

Simple Firewall

Cornelius Diekmann, Julius Michaelis, Max Haslbeck

September 23, 2023

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	5
3.1	TCP Flags to String	5
4	Simple Packet	6
5	The state of a firewall, abstracted only to the packet filtering outcome	7
6	Network Interfaces	7
6.1	Helpers for the interface name (<i>string</i>)	9
6.2	Matching	10
6.3	Enumerating Interfaces	13
6.4	Negating Interfaces	13

7	Simple Firewall Syntax	14
8	Simple Firewall Semantics	16
8.1	Simple Ports	17
8.2	Simple IPs	18
8.3	Merging Simple Matches	18
8.4	Further Properties of a Simple Firewall	19
8.5	Reality check: Validity of Simple Matches	19
9	List Product Helpers	21
10	Option to List and Option to Set	22
11	Generalize Simple Firewall	22
11.1	Semantics	22
11.2	Lemmas	23
11.3	Equality with the Simple Firewall	23
11.4	Joining two firewalls, i.e. a packet is send through both sequentially.	24
11.5	Validity	26
12	Shadowed Rules	26
12.1	Removing Shadowed Rules	26
12.1.1	Soundness	27
13	Partition a Set by a Specific Constraint	28
14	Group by Function	33
15	Helper: Pretty Printing Word Intervals which correspond to IP address Ranges	35
16	toString Functions for Primitives	36
17	Service Matrices	37
17.1	IP Address Space Partition	37
17.2	Service Matrix over an IP Address Space Partition	49
18	Simple Firewall toString Functions	53

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*
<proof>

lemma *enum-set-get-one-None*: *S* \subseteq *set* *ss* \Longrightarrow *enum-set-get-one* *ss* *S* = *None*
 \longleftrightarrow *S* = {}
<proof>

lemma *enum-set-get-one-Some*: *S* \subseteq *set* *ss* \Longrightarrow *enum-set-get-one* *ss* *S* = *Some* *x*
 \Longrightarrow *x* \in *S*
<proof>

corollary *enum-set-get-one-enum-Some*: *enum-set-get-one* *enum-class.enum* *S* =
Some *x* \Longrightarrow *x* \in *S*
<proof>

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*
<proof>

corollary *enum-set-get-one-enum-Ex-Some*:
S \neq {} \Longrightarrow \exists *x*. *enum-set-get-one* *enum-class.enum* *S* = *Some* *x*
<proof>

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* \Rightarrow *a#enum-set-to-list* (*S* - {*a*}))
<proof>

termination *enum-set-to-list*
<proof>

lemma *enum-set-to-list-simps*: *enum-set-to-list* *S* =
(case *enum-set-get-one* (*Enum.enum*) *S* of *None* \Rightarrow []
| *Some* *a* \Rightarrow *a#enum-set-to-list* (*S* - {*a*}))
<proof>

declare *enum-set-to-list.simps*[*simp del*]

lemma *enum-set-to-list*: *set* (*enum-set-to-list* *A*) = *A*
<proof>

lemma *list-toString bool-toString (enum-set-to-list {True, False}) = "[False, True]"*
 <proof>

end

theory *L4-Protocol*

imports *../Common/Lib-Enum-toString HOL-Library.Word*

begin

2 Transport Layer Protocols

type-synonym *primitive-protocol = 8 word*

definition *ICMP ≡ 1 :: 8 word*

definition *TCP ≡ 6 :: 8 word*

definition *UDP ≡ 17 :: 8 word*

context begin

qualified definition *SCTP ≡ 132 :: 8 word*

qualified definition *IGMP ≡ 2 :: 8 word*

qualified definition *GRE ≡ 47 :: 8 word*

qualified definition *ESP ≡ 50 :: 8 word*

qualified definition *AH ≡ 51 :: 8 word*

qualified definition *IPv6ICMP ≡ 58 :: 8 word*

end

datatype *protocol = ProtoAny | Proto primitive-protocol*

fun *match-protocol :: protocol ⇒ primitive-protocol ⇒ bool where*

match-protocol ProtoAny - ↔ True |

match-protocol (Proto (p)) p-p ↔ p-p = p

fun *simple-protocol-conjunct :: protocol ⇒ protocol ⇒ protocol option where*

simple-protocol-conjunct ProtoAny proto = Some proto |

simple-protocol-conjunct proto ProtoAny = Some proto |

simple-protocol-conjunct (Proto p1) (Proto p2) = (if p1 = p2 then Some (Proto p1) else None)

lemma *simple-protocol-conjunct-asimp[simp]: simple-protocol-conjunct proto ProtoAny = Some proto*

<proof>

lemma *simple-protocol-conjunct-correct: match-protocol p1 pkt ∧ match-protocol p2 pkt ↔*

(case simple-protocol-conjunct p1 p2 of None ⇒ False | Some proto ⇒ match-protocol proto pkt)

<proof>

lemma *simple-protocol-conjunct-Some: simple-protocol-conjunct p1 p2 = Some proto ⇒*

match-proto proto pkt \longleftrightarrow *match-proto p1 pkt* \wedge *match-proto p2 pkt*
 \langle proof \rangle

lemma *simple-proto-conjunct-None*: *simple-proto-conjunct p1 p2 = None* \implies
 \neg (*match-proto p1 pkt* \wedge *match-proto p2 pkt*)
 \langle proof \rangle

lemma *conjunctProtoD*:

simple-proto-conjunct a (Proto b) = Some x \implies *x = Proto b* \wedge (*a = ProtoAny*
 \vee *a = Proto b*)
 \langle proof \rangle

lemma *conjunctProtoD2*:

simple-proto-conjunct (Proto b) a = Some x \implies *x = Proto b* \wedge (*a = ProtoAny*
 \vee *a = Proto b*)
 \langle proof \rangle

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-neq*: *p = Proto pi* \implies $\exists p'. p' \neq pi$ **for** *pi*
 \langle proof \rangle

lemma *protocol-Ex-neq*: $\exists p'. Proto p' \neq p$
 \langle proof \rangle

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*} \langle proof \rangle

instance *tcp-flag* :: *finite*
 \langle proof \rangle

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 \langle proof \rangle

end

3.1 TCP Flags to String

fun *tcp-flag-toString* :: *tcp-flag* \Rightarrow *string* **where**

```

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

```

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

lemma *ipt-tcp-flags-toString* {*TCP-SYN, TCP-SYN, TCP-ACK*} = "[*TCP-SYN, TCP-ACK*]" *<proof>*

end

4 Simple Packet

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

Packet constants are prefixed with *p*

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

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

```

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

```

```

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

```

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

definition *simple-packet-unext* :: (*'i::len, 'a*) *simple-packet-scheme* \Rightarrow *'i simple-packet* **where**

```

simple-packet-unext p  $\equiv$ 
  (p-iface = p-iface 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* ⇒ *iface* ⇒ *bool* **where**
 (*Iface* []) ≤ (*Iface* -) ↔ *True* |
 (*Iface* -) ≤ (*Iface* []) ↔ *False* |
 (*Iface* (a#as)) ≤ (*Iface* (b#bs)) ↔ (if a = b then *Iface* as ≤ *Iface* bs else a ≤ b)
 ⟨*proof*⟩

termination *less-eq* :: *iface* ⇒ - ⇒ *bool*
 ⟨*proof*⟩

lemma *Iface-less-eq-empty*: *Iface* x ≤ *Iface* [] ⇒ x = []
 ⟨*proof*⟩

lemma *less-eq-empty*: *Iface* [] ≤ q
 ⟨*proof*⟩

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

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

termination *less* :: *iface* ⇒ - ⇒ *bool*
 ⟨*proof*⟩

instance
 ⟨*proof*⟩

end

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

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

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

- lo
- lo+
- l+

- +

It is not matched by the following expressions

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

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

context
begin

6.1 Helpers for the interface name (*string*)

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

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

<proof><proof><proof><proof><proof><proof><proof><proof><proof>

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

```

    {(butlast i)@cs | cs. True}
private lemma {(butlast i)@cs | cs. True} = (λs. (butlast i)@s) ‘ (UNIV::string
set) <proof> lemma internal-iface-name-to-set: internal-iface-name-match i p-iface
↔ p-iface ∈ internal-iface-name-to-set i
  <proof> lemma internal-iface-name-to-set2: internal-iface-name-to-set ifce =
{i. internal-iface-name-match ifce i}
  <proof> lemma internal-iface-name-match-refl: internal-iface-name-match i i
  <proof>

```

6.2 Matching

