Abstract

We present LOFT — Linux firewall OpenFlow Translator, a system that transforms the main routing table and FORWARD chain of iptables of a Linux-based firewall into a set of static OpenFlow rules. Our implementation is verified against a model of a simplified Linux-based router and we can directly show how much of the original functionality is preserved.

Please note that this document is organized in two distinct parts. The first part contains the necessary definitions, helper lemmas and proofs in all their technicality as made in the theory code. The second part reiterates the most important definitions and proofs in a manner that is more suitable for human readers and enriches them with detailed explanations in natural language. Any interested reader should start from there.

Many of the considerations that have led to the definitions made here have been explained in [8].

Contents

I Code 2

II Documentation 44

1 Configuration Translation 44

1.1 Linux Firewall Model .......................................................... 44
1.1.1 Routing Table .................................................................. 45
1.1.2 iptables Firewall .............................................................. 46
1.2 OpenFlow Switch Model .......................................................... 46
1.2.1 Matching Flow Table entries .............................................. 47
1.2.2 Evaluating a Flow Table ..................................................... 47
1.3 Translation Implementation ...................................................... 49
1.3.1 Chaining Firewalls ............................................................ 49
1.3.2 Translation Implementation ............................................... 50
1.3.3 Comparison to Exodus ..................................................... 52

2 Evaluation 52

2.1 Mininet Examples ................................................................. 52
2.2 Performance Evaluation ......................................................... 54

3 Conclusion and Future Work 55
Part I

Code

theory OpenFlow-Matches
imports IP-Addresses.Prefix-Match
  Simple-Firewall.Simple-Packet
  HOL-Library.Monad-Syntax
  HOL-Library.List-Lexorder
  HOL-Library.Char-ord
begin

datatype of-match-field =
  IngressPort string
| EtherSrc 48 word
| EtherDst 48 word
| EtherType 16 word
| VlanId 16 word
| VlanPriority 16 word
| IPv4Src 32 prefix-match
| IPv4Dst 32 prefix-match
| IPv4Proto 8 word
| L4Src 16 word 16 word
| L4Dst 16 word 16 word

schematic-goal of-match-field-typeset: (field-match :: of-match-field) ∈ {
  IngressPort (s::string),
  EtherSrc (as::48 word), EtherDst (ad::48 word),
  EtherType (t::16 word),
  VlanId (i::16 word), VlanPriority (p::16 word),
  IPv4Src (pms::32 prefix-match), IPv4Dst (pmd::32 prefix-match),
  IPv4Proto (ipp :: 8 word),
  L4Src (ps :: 16 word) (ms :: 16 word),
  L4Dst (pd :: 16 word) (md :: 16 word)
}

proof((cases field-match; clarsimp),goal-cases)
  next case (IngressPort s)  thus s = (case field-match of IngressPort s ⇒ s) unfolding IngressPort of-match-field.simps by rule
  next case (EtherSrc s)  thus s = (case field-match of EtherSrc s ⇒ s) unfolding EtherSrc of-match-field.simps by rule
  next case (EtherDst s)  thus s = (case field-match of EtherDst s ⇒ s) unfolding EtherDst of-match-field.simps by rule
  next case (EtherType s)  thus s = (case field-match of EtherType s ⇒ s) unfolding EtherType of-match-field.simps by rule
  next case (VlanId s)  thus s = (case field-match of VlanId s ⇒ s) unfolding VlanId of-match-field.simps by rule
  next case (VlanPriority s)  thus s = (case field-match of VlanPriority s ⇒ s) unfolding VlanPriority of-match-field.simps by rule

2
next case (IPv4Src s)  thus s = (case field-match of IPv4Src s ⇒ s) unfolding IPv4Src of-match-field.simps by rule
next case (IPv4Dst s) thus s = (case field-match of IPv4Dst s ⇒ s) by simp
next case (IPv4Proto s) thus s = (case field-match of IPv4Proto s ⇒ s) by simp
next case (L4Src p l) thus p = (case field-match of L4Src p m ⇒ p) ∧ l = (case field-match of L4Src p m ⇒ m) by simp
next case (L4Dst p l) thus p = (case field-match of L4Dst p m ⇒ p) ∧ l = (case field-match of L4Dst p m ⇒ m) by simp
qed

function prerequisites :: of-match-field ⇒ of-match-field set ⇒ bool where
prerequisites (IngressPort -) = True |
prerequisites (EtherDst -) = True |
prerequisites (EtherSrc -) = True |
prerequisites (EtherType -) = True |
prerequisites (VlanId -) = True |
prerequisites (VlanPriority -) = (∃ id. let v = VlanId id in v ∈ m ∧ prerequisites v m) |
prerequisites (IPv4Proto -) = (let v = EtherType 0x0800 in v ∈ m ∧ prerequisites v m) |
prerequisites (IPv4Src -) = (let v = EtherType 0x0800 in v ∈ m ∧ prerequisites v m) |
prerequisites (IPv4Dst -) = (let v = EtherType 0x0800 in v ∈ m ∧ prerequisites v m) |
prerequisites (L4Src -) = (∀ proto ∈ {TCP,UDP,L4-Protocol.SCTP}. let v = IPv4Proto proto in v ∈ m ∧ prerequisites v m) |
prerequisites (L4Dst -) = prerequisites (L4Src undefined undefined) m by pat-completeness auto

fun match-sorter :: of-match-field ⇒ nat where
match-sorter (IngressPort -) = 1 |
match-sorter (VlanId -) = 2 |
match-sorter (VlanPriority -) = 3 |
match-sorter (EtherType -) = 4 |
match-sorter ( EtherSrc -) = 5 |
match-sorter (EtherDst -) = 6 |
match-sorter (IPv4Proto -) = 7 |
match-sorter (IPv4Src -) = 8 |
match-sorter (IPv4Dst -) = 9 |
match-sorter (L4Src -) = 10 |
match-sorter (L4Dst -) = 11

termination prerequisites by (relation measure (match-sorter ◦ fst), simp-all)
definition less-eq-of-match-field1 :: of-match-field ⇒ of-match-field ⇒ bool
where \( \text{less-eq-of-match-field1} \ (a::\text{of-match-field}) \ (b::\text{of-match-field}) \iff (\text{case} \ (a, b) \ of \)
\begin{align*}
(\text{IngressPort} \ a, \ \text{IngressPort} \ b) & \Rightarrow a \leq b \\
(\text{VlanId} \ a, \ \text{VlanId} \ b) & \Rightarrow a \leq b \\
(\text{EtherDst} \ a, \ \text{EtherDst} \ b) & \Rightarrow a \leq b \\
(\text{EtherSrc} \ a, \ \text{EtherSrc} \ b) & \Rightarrow a \leq b \\
(\text{EtherType} \ a, \ \text{EtherType} \ b) & \Rightarrow a \leq b \\
(\text{VlanPriority} \ a, \ \text{VlanPriority} \ b) & \Rightarrow a \leq b \\
(\text{IPv4Proto} \ a, \ \text{IPv4Proto} \ b) & \Rightarrow a \leq b \\
(\text{IPv4Src} \ a, \ \text{IPv4Src} \ b) & \Rightarrow a \leq b \\
(\text{IPv4Dst} \ a, \ \text{IPv4Dst} \ b) & \Rightarrow a \leq b \\
(\text{L4Src} \ a1 \ a2, \ L4Src \ b1 \ b2) & \Rightarrow \text{if } a2 = b2 \text{ then } a1 \leq b1 \text{ else } a2 \leq b2 \\
(\text{L4Dst} \ a1 \ a2, \ L4Dst \ b1 \ b2) & \Rightarrow \text{if } a2 = b2 \text{ then } a1 \leq b1 \text{ else } a2 \leq b2 \\
(a, b) & \Rightarrow \text{match-sorter } a < \text{-match-sorter } b
\end{align*}

\begin{align*}
\text{instantiation of-match-field} & \colon \text{linorder} \\
\begin{align*}
\text{definition} & \quad \text{less-eq-of-match-field} \ (a::\text{of-match-field}) \ (b::\text{of-match-field}) \iff \text{less-eq-of-match-field1} \ a \ b \\
\text{definition} & \quad \text{less-of-match-field} \ (a::\text{of-match-field}) \ (b::\text{of-match-field}) \iff a \neq b \land \text{less-eq-of-match-field1} \ a \ b
\end{align*}
\end{align*}

\begin{align*}
\text{instance} & \quad \text{by standard (auto simp add: less-eq-of-match-field-def less-of-match-field-def less-eq-of-match-field1-def split: prod.splits of-match-field.splits if-splits)}
\end{align*}

end

fun match-no-prereq :: \text{of-match-field} \Rightarrow (32, 'a) \text{simple-packet-ext-scheme} \Rightarrow \text{bool}
\begin{align*}
& \text{where} \\
& \text{match-no-prereq} \ (\text{IngressPort} \ i) \ p = (p\text{-iface } p = i) \\
& \text{match-no-prereq} \ (\text{EtherDst} \ i) \ p = (p\text{-l2src } p = i) \\
& \text{match-no-prereq} \ (\text{EtherSrc} \ i) \ p = (p\text{-l2dst } p = i) \\
& \text{match-no-prereq} \ (\text{EtherType} \ i) \ p = (p\text{-l2type } p = i) \\
& \text{match-no-prereq} \ (\text{VlanId} \ i) \ p = (p\text{-vlanid } p = i) \\
& \text{match-no-prereq} \ (\text{VlanPriority} \ i) \ p = (p\text{-vlanprio } p = i) \\
& \text{match-no-prereq} \ (\text{IPv4Proto} \ i) \ p = (p\text{-proto } p = i) \\
& \text{match-no-prereq} \ (\text{IPv4Src} \ i) \ p = (p\text{-prefix-match-semantics } i \ (p\text{-src } p)) \\
& \text{match-no-prereq} \ (\text{IPv4Dst} \ i) \ p = (p\text{-prefix-match-semantics } i \ (p\text{-dst } p)) \\
& \text{match-no-prereq} \ (\text{L4Src} \ i m) \ p = (p\text{-sport } p \&\& m = i) \\
& \text{match-no-prereq} \ (\text{L4Dst} \ i m) \ p = (p\text{-dport } p \&\& m = i)
\end{align*}

\begin{align*}
\text{definition} & \quad \text{match-prereq} :: \text{of-match-field} \Rightarrow \text{of-match-field set} \\
& \quad \Rightarrow (32, 'a) \text{simple-packet-ext-scheme} \Rightarrow \text{bool} \\
& \text{where} \\
& \text{match-prereq } i \ s \ p \ = (\text{if prerequisite } i \ s \ \text{then Some (match-no-prereq } i \ p) \ \text{else None)}
\end{align*}

\begin{align*}
\text{definition} & \quad \text{set-seq } s \equiv \ (\forall x \in s. \ x \neq \text{None}) \text{ then Some (the } s) \text{ else None} \\
\text{definition} & \quad \text{all-true } s \equiv \ (\forall x \in s. \ x \\
\text{term map-option} \quad \text{definition} & \quad \text{OF-match-fields :: of-match-field set} \Rightarrow (32, 'a) \text{simple-packet-ext-scheme} \Rightarrow \text{bool} \\
& \quad \text{where} \ \text{OF-match-fields} \\
& \quad m \ p = \text{map-option all-true (set-seq } ((\lambda f. \text{match-prereq } f \ m \ p) \ 'm))
\end{align*}

\begin{align*}
\text{definition} & \quad \text{OF-match-fields-unsafe :: of-match-field set} \Rightarrow (32, 'a) \text{simple-packet-ext-scheme} \Rightarrow \text{bool} \\
& \quad \text{where} \ \text{OF-match-fields-unsafe} \\
& \quad m \ p = (\forall f \in m. \ \text{match-no-prereq } f \ p)
\end{align*}

\begin{align*}
\text{definition} & \quad \text{OF-match-fields-safe } m \equiv \ (\forall f \in \text{OF-match-fields} \ m)
\end{align*}
definition all-prerequisites \( m \equiv \forall f \in m. \text{prerequisites } f \text{ } m \)

lemma all-prerequisites \( p \implies L4Src \ x \ y \in p \implies \text{IPProt} \{\text{TCP}, \text{UDP}, \text{L4-Protocol.SCTP}\} \cap p \neq \{\} \)

unfolding all-prerequisites-def by auto

lemma of-safe-unsafe-match-eq: all-prerequisites \( m \implies \text{OF-match-fields } m \ p \ = \ \text{Some } (\text{OF-match-fields-unsafe } m \ p) \)

unfolding OF-match-fields-def OF-match-fields-unsafe-def comp-def set-seq-def match-prereq-def all-prerequisites-def

proof goal-cases
  case 1
  have 2: \( (\lambda f. \text{if prerequisites } f \text{ } m \text{ then Some } (\text{match-no-prereq } f \text{ } p) \text{ else None) } \ ' m \ \ = \ (\lambda f. \text{Some } (\text{match-no-prereq } f \text{ } p)) \ ' m \)
  using 1 by fastforce
  have 3: \( \forall x \in (\lambda f. \text{Some } (\text{match-no-prereq } f \text{ } p)) \ ' m. \ x \neq \text{None} \ \text{by blast} \)
  show \( \text{false} \)
  unfolding 2 unfolding eqTrueI \[OF 3\] unfolding if-True unfolding image-comp comp-def unfolding option sel by(simp add: all-true-def)
  qed

lemma of-match-fields-safe-eq: assumes all-prerequisites \( m \)

shows \( \text{OF-match-fields-safe } m \ p \ = \ \text{OF-match-fields-unsafe } m \ p \)

unfolding OF-match-fields-safe-def \[abs-def\] fun-eq-iff comp-def unfolding of-safe-unsafe-match-eq[OF assms] unfolding option.sel by clarify

lemma OF-match-fields-alt: OF-match-fields \( m \ p \ = \)
  \( \text{if } \exists f \in m. \ \neg \text{prerequisites } f \text{ } m \text{ then None else} \)
  \( \text{if } \forall f \in m. \ \text{match-no-prereq } f \text{ } p \text{ then Some True else Some False} \)

unfolding OF-match-fields-def all-true-def[abs-def] set-seq-def match-prereq-def
by(auto simp add: all-true-def)

lemma of-match-fields-safe-eq2: assumes all-prerequisites \( m \)

shows \( \text{OF-match-fields-safe } m \ p \ \longleftrightarrow \ \text{OF-match-fields } m \ p \)


end

theory OpenFlow-Action

imports
  OpenFlow-Matches

begin

datatype of-action = Forward (oiface-sel: string) | ModifyField-l2dst 48 word

fun of-action-semantics where
of-action-semantics \( p \ [\] = \{\} \ |
of-action-semantics \( p \ (a\#as) = \text{(case } a \text{ } of \)
Forward i ⇒ insert (i,p) (of-action-semantics p as) |
ModifyField-l2dst a ⇒ of-action-semantics (p[p-l2dst := a]) as)

value of-action-semantics p []
value of-action-semantics p [ModifyField-l2dst 66, Forward "of"]

end
theory Semantics-OpenFlow
imports List-Group Sort-Descending
  IP-Addresses.IPv4
  OpenFlow-Helpers
begin

datatype 'a flowtable-behavior = Action 'a | NoAction | Undefined

definition option-to-ftb b ≡ case b of Some a ⇒ Action a | None ⇒ NoAction

definition ftb-to-option b ≡ case b of Action a ⇒ Some a | NoAction ⇒ None

datatype ('m, 'a) flow-entry-match = OFEntry (ofe-prio: 16 word) (ofe-fields: 'm set) (ofe-action: 'a)

find-consts (''a × ''b) ⇒ ''c ⇒ ''a ⇒ ''b ⇒ ''c

find-consts (''a ⇒ ''b ⇒ ''c) ⇒ (''a × ''b) ⇒ ''c

definition split3 f p ≡ case p of (a,b,c) ⇒ f a b c

find-consts (''a ⇒ ''b ⇒ ''c ⇒ ''d) ⇒ (''a × ''b × ''c) ⇒ ''d

type-synonym (''m, 'a) flowtable = ((''m, 'a) flow-entry-match) list

find-consts (''m, 'a) flowtable = (''m, 'a) flow-entry-match list

definition OF-same-priority-match2 :: (''m, 'p) field-matcher ⇒ (''m, 'a) flowtable ⇒ 'p ⇒ 'a flowtable-behavior where
OF-same-priority-match2 γ flow-entries packet ≡ let s =
  {ofe-action f | f ∈ set flow-entries ∧ γ (ofe-fields f) packet ∧
   (∀ fo ∈ set flow-entries. ofe-prio fo > ofe-prio f → ¬γ (ofe-fields fo) packet)} in
  case card s of 0 ⇒ NoAction
  | (Suc 0) ⇒ Action (the-elem s)
  | _ ⇒ Undefined
definition check-no-overlap $\gamma \ ft = (\forall a \in \text{set ft.} \ \forall b \in \text{set ft.} \ \forall p \in \text{UNIV.} \ (\text{ofe-prio a = ofe-prio b \land} \ \gamma \ (\text{ofe-fields a} p \land a \neq b) \ \rightarrow \ \neg \gamma \ (\text{ofe-fields b} p))$
definition check-no-overlap2 $\gamma \ ft = (\forall a \in \text{set ft.} \ \forall b \in \text{set ft.} \ (a \neq b \land \gamma \ (\text{ofe-fields a} p \land a \neq b) \rightarrow \neg(\exists p \in \text{UNIV.} \ \gamma \ (\text{ofe-fields b} p)))$

lemma check-no-overlap-alt: check-no-overlap $\gamma \ ft = \text{check-no-overlap2} \ \gamma \ ft$

unfolding check-no-overlap-alt: check-no-overlap2 $\gamma \ ft$

by blast

lemma no-overlap-not-undefined: check-no-overlap $\gamma \ ft \Longrightarrow \text{OF-same-priority-match2} \ \gamma \ ft \ p \neq \text{Undefined}$

proof

assume goal1: check-no-overlap $\gamma \ ft \ \text{OF-same-priority-match2} \ \gamma \ ft \ p = \text{Undefined}$

let $\forall a s = \{f. \ f \in \text{set ft} \land \gamma \ (\text{oefields f} p) \land (\forall f o \in \text{set ft.} \ \text{oef-prio f < oef-prio f o} \rightarrow \neg \gamma \ (\text{oefields f o}))\}$

have fin: finite $\forall a s$ by simp

note goal1(2)[unfolded OF-same-priority-match2-def]

then have $2 \leq \text{card} \ (\text{oef-action} \ f) \ \text{unfolding f-img-ex-set}$

unfolding Let-def

by (cases card (oef-action $\ f$) $\forall a s$, simp) (rename-tac nat1, case-tac nat1, simp add: image-Collect, presburger)

then have $2 \leq \text{card} \ f$ using card-image-le [OF fin, of oef-action] by linarith

then obtain a b where ab: a $\neq$ b a $\in$ $\forall a s b \in$ $\forall a s$ using card2-eI by blast

then have ab2: a $\in$ set ft $\gamma$ (oef-fields a p) $\gamma$ (oef-fields a p $<$ oef-prio p o $\rightarrow$ $\neg \gamma$ (oef-fields p)) b $\in$ set ft $\gamma$ (oef-fields b p) $\gamma$ (oef-fields b p $<$ oef-prio b p $\rightarrow$ $\neg \gamma$ (oef-fields p)) by simp-all

then have oef-prio a = oef-prio b by fastforce

note goal1(1)[unfolded check-no-overlap-def] ab2(1) ab2(4) this ab2(2) ab(1) ab2(5)

then show False by blast

qed

fun OF-match-linear :: (m::'m, p::'p) field-matcher $\Rightarrow$ (m, p) flowtable $\Rightarrow$ p $\Rightarrow$ 'a flowtable-behavior where

OF-match-linear [] $\cdot$ = NoAction |

OF-match-linear $\gamma$ (a $\neq$ a) p = (if $\gamma$ (oef-fields a) p then Action (oef-action a) else OF-match-linear $\gamma$ as p)

lemma OF-match-linear-ne-Undefined: OF-match-linear $\gamma \ ft \ p \neq \text{Undefined}$

by (induction ft) auto

lemma OF-match-linear-append: OF-match-linear $\gamma$ (a $\oplus$ b) p = (case OF-match-linear $\gamma$ a p of NoAction $\Rightarrow$ OF-match-linear $\gamma$ b p | x $\Rightarrow$ x)

by (induction a) simp-all

lemma OF-match-linear-match-allsameaction: $[\forall r \in \text{oms}\;; \gamma \ r \ p = \text{True}]$ $\Longrightarrow$ OF-match-linear $\gamma$ (map ($\lambda$x. split3 OFEntry (pri, x, act)) oms) p = Action act

by (induction oms) (auto simp add: split3-def)

lemma OF-lm-noa-none-iff: OF-match-linear $\gamma \ ft \ p = \text{NoAction} \iff (\forall e \in \text{set ft.} \ \neg \gamma \ (\text{oefields e} e) p)$

by (induction ft) (simp-all split: if-splits)

lemma set-eq-rule: ($\land x. \ x \in a \Longrightarrow x \in b$) $\Longrightarrow$ ($\land x. \ x \in b \Longrightarrow x \in a$) $\Longrightarrow$ a = b by (rule antisym[OF subset1 subset2])

lemma unmatching-insert-agnostic: $\neg \gamma$ (oefields a) p $\Longrightarrow$ OF-same-priority-match2 $\gamma$ (a $\neq$ ft) p = OF-same-priority-match2 $\gamma$ ft p

proof

let $\forall a s = \{f. \ f \in \text{set ft} \land \gamma \ (\text{oefields f} p) \land (\forall f o \in \text{set ft.} \ \text{oef-prio f < oef-prio f o} \rightarrow \neg \gamma \ (\text{oefields f o}))\}$

let $\forall a s = \{f. \ f \in \text{set (a $\neq$ ft) \land} \gamma \ (\text{oefields f} p) \land (\forall f o \in \text{set (a $\neq$ ft).} \ \text{oef-prio f < oef-prio f o} \rightarrow \neg \gamma \ (\text{oefields f o})\}$

assume nm: $\neg \gamma$ (oefields a) p

by blast
have aa: ?aas = ?as
proof (rule set-eq-rule)
  fix x
  assume x ∈ \{ f | f ∈ set (a ≠ ft) ∧ γ (ofe-fields f) p \∧ (\forall fo∈set (a ≠ ft). ofe-prio f < ofe-prio fo → ¬ γ (ofe-fields fo) p)\}
  hence as: x ∈ set (a ≠ ft) ∧ γ (ofe-fields x) p \∧ (\forall fo∈set (a ≠ ft). ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p) by simp
  with nm have x ∈ set ft by fastforce
  moreover from as have (\forall fo∈set ft. ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p) by simp
  ultimately show x ∈ \{ f ∈ set ft. γ (ofe-fields f) p \∧ (\forall fo∈set ft. ofe-prio f < ofe-prio fo → ¬ γ (ofe-fields fo) p)\} using as by force
next
  fix x
  assume x ∈ \{ f ∈ set ft. γ (ofe-fields f) p \∧ (\forall fo∈set ft. ofe-prio f < ofe-prio fo → ¬ γ (ofe-fields fo) p)\}
  hence as: x ∈ set ft γ (ofe-fields x) p \∧ (\forall fo∈set ft. ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p) by simp-all
  from as(1) have x ∈ set (a ≠ ft) by simp
  moreover from as(3) have (\forall fo∈set (a ≠ ft). ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p) using nm by simp
  ultimately show x ∈ \{ f | f ∈ set (a ≠ ft) ∧ γ (ofe-fields f) p \∧ (\forall fo∈set (a ≠ ft). ofe-prio f < ofe-prio fo → ¬ γ (ofe-fields fo) p)\} using as(2) by blast
qed

note uf = arg-cong[OF aa, of (') ofe-action, unfolded image-Collect]

show ?thesis unfolding OF-same-priority-match2-def using uf by presburger

qed

lemma OF-match-eq: sorted-descending (map ofe-prio ft) ⇒ check-no-overlap γ ft ⇒
OF-same-priority-match2 γ ft p = OF-match-linear γ ft p
proof (induction ft)
  case (Cons a ft)
  have 1: sorted-descending (map ofe-prio ft) using Cons(2) by simp
  have 2: check-no-overlap γ ft using Cons(3) unfolding check-no-overlap-def using set-subset-Cons by fast
  note mIH = Cons(1)(OF 1 2]
  show ?thesis is ?kees
  proof (cases (ofe-fields a) p)
  case False thus ?kees
    by (simp only: OF-match-linear.simps if False mIH[symmetric] unmatching-insert-agnostic[of γ, OF False])
next
  note sorted-descending-split[OF Cons(2)]
  then obtain m n where mn: a ≠ ft = m ⊗ n \∀ e∈set m. ofe-prio a = ofe-prio e \∀ e∈set n. ofe-prio e < ofe-prio a
  unfolding list.sel by blast
  hence aem: a ∈ set m
    by (metis UnE less_imp_neg list.intros(1) set.append)
  have mover: check-no-overlap γ m using Cons(3) unfolding check-no-overlap-def
    by (metis Un iff mn(1) set.append)
  let ?fc = \λ s.
    \{ f. f ∈ set s ∧ γ (ofe-fields f) p \∧ (\forall fo∈set (a ≠ ft). ofe-prio f < ofe-prio fo → ¬ γ (ofe-fields fo) p)\})
  case True
  have ?fc (m ⊗ n) = ?fc m ∪ ?fc n by auto
  moreover have ?fc n = {}
  proof (rule set-eq-rule, rule conatr, goal-cases)
    case (1 x)
    hence gl: x ∈ set n γ (ofe-fields x) p
      (\forall fo∈set m. ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p)
      (\forall fo∈set n. ofe-prio x < ofe-prio fo → ¬ γ (ofe-fields fo) p)
    qed
unfolding \text{mn}(1) \ by\ (\text{simp-all})
from \text{g1}(1) \ \text{mn}(3) \ have \ le: \ \text{ofe-prio} \ x < \ \text{ofe-prio} \ a \ \text{by} \ \text{simp}
\text{note} \ le \ g1(3) \ aem \ True
\text{then} \ show \ False \ \text{by} \ \text{blast}
\text{qed} \ \text{simp}
ultimately \ have \ cc: \ ?fc (m \ @ n) = ?fc m \ \text{by} \ \text{blast}
have cm: \ ?fc m = \{a\}
proof –
have \ \forall f \in \text{set} m. (\forall fo \in \text{set} \ (a \# ft). \ \text{ofe-prio} \ f < \ \text{ofe-prio} \ fo \ \rightarrow \ \neg \gamma (\text{ofe-fields} \ fo) \ p)
by \ (\text{metis} \ \text{UnE} \ \text{less-asym} \ \text{mn} \ \text{set-append})
hence 1: ?fc m = \{f \in \text{set} m. \ \gamma (\text{ofe-fields} \ f) \ p\} \ \text{by} \ \text{blast}
show \ \{f \in \text{set} m. \ \gamma (\text{ofe-fields} \ f) \ p \ \land \ (\forall fo \in \text{set} \ (a \# ft). \ \text{ofe-prio} \ f < \ \text{ofe-prio} \ fo \ \rightarrow \ \neg \gamma (\text{ofe-fields} \ fo) \ p)\} = \{a\}
unfolding 1
proof \ (\text{rule} \ \text{set-eq-rule,} \ \text{goal-cases} \ \text{fwd bwd})
case \ (\text{bwd} \ x)
have \ a \in \{f \in \text{set} m. \ \gamma (\text{ofe-fields} \ f) \ p\} \ \text{using} \ \text{True} \ \text{aem} \ \text{by} \ \text{simp}
thus \ ?case \ \text{using} \ \text{bwd} \ \text{by} \ \text{simp}
next
case \ (\text{fwd} \ x) \ show \ ?case \ \text{proof} \ (\text{rule} \ \text{ccontr})
assume \ x \ / \in \{a\} \ hence: \ x \neq a \ \text{by} \ \text{simp}
from \ \text{fwd} \ \text{have} \ 1: x \in \text{set} m \ \gamma (\text{ofe-fields} \ x) \ p \ \text{by} \ \text{simp-all}
have 2: \ \text{ofe-prio} \ x = \ \text{ofe-prio} \ a \ \text{using} \ 1(1) \ \text{mn}(2) \ \text{by} \ \text{simp}
show \ False \ \text{using} \ 1 \ \text{ne} \ \text{mover} \ \text{aem} \ True \ 2 \ \text{unfolding} \ \text{check-no-overlap-def} \ \text{by} \ \text{blast}
\text{qed}
\text{qed}
show \ ?kees
unfolding \ \text{mn}(1)
unfolding \ \text{OF-same-priority-match2-def}
unfolding \ \text{f-Img-ex-set}
unfolding \ \text{cc[unfolded \ mn(1)]}
unfolding \ \text{cm[unfolded \ mn(1)]}
unfolding \ \text{Let-def}
by \ (\text{simp only:} \ \text{mn}(1)[\text{symmetric}] \ \text{OF-match-linear}.\text{simps True if-True, simp})
\text{qed}
\text{qed} \ (\text{simp add:} \ \text{OF-same-priority-match2-def})

\text{lemma} \ \text{overlap-sort-invar[simp]}: \ \text{check-no-overlap} \ \gamma (\text{sort-descending-key} \ k \ ft) = \ \text{check-no-overlap} \ \gamma \ ft
\text{unfolding} \ \text{check-no-overlap-def}
\text{unfolding} \ \text{sort-descending-set-inv}
.. 

\text{lemma} \ \text{OF-match-eq2}:
\text{assumes} \ \text{check-no-overlap} \ \gamma \ ft
\text{shows} \ \text{OF-same-priority-match2} \ \gamma \ ft \ p = \ \text{OF-match-linear} \ \gamma (\text{sort-descending-key} \ \text{ofe-prio} \ ft) \ p
\text{proof –}
\text{have} \ \text{sorted-descending} \ (\text{map} \ \text{ofe-prio} \ (\text{sort-descending-key} \ \text{ofe-prio} \ ft)) \ \text{by} \ (\text{simp add:} \ \text{sorted-descending-sort-descending-key})
\text{note} \ \text{ceq} = \ \text{OF-match-eq2}[\text{OF this, unfolded overlap-sort-invar,} \ \text{OF} \ \text{check-no-overlap} \ \gamma \ ft, \ \text{symmetric}]
\text{show} \ ?thesis
\text{unfolding} \ \text{ceq}
\text{unfolding} \ \text{OF-same-priority-match2-def}
\text{unfolding} \ \text{sort-descending-set-inv}
..
\begin{proof}
\end{proof}

**lemma** \texttt{prio-match-matcher-alt}: \( \{ f.\ f \in \text{set flow-entries} \land \gamma (\text{ofe-fields} f) \text{ packet} \land \\
(\forall f_i \in \text{set flow-entries}. \ ofe-prio f_i > ofe-prio f \rightarrow \lnot \gamma (\text{ofe-fields} f_i) \text{ packet}) \} \)

\begin{enumerate}
\item let matching = \( \{ f.\ f \in \text{set flow-entries} \land \gamma (\text{ofe-fields} f) \text{ packet} \} \)
\item in \( \{ f.\ f \in \text{matching} \land (\forall f_i \in \text{matching}. \ ofe-prio f_i \leq ofe-prio f) \} \)
\end{enumerate}

\begin{proof}
\end{proof}

**lemma** \texttt{prio-match-matcher-alt2}: \( \{ f.\ f \in \text{matching} \land (\forall f_i \in \text{matching}. \ ofe-prio f_i \leq ofe-prio f) \} \)

\begin{proof}
\end{proof}

**definition** \texttt{OF-priority-match} where
\begin{align*}
\text{OF-priority-match} \gamma \text{ flow-entries packet} & \equiv \\
\text{let } m = & \text{filter} (\lambda f. \gamma (\text{ofe-fields} f) \text{ packet}) \text{ flow-entries}; \\
m' = & \text{filter} (\lambda f. (\forall f_i \in \text{set m}. \ ofe-prio f_i \leq ofe-prio f) \text{ m in} \\
\text{case } m' \text{ of} & \Rightarrow \text{NoAction} \\
\mid [s] & \Rightarrow \text{Action (ofe-action s)} \\
\mid - & \Rightarrow \text{Undefined}
\end{align*}

**definition** \texttt{OF-priority-match-ana} where
\begin{align*}
\text{OF-priority-match-ana} \gamma \text{ flow-entries packet} & \equiv \\
\text{let } m = & \text{filter} (\lambda f. \gamma (\text{ofe-fields} f) \text{ packet}) \text{ flow-entries}; \\
m' = & \text{filter} (\lambda f. (\forall f_i \in \text{set m}. \ ofe-prio f_i \leq ofe-prio f) \text{ m in} \\
\text{case } m' \text{ of} & \Rightarrow \text{NoAction} \\
\mid [s] & \Rightarrow \text{Action s} \\
\mid - & \Rightarrow \text{Undefined}
\end{align*}

**lemma** \texttt{filter-singleton}: \([x \leftarrow s. \ f x] = [y] \Rightarrow f y \land y \in \text{set s} \) by \( \text{metis filter-eq-Cons-iff in-set-conv-decomp} \)

**lemma** \texttt{OF-spm3-get-fe}: \( \text{OF-priority-match} \gamma \ ft \ p = \text{Action} a \Rightarrow \exists fe. \ ofe-action fe = a \land fe \in \text{set ft} \land \\
\text{OF-priority-match-ana} \gamma \ ft \ p = \text{Action} fe \)

\begin{proof}
\end{proof}

**fun** \texttt{no-overlaps} where
\begin{enumerate}
\item \texttt{no-overlaps - []} = True
\item \texttt{no-overlaps }\gamma (a\#as) = (\texttt{no-overlaps }\gamma as \land ( \\
\forall b \in \text{set as}. \ ofe-prio a = ofe-prio b \rightarrow \lnot(\exists p \in \text{UNIV}. \gamma (\text{ofe-fields} a) p \land \gamma (\text{ofe-fields} b) p)))
\end{enumerate}

**lemma** \texttt{no-overlap-Cons1}: \texttt{check-no-overlap2 }\gamma (x\#xs) \Rightarrow \texttt{check-no-overlap2 }\gamma xs 

\begin{proof}
\end{proof}

**lemma** \texttt{no-overlaps1}: \texttt{check-no-overlap }\gamma t \Rightarrow \texttt{distinct }t \Rightarrow \texttt{no-overlaps }\gamma t 

\begin{proof}
\end{proof}

**case** \texttt{(Cons a t)}
from no-overlap-ConsI[OF Cons(2)] Cons(3,1)

have no-overlaps γ t by simp 
thus ?case using Cons(2,3) unfolding check-no-overlap2-def by auto 
qed (simp add: check-no-overlap2-def)

lemma check-no-overlapI: no-overlaps γ t =⇒ check-no-overlap γ t

unfolding check-no-overlap-alt

proof(induction t)

case (Cons a t)

from Cons(1)[OF conjunct1[OF Cons(2)]unfolded no-overlaps.simps]]

show ?case using conjunct2[OF Cons(2)]unfolded no-overlaps.
simps]

unfolding check-no-overlap2-def by auto
qed (simp add: check-no-overlap2-def)

lemma (⋀ e p. e ∈ set t =⇒ ¬ γ (ofe-fields e) p) =⇒ no-overlaps γ t

by (induction t) simp-all

lemma no-overlaps-append: no-overlaps γ (x @ y) =⇒ no-overlaps γ y

by(induction x) simp-all

lemma no-overlaps-ne1: no-overlaps γ (x @ a # y @ b # z) =⇒ (∃ p. γ (ofe-fields a) p) ∨ (∃ p. γ (ofe-fields b) p)) =⇒ a ≠ b

proof (rule notI, goal-cases contr)

case contr

from contr(1) no-overlaps-append have no-overlaps γ (a # y @ b # z) by blast

note this[unfolded no-overlaps.simps]

with contr(3) have ¬ (∃ p∈UNIV. γ (ofe-fields a) p ∧ γ (ofe-fields b) p) by simp

with contr(2) show False unfolding contr(3) by simp

qed

lemma no-overlaps-defeq: no-overlaps γ fe =⇒ OF-same-priority-match2 γ fe p = OF-priority-match γ fe p

unfolding OF-same-priority-match2-def OF-priority-match-def

unfolding f-Img-ex-set

unfolding prio-match-matcher-alt

unfolding prio-match-matcher-alt2

proof (goal-cases uf)

case uf

let ?m' = let m = [f←fe . γ (ofe-fields f) p] in [f←m . ∀ fo∈set m. ofe-prio fo ≤ ofe-prio f]

let ?s = ofe-action ' set ?m'

from uf show ?case

proof(cases ?m')

case Nil

moreover then have card ?s = 0 by force

ultimately show ?thesis by(simp add: Let-def)

next

case (Cons a as)

have as = []

proof(rule ccontr)

assume as ≠ []

then obtain b bs where bs: as = b # bs by (meson neq-Nil-conv)

note no = Cons[unfolded Let-def filter-filter]

have f1: a ∈ set ?m' b ∈ set ?m' unfolding bs local.Cons by simp-all

hence ofe-prio a = ofe-prio b by (simp add: antisym)
moreover have \( ms \colon \gamma \) (ofe-fields \( a \)) \( p \) \( \gamma \) (ofe-fields \( b \)) \( p \) using no[symmetric] unfolding \( bbs \) by(blast dest: Cons-eq-filterD)+

moreover have \( a b \in \text{set} \text{fe} b \in \text{set} \text{fe} \) using \( f1 \) by auto

moreover have \( a \neq b \) proof(cases \( \exists x y z. \text{fe} = x \oplus a \# y \oplus b \# z \))
case True
then obtain \( x y z \) where \( \text{xyz} : \text{fe} = x \oplus a \# y \oplus b \# z \) by blast
from no-overlaps-ne1 \( ms(1) \) uf[unfolded \text{xyz}]
show ?thesis by blast
next
case False
then obtain \( x y z \) where \( \text{xyz} : \text{fe} = x \oplus b \# y \oplus a \# z \)
using no unfolding bbs by (metis (no-types, lifting) Cons-eq-filterD)
from no-overlaps-ne1 \( ms(1) \) uf[unfolded \text{xyz}]
show ?thesis by blast
qed
ultimately show False using check-no-overlapI[of uf, unfolded check-no-overlap-def] by blast
qed
then have \( oe : a \# as = [a] \) by simp
show ?thesis using Cons[unfolded oe] by force
qed
qed

lemma distinct \( \text{fe} \implies \text{check-no-overlap} \gamma \text{ fe} \implies \text{OF-same-priority-match2} \gamma \text{ fe} \text{ p} = \text{OF-priority-match} \gamma \text{ fe} \text{ p} \)
by(rule no-overlaps-defeq) (drule (2) no-overlapsI)

theorem \( \text{OF-eq} \);
assumes no: \( \text{no-overlaps} \gamma \text{ f} \)
and so: \( \text{sorted-descending} \ (\text{map} \ \text{ofe-prio} \ \text{f}) \)
shows \( \text{OF-match-linear} \ \gamma \text{ f} \text{ p} = \text{OF-priority-match} \gamma \text{ f} \text{ p} \)
unfolding no-overlaps-defeq[ symmetric,OF no] OF-match-eq[OF so check-no-overlapI[OF no]]

..  
corollary \( \text{OF-eq-sort} \);
assumes no: \( \text{no-overlaps} \gamma \text{ f} \)
shows \( \text{OF-priority-match} \ \gamma \text{ f} \text{ p} = \text{OF-match-linear} \ \gamma \text{ (sort-descending-key ofe-prio f) p} \)
using \( \text{OF-match-eq2} \) check-no-overlapI no no-overlaps-defeq by fastforce

lemma \( \text{OF-lm-noa-none} : \text{OF-match-linear} \ \gamma \text{ ft} \text{ p} = \text{NoAction} \implies \forall e \in \text{set} \text{ ft}. \neg \gamma \ (\text{ofe-fields} \ e) \text{ p} \)
by(induction ft) (simp-all split: if-splits)

lemma \( \text{OF-spm3-noa-none} : \)
assumes no: \( \text{no-overlaps} \gamma \text{ ft} \)
shows \( \text{OF-priority-match} \ \gamma \text{ ft} \text{ p} = \text{NoAction} \implies \forall e \in \text{set} \text{ ft}. \neg \gamma \ (\text{ofe-fields} \ e) \text{ p} \)
unfolding \( \text{OF-eq-sort}[OF no] \) by(drule \( \text{OF-lm-noa-none} \) simp

lemma no-overlaps-not-undefined: \( \text{no-overlaps} \ \gamma \text{ ft} \implies \text{OF-priority-match} \ \gamma \text{ ft} \text{ p} \neq \text{Undefined} \)
using check-no-overlapI no-overlaps-not-undefined no-overlaps-defeq by fastforce

end

theory OpenFlow-Serialize
imports OpenFlow-Matches
begin

definition serialization-test-entry ≡ OFEntry 7 {EtherDst 0x1, IPv4Dst (PrefixMatch 0xA000201 32), IngressPort "s1-lan", L4Dst 0x50 0, L4Src 0x400 0x3FF, IPv4Proto 6, EtherType 0x800} [ModifyField-l2dst 0xA641F185E862, Forward "s1-wan"]

definition serialize-mac (m::hex-string-of-word 3 m) = (case m of
  ofe-action serialization-test-entry ⇒ "de:ad:be:ef:ca:fe" by eval

lemma serialize-mac 0xdeadbeefcafe = "de:ad:be:ef:ca:fe" by eval

lemma serialize-actions pids a ≡ case a of
  Forward oif ⇒ "output:" @ pids oif |
  ModifyField-l2dst na ⇒ "mod-dl-dst:" @ serialize-mac na

lemma serialize-actions anything [] = "drop" by(simp add: serialize-actions-def)


value (map ((<>) (1::8 word) ∘ ((*) 8) ∘ rev) [0..<6])

value serialize-mac (m::hex-string-of-word 3 m) ≡ (intersperse (CHR "-"::CHR) ∘ map (hex-string-of-word 1 ∘ (λh. (m >> h * 8) & & (0xff)) ∘ rev) [0..<6])

lemma serialize-mac 0xdeabbeefcafe = "de:ad:be:ef:ca:fe" by eval

lemma serialize-actions pids a ≡ if length a = 0 then "drop" else (intersperse (CHR ","::CHR) ∘ map (serialize-action pids)) a


lemma serialize-actions anything [] = "drop" by(simp add: serialize-actions-def)

definition prefix-to-string pfx ≡ ipv4-cidr-toString (pfxm-prefix pfx, pfxm-length pfx)

primrec serialize-of-match where
  serialize-of-match (IngressPort p) = "in-port=" @ pids p |
  serialize-of-match - (VlanId i) = "dl-vlan=" @ dec-string-of-word0 i |
  serialize-of-match - (VlanPriority -) = undefined |
  serialize-of-match - (EtherType i) = "dl-type=0x" @ hex-string-of-word0 i |
  serialize-of-match - (EtherSrc m) = "dl-src=" @ serialize-mac m |
  serialize-of-match - (EtherDst m) = "dl-dst=" @ serialize-mac m |
  serialize-of-match - (IPv4Proto i) = "nw-proto=" @ dec-string-of-word0 i |
  serialize-of-match - (IPv4Src p) = "nw-src=" @ prefix-to-string p |
  serialize-of-match - (IPv4Dst p) = "nw-dst=" @ prefix-to-string p |
  serialize-of-match - (L4Src i m) = "lp-src=" @ dec-string-of-word0 i @ (if m = max-word then [] else "/0x" @ hex-string-of-word3 m) |
  serialize-of-match - (L4Dst i m) = "lp-dst=" @ dec-string-of-word0 i @ (if m = max-word then [] else "/0x" @ hex-string-of-word3 m)

definition serialize-of-matches :: (string ⇒ string) ⇒ of-match-field set ⇒ string
  where
    serialize-of-matches pids ≡ (λx."hard-timeout=0,idle-timeout=0," ∘ intersperse (CHR ","::CHR) ∘ map (serialize-of-match pids) ∘ sorted-list-of-set)

lemma serialize-of-matches pids of-matches=
  (List.append "hard-timeout=0,idle-timeout=0,")


(intersperse (CHR ".") (map (serialize-of-match pids) (sorted-list-of-set-of-matches)))
by (simp add: serialize-of-matches-def)

export-code serialize-of-matches checking SML

lemma serialize-of-matches (lofif "42") (ofe-fields serialization-test-entry) =
"hard-timeout=0, idle-timeout=0, in-port=42, dl-type=0x800, dl-dst=00:00:00:00:00:01, nw-proto=6, nw-dst=10.0.2.1/32, tp-src=1024"
by eval

definition serialize-of-entry pids e (case e of (OFEntry p f a) ⇒ "priority=" @ dec-string-of-word0 p @ "," @ serialize-of-matches pids f @ "," @ "action=" @ serialize-actions pids a)

lemma serialize-of-entry (the o map-of [("s1-lan", "42"),("s1-wan", "1337")]) serialization-test-entry =
"priority=7, hard-timeout=0, idle-timeout=0, in-port=42, dl-type=0x800, dl-dst=00:00:00:00:00:01, nw-proto=6, nw-dst=10.0.2.1/32, tp-src=1024"
by eval

end

theory Featherweight-OpenFlow-Comparison

imports Semantics-OpenFlow

begin

inductive guha-table-semantics :: ("m", "p") field-matcher ⇒ ("m", "a") flowtable ⇒ "p" ⇒ "a option ⇒ bool where

| guha-matched: γ (ofe-fields fe) p = True ⇒
| ∀ fe' ∈ set (ft1 @ ft2). ofe-prio fe' > ofe-prio fe ⇒ γ (ofe-fields fe') p = False ⇒
| guha-table-semantics γ (ft1 @ fe # ft2) p (Some (ofe-action fe)) |

| guha-unmatched: ∀ fe ∈ set ft. γ (ofe-fields fe) p = False ⇒
| guha-table-semantics γ ft p None |

lemma guha-table-semantics-ex2res:

| assumes ms: CARD("a") ≥ 2
| assumes to: CARD("a") ≥ 2
| assumes ms: CARD("a") ≥ 2

| shows ∃ ff. γ ff p
| shows ∃ ft (a1 :: "a" (a2 :: "a"). a1 ≠ a2 ∧ guha-table-semantics γ ft p (Some a1) ∧ guha-table-semantics γ ft p (Some a2)

| proof –
| from ms obtain ff where: m. γ ff p .
| from to obtain a1 a2 :: "a where as: a1 ≠ a2 using card2-e1 by blast
| let ?fe1 = OFEntry 0 ff a1
| let ?fe2 = OFEntry 0 ff a2
| let ?ft = [?fe1, ?fe2]
| have guha-table-semantics γ ?ft p (Some a1) guha-table-semantics γ ?ft p (Some a2)
| by (rule guha-table-semantics.intros(1)[of γ ?fe1 p [] ?fe2, unfolded append-Nil flow-entry-match.set] |
| rule guha-table-semantics.intros(1)[of γ ?fe2 p [?fe1] [], unfolded append-Nil2 flow-entry-match.set append.simps] |
| simp add: m)+
| thus ?thesis using as by (intro exI conjI)

qed

lemma guha-amstaendlich:

| assumes ae: a = ofe-action fe
| assumes elec: fe ∈ set ft
| assumes rest: γ (ofe-fields fe) p
| ∀ fe' ∈ set ft. ofe-prio fe' > ofe-prio fe ⇒ γ (ofe-fields fe') p

| shows guha-table-semantics γ ft p (Some a)
proof

from ele obtain ft1 ft2 where ftsp1: ft = ft1 @ fe # ft2 using split-list by fastforce
show ?thesis unfolding ae ftsp1
  apply (rule guha-table-semantics.intros(1))
  using rest(1) apply (simp)
  using rest(2)[unfolded ftsp1] apply (simp)
done

qed

lemma guha-matched-rule-inversion:
  assumes guha-table-semantics γ ft p (Some a)
  shows ∃fe ∈ set ft. a = ofe-action fe ∧ γ (ofe-fields fe) p ∧ (∀fe' ∈ set ft. ofe-prio fe' > ofe-prio fe → ¬γ (ofe-fields fe')) p)
proof −

{ fix d
  assume guha-table-semantics γ ft p d
  hence Some a = d → (∃fe ∈ set ft. a = ofe-action fe ∧ γ (ofe-fields fe) p ∧ (∀fe' ∈ set ft. ofe-prio fe' > ofe-prio fe → ¬γ (ofe-fields fe')) p)
  by (induction rule: guha-table-semantics.induct) simp-all
}
from this[OF assms refl]
show ?thesis .

qed

lemma guha-equal-Action:
  assumes no: no-overlaps γ ft
  assumes spm: OF-priority-match γ ft p = Action a
  shows guha-table-semantics γ ft p (Some a)
proof −

  show ?thesis
  apply (rule guha-amstaendlich)
  apply (rule a[symmetric])
  apply (rule fein)
  using feana unfolding OF-priority-match-ana-def
  apply (auto dest!: filter-singleton split: list.splits)
done

qed

lemma guha-equal-NoAction:
  assumes no: no-overlaps γ ft
  assumes spm: OF-priority-match γ ft p = NoAction
  shows guha-table-semantics γ ft p None
using spm unfolding OF-priority-match-def
by (auto simp add: filter-empty-cone OF-spm3-noa-none[OF no spm] intro: guha-table-semantics.intros(2) split: list.splits)

lemma guha-equal-hlp:
  assumes no: no-overlaps γ ft
  shows guha-table-semantics γ ft p (ftb-to-option (OF-priority-match γ ft p))
unfolding ftb-to-option-def
apply (cases (OF-priority-match γ ft p))
apply (simp add: guha-equal-Action[OF no])
apply(simp add: guha-equal-NoAction[OF no])
apply(subgoal-tac False, simp)
apply(simp add: no no-overlaps-not-undefined)
done

lemma guha-deterministic1: guha-table-semantics γ ft p (Some x1) ⇒¬ guha-table-semantics γ ft p None
by(auto simp add: guha-table-semantics.simps)

lemma guha-deterministic2: [no-overlaps γ ft; guha-table-semantics γ ft p (Some x1); guha-table-semantics γ ft p (Some a)] ] ⇒ x1 = a
proof(rule ccontr, goal-cases)
case 1
note 1(2−3)[THEN guha-matched-rule-inversion] then obtain fe1 fe2 where fes:
fe1∈set ft x1 = ofe-action fe1 γ (ofe-fields fe1) p (∀ fe∈set ft. ofe-prio fe1 < ofe-prio fe′ ←→ γ (ofe-fields fe′) p)
fe2∈set ft a = ofe-action fe2 γ (ofe-fields fe2) p (∀ fe′∈set ft. ofe-prio fe2 < ofe-prio fe′ ←→ γ (ofe-fields fe′) p)
by blast
from ⟨x1 ≠ a⟩ have fene: fe1 ≠ fe2 using fes
have pe: ofe-prio fe1 = ofe-prio fe2 using fes(1,3−4,5,7−8) less-linear by blast
note ⟨no-overlaps γ ft⟩[THEN check-no-overlapI, unfolded check-no-overlap-def]
thus False by blast
qed

lemma guha-equal:
assumes no: no-overlaps γ ft
shows OF-priority-match γ ft p = option-to-ftb d ←→ guha-table-semantics γ ft p d
using guha-equal-hlp[OF no, of p] unfolding ftb-to-option-def option-to-ftb-def
apply(cases OF-priority-match γ ft p; cases d)
apply(simp-all)
using guha-deterministic1 apply fast
using guha-deterministic2[OF no] apply blast
using guha-deterministic1 apply fast
using no-overlaps-not-undefined[OF no] apply fastforce
using no-overlaps-not-undefined[OF no] apply fastforce
done

lemma guha-nondeterministicD:
assumes ¬check-no-overlap γ ft
shows ∃ fe1 fe2 p. fe1 ∈ set ft ∧ fe2 ∈ set ft
∧ guha-table-semantics γ ft p (Some (ofe-action fe1))
∧ guha-table-semantics γ ft p (Some (ofe-action fe2))
using assms
apply(unfold check-no-overlap-def)
apply(clarsimp)
apply(rename-tac fe1 fe2 p)
apply(rule_tac x = fe1 in exI)
apply(simp)
apply(rule_tac x = fe2 in exI)
apply(simp)
apply(rule_tac x = p in exI)
apply(rule conjI)
apply(subst guha-table-semantics.simps)
apply(rule disjI1)

16
apply(clarsimp)
apply(rule-tac x = fe1 in exI)
apply(drule split-list)
apply(clarify)
apply(rename-tac ft1 ft2)
apply(rule-tac x = ft1 in exI)
apply(rule-tac x = ft2 in exI)
apply(simp)
oops

The above lemma does indeed not hold, the reason for this are (possibly partially) shadowed overlaps. This is exemplified below: If there are at least three different possible actions (necessary assumption) and a match expression that matches all packets (convenience assumption), it is possible to construct a flow table that is admonished by check-no-overlap but still will never run into undefined behavior.

lemma
assumes CARD('action) ≥ 3
assumes ∀ p. γ x p
shows ∃ ft :: ('a, 'action) flow-entry-match list. ