in (\lambda xs. \text{wordinterval-sort } (\text{wordinterval-compress } (\text{foldr } \text{wordinterval-union } xs \text{ Empty-WordInterval}))) \text{ ' set } (\text{groupWIs } c \text{ rs})$
by (*simp add: build-ip-partition-def groupWIs3*)
from *assm* **obtain** *wi-orig* **where** 1: $wi\text{-orig} \in \text{set } (\text{groupWIs } c \text{ rs})$ **and**
2: $wi = \text{wordinterval-sort } (\text{wordinterval-compress } (\text{foldr } \text{wordinterval-union } wi\text{-orig } \text{Empty-WordInterval}))$ **by** *blast*
from 1 *groupWIs-not-empty-elem* **have** *i1*: $wi\text{-orig} \neq \{\}$ **by** *blast*
from 1 *groupWIs-not-empty-elems* **have** *i2*: $\bigwedge w. w \in \text{set } wi\text{-orig} \implies \neg \text{wordinterval-empty } w$ **by** *simp*
from *i1 i2* **have** $\text{wordinterval-to-set } (\text{foldr } \text{wordinterval-union } wi\text{-orig } \text{Empty-WordInterval}) \neq \{\}$
by (*induction wi-orig*) *simp-all*
with 2 **show** *?thesis* **by** (*simp add: wordinterval-compress wordinterval-sort*)
qed

lemma *build-ip-partition-disjoint*:
 $V1 \in \text{set } (\text{build-ip-partition } c \text{ rs}) \implies V2 \in \text{set } (\text{build-ip-partition } c \text{ rs}) \implies V1 \neq V2 \implies \text{wordinterval-to-set } V1 \cap \text{wordinterval-to-set } V2 = \{\}$
by (*metis build-ip-partition-same-fw build-ip-partition-same-fw-min disjoint-iff*)

lemma *map-wordinterval-to-set-distinct*:
assumes *distinct*: *distinct xs*
and *disjoint*: $(\forall x1 \in \text{set } xs. \forall x2 \in \text{set } xs. x1 \neq x2 \longrightarrow \text{wordinterval-to-set } x1 \cap \text{wordinterval-to-set } x2 = \{\})$
and *notempty*: $\forall x \in \text{set } xs. \neg \text{wordinterval-empty } x$
shows *distinct* (*map wordinterval-to-set xs*)
proof –
have $\neg \text{wordinterval-empty } x1 \implies \text{wordinterval-to-set } x1 \cap \text{wordinterval-to-set } x2 = \{\} \implies \text{wordinterval-to-set } x1 \neq \text{wordinterval-to-set } x2$ **for** $x1::('b::\text{len}) \text{ wordinterval}$
and $x2$
by *auto*
with *disjoint notempty* **have** $(\forall x1 \in \text{set } xs. \forall x2 \in \text{set } xs. x1 \neq x2 \longrightarrow \text{wordinterval-to-set } x1 \neq \text{wordinterval-to-set } x2)$
by *force*
with *distinct* **show** *distinct* (*map wordinterval-to-set xs*)
proof (*induction xs*)
case *Cons* **thus** *?case* **by** *simp fast*
qed (*simp*)

qed

lemma *map-getOneIp-distinct*: **assumes**

distinct: *distinct xs*

and *disjoint*: $(\forall x1 \in \text{set } xs. \forall x2 \in \text{set } xs. x1 \neq x2 \longrightarrow \text{wordinterval-to-set } x1 \cap \text{wordinterval-to-set } x2 = \{\})$

and *notempty*: $\forall x \in \text{set } xs. \neg \text{wordinterval-empty } x$

shows *distinct* (*map getOneIp xs*)

proof –

have $\neg \text{wordinterval-empty } x \implies \neg \text{wordinterval-empty } xa \implies$

$\text{wordinterval-to-set } x \cap \text{wordinterval-to-set } xa = \{\} \implies \text{getOneIp } x \neq$

getOneIp xa

for $x \text{ } xa::'b::\text{len wordinterval}$

by(*fastforce dest: getOneIp-elem*)

with *disjoint notempty* **have** $(\forall x1 \in \text{set } xs. \forall x2 \in \text{set } xs. x1 \neq x2 \longrightarrow \text{getOneIp } x1 \neq \text{getOneIp } x2)$

by *metis*

with *distinct* **show** *?thesis*

proof(*induction xs*)

case *Cons* **thus** *?case* **by** *simp fast*

qed(*simp*)

qed

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

proof–

have *disjoint-list-partitioningIps*:

$\{\} \notin \text{set } (\text{map wordinterval-to-set } ts) \implies \text{disjoint-list } (\text{map wordinterval-to-set } ts) \implies$

$(\text{wordinterval-list-to-set } ss) \subseteq (\text{wordinterval-list-to-set } ts) \implies$

disjoint-list (map wordinterval-to-set (partitioningIps ss ts))

for $ts::'a::\text{len wordinterval list}$ **and** ss

by (*simp add: partitioning1-disjoint-list partitioningIps-equi wordinterval-list-to-set-def*)

have $\{\} \notin \text{set } (\text{map wordinterval-to-set } [\text{wordinterval-UNIV}])$

and *disjoint-list (map wordinterval-to-set [wordinterval-UNIV])*

and $\text{wordinterval-list-to-set } (\text{extract-IPSets } rs) \subseteq \text{wordinterval-list-to-set } [\text{wordinterval-UNIV}]$

by(*simp add: wordinterval-list-to-set-def disjoint-list-def disjoint-def*)+

thus *?thesis*

unfolding *getParts-def* **by**(*rule disjoint-list-partitioningIps*)

qed

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

proof –

have

$(\text{wordinterval-to-set} \circ (\lambda xs. \text{wordinterval-sort } (\text{wordinterval-compress } (\text{foldr wordinterval-union } xs \text{ Empty-WordInterval})))) \text{ } ws$

$= \bigcup (\text{set } (\text{map wordinterval-to-set } ws))$ **for** $ws::'a::\text{len wordinterval list}$

proof(*induction ws*)

qed(*simp-all add: wordinterval-compress wordinterval-sort*)
hence *hlp1*: *map wordinterval-to-set (build-ip-partition c rs) =*
map (λx. ⋃(set (map wordinterval-to-set x))) (groupWIs c rs)
unfolding *build-ip-partition-def groupWIs3* **by** *auto*

— generic rule

have $\forall x \in \text{set } xs. \neg \text{wordinterval-empty } x \implies$
disjoint-list (map wordinterval-to-set xs) \implies
distinct (map (λx. ⋃(set (map wordinterval-to-set x))) (groupF f xs))
for *f::'x::len wordinterval \Rightarrow 'y and xs::'x::len wordinterval list*
proof(*induction f xs rule: groupF.induct*)
case 1 thus ?case by simp
next
case (*2 f x xs*)
have *hlp-internal*:
 $\bigcup (\text{set } (\text{map } (\lambda x. \bigcup (\text{set } (\text{map } \text{wordinterval-to-set } x))) (\text{groupF } f \text{ xs}))) =$
 $\bigcup (\text{set } (\text{map } \text{wordinterval-to-set } xs))$ **for** *f::'x wordinterval \Rightarrow 'y and xs*
by(*induction f xs rule: groupF.induct*) (*auto*)

from *2(2,3)* **have** *wordinterval-to-set x $\cap \bigcup (\text{wordinterval-to-set ' set xs}) =$*
 $\{\}$
by(*auto simp add: disjoint-def disjoint-list-def*)
hence $\neg (\text{wordinterval-to-set } x) \subseteq \bigcup (\text{wordinterval-to-set ' set xs})$ **using** *2(2)*
by *auto*
hence $\neg \text{wordinterval-to-set } x \subseteq \bigcup (\text{set } (\text{map } \text{wordinterval-to-set } [y \leftarrow xs . f x \neq f y]))$ **by** *auto*
hence $\neg \text{wordinterval-to-set } x \cup (\bigcup_{x \in \{xa \in \text{set } xs. f x = f xa\}} \text{wordinterval-to-set } x) \subseteq \bigcup (\text{set } (\text{map } (\lambda x. \bigcup (\text{set } (\text{map } \text{wordinterval-to-set } x)))) (\text{groupF } f [y \leftarrow xs . f x \neq f y]))$
unfolding *hlp-internal* **by** *blast*
hence *g1*: *wordinterval-to-set x $\cup (\bigcup_{x \in \{xa \in \text{set } xs. f x = f xa\}} \text{wordinterval-to-set } x) \notin (\lambda x. \bigcup_{x \in \text{set } x. \text{wordinterval-to-set } x) ' \text{set } (\text{groupF } f [y \leftarrow xs . f x \neq f y])$*
by *force*

from *2(3)* **have** *distinct (map wordinterval-to-set [y ← xs . f x ≠ f y])*
by (*simp add: disjoint-list-def distinct-map-filter*)
moreover from *2* **have** *disjoint (wordinterval-to-set ' {xa ∈ set xs. f x ≠ f xa})*
by(*simp add: disjoint-def disjoint-list-def*)
ultimately have *g2*: *distinct (map (λx. ⋃_{x ∈ set x. wordinterval-to-set x} (groupF f [y ← xs . f x ≠ f y]))*
using *2(1,2) unfolding disjoint-list-def* **by**(*simp*)

from *g1 g2* **show** *?case by simp*
qed
with *getParts-disjoint-list getParts-nonempty-elems* **have**
distinct
(map (λx. ⋃(set (map wordinterval-to-set x)))

```

      (groupF (λwi. (map (λd. runFw (getOneIp wi) d c rs) (map getOneIp (getParts
rs))),
              map (λs. runFw s (getOneIp wi) c rs) (map getOneIp (getParts
rs))))
      (getParts rs)) by blast

```

thus *?thesis* **unfolding** *hlp1 groupWIs-def Let-def* **by** *presburger*
qed

lemma *build-ip-partition-distinct'*: *distinct (build-ip-partition c rs)*
using *build-ip-partition-distinct distinct-mapI* **by** *blast*

17.2 Service Matrix over an IP Address Space Partition

definition *simple-firewall-without-interfaces* :: *'i::len simple-rule list ⇒ bool* **where**
simple-firewall-without-interfaces rs ≡ ∀ m ∈ match-sel ' set rs. iface m =
ifaceAny ∧ 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-iface:= x, p-oiface:= y))*

proof –

from *no-ifaces* **have** $\forall r \in \text{set } rs. \text{iface } (\text{match-sel } r) = \text{ifaceAny} \wedge \text{oiface } (\text{match-sel } r) = \text{ifaceAny}$

by (*simp add: simple-firewall-without-interfaces-def*)

thus *?thesis* **apply** (*induction rs p rule:simple-fw.induct*)

by (*simp-all add: simple-matches.simps match-ifaceAny*)

qed

lemma *runFw-no-interfaces*:

assumes *no-ifaces: simple-firewall-without-interfaces rs*

shows *runFw s d c rs = runFw s d (c(| pc-iface:= x, pc-oiface:= y)) rs*

apply (*simp add: runFw-def*)

apply (*subst simple-fw-no-interfaces[OF no-ifaces]*)

by (*simp*)

lemma[*code-unfold*]: *simple-firewall-without-interfaces rs ≡*

$\forall m \in \text{set } rs. \text{iface } (\text{match-sel } m) = \text{ifaceAny} \wedge \text{oiface } (\text{match-sel } m) = \text{ifaceAny}$

by (*simp add: simple-firewall-without-interfaces-def*)

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 \implies (s, d) ∈ *set* E \implies s ∈ *dom* (*map-of* V) **and**

(V, E) = *access-matrix* c rs \implies (s, d) ∈ *set* E \implies d ∈ *dom* (*map-of* V)

by(*auto simp add: access-matrix-def Let-def all-pairs-def*)

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

lemma (V, E) = *access-matrix* c rs \implies (s, d) ∈ *set* E \implies *runFw* s d c rs = *Decision FinalAllow*

by(*auto simp add: access-matrix-def Let-def*)

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 \implies k = f v

apply(*induction* rs)

apply(*simp*)

apply(*simp split: if-split-asm*)

done

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 –

let ?part=(*build-ip-partition* c rs)

have V: V = (*zip* (*map* *getOneIp* ?part) ?part)

using *matrix* **by**(*simp add: access-matrix-def Let-def*)

from *matrix repr* **have** *repr-Allow*: *runFw* s-repr d-repr c rs = *Decision FinalAllow*

by(*auto simp add: access-matrix-def Let-def*)

have *s-range-in-part*: *s-range* ∈ *set* ?part **using** V *s-range* **by** (*fastforce elim: in-set-zipE dest: map-of-SomeD*)

with *build-ip-partition-no-empty-elems* **have** \neg *wordinterval-empty* s-range **by** *simp*

have *d-range-in-part*: *d-range* ∈ *set* ?part **using** V *d-range* **by** (*fastforce elim: in-set-zipE dest: map-of-SomeD*)

with *build-ip-partition-no-empty-elems* **have** \neg *wordinterval-empty* d-range **by** *simp*

from *map-of-zip-map* V *s-range* **have** s-repr = *getOneIp* s-range **by** *fast*

```

with (¬ wordinterval-empty s-range) getOneIp-elem wordinterval-element-set-eq

have s-repr ∈ wordinterval-to-set s-range by blast

from map-of-zip-map V d-range have d-repr = getOneIp d-range by fast
with (¬ wordinterval-empty d-range) getOneIp-elem wordinterval-element-set-eq

have d-repr ∈ wordinterval-to-set d-range by blast

from s-range-in-part have s-range-in-part': s-range ∈ set (build-ip-partition c
rs) by simp
from d-range-in-part have d-range-in-part': d-range ∈ set (build-ip-partition c
rs) by simp

from build-ip-partition-same-fw[OF s-range-in-part', unfolded same-fw-behaviour-one-def]
s
(s-repr ∈ wordinterval-to-set s-range)

have
  ∀ d. runFw s-repr d c rs = runFw s d c rs by blast
  with repr-Allow have 1: runFw s d-repr c rs = Decision FinalAllow by simp

from build-ip-partition-same-fw[OF d-range-in-part', unfolded same-fw-behaviour-one-def]
d
(d-repr ∈ wordinterval-to-set d-range)

have
  ∀ s. runFw s d-repr c rs = runFw s d c rs by blast
  with 1 have 2: runFw s d c rs = Decision FinalAllow by simp
  thus ?thesis .
qed

```

```

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(induction xs)
case Nil thus ?case by simp
next
case (Cons x xs)
  consider v = x | v ∈ set xs using Cons.prem1 by fastforce
  thus ?case
  proof(cases)
    case 1 thus ?thesis by simp blast
  next
    case 2 with Cons.IH Cons.prem2 obtain s-repr where
      s-repr: map-of (zip (map getOneIp xs) xs) s-repr = Some v by force
    show ?thesis
    proof(cases s-repr ≠ getOneIp x)
      case True with Cons.prem2 s-repr show ?thesis by(rule-tac x=s-repr in
exI, simp)

```

```

      next
      case False with Cons.premis s-repr show ?thesis by(fastforce elim:
in-set-zipE)
    qed
  qed
qed

```

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 -
    let ?part=(build-ip-partition c rs)
    have V: V = zip (map getOneIp ?part) ?part
      using matrix by(simp add: access-matrix-def Let-def)
    have E: E = [(s, d) $\leftarrow$ all-pairs (map getOneIp ?part). runFw s d c rs = Decision
FinalAllow]
      using matrix by(simp add: access-matrix-def Let-def)

```

have *build-ip-partition-obtain*:

```

 $\exists$  V. V  $\in$  set (build-ip-partition c rs)  $\wedge$  s  $\in$  wordinterval-to-set V for s
using build-ip-partition-complete by blast

```

have *distinct-map-getOneIp-build-ip-partition-obtain*:

```

  v  $\in$  set (build-ip-partition c rs)  $\implies$ 
 $\exists$  s-repr. map-of (zip (map getOneIp (build-ip-partition c rs)) (build-ip-partition
c rs)) s-repr = Some v

```

for v and rs :: 'i::len simple-rule list

proof(erule *distinct-map-getOneIp-obtain*)

show *distinct* (map getOneIp (build-ip-partition c rs))

apply(rule *map-getOneIp-distinct*)

subgoal using *build-ip-partition-distinct'* by blast

subgoal using *build-ip-partition-disjoint build-ip-partition-distinct'* by blast

subgoal using *build-ip-partition-no-empty-elems[simplified]* by auto

done

qed

from *build-ip-partition-obtain* obtain s-range where

s-range \in set ?part and s \in wordinterval-to-set s-range by blast

from this *distinct-map-getOneIp-build-ip-partition-obtain* V obtain s-repr

where

ex-s1: (map-of V) s-repr = Some s-range and ex-s2: s \in wordinterval-to-set s-range

by blast

from *build-ip-partition-obtain* **obtain** *d-range* **where**
d-range \in *set ?part* **and** *d* \in *wordinterval-to-set d-range* **by** *blast*
from *this distinct-map-getOneIp-build-ip-partition-obtain V* **obtain** *d-repr*
where
ex-d1: (*map-of V*) *d-repr* = *Some d-range* **and** *ex-d2*: *d* \in *wordinterval-to-set d-range*
by *blast*

have *1*: *s-repr* \in *getOneIp 'set (build-ip-partition c rs)*
using *V (map-of V s-repr = Some s-range)* **by** (*fastforce elim: in-set-zipE*
dest: map-of-SomeD)
have *2*: *d-repr* \in *getOneIp 'set (build-ip-partition c rs)*
using *V (map-of V d-repr = Some d-range)* **by** (*fastforce elim: in-set-zipE*
dest: map-of-SomeD)

have *runFw s-repr d-repr c rs = Decision FinalAllow*
proof –
have *f1*: ($\forall w wa p ss. \neg$ *same-fw-behaviour-one w wa (p::parts-connection) ss*
 \vee
 $(\forall wb wc. runFw w wb p ss = runFw wa wb p ss \wedge runFw wc w p ss =$
 $runFw wc wa p ss)) \wedge$
 $(\forall w wa p ss. (\exists wb wc. runFw w wb (p::parts-connection) ss \neq runFw$
 $wa wb p ss \vee runFw wc w p ss \neq runFw wc wa p ss) \vee$
 $same-fw-behaviour-one w wa p ss)$
unfolding *same-fw-behaviour-one-def* **by** *blast*
from (*s-range* \in *set (build-ip-partition c rs)*) **have** *f2*: *same-fw-behaviour-one*
s s-repr c rs
by (*metis (no-types) map-of-zip-map V build-ip-partition-no-empty-elems*
build-ip-partition-same-fw ex-s1 ex-s2 getOneIp-elem wordinterval-element-set-eq)
from (*d-range* \in *set (build-ip-partition c rs)*) **have** *same-fw-behaviour-one*
d-repr d c rs
by (*metis (no-types) map-of-zip-map V build-ip-partition-no-empty-elems*
build-ip-partition-same-fw ex-d1 ex-d2 getOneIp-elem wordinterval-element-set-eq)
with *f1 f2* **show** *?thesis*
using *allow* **by** *metis*
qed

hence *ex1*: (*s-repr, d-repr*) \in *set E* **by**(*simp add: E all-pairs-set 1 2*)

thus *?thesis* **using** *ex1 ex-s1 ex-s2 ex-d1 ex-d2* **by** *blast*
qed

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

$$\begin{aligned}
& (\text{map-of } V) \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge d \in \text{wordinterval-to-set } d\text{-range}) \\
& \longleftrightarrow \\
& \text{runFw } s \text{ } d \text{ } c \text{ } rs = \text{Decision } \text{FinalAllow}
\end{aligned}$$

using *matrix access-matrix-sound access-matrix-complete* **by** *blast*

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: $V \in \text{set } (\text{build-ip-partition } c \text{ } rs) \implies \forall ip1 \in \text{wordinterval-to-set } V. \forall ip2 \in \text{wordinterval-to-set } V. \text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$

Minimal: $\llbracket A \in \text{set } (\text{build-ip-partition } c \text{ } rs); B \in \text{set } (\text{build-ip-partition } c \text{ } rs); A \neq B \rrbracket \implies \forall ip1 \in \text{wordinterval-to-set } A. \forall ip2 \in \text{wordinterval-to-set } B. \neg \text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$

The resulting access control matrix is sound and complete:

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

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*

$:: \text{parts-connection} \Rightarrow 32 \text{ simple-rule list} \Rightarrow (\text{string} \times \text{string}) \text{ list} \times (\text{string} \times \text{string}) \text{ list}$

where

$\text{access-matrix-pretty-ipv4 } c \text{ } rs \equiv$

$\text{if } \neg \text{simple-firewall-without-interfaces } rs \text{ then undefined else}$

$(\text{let } (V,E) = (\text{access-matrix } c \text{ } rs);$

$\text{formatted-nodes} =$

$\text{map } (\lambda(v\text{-repr}, v\text{-range}). (\text{ipv4addr-toString } v\text{-repr}, \text{ipv4addr-wordinterval-toString } v\text{-range})) \text{ } V;$

$\text{formatted-edges} =$

$\text{map } (\lambda(s,d). (\text{ipv4addr-toString } s, \text{ipv4addr-toString } d)) \text{ } E$

in

$(\text{formatted-nodes}, \text{formatted-edges})$

$)$

definition *access-matrix-pretty-ipv4-code*
 $\text{:: parts-connection} \Rightarrow 32 \text{ simple-rule list} \Rightarrow (\text{string} \times \text{string}) \text{ list} \times (\text{string} \times \text{string}) \text{ list}$
where
access-matrix-pretty-ipv4-code $c \text{ rs} \equiv$
if $\neg \text{simple-firewall-without-interfaces } rs$ *then undefined else*
 $(\text{let } W = \text{build-ip-partition } c \text{ rs};$
 $R = \text{map } \text{getOneIp } W;$
 $U = \text{all-pairs } R$
in
 $(\text{zip } (\text{map } \text{ipv4addr-toString } R) (\text{map } \text{ipv4addr-wordinterval-toString } W),$
 $\text{map } (\lambda(x,y). (\text{ipv4addr-toString } x, \text{ipv4addr-toString } y)) [(s, d) \leftarrow \text{all-pairs } R].$
 $\text{runFw } s \text{ d } c \text{ rs} = \text{Decision FinalAllow}])$

lemma *access-matrix-pretty-ipv4-code*[code]: *access-matrix-pretty-ipv4* = *access-matrix-pretty-ipv4-code*
by (*simp add: fun-eq-iff access-matrix-pretty-ipv4-def access-matrix-pretty-ipv4-code-def*
Let-def access-matrix-def map-prod-fun-zip)

definition *access-matrix-pretty-ipv6*
 $\text{:: parts-connection} \Rightarrow 128 \text{ simple-rule list} \Rightarrow (\text{string} \times \text{string}) \text{ list} \times (\text{string} \times \text{string}) \text{ list}$
where
access-matrix-pretty-ipv6 $c \text{ rs} \equiv$
if $\neg \text{simple-firewall-without-interfaces } rs$ *then undefined else*
 $(\text{let } (V,E) = (\text{access-matrix } c \text{ rs});$
 $\text{formatted-nodes} =$
 $\text{map } (\lambda(v\text{-repr}, v\text{-range}). (\text{ipv6addr-toString } v\text{-repr}, \text{ipv6addr-wordinterval-toString } v\text{-range})) V;$
 $\text{formatted-edges} =$
 $\text{map } (\lambda(s,d). (\text{ipv6addr-toString } s, \text{ipv6addr-toString } d)) E$
in
 $(\text{formatted-nodes}, \text{formatted-edges})$
 $)$

definition *access-matrix-pretty-ipv6-code*
 $\text{:: parts-connection} \Rightarrow 128 \text{ simple-rule list} \Rightarrow (\text{string} \times \text{string}) \text{ list} \times (\text{string} \times \text{string}) \text{ list}$
where
access-matrix-pretty-ipv6-code $c \text{ rs} \equiv$
if $\neg \text{simple-firewall-without-interfaces } rs$ *then undefined else*
 $(\text{let } W = \text{build-ip-partition } c \text{ rs};$
 $R = \text{map } \text{getOneIp } W;$
 $U = \text{all-pairs } R$
in
 $(\text{zip } (\text{map } \text{ipv6addr-toString } R) (\text{map } \text{ipv6addr-wordinterval-toString } W),$
 $\text{map } (\lambda(x,y). (\text{ipv6addr-toString } x, \text{ipv6addr-toString } y)) [(s, d) \leftarrow \text{all-pairs } R].$
 $\text{runFw } s \text{ d } c \text{ rs} = \text{Decision FinalAllow}])$

lemma *access-matrix-pretty-ipv6-code*[code]: *access-matrix-pretty-ipv6* = *access-matrix-pretty-ipv6-code*
by(*simp add: fun-eq-iff access-matrix-pretty-ipv6-def access-matrix-pretty-ipv6-code-def*
Let-def access-matrix-def map-prod-fun-zip)

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*
by(*simp-all add: mk-parts-connection-TCP-def parts-connection-ssh-def parts-connection-http-def*)

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

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* @ " " @

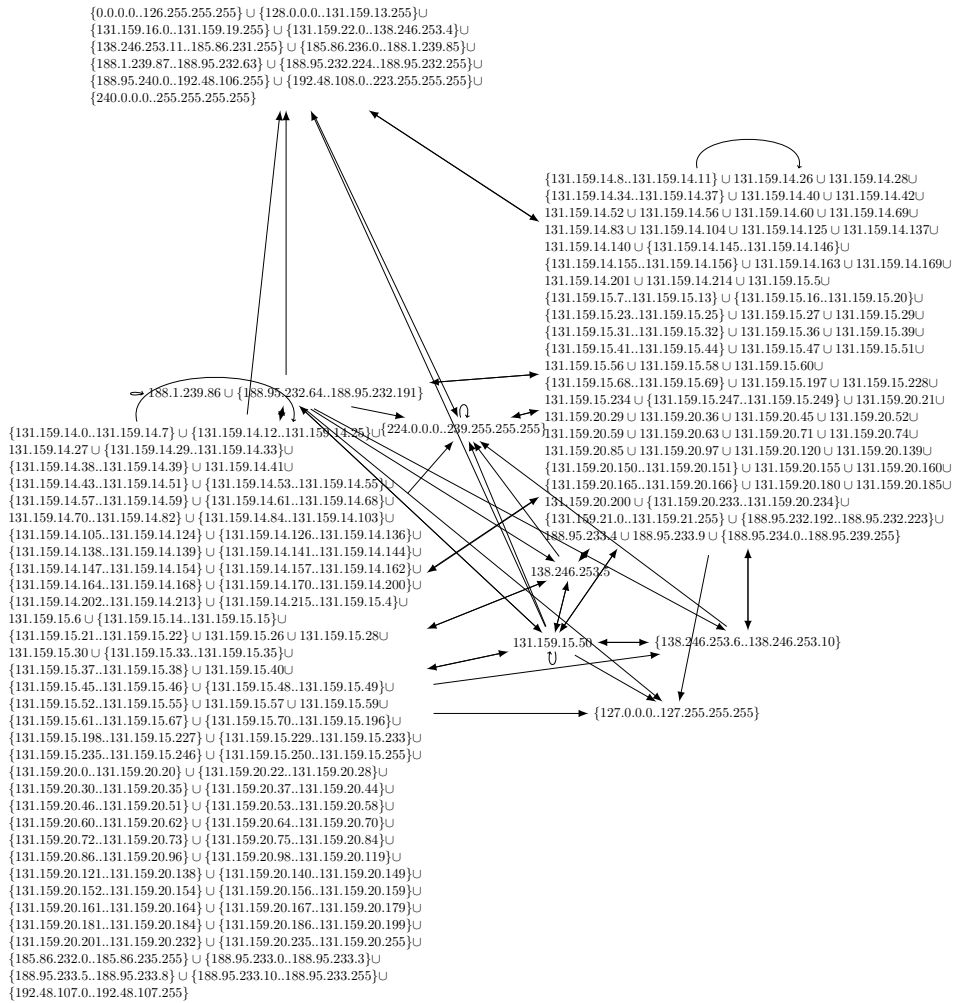


Figure 1: TUM ssh Service Matrix

```

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.