```

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

```

— Examples

```

lemma match-iface (Iface "lo") "lo"
  match-iface (Iface "lo+") "lo"
  match-iface (Iface "l+") "lo"
  match-iface (Iface "+") "lo"
  ¬ match-iface (Iface "lo++") "lo"
  ¬ match-iface (Iface "lo+++") "lo"
  ¬ match-iface (Iface "lo1+") "lo"
  ¬ match-iface (Iface "lo1") "lo"
  match-iface (Iface "+") "eth0"
  match-iface (Iface "+") "eth0"
  match-iface (Iface "eth+") "eth0"
  ¬ match-iface (Iface "lo+") "eth0"
  match-iface (Iface "lo+") "loX"
  ¬ match-iface (Iface "'") "loX"
  <proof>
lemma match-ifaceAny: match-iface ifaceAny i
  <proof>
lemma match-IfaceFalse: ¬(∃ IfaceFalse. (∀ i. ¬ match-iface IfaceFalse i))
  <proof>
lemma match-iface-case-nowildcard: ¬ iface-name-is-wildcard i ⇒ match-iface
(Iface i) p-i ↔ i = p-i
  <proof>
lemma match-iface-case-wildcard-prefix:
  iface-name-is-wildcard i ⇒ match-iface (Iface i) p-i ↔ butlast i = take
(length i - 1) p-i
  <proof>
lemma match-iface-case-wildcard-length: iface-name-is-wildcard i ⇒ match-iface
(Iface i) p-i ⇒ length p-i ≥ (length i - 1)
  <proof>
corollary match-iface-case-wildcard:
  iface-name-is-wildcard i ⇒ match-iface (Iface i) p-i ↔ butlast i = take
(length i - 1) p-i ∧ length p-i ≥ (length i - 1)
  <proof>

```

lemma *match-iface-set*: *match-iface (Iface i) p-iface* \longleftrightarrow *p-iface* \in *internal-iface-name-to-set i*
 <proof> **definition** *internal-iface-name-wildcard-longest* :: *string* \Rightarrow *string* \Rightarrow *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 \leq length i2 then i2 else i1)
 else
 None)
private lemma *internal-iface-name-wildcard-longest "eth+" "eth3+"* = Some "eth3+" <proof> **lemma** *internal-iface-name-wildcard-longest "eth+" "e+"* = Some "eth+" <proof> **lemma** *internal-iface-name-wildcard-longest "eth+" "lo"* = None <proof> **lemma** *internal-iface-name-wildcard-longest-commute*: *iface-name-is-wildcard i1* \implies *iface-name-is-wildcard i2* \implies
internal-iface-name-wildcard-longest i1 i2 = *internal-iface-name-wildcard-longest i2 i1*
 <proof> **lemma** *internal-iface-name-wildcard-longest-refl*: *internal-iface-name-wildcard-longest i i* = Some *i*
 <proof> **lemma** *internal-iface-name-wildcard-longest-correct*:
iface-name-is-wildcard i1 \implies *iface-name-is-wildcard i2* \implies
match-iface (Iface i1) p-i \wedge *match-iface (Iface i2) p-i* \longleftrightarrow
 (case *internal-iface-name-wildcard-longest i1 i2* of None \Rightarrow False | Some *x* \Rightarrow *match-iface (Iface x) p-i*)
 <proof>
fun *iface-conjunct* :: *iface* \Rightarrow *iface* \Rightarrow *iface option* **where**
iface-conjunct (Iface i1) (Iface i2) = (case (*iface-name-is-wildcard i1*, *iface-name-is-wildcard i2*) of
 (True, True) \Rightarrow *map-option Iface (internal-iface-name-wildcard-longest i1 i2)* |
 (True, False) \Rightarrow (if *match-iface (Iface i1) i2* then Some (Iface i2) else None)
 |
 (False, True) \Rightarrow (if *match-iface (Iface i2) i1* then Some (Iface i1) else None)
 |
 (False, False) \Rightarrow (if *i1 = i2* then Some (Iface i1) else None))
lemma *iface-conjunct-Some*: *iface-conjunct i1 i2* = Some *x* \implies
match-iface x p-i \longleftrightarrow *match-iface i1 p-i* \wedge *match-iface i2 p-i*
 <proof>
lemma *iface-conjunct-None*: *iface-conjunct i1 i2* = None \implies \neg (*match-iface i1 p-i* \wedge *match-iface i2 p-i*)
 <proof>
lemma *iface-conjunct*: *match-iface i1 p-i* \wedge *match-iface i2 p-i* \longleftrightarrow
 (case *iface-conjunct i1 i2* of None \Rightarrow False | Some *x* \Rightarrow *match-iface x p-i*)

$\langle \text{proof} \rangle$

lemma *match-iface-refl*: *match-iface* (*Iface* *x*) *x* $\langle \text{proof} \rangle$

lemma *match-iface-eqI*: **assumes** *x* = *Iface* *y* **shows** *match-iface* *x* *y*
 $\langle \text{proof} \rangle$

lemma *iface-conjunct-ifaceAny*: *iface-conjunct* *ifaceAny* *i* = *Some* *i*
 $\langle \text{proof} \rangle$

lemma *iface-conjunct-commute*: *iface-conjunct* *i1* *i2* = *iface-conjunct* *i2* *i1*

$\langle \text{proof} \rangle$ **definition** *internal-iface-name-subset* :: *string* \Rightarrow *string* \Rightarrow *bool* **where**

internal-iface-name-subset *i1* *i2* = (case (*iface-name-is-wildcard* *i1*, *iface-name-is-wildcard*

i2) of

(*True*, *True*) \Rightarrow *length* *i1* \geq *length* *i2* \wedge *take* ((*length* *i2*) - 1) *i1* = *butlast*

i2 |

(*True*, *False*) \Rightarrow *False* |

(*False*, *True*) \Rightarrow *take* (*length* *i2* - 1) *i1* = *butlast* *i2* |

(*False*, *False*) \Rightarrow *i1* = *i2*

)

private lemma *butlast-take-length-helper*:

fixes *x* :: *char* *list*

assumes *a1*: *length* *i2* \leq *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*

$\langle \text{proof} \rangle$ **lemma** *internal-iface-name-subset*: *internal-iface-name-subset* *i1* *i2* \longleftrightarrow

{*i*. *internal-iface-name-match* *i1* *i*} \subseteq {*i*. *internal-iface-name-match* *i2* *i*}

$\langle \text{proof} \rangle$

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*. *match-iface* *i1* *i*} \subseteq {*i*. *match-iface*
i2 *i*}

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

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*)
 ⟨*proof*⟩

we can compute this, but its horribly inefficient!

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

Non-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* = [""]
 ⟨*proof*⟩ **lemma** *non-wildcard-ifaces: set (non-wildcard-ifaces n)* = {*s::string. length s = n* \wedge \neg *iface-name-is-wildcard s*}
 ⟨*proof*⟩ **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*}
 ⟨*proof*⟩

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*}
 ⟨*proof*⟩

6.4 Negating Interfaces

private lemma *inv-iface-name-set: - (internal-iface-name-to-set i)* = (
if *iface-name-is-wildcard i*
then
 {*c | c. length c < length (butlast i)*} \cup {*c @ cs | c cs. length c = length (butlast i) \wedge c \neq butlast i*}
else
 {*c | c. length c < length i*} \cup {*c@cs | c cs. length c \geq length i \wedge c \neq i*}
)
 ⟨*proof*⟩

Negating is really not intuitive. The Interface "*et*" is in the negated set of "*eth+*". And the Interface "*et+*" is also in this set! This is because "+" is a normal interface character and not a wildcard here! In contrast, the set

described by `"et+"` (with `"+"` a wildcard) is not a subset of the previously negated set.

lemma `"et" ∈ - (internal-iface-name-to-set "eth+")` *<proof>*

lemma `"et+" ∈ - (internal-iface-name-to-set "eth+")` *<proof>*

lemma `"+" ∈ - (internal-iface-name-to-set "eth+")` *<proof>*

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

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

lemma `"+" ∈ - (internal-iface-name-to-set "eth+")` *<proof>*

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

lemma `compress-pos-interfaces-Some: compress-pos-interfaces ifces = Some ifce`
 \implies
`match-iface ifce p-i ⟷ (∀ i ∈ set ifces. match-iface i p-i)`
<proof>

lemma `compress-pos-interfaces-None: compress-pos-interfaces ifces = None` \implies
 $\neg (\forall i \in \text{set } ifces. \text{match-iface } i \text{ p-}i)$
<proof>

end

end

7 Simple Firewall Syntax

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

begin

For for IP addresses of arbitrary length

datatype `simple-action = Accept | Drop`

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

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

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

But negation of interfaces is not supported. Since interfaces support a 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 =
  iface :: 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*
 \langle *proof* \rangle

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-ipcidr-tuple-to-wordinterval-simple-match-ip-set*:
wordinterval-to-set (ipcidr-tuple-to-wordinterval ip) = {*d. simple-match-ip ip d*}
 \langle *proof* \rangle

lemma {(253::8 *word*) .. 8} = {} \langle *proof* \rangle

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 (iface m) (p-iface 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* |

$simple\text{-}fw ((SimpleRule\ m\ Accept)\#rs)\ p = (if\ simple\text{-}matches\ m\ p\ then\ Decision\ FinalAllow\ else\ simple\text{-}fw\ rs\ p) \mid$

$simple\text{-}fw ((SimpleRule\ m\ Drop)\#rs)\ p = (if\ simple\text{-}matches\ m\ p\ then\ Decision\ FinalDeny\ else\ simple\text{-}fw\ rs\ p)$

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 FinalDeny) else *simple-fw-alt* rs p)

lemma *simple-fw-alt*: *simple-fw* r p = *simple-fw-alt* r p <proof>

definition *simple-match-any* :: 'i::len *simple-match* **where**

simple-match-any \equiv (iiface=iifaceAny, oiface=iifaceAny, src=(0,0), dst=(0,0), proto=ProtoAny, sports=(0,65535), dports=(0,65535) |)

lemma *simple-match-any*: *simple-matches* *simple-match-any* p

<proof>

we specify only one empty port range

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

simple-match-none \equiv

(iiface=iifaceAny, oiface=iifaceAny, 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>

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>

lemma *nomatch*: \neg *simple-matches* m p \implies *simple-fw* (SimpleRule m a # rs) p

= *simple-fw* rs p

<proof>

8.1 Simple Ports

fun *simpl-ports-conjunct* :: (16 word \times 16 word) \Rightarrow (16 word \times 16 word) \Rightarrow (16 word \times 16 word) **where**

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

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

<proof>

lemma *simple-ports-conjunct-correct*:
 $simple-match-port\ p1\ pkt \wedge simple-match-port\ p2\ pkt \longleftrightarrow simple-match-port$
 $(simple-ports-conjunct\ p1\ p2)\ pkt$
 $\langle proof \rangle$

lemma *simple-match-port-code*[code] : $simple-match-port\ (s,e)\ p-p = (s \leq p-p \wedge$
 $p-p \leq e) \langle proof \rangle$

lemma *simple-match-port-UNIV*: $\{p.\ simple-match-port\ (s,e)\ p\} = UNIV \longleftrightarrow$
 $(s = 0 \wedge e = -1)$
 $\langle proof \rangle$

8.2 Simple IPs

lemma *simple-match-ip-conjunct*:
fixes $ip1 :: 'i::len\ word \times nat$
shows $simple-match-ip\ ip1\ p-ip \wedge simple-match-ip\ ip2\ p-ip \longleftrightarrow$
 $(case\ ipcidr-conjunct\ ip1\ ip2\ of\ None \Rightarrow False \mid Some\ ipx \Rightarrow simple-match-ip$
 $ipx\ p-ip)$
 $\langle proof \rangle$

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\) =$
 $(case\ ipcidr-conjunct\ sip1\ sip2\ of\ None \Rightarrow None \mid Some\ sip \Rightarrow$
 $(case\ ipcidr-conjunct\ dip1\ dip2\ of\ None \Rightarrow None \mid Some\ dip \Rightarrow$
 $(case\ iface-conjunct\ iif1\ iif2\ of\ None \Rightarrow None \mid Some\ iif \Rightarrow$
 $(case\ iface-conjunct\ oif1\ oif2\ of\ None \Rightarrow None \mid Some\ oif \Rightarrow$
 $(case\ simple-proto-conjunct\ p1\ p2\ of\ None \Rightarrow None \mid Some\ p \Rightarrow$
 $Some\ (\iiface=iif,\ oiface=oif,\ src=sip,\ dst=dip,\ proto=p,$
 $sports=simpl-ports-conjunct\ sps1\ sps2,\ dports=simpl-ports-conjunct\ dps1$
 $dps2\)))$

lemma *simple-match-and-correct*: $simple-matches\ m1\ p \wedge simple-matches\ m2\ p$
 \longleftrightarrow
 $(case\ simple-match-and\ m1\ m2\ of\ None \Rightarrow False \mid Some\ m \Rightarrow simple-matches$
 $m\ p)$
 $\langle proof \rangle$

lemma *simple-match-and-SomeD*: $simple-match-and\ m1\ m2 = Some\ m \Longrightarrow$

simple-matches $m\ p \longleftrightarrow (\text{simple-matches } m1\ p \wedge \text{simple-matches } m2\ p)$
 ⟨proof⟩

lemma *simple-match-and-NoneD*: *simple-match-and* $m1\ m2 = \text{None} \implies$
 $\neg(\text{simple-matches } m1\ p \wedge \text{simple-matches } m2\ p)$
 ⟨proof⟩

lemma *simple-matches-andD*: *simple-matches* $m1\ p \implies \text{simple-matches } m2\ p$
 \implies
 $\exists m. \text{simple-match-and } m1\ m2 = \text{Some } m \wedge \text{simple-matches } m\ p$
 ⟨proof⟩

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* $m\ -$)] = ($m = \text{simple-match-any}$) |
has-default-policy ($-\#rs$) = *has-default-policy* rs

lemma *has-default-policy*: *has-default-policy* $rs \implies$
 $\text{simple-fw } rs\ p = \text{Decision } \text{FinalAllow} \vee \text{simple-fw } rs\ p = \text{Decision } \text{FinalDeny}$
 ⟨proof⟩

lemma *has-default-policy-fst*: *has-default-policy* $rs \implies \text{has-default-policy } (r\#rs)$
 ⟨proof⟩

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

fun *cut-off-after-match-any* :: 'i::len *simple-rule list* \Rightarrow 'i *simple-rule list* **where**
cut-off-after-match-any [] = [] |
cut-off-after-match-any (*SimpleRule* $m\ a\ \#rs$) =
 (*if* $m = \text{simple-match-any}$ *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) $p = \text{simple-fw } rs\ p$
 ⟨proof⟩

lemma *simple-fw-not-matches-removeAll*: $\neg \text{simple-matches } m\ p \implies$
 $\text{simple-fw } (\text{removeAll } (\text{SimpleRule } m\ a)\ rs)\ p = \text{simple-fw } rs\ p$
 ⟨proof⟩

8.5 Reality check: Validity of Simple Matches

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

definition *valid-prefix-fw* $m = \text{valid-prefix } (\text{uncurry } \text{PrefixMatch } m)$

lemma *ipcidr-conjunct-valid*:
 $\llbracket \text{valid-prefix-fw } p1; \text{valid-prefix-fw } p2; \text{ipcidr-conjunct } p1 \ p2 = \text{Some } p \rrbracket \implies$
 $\text{valid-prefix-fw } p$
 $\langle \text{proof} \rangle$

definition *simple-match-valid* :: ('i::len, 'a) *simple-match-scheme* \implies bool **where**
simple-match-valid $m \equiv$
 $(\{p. \text{simple-match-port } (\text{sports } m) \ p\} \neq \text{UNIV} \vee \{p. \text{simple-match-port } (\text{dports}$
 $m) \ p\} \neq \text{UNIV} \longrightarrow$
 $\text{proto } m \in \text{Proto } \{TCP, UDP, L4-Protocol.SCTP\}) \wedge$
 $\text{valid-prefix-fw } (\text{src } m) \wedge \text{valid-prefix-fw } (\text{dst } m)$

lemma *simple-match-valid-alt*[code-unfold]: *simple-match-valid* = $(\lambda m.$
 $(\text{let } c = (\lambda(s,e). (s \neq 0 \vee e \neq -1)) \text{ in } ($
 $\text{if } c (\text{sports } m) \vee c (\text{dports } m) \text{ then } \text{proto } m = \text{Proto } TCP \vee \text{proto } m = \text{Proto}$
 $UDP \vee \text{proto } m = \text{Proto } L4-Protocol.SCTP \text{ else True})) \wedge$
 $\text{valid-prefix-fw } (\text{src } m) \wedge \text{valid-prefix-fw } (\text{dst } m))$
 $\langle \text{proof} \rangle$

Example:

context
begin
private definition *example-simple-match1* \equiv
 $(\llbracket \text{iiface} = \text{Iface } "+", \text{oiface} = \text{Iface } "+", \text{src} = (0::32 \text{ word}, 0), \text{dst} = (0, 0),$
 $\text{proto} = \text{Proto } TCP, \text{sports} = (0, 1024), \text{dports} = (0, 1024) \rrbracket)$

lemma *simple-fw* [SimpleRule *example-simple-match1 Drop*]
 $(\llbracket p\text{-iiface} = "", p\text{-oiface} = "", p\text{-src} = (1::32 \text{ word}), p\text{-dst} = 2, p\text{-proto} =$
 $TCP, p\text{-sport} = 8,$
 $p\text{-dport} = 9, p\text{-tcp-flags} = \{\}, p\text{-payload} = "" \rrbracket) =$
 $\text{Decision } \text{FinalDeny } \langle \text{proof} \rangle$ **definition** *example-simple-match2* \equiv *exam-*
ple-simple-match1 $(\llbracket \text{proto} := \text{ProtoAny} \rrbracket)$

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* $\langle \text{proof} \rangle$ **lemma** \neg
simple-match-valid example-simple-match2 $\langle \text{proof} \rangle$
end

lemma *simple-match-and-valid*:
fixes $m1 :: 'i::len \text{ simple-match}$
assumes $mv: \text{simple-match-valid } m1 \text{ simple-match-valid } m2$
assumes $mj: \text{simple-match-and } m1 \ m2 = \text{Some } m$
shows *simple-match-valid* m
 $\langle \text{proof} \rangle$

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

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 = iiface,
   oiface = oiface,
   p-src = s, dst = d,
   p-prot = prot,
   p-sport = sport, p-dport = dport,
   tcp-flags = tcp-flags, p-payload = payload1)
 $\longleftrightarrow$ 
simple-matches m
  (p-iiface = iiface,
   oiface = oiface,
   p-src = s, p-dst = d,
   p-prot = prot,
   p-sport = sport, p-dport = dport,
   p-tcp-flags = tcp-flags2, p-payload = payload2, ... = aux)

```

<proof>

end

9 List Product Helpers

theory *List-Product-More*

imports *Main*

begin

lemma *List-product-concat-map*: $List.product\ xs\ ys = concat\ (map\ (\lambda x. map\ (\lambda y. (x,y))\ ys)\ xs)$

<proof>

definition *all-pairs* :: $'a\ list \Rightarrow ('a \times 'a)\ list$ **where**

$all-pairs\ xs \equiv concat\ (map\ (\lambda x. map\ (\lambda y. (x,y))\ xs)\ xs)$

lemma *all-pairs-list-product*: $all-pairs\ xs = List.product\ xs\ xs$

<proof>

lemma *all-pairs*: $\forall (x,y) \in (set\ xs \times set\ xs). (x,y) \in set\ (all-pairs\ xs)$

<proof>

lemma *all-pairs-set*: $set\ (all-pairs\ xs) = set\ xs \times set\ xs$

<proof>

end

10 Option to List and Option to Set

```
theory Option-Helpers
imports Main
begin
```

Those are just syntactic helpers.

```
definition option2set :: 'a option  $\Rightarrow$  '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
  <proof>
```

```
lemma option2list-simps[simp]: option2list (Some x) = [x] option2list (None) = []
  <proof>
```

```
lemma option2set-None: option2set None = {}
  <proof>
```

```
lemma option2list-map: option2list (map-option f n) = map f (option2list n)
  <proof>
```

```
lemma option2set-map: option2set (map-option f n) = f ` option2set n
  <proof>
```

```
end
```

11 Generalize Simple Firewall

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

11.1 Semantics

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

```
definition generalized-sfw
  :: ('i::len simple-match  $\times$  '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,-) ⇒ simple-matches m p) then Some a else generalized-sfw as p)

<proof>

lemma *generalized-sfw-append:*

*generalized-sfw (a @ b) p = (case generalized-sfw a p of Some x ⇒ Some x
| None ⇒ generalized-sfw b p)*

<proof>

lemma *simple-generalized-undecided:*

simple-fw fw p ≠ Undecided ⇒ generalized-sfw (map simple-rule-dtor fw) p ≠ None

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *generalized-sfw-filterD:*

generalized-sfw (filter f fw) p = Some (r,d) ⇒ simple-matches r p ∧ f (r,d)

<proof>

lemma *generalized-sfw-apsnd:*

generalized-sfw (map (apsnd f) fw) p = map-option (apsnd f) (generalized-sfw fw p)

<proof>

11.3 Equality with the Simple Firewall

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

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

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

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

lemma *simple-fw-iff-generalized-fw*:
 $simple\text{-}fw\ fw\ p = simple\text{-}action\text{-}to\text{-}decision\ a \longleftrightarrow (\exists r. generalized\text{-}sfw\ (map\ simple\text{-}rule\text{-}dtdor\ fw)\ p = Some\ (r,a))$
 ⟨proof⟩

lemma *simple-fw-iff-generalized-fw-accept*:
 $simple\text{-}fw\ fw\ p = Decision\ FinalAllow \longleftrightarrow (\exists r. generalized\text{-}sfw\ (map\ simple\text{-}rule\text{-}dtdor\ fw)\ p = Some\ (r,\ Accept))$
 ⟨proof⟩

lemma *simple-fw-iff-generalized-fw-drop*:
 $simple\text{-}fw\ fw\ p = Decision\ FinalDeny \longleftrightarrow (\exists r. generalized\text{-}sfw\ (map\ simple\text{-}rule\text{-}dtdor\ fw)\ p = Some\ (r,\ Drop))$
 ⟨proof⟩

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

definition *generalized-fw-join*
 $:: ('i::len\ simple\text{-}match \times 'a)\ list \Rightarrow ('i\ simple\text{-}match \times 'b)\ list \Rightarrow ('i\ simple\text{-}match \times 'a \times 'b)\ list$

where
 $generalized\text{-}fw\text{-}join\ l1\ l2 \equiv [(u,(a,b)). (m1,a) \leftarrow l1, (m2,b) \leftarrow l2, u \leftarrow option2list\ (simple\text{-}match\text{-}and\ m1\ m2)]$

lemma *generalized-fw-join-1-Nil[simp]*: $generalized\text{-}fw\text{-}join\ []\ f2 = []$
 ⟨proof⟩

lemma *generalized-fw-join-2-Nil[simp]*: $generalized\text{-}fw\text{-}join\ f1\ [] = []$
 ⟨proof⟩

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

lemma *generalized-fw-join-1-nomatch*:
 $\neg simple\text{-}matches\ am\ p \Longrightarrow generalized\text{-}sfw\ [(u,(ad,b)). (m2,b) \leftarrow l2, u \leftarrow option2list\ (simple\text{-}match\text{-}and\ am\ m2)]\ p = None$
 ⟨proof⟩

lemma *generalized-fw-join-2-nomatch*:
 $\neg simple\text{-}matches\ bm\ p \Longrightarrow generalized\text{-}sfw\ (generalized\text{-}fw\text{-}join\ as\ ((bm,\ bd) \# bs))\ p = generalized\text{-}sfw\ (generalized\text{-}fw\text{-}join\ as\ bs)\ p$
 ⟨proof⟩

lemma *generalized-fw-joinI*:

$$\llbracket \text{generalized-sfw } f1 \text{ } p = \text{Some } (r1, d1); \text{ generalized-sfw } f2 \text{ } p = \text{Some } (r2, d2) \rrbracket$$

$$\implies$$

$$\text{generalized-sfw } (\text{generalized-fw-join } f1 \text{ } f2) \text{ } p = \text{Some } (\text{the } (\text{simple-match-and } r1 \text{ } r2), d1, d2)$$

$$\langle \text{proof} \rangle$$

lemma *generalized-fw-joinD*:

$$\text{generalized-sfw } (\text{generalized-fw-join } f1 \text{ } f2) \text{ } p = \text{Some } (u, d1, d2) \implies$$

$$\exists r1 \text{ } r2. \text{ generalized-sfw } f1 \text{ } p = \text{Some } (r1, d1) \wedge \text{ generalized-sfw } f2 \text{ } p = \text{Some } (r2, d2) \wedge \text{Some } u = \text{simple-match-and } r1 \text{ } r2$$

$$\langle \text{proof} \rangle$$

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

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

shows

$$\text{simple-fw } rs1 \text{ } p = \text{Decision } \text{FinalAllow} \wedge \text{simple-fw } rs2 \text{ } p = \text{Decision } \text{FinalAllow}$$

$$\longleftrightarrow$$

$$\text{simple-fw } (\text{rule-translate } (\text{generalized-fw-join } (\text{map } \text{simple-rule-dtor } rs1) (\text{map } \text{simple-rule-dtor } rs2))) \text{ } p = \text{Decision } \text{FinalAllow}$$

$$\langle \text{proof} \rangle$$

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
$$\text{map } (\text{apsnd } (\lambda(a, b) \Rightarrow (\text{case } a \text{ of } \text{Accept} \Rightarrow b \mid \text{Drop} \Rightarrow \text{Drop})))$$

shows $\text{simple-fw } rs1 \text{ } p = \text{Decision } \text{FinalAllow} \wedge \text{simple-fw } rs2 \text{ } p = \text{Decision } \text{FinalAllow} \longleftrightarrow$

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

$$\langle \text{proof} \rangle$$

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

$$\text{generalized-fw-join } [(m1, d1)] \text{ } fw2 = \text{foldr } (\lambda(m2, d2). (@) (\text{case } \text{simple-match-and } m1 \text{ } m2 \text{ of } \text{None} \Rightarrow [] \mid \text{Some } mu \Rightarrow [(mu, d1, d2)])) \text{ } fw2 \text{ } []$$

$$\langle \text{proof} \rangle$$

lemma *generalized-sfw-2-join-None*:
 $generalized-sfw\ fw2\ p = None \implies generalized-sfw\ (generalized-fw-join\ fw1\ fw2)$
 $p = None$
 $\langle proof \rangle$

lemma *generalized-sfw-1-join-None*:
 $generalized-sfw\ fw1\ p = None \implies generalized-sfw\ (generalized-fw-join\ fw1\ fw2)$
 $p = None$
 $\langle proof \rangle$

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)
 $\langle proof \rangle$

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*: $gsfw-valid\ f1 \implies gsfw-valid\ f2 \implies gsfw-valid\ (generalized-fw-join\ f1\ f2)$
 $\langle proof \rangle$

lemma *gsfw-validI*: $simple-fw-valid\ fw \implies gsfw-valid\ (map\ simple-rule-dtor\ fw)$
 $\langle proof \rangle$

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 ∈ P. ¬ simple-matches m p}))

```

12.1.1 Soundness

lemma *rmshadow-sound*:

```

  p ∈ P ⇒ simple-fw (rmshadow rs P) p = simple-fw rs p
⟨proof⟩

```

corollary *rmshadow*:

```

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

```

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

fun *rmshadow'* :: 'i::len simple-rule list ⇒ 'i simple-packet set ⇒ 'i simple-rule list
where

```

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

```

lemma *rmshadow'-sound*:

```

  p ∉ P ⇒ simple-fw (rmshadow' rs P) p = simple-fw rs p
⟨proof⟩

```

corollary

```

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

```

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

context

begin

```

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

```

private definition *simple-packet-set-toSet* :: 'i::len simple-packet-set ⇒ 'i simple-packet set **where**

```

  simple-packet-set-toSet ms = {p. ∃ m ∈ set ms. simple-matches m p}

```

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

private lemma $simple\text{-}packet\text{-}set\text{-}toSet\ (simple\text{-}packet\text{-}set\text{-}union\ ps\ m) = simple\text{-}packet\text{-}set\text{-}toSet\ ps \cup \{p. simple\text{-}matches\ m\ p\}$
 ⟨proof⟩ **lemma** $(\exists m' \in set\ ms.$
 $\{i. match\text{-}iface\ iif\ i\} \subseteq \{i. match\text{-}iface\ (iiface\ m')\ i\} \wedge$
 $\{i. match\text{-}iface\ oif\ i\} \subseteq \{i. match\text{-}iface\ (oiface\ m')\ i\} \wedge$
 $\{ip. simple\text{-}match\text{-}ip\ sip\ ip\} \subseteq \{ip. simple\text{-}match\text{-}ip\ (src\ m')\ ip\} \wedge$
 $\{ip. simple\text{-}match\text{-}ip\ dip\ ip\} \subseteq \{ip. simple\text{-}match\text{-}ip\ (dst\ m')\ ip\} \wedge$
 $\{p. match\text{-}proto\ protocol\ p\} \subseteq \{p. match\text{-}proto\ (proto\ m')\ p\} \wedge$
 $\{p. simple\text{-}match\text{-}port\ sps\ p\} \subseteq \{p. simple\text{-}match\text{-}port\ (sports\ m')\ p\} \wedge$
 $\{p. simple\text{-}match\text{-}port\ dps\ p\} \subseteq \{p. simple\text{-}match\text{-}port\ (dports\ m')\ p\}$
 $)$
 $\Rightarrow \{p. simple\text{-}matches\ (iiface=iif, oiface=oif, src=sip, dst=dip, proto=protocol,$
 $sports=sps, dports=dps)\ p\} \subseteq (simple\text{-}packet\text{-}set\text{-}toSet\ ms)$
 ⟨proof⟩

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

private lemma $\{p:: 'i::len\ simple\text{-}packet. simple\text{-}matches\ m\ p\} \subseteq (simple\text{-}packet\text{-}set\text{-}toSet\ ms) \iff$
 $\{p:: 'i::len\ simple\text{-}packet. simple\text{-}matches\ m\ p\} \cap (\bigcap m \in set\ ms. \{p. \neg$
 $simple\text{-}matches\ m\ p\}) = \{\}$ (**is** $?l \iff ?r$)
 ⟨proof⟩

end
end

13 Partition a Set by a Specific Constraint

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

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

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

definition *disjoint* :: $'a\ set\ set \Rightarrow bool$ **where**
 $disjoint\ ts \equiv \forall A \in ts. \forall B \in ts. A \neq B \longrightarrow A \cap B = \{\}$ *We will call two partitioned sets complete iff $\bigcup ss = \bigcup ts$.*

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

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

context begin

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

private lemma $disjoint-equi: disjoint-list-rec ts \Longrightarrow disjoint (set ts)$
 $\langle proof \rangle$ **lemma** $disjoint-list-disjoint-list-rec: disjoint-list ts \Longrightarrow disjoint-list-rec ts$

$\langle proof \rangle$ **definition** $addSubsetSet :: 'a \text{ set} \Rightarrow 'a \text{ set set} \Rightarrow 'a \text{ set set}$ **where**
 $addSubsetSet s ts = insert (s - \bigcup ts) (((\cap) s) ' ts) \cup ((\lambda x. x - s) ' ts)$

private fun $partitioning :: 'a \text{ set list} \Rightarrow 'a \text{ set set} \Rightarrow 'a \text{ set set}$ **where**
 $partitioning [] ts = ts \mid$
 $partitioning (s\#ss) ts = partitioning ss (addSubsetSet s ts)$

simple examples

lemma $partitioning [\{1::nat,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}] \{\} = \{\{10\}, \{6\}, \{5,7\}, \{\}, \{3,4\}, \{1,2\}\}$ $\langle proof \rangle$

lemma $\bigcup \{\{1::nat,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}\} = \bigcup (partitioning [\{1,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}]) \{\}$ $\langle proof \rangle$

lemma $disjoint (partitioning [\{1::nat,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}]) \{\}$ $\langle proof \rangle$

lemma $ipPartition \{\{1::nat,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}\} (partitioning [\{1::nat,2\},\{3,4\},\{5,6,7\},\{6\},\{10\}]) \{\}$ $\langle proof \rangle$

lemma $ipPartition A \{\}$ $\langle proof \rangle$

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

$\langle proof \rangle$ **lemma** $disjointAddSubset: disjoint ts \Longrightarrow disjoint (addSubsetSet a ts)$

$\langle proof \rangle$ **lemma** $coversallAddSubset: \bigcup (insert a ts) = \bigcup (addSubsetSet a ts)$

$\langle proof \rangle$ **lemma** $ipPartitioningAddSubset0: disjoint ts \Longrightarrow ipPartition ts (addSubsetSet a ts)$

$\langle proof \rangle$ **lemma** $ipPartitioningAddSubset1: disjoint ts \Longrightarrow ipPartition (insert a ts) (addSubsetSet a ts)$

$\langle proof \rangle$ **lemma** $addSubsetSetI:$

$s - \bigcup ts \in addSubsetSet s ts$

$t \in ts \Longrightarrow s \cap t \in addSubsetSet s ts$

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

private fun *intersection-list-opt* :: 'a set \Rightarrow 'a set list \Rightarrow 'a set list **where**
 $\text{intersection-list-opt } - [] = [] \mid$
 $\text{intersection-list-opt } s \ (t\#ts) = (s \cap t)\#(\text{intersection-list-opt } (s - t) \ ts)$

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

private fun *difference-list-opt* :: 'a set \Rightarrow 'a set list \Rightarrow 'a set list **where**
 $\text{difference-list-opt } - [] = [] \mid$
 $\text{difference-list-opt } s \ (t\#ts) = (t - s)\#(\text{difference-list-opt } (s - t) \ ts)$

private lemma *difList-equi*: $\text{disjoint-list-rec } ts \implies \text{difference-list } s \ ts = \text{difference-list-opt } s \ ts$

ence-list-opt s ts

$\langle \text{proof} \rangle$ **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*: $\text{set}(\text{partList0 } s \text{ ts}) = (((\cap) s) \text{ ` } (\text{set } ts)) \cup ((\lambda x. x - s) \text{ ` } (\text{set } ts))$

$\langle \text{proof} \rangle$ **lemma** *partList-sub-equi0*: $\text{set}(\text{partList0 } s \text{ ts}) = \text{set}(\text{difference-list } s \text{ ts}) \cup \text{set}(\text{intersection-list } s \text{ ts})$

$\langle \text{proof} \rangle$ **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*: $\text{set}(\text{partList1 } s \text{ ts}) = \text{set}(\text{difference-list-opt } s \text{ ts}) \cup \text{set}(\text{intersection-list-opt } s \text{ ts})$
 $\langle \text{proof} \rangle$ **lemma** *partList0-partList1-equi*: $\text{disjoint-list-rec } ts \Longrightarrow \text{set}(\text{partList0 } s \text{ ts}) = \text{set}(\text{partList1 } s \text{ ts})$

$\langle \text{proof} \rangle$ **fun** *partList2* :: 'a set \Rightarrow 'a set list \Rightarrow '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*: $\text{partList2 } \{\} \text{ ts} = \text{ts}$
 $\langle \text{proof} \rangle$ **lemma** *partList1-partList2-equi*: $\text{set}(\text{partList1 } s \text{ ts}) - \{\{\}\} = \text{set}(\text{partList2 } s \text{ ts}) - \{\{\}\}$

$\langle \text{proof} \rangle$ **fun** *partList3* :: 'a set \Rightarrow 'a set list \Rightarrow '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*: $\text{set}(\text{partList2 } s \text{ ts}) - \{\{\}\} = \text{set}(\text{partList3 } s \text{ ts}) - \{\{\}\}$
 $\langle \text{proof} \rangle$

fun *partList4* :: 'a set \Rightarrow 'a set list \Rightarrow '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*: $\text{partList4 } s \text{ ts} = \text{partList3 } s \text{ ts}$
 $\langle \text{proof} \rangle$ **lemma** *partList0-addSubsetSet-equi*: $s \subseteq \bigcup (\text{set } ts) \Longrightarrow \text{addSubsetSet } s (\text{set } ts) - \{\{\}\} = \text{set}(\text{partList0 } s \text{ ts}) - \{\{\}\}$

\langle proof \rangle **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) = partitioning ss (addSubsetSet a ts)*

\langle proof \rangle **lemma** *partitioning-nottail-equi*: *partitioning-nontail ss ts = partitioning ss ts*

\langle proof \rangle

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*: $\{\} \notin \text{set } ts \Longrightarrow \{\} \notin \text{set } (\text{partList4 } s \text{ } ts)$

\langle proof \rangle

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

\langle proof \rangle

lemma *partitioning1-empty1*: $\{\} \notin \text{set } ts \Longrightarrow$

$\text{set}(\text{partitioning1 } ss \text{ } ts) - \{\{\}\} = \text{set}(\text{partitioning1 } ss \text{ } ts)$

\langle proof \rangle

lemma *partList4-subset*: $a \subseteq \bigcup (\text{set } ts) \Longrightarrow a \subseteq \bigcup (\text{set } (\text{partList4 } b \text{ } ts))$

\langle proof \rangle **lemma** $a \neq \{\} \Longrightarrow \text{disjoint-list-rec } (a \# ts) \longleftrightarrow \text{disjoint-list-rec } ts \wedge a \cap \bigcup (\text{set } ts) = \{\}$ \langle proof \rangle

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

\langle proof \rangle **lemma** *partList4-disjoint*: $s \subseteq \bigcup (\text{set } ts) \Longrightarrow \text{disjoint-list-rec } ts \Longrightarrow$

$\text{disjoint-list-rec } (\text{partList4 } s \text{ } ts)$

\langle proof \rangle

lemma *union-set-partList4*: $\bigcup (\text{set } (\text{partList4 } s \text{ } ts)) = \bigcup (\text{set } ts)$

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

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

\langle proof \rangle **lemma** *partList4-distinct*: $\{\} \notin \text{set } ts \Longrightarrow \text{disjoint-list } ts \Longrightarrow \text{distinct } (\text{partList4 } s \text{ } ts)$

\langle proof \rangle

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

shows *disjoint-list* (*partList4 s ts*)

\langle proof \rangle

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

\langle proof \rangle

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

lemma *ipPartitioning-helper-opt*: $\{\} \notin \text{set } ts \implies \text{disjoint-list } ts \implies \bigcup (\text{set } ss) \subseteq \bigcup (\text{set } ts)$
 $\implies \text{ipPartition } (\text{set } ss) \ (\text{set } (\text{partitioning1 } ss \ ts))$
 <proof>

lemma *complete-helper*: $\{\} \notin \text{set } ts \implies \bigcup (\text{set } ss) \subseteq \bigcup (\text{set } ts) \implies$
 $\bigcup (\text{set } ts) = \bigcup (\text{set } (\text{partitioning1 } ss \ ts))$
 <proof>

lemma *partitioning1* $[\{1::\text{nat}\},\{2\},\{\}] [\{1\},\{\},\{2\},\{3\}] = [\{1\}, \{\}, \{2\}, \{3\}]$
 <proof>

lemma *partitioning-foldr*: $\text{partitioning } X \ B = \text{foldr } \text{addSubsetSet } X \ B$
 <proof>

lemma *ipPartition* $(\text{set } X) \ (\text{foldr } \text{addSubsetSet } X \ \{\})$
 <proof>

lemma $\bigcup (\text{set } X) = \bigcup (\text{foldr } \text{addSubsetSet } X \ \{\})$
 <proof>

lemma *partitioning1* $X \ B = \text{foldr } \text{partList4 } X \ B$
 <proof>

lemma *ipPartition* $(\text{set } X) \ (\text{set } (\text{partitioning1 } X \ [\text{UNIV}]))$
 <proof>

lemma $(\bigcup (\text{set } (\text{partitioning1 } X \ [\text{UNIV}]))) = \text{UNIV}$
 <proof>

end
end

14 Group by Function

theory *GroupF*
imports *Main*
begin

Grouping elements of a list according to a function.

```
fun groupF :: ('a ⇒ '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> **lemma**

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

<proof> **function** groupF-code :: ('a ⇒ '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))
```

<proof> **termination** groupF-code

<proof>

```
lemma groupF-code[code]: groupF f as = groupF-code f as
```

<proof>

export-code groupF **checking** SML

end

```
lemma groupF-concat-set: set (concat (groupF f xs)) = set xs
```

<proof>

```
lemma groupF-Union-set: (⋃ x ∈ set (groupF f xs). set x) = set xs
```

<proof>

```
lemma groupF-set: ∀ X ∈ set (groupF f xs). ∀ x ∈ set X. x ∈ set xs
```

<proof>

lemma groupF-equality:

```
defines same f A ≡ ∀ a1 ∈ set A. ∀ a2 ∈ set A. f a1 = f a2
```

shows $\forall A \in \text{set } (\text{groupF } f \text{ } xs). \text{ same } f \text{ } A$
 <proof>

lemma *groupF-inequality*: $A \in \text{set } (\text{groupF } f \text{ } xs) \implies B \in \text{set } (\text{groupF } f \text{ } xs) \implies A \neq B \implies$
 $\forall a \in \text{set } A. \forall b \in \text{set } B. f \text{ } a \neq f \text{ } b$
 <proof>

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

lemma *groupF-empty*: $\text{groupF } f \text{ } xs \neq [] \longleftrightarrow xs \neq []$
 <proof>

lemma *groupF-empty-elem*: $x \in \text{set } (\text{groupF } f \text{ } xs) \implies x \neq []$
 <proof>

lemma *groupF-distinct*: $\text{distinct } xs \implies \text{distinct } (\text{concat } (\text{groupF } f \text{ } xs))$
 <proof>

It is possible to use $\text{map } (\text{map } fst) (\text{groupF } snd (\text{map } (\lambda x. (x, f \text{ } x)) \text{ } P))$ instead of $\text{groupF } f \text{ } P$ for the following reasons: *groupF* executes its compare function (first parameter) very often; it always tests for $f \text{ } x = f \text{ } y$. 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*: $\text{groupF } f \text{ } xs = \text{map } (\text{map } fst) (\text{groupF } snd (\text{map } (\lambda x. (x, f \text{ } x)) \text{ } xs))$
 <proof>
end

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

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

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

```

    ipv4addr-wordinterval-toString a @ " u "@ipv4addr-wordinterval-toString b

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

end

```

16 toString Functions for Primitives

```

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

```

```

definition ipv4-cidr-toString :: (ipv4addr × nat) ⇒ string where
    ipv4-cidr-toString ip-n = (case ip-n of (base, n) ⇒ (ipv4addr-toString base
@"/"@" string-of-nat n))

```

```

lemma ipv4-cidr-toString (ipv4addr-of-dotdecimal (192,168,0,1), 22) = "192.168.0.1/22"
<proof>

```

```

definition ipv6-cidr-toString :: (ipv6addr × nat) ⇒ string where
    ipv6-cidr-toString ip-n = (case ip-n of (base, n) ⇒ (ipv6addr-toString base
@"/"@" string-of-nat n))

```

```

lemma ipv6-cidr-toString (42540766411282592856906245548098208122, 64) = "2001:db8::8:800:200c:417a/"
<proof>

```

```

definition primitive-protocol-toString :: primitive-protocol ⇒ string where

```

```

    primitive-protocol-toString protid ≡ (
    if protid = TCP then "tcp" else
    if protid = UDP then "udp" else
    if protid = ICMP then "icmp" else
    if protid = L4-Protocol.SCTP then "sctp" else
    if protid = L4-Protocol.IGMP then "igmp" else
    if protid = L4-Protocol.GRE then "gre" else
    if protid = L4-Protocol.ESP then "esp" else
    if protid = L4-Protocol.AH then "ah" else
    if protid = L4-Protocol.IPv6ICMP then "ipv6-icmp" else
    "protocolid:"@"dec-string-of-word0 protid)

```

```

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

```

```

definition iface-toString :: string ⇒ iface ⇒ string where
  iface-toString descr iface = (if iface = ifaceAny then "" else
    (case iface of (Iface name) ⇒ descr@name))
lemma iface-toString "in: " (Iface "+") = "" <proof>
lemma iface-toString "in: " (Iface "eth0") = "in: eth0" <proof>

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

fun ports-toString :: string ⇒ (16 word × 16 word) ⇒ string where
  ports-toString descr (s,e) = (if s = 0 ∧ e = - 1 then "" else descr @ (if s=e
  then port-toString s else port-toString s@":"@port-toString e))
lemma ports-toString "spt: " (0,65535) = "" <proof>
lemma ports-toString "spt: " (1024,2048) = "spt: 1024:2048" <proof>
lemma ports-toString "spt: " (1024,1024) = "spt: 1024" <proof>

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

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

end

```

17 Service Matrices

```

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

```

17.1 IP Address Space Partition

```

fun extract-IPSets-generic0
  :: ('i::len simple-match ⇒ 'i word × nat) ⇒ 'i simple-rule list ⇒ ('i wordinterval)
  list
  where
  extract-IPSets-generic0 - [] = [] |
  extract-IPSets-generic0 sel ((SimpleRule m -)#ss) = (ipcidr-tuple-to-wordinterval
  (sel m)) #
  (extract-IPSets-generic0 sel ss)

```

lemma *extract-IPSets-generic0-length*: $\text{length } (\text{extract-IPSets-generic0 } \text{sel } rs) = \text{length } rs$
 ⟨proof⟩

lemma *mergesort-remdups* [(1::ip4addr, 2::nat), (8,0), (8,1), (2,2), (2,4), (1,2), (2,2)] = [(1, 2), (2, 2), (2, 4), (8, 0), (8, 1)] ⟨proof⟩

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

lemma *extract-src-dst-ips-length*: $\text{length } (\text{extract-src-dst-ips } rs \text{ acc}) = 2 * \text{length } rs + \text{length } acc$
 ⟨proof⟩

definition *extract-IPSets*
 :: 'i::len simple-rule list ⇒ ('i wordinterval) list **where**
 extract-IPSets rs ≡ map ipcidr-tuple-to-wordinterval (mergesort-remdups (extract-src-dst-ips rs []))

lemma *extract-IPSets*:
 set (extract-IPSets rs) = set (extract-IPSets-generic0 src rs) ∪ set (extract-IPSets-generic0 dst rs)
 ⟨proof⟩

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

lemma *merge-length*: $\text{length } (\text{merge } l1 \ l2) \leq \text{length } l1 + \text{length } l2$
 ⟨proof⟩

lemma *merge-list-length*: $\text{length } (\text{merge-list } as \ ls) \leq \text{length } (\text{concat } (as @ ls))$
 ⟨proof⟩

lemma *mergesort-remdups-length*: $\text{length } (\text{mergesort-remdups } as) \leq \text{length } as$
 ⟨proof⟩

lemma *extract-IPSets-length*: $\text{length } (\text{extract-IPSets } rs) \leq 2 * \text{length } rs$
 ⟨proof⟩

lemma *extract-equi0*:

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

lemma *src-ipPart-motivation*:

fixes *rs*

defines $X \equiv (\lambda(\text{base}, \text{len}). \text{ipset-from-cidr base len}) \text{ ‘ src ‘ match-sel ‘ set rs}$

assumes $\forall A \in X. B \subseteq A \vee B \cap A = \{\}$ **and** $s1 \in B$ **and** $s2 \in B$

shows $\text{simple-fw rs } (p(p\text{-src}:=s1)) = \text{simple-fw rs } (p(p\text{-src}:=s2))$
⟨proof⟩

lemma *src-ipPart*:

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

$B \in A \ s1 \in B \ s2 \in B$

shows $\text{simple-fw rs } (p(p\text{-src}:=s1)) = \text{simple-fw rs } (p(p\text{-src}:=s2))$
⟨proof⟩

lemma *dst-ipPart*:

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

$B \in A \ s1 \in B \ s2 \in B$

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

definition *wordinterval-list-to-set* :: $'a::\text{len}$ *wordinterval list* $\Rightarrow 'a::\text{len}$ *word set*

where

$\text{wordinterval-list-to-set ws} = \bigcup (\text{set } (\text{map wordinterval-to-set ws}))$

lemma *wordinterval-list-to-set-compressed*:

$\text{wordinterval-to-set } (\text{wordinterval-compress } (\text{foldr wordinterval-union xs Empty-WordInterval}))$
 $=$

$\text{wordinterval-list-to-set xs}$

⟨proof⟩

fun *partIps* :: $'a::\text{len}$ *wordinterval* $\Rightarrow 'a::\text{len}$ *wordinterval list*

$\Rightarrow 'a::\text{len}$ *wordinterval list* **where**

$\text{partIps } [] = []$ |

$\text{partIps } s \ (t\#ts) = (\text{if wordinterval-empty } s \text{ then } (t\#ts) \text{ else}$

```

      (if wordinterval-empty (wordinterval-intersection s t)
       then (t#(partIps s ts))
       else
        (if wordinterval-empty (wordinterval-setminus t s)
         then (t#(partIps (wordinterval-setminus s t) ts))
         else (wordinterval-intersection t s)#((wordinterval-setminus
t s)#
          (partIps (wordinterval-setminus s t) ts))))))

```

lemma *partIps* (WordInterval (1::ipv4addr) 1) [WordInterval 0 1] = [WordInterval 1 1, WordInterval 0 0] *<proof>*

lemma *partIps-length*: length (partIps s ts) ≤ (length ts) * 2
<proof>

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

lemma *partitioningIps-length*: length (partitioningIps ss ts) ≤ (2^{length ss}) * length
ts
<proof>

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

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

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

lemma *partitioningIps-foldr*: partitioningIps ss ts = foldr partIps ss ts
<proof>

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

lemma *getParts-length*: length (getParts rs) ≤ 2^(2 * length rs)
<proof>

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

(set (map wordinterval-to-set (getParts rs)))

⟨proof⟩

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

theorem *getParts-samefw*:
assumes $A \in \text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts } rs))$ $s1 \in A$ $s2 \in A$
shows $\text{simple-fw } rs \ (p(p\text{-src}:=s1)) = \text{simple-fw } rs \ (p(p\text{-src}:=s2)) \wedge$
 $\text{simple-fw } rs \ (p(p\text{-dst}:=s1)) = \text{simple-fw } rs \ (p(p\text{-dst}:=s2))$
 ⟨proof⟩

lemma *partIps-nonempty*: $ts \neq [] \implies \text{partIps } s \ ts \neq []$
 ⟨proof⟩

lemma *partitioningIps-nonempty*: $ts \neq [] \implies \text{partitioningIps } ss \ ts \neq []$
 ⟨proof⟩

lemma *getParts-nonempty*: $\text{getParts } rs \neq []$ ⟨proof⟩

lemma *getParts-nonempty-elems*: $\forall w \in \text{set } (\text{getParts } rs). \neg \text{wordinterval-empty } w$
 ⟨proof⟩

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*: $\neg \text{wordinterval-empty } W \implies \text{wordinterval-element } (\text{getOneIp } W)$ W
 ⟨proof⟩

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::len word \Rightarrow 'i word \Rightarrow 'i sim-

ple-rule list \Rightarrow bool **where**

same-fw-behaviour ~~*TYPP/E(/p/K/et/)*~~ *a b rs* \equiv

$\forall (p:: 'i::len \text{ simple-packet}).$

$\text{simple-fw } rs (p(p\text{-src}=a)) = \text{simple-fw } rs (p(p\text{-src}=b)) \wedge$

$\text{simple-fw } rs (p(p\text{-dst}=a)) = \text{simple-fw } rs (p(p\text{-dst}=b))$

lemma *getParts-same-fw-behaviour*:

$A \in \text{set } (\text{map } \text{wordinterval-to-set } (\text{getParts } rs)) \Longrightarrow s1 \in A \Longrightarrow s2 \in A \Longrightarrow$

same-fw-behaviour *s1 s2 rs*

<proof>

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

$p\text{-src}=s, p\text{-dst}=d,$

$p\text{-proto}=pc\text{-proto } c,$

$p\text{-sport}=pc\text{-sport } c, p\text{-dport}=pc\text{-dport } c,$

$p\text{-tcp-flags}=\{TCP\text{-SYN}\},$

$p\text{-payload}=""$)

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

definition *runFw-scheme* :: *'i::len word* \Rightarrow *'i word* \Rightarrow *'b 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 = *simple-fw rs*

$(p(p\text{-iiface}=pc\text{-iiface } c,$

$p\text{-oiface}=pc\text{-oiface } c,$

$p\text{-src}=s,$

$p\text{-dst}=d,$

$p\text{-proto}=pc\text{-proto } c,$

$p\text{-sport}=pc\text{-sport } c,$

$p\text{-dport}=pc\text{-dport } c))$

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

<proof>

lemma *has-default-policy-runFw*: *has-default-policy rs* \Longrightarrow *runFw s d c rs* = *Decision FinalAllow* \vee *runFw s d c rs* = *Decision FinalDeny*

<proof>

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* \Longrightarrow *same-fw-behaviour-one ip1*

$ip2\ c\ rs$
 ⟨proof⟩

Is an equivalence relation

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

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

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

lemma *runFw-sameFw-behave:*

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

lemma *sameFw-behave-sets:*

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

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

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

lemma *groupWIs-not-empty*: $\text{groupWIs } c \text{ rs} \neq []$

<proof>

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

<proof>

lemma *groupWIs-not-empty-elems*:

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

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

<proof>

lemma *groupParts-same-fw-wi0*:

assumes $V \in \text{set } (\text{groupWIs } c \text{ rs})$

shows $\forall w \in \text{set } (\text{map wordinterval-to-set } V). \forall a1 \in w. \forall a2 \in w. \text{same-fw-behaviour-one } a1 \ a2 \ c \ \text{rs}$

<proof>

lemma *groupWIs-same-fw-not*: $A \in \text{set } (\text{groupWIs } c \text{ rs}) \implies B \in \text{set } (\text{groupWIs } c \text{ rs}) \implies$

$A \neq B \implies$

$\forall aw \in \text{set } (\text{map wordinterval-to-set } A).$

$\forall bw \in \text{set } (\text{map wordinterval-to-set } B).$

$\forall a \in aw. \forall b \in bw. \neg \text{same-fw-behaviour-one } a \ b \ c \ \text{rs}$

<proof>

lemma *groupParts-same-fw-wi1*:

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

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

<proof>

lemma *groupParts-same-fw-wi2*: $V \in \text{set } (\text{groupWIs } c \text{ rs}) \implies$

$\forall ip1 \in \text{wordinterval-list-to-set } V.$

$\forall ip2 \in \text{wordinterval-list-to-set } V.$

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

<proof>

lemma *groupWIs-same-fw-not2*: $A \in \text{set } (\text{groupWIs } c \text{ rs}) \implies B \in \text{set } (\text{groupWIs } c \text{ rs}) \implies$

$A \neq B \implies$

$\forall ip1 \in \text{wordinterval-list-to-set } A.$

$\forall ip2 \in \text{wordinterval-list-to-set } B.$

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

<proof>

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

<proof>

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

definition *groupWIs1* :: 'a parts-connection-scheme \Rightarrow 'i::len simple-rule list \Rightarrow
'i wordinterval list list **where**
 $\text{groupWIs1 } c \ \text{rs} = (\text{let } P = \text{getParts } \text{rs} \ \text{in}$
 $(\text{let } W = \text{map } \text{getOneIp } P \ \text{in}$
 $(\text{let } f = (\lambda wi. (\text{map } (\lambda d. \text{runFw } (\text{getOneIp } wi) \ d \ c \ \text{rs}) \ W,$
 $\text{map } (\lambda s. \text{runFw } s \ (\text{getOneIp } wi) \ c \ \text{rs}) \ W)) \ \text{in}$
 $\text{map } (\text{map } \text{fst}) \ (\text{groupF } \text{snd} \ (\text{map } (\lambda x. (x, f \ x)) \ P))))))$

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

definition *simple-conn-matches* :: 'i::len simple-match \Rightarrow parts-connection \Rightarrow
bool **where**
 $\text{simple-conn-matches } m \ c \ \longleftrightarrow$
 $(\text{match-iface } (\text{iiface } m) \ (\text{pc-iiface } c)) \ \wedge$
 $(\text{match-iface } (\text{oiface } m) \ (\text{pc-oiface } c)) \ \wedge$
 $(\text{match-proto } (\text{proto } m) \ (\text{pc-proto } c)) \ \wedge$
 $(\text{simple-match-port } (\text{sports } m) \ (\text{pc-sport } c)) \ \wedge$
 $(\text{simple-match-port } (\text{dports } m) \ (\text{pc-dport } c))$

lemma *simple-conn-matches-simple-match-any*: $\text{simple-conn-matches } \text{simple-match-any}$
 c
<proof>

lemma *has-default-policy-simple-conn-matches*:
 $\text{has-default-policy } \text{rs} \implies \text{has-default-policy } [\text{r} \leftarrow \text{rs} . \text{simple-conn-matches } (\text{match-sel}$
 $\text{r}) \ c]$
<proof>

lemma *filter-conn-fw-lem*:
 $\text{runFw } s \ d \ c \ (\text{filter } (\lambda r. \text{simple-conn-matches } (\text{match-sel } r) \ c) \ \text{rs}) = \text{runFw } s \ d$
 $c \ \text{rs}$
<proof>

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

```

  groupWIs2 c rs = (let P = getParts rs in
                    (let W = map getOneIp P in
                     (let filterW = (filter (λr. 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
 ⟨proof⟩

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

fun *matching-dsts* :: 'i::len *word* ⇒ 'i *simple-rule list* ⇒ 'i *wordinterval* ⇒ 'i *wordinterval* **where**

```

  matching-dsts - [] - = Empty-WordInterval |
  matching-dsts s ((SimpleRule m Accept)#rs) acc-dropped =
    (if simple-match-ip (src m) s then
     wordinterval-union (wordinterval-setminus (ipcidr-tuple-to-wordinterval
(dst m)) acc-dropped) (matching-dsts s rs acc-dropped)
    else
     matching-dsts s rs acc-dropped) |
  matching-dsts s ((SimpleRule m Drop)#rs) acc-dropped =
    (if simple-match-ip (src m) s then
     matching-dsts s rs (wordinterval-union (ipcidr-tuple-to-wordinterval (dst
m)) acc-dropped)
    else
     matching-dsts s rs acc-dropped)

```

lemma *matching-dsts-pull-out-accu*:

```

  wordinterval-to-set (matching-dsts s rs (wordinterval-union a1 a2)) = wordinter-
val-to-set (matching-dsts s rs a2) - wordinterval-to-set a1
  ⟨proof⟩

```

fun *matching-srcs* :: 'i::len *word* ⇒ 'i *simple-rule list* ⇒ 'i *wordinterval* ⇒ 'i *wordinterval* **where**

```

  matching-srcs - [] - = Empty-WordInterval |

```

$matching_srcs\ d\ ((SimpleRule\ m\ Accept)\#rs)\ acc_dropped =$
 (if simple-match-ip (dst m) d then
 wordinterval-union (wordinterval-setminus (ipcidr-tuple-to-wordinterval
 (src m)) acc-dropped) (matching-srcs d rs acc-dropped)
 else
 matching-srcs d rs acc-dropped) |
 $matching_srcs\ d\ ((SimpleRule\ m\ Drop)\#rs)\ acc_dropped =$
 (if simple-match-ip (dst m) d then
 matching-srcs d rs (wordinterval-union (ipcidr-tuple-to-wordinterval (src
 m)) acc-dropped)
 else
 matching-srcs d rs acc-dropped)

lemma *matching-srcs-pull-out-accu:*

$wordinterval\ to\ set\ (matching_srcs\ d\ rs\ (wordinterval\ union\ a1\ a2)) = wordinter\ val\ to\ set\ (matching_srcs\ d\ rs\ a2) - wordinterval\ to\ set\ a1$
 <proof>

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

$wordinterval\ to\ set\ (matching_dsts\ s\ rs\ Empty\ WordInterval) = \{d.\ runFw\ s\ d\ c\ rs = Decision\ FinalAllow\}$
 <proof>

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

$wordinterval\ to\ set\ (matching_srcs\ d\ rs\ Empty\ WordInterval) = \{s.\ runFw\ s\ d\ c\ rs = Decision\ FinalAllow\}$
 <proof>

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 $\text{groupWIs2 } c \text{ } rs = \text{groupWIs3-default-policy } c \text{ } rs$
 ⟨*proof*⟩

definition $\text{groupWIs3} :: \text{parts-connection} \Rightarrow 'i::\text{len simple-rule list} \Rightarrow 'i \text{ wordinterval list list}$ **where**
 $\text{groupWIs3 } c \text{ } rs = (\text{if } \text{has-default-policy } rs \text{ then } \text{groupWIs3-default-policy } c \text{ } rs$
 $\text{else } \text{groupWIs2 } c \text{ } rs)$

lemma $\text{groupWIs3: groupWIs3} = \text{groupWIs}$
 ⟨*proof*⟩

definition $\text{build-ip-partition} :: \text{parts-connection} \Rightarrow 'i::\text{len simple-rule list} \Rightarrow 'i \text{ wordinterval list}$ **where**
 $\text{build-ip-partition } c \text{ } rs = \text{map}$
 $(\lambda xs. \text{wordinterval-sort } (\text{wordinterval-compress } (\text{foldr } \text{wordinterval-union } xs$
 $\text{Empty-WordInterval})))$
 $(\text{groupWIs3 } c \text{ } rs)$

theorem $\text{build-ip-partition-same-fw: } V \in \text{set } (\text{build-ip-partition } c \text{ } rs) \Longrightarrow$
 $\forall ip1::'i::\text{len word} \in \text{wordinterval-to-set } V.$
 $\forall ip2::'i::\text{len word} \in \text{wordinterval-to-set } V.$
 $\text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$
 ⟨*proof*⟩

theorem $\text{build-ip-partition-same-fw-min: } A \in \text{set } (\text{build-ip-partition } c \text{ } rs) \Longrightarrow B$
 $\in \text{set } (\text{build-ip-partition } c \text{ } rs) \Longrightarrow$
 $A \neq B \Longrightarrow$
 $\forall ip1::'i::\text{len word} \in \text{wordinterval-to-set } A.$
 $\forall ip2::'i::\text{len word} \in \text{wordinterval-to-set } B.$
 $\neg \text{same-fw-behaviour-one } ip1 \text{ } ip2 \text{ } c \text{ } rs$
 ⟨*proof*⟩

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

lemma $\text{build-ip-partition-no-empty-elems: } wi \in \text{set } (\text{build-ip-partition } c \text{ } rs) \Longrightarrow \neg$
 $\text{wordinterval-empty } wi$
 ⟨*proof*⟩

lemma *build-ip-partition-disjoint*:

$V1 \in \text{set } (\text{build-ip-partition } c \text{ } rs) \implies V2 \in \text{set } (\text{build-ip-partition } c \text{ } rs) \implies$
 $V1 \neq V2 \implies$
 $\text{wordinterval-to-set } V1 \cap \text{wordinterval-to-set } V2 = \{\}$
{proof}

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}

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}

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

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

lemma *build-ip-partition-distinct'*: *distinct* (*build-ip-partition* *c* *rs*)
{proof}

17.2 Service Matrix over an IP Address Space Partition

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

lemma *simple-fw-no-interfaces*:

assumes *no-ifaces*: *simple-firewall-without-interfaces* *rs*
shows *simple-fw* *rs* *p* = *simple-fw* *rs* (*p* | *p*-iiface:= *x*, *p*-oiface:= *y*)
{proof}

lemma *runFw-no-interfaces*:

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

lemma[code-unfold]: *simple-firewall-without-interfaces* $rs \equiv$
 $\forall m \in \text{set } rs. \text{iiface } (\text{match-sel } m) = \text{ifaceAny} \wedge \text{oiface } (\text{match-sel } m) = \text{ifaceAny}$
 ⟨proof⟩

definition *access-matrix*

$:: \text{parts-connection} \Rightarrow 'i::\text{len simple-rule list} \Rightarrow ('i \text{ word} \times 'i \text{ wordinterval}) \text{ list} \times$
 $('i \text{ word} \times 'i \text{ word}) \text{ list}$

where

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

(let $W = \text{build-ip-partition } c \text{ } rs;$

$R = \text{map } \text{getOneIp } W$

in

(zip $R \ W, [(s, d) \leftarrow \text{all-pairs } R. \text{runFw } s \ d \ c \ rs = \text{Decision } \text{FinalAllow}]])$)

lemma *access-matrix-nodes-defined*:

$(V, E) = \text{access-matrix } c \text{ } rs \Rightarrow (s, d) \in \text{set } E \Rightarrow s \in \text{dom } (\text{map-of } V)$ **and**

$(V, E) = \text{access-matrix } c \text{ } rs \Rightarrow (s, d) \in \text{set } E \Rightarrow d \in \text{dom } (\text{map-of } V)$

⟨proof⟩

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

lemma $(V, E) = \text{access-matrix } c \text{ } rs \Rightarrow (s, d) \in \text{set } E \Rightarrow \text{runFw } s \ d \ c \ rs =$
 $\text{Decision } \text{FinalAllow}$

⟨proof⟩

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

lemma *map-of-zip-map*: $\text{map-of } (\text{zip } (\text{map } f \ rs) \ rs) \ k = \text{Some } v \Rightarrow k = f \ v$

⟨proof⟩

lemma *access-matrix-sound*: **assumes** *matrix*: $(V, E) = \text{access-matrix } c \text{ } rs$ **and**

repr: $(s\text{-repr}, d\text{-repr}) \in \text{set } E$ **and**

s-range: $(\text{map-of } V) \ s\text{-repr} = \text{Some } s\text{-range}$ **and** $s: s \in \text{wordinterval-to-set}$
s-range **and**

d-range: $(\text{map-of } V) \ d\text{-repr} = \text{Some } d\text{-range}$ **and** $d: d \in \text{wordinterval-to-set}$
d-range

shows $\text{runFw } s \ d \ c \ rs = \text{Decision } \text{FinalAllow}$

⟨proof⟩

lemma *distinct-map-getOneIp-obtain*: $v \in \text{set } xs \Rightarrow \text{distinct } (\text{map } \text{getOneIp } xs)$

\Rightarrow

$\exists s\text{-repr}. \text{map-of } (\text{zip } (\text{map } \text{getOneIp } xs) \ xs) \ s\text{-repr} = \text{Some } v$

⟨proof⟩

lemma *access-matrix-complete*:

fixes $rs :: 'i::\text{len simple-rule list}$

assumes *matrix*: $(V,E) = \text{access-matrix } c \text{ } rs$ **and**
allow: $\text{runFw } s \text{ } d \text{ } c \text{ } rs = \text{Decision FinalAllow}$
shows $\exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range. } (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge$
 $(\text{map-of } V) \text{ } s\text{-repr} = \text{Some } s\text{-range} \wedge s \in \text{wordinterval-to-set } s\text{-range} \wedge$
 $(\text{map-of } V) \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge d \in \text{wordinterval-to-set } d\text{-range}$
 $\langle \text{proof} \rangle$

theorem *access-matrix*:

fixes $rs :: 'i::\text{len simple-rule list}$

assumes *matrix*: $(V,E) = \text{access-matrix } c \text{ } rs$

shows $(\exists s\text{-repr } d\text{-repr } s\text{-range } d\text{-range. } (s\text{-repr}, d\text{-repr}) \in \text{set } E \wedge$

$(\text{map-of } V) \text{ } s\text{-repr} = \text{Some } s\text{-range} \wedge s \in \text{wordinterval-to-set } s\text{-range} \wedge$

$(\text{map-of } V) \text{ } d\text{-repr} = \text{Some } d\text{-range} \wedge d \in \text{wordinterval-to-set } d\text{-range})$

\longleftrightarrow

$\text{runFw } s \text{ } d \text{ } c \text{ } rs = \text{Decision FinalAllow}$

$\langle \text{proof} \rangle$

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:

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

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

```
access-matrix-pretty-ipv4 c rs ≡
  if ¬ simple-firewall-without-interfaces rs then undefined else
    (let (V,E) = (access-matrix c rs);
        formatted-nodes =
          map (λ(v-repr, v-range). (ipv4addr-toString v-repr, ipv4addr-wordinterval-toString
v-range)) V;
        formatted-edges =
          map (λ(s,d). (ipv4addr-toString s, ipv4addr-toString d)) E
    in
      (formatted-nodes, formatted-edges)
  )
```

definition *access-matrix-pretty-ipv4-code*

:: parts-connection ⇒ 32 simple-rule list ⇒ (string × string) list × (string × string) list

where

```
access-matrix-pretty-ipv4-code c rs ≡
  if ¬ simple-firewall-without-interfaces rs then undefined else
    (let W = build-ip-partition c rs;
        R = map getOneIp W;
        U = all-pairs R
    in
      (zip (map ipv4addr-toString R) (map ipv4addr-wordinterval-toString W),
        map (λ(x,y). (ipv4addr-toString x, ipv4addr-toString y)) [(s, d)←all-pairs R.
runFw s d c rs = Decision FinalAllow]))
```

lemma *access-matrix-pretty-ipv4-code[code]: access-matrix-pretty-ipv4 = access-matrix-pretty-ipv4-code*
(*proof*)

definition *access-matrix-pretty-ipv6*

:: parts-connection ⇒ 128 simple-rule list ⇒ (string × string) list × (string × string) list

where

```
access-matrix-pretty-ipv6 c rs ≡
  if ¬ simple-firewall-without-interfaces rs then undefined else
    (let (V,E) = (access-matrix c rs);
        formatted-nodes =
          map (λ(v-repr, v-range). (ipv6addr-toString v-repr, ipv6addr-wordinterval-toString
v-range)) V;
        formatted-edges =
          map (λ(s,d). (ipv6addr-toString s, ipv6addr-toString d)) E
    in
      (formatted-nodes, formatted-edges)
  )
```

definition *access-matrix-pretty-ipv6-code*

:: parts-connection ⇒ 128 simple-rule list ⇒ (string × string) list × (string ×

```

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

```

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

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

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

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

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

value[code] *partitioningIps* [WordInterval (0::ipv4addr) 0] [WordInterval 0 2, WordInter-
 val 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" |

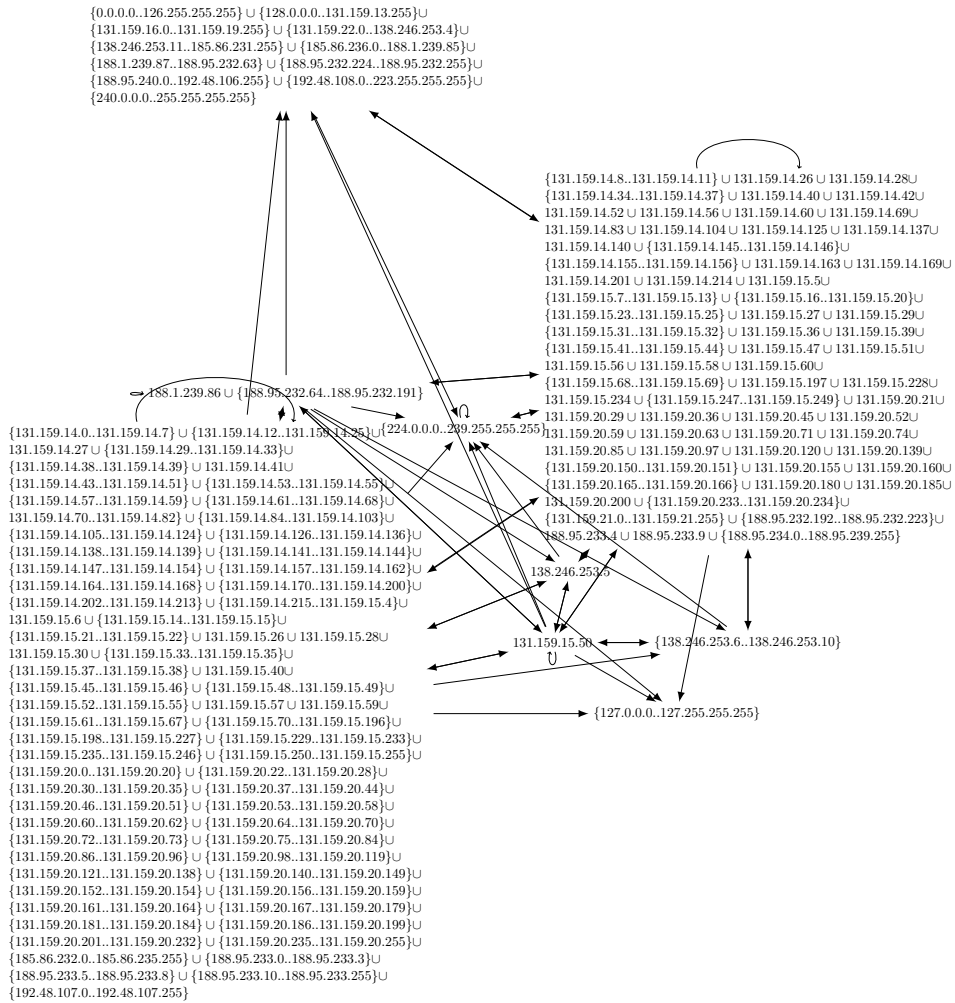


Figure 1: TUM ssh Service Matrix

```
simple-action-toString Drop = "DROP"
```

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

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

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

```
end
```

References

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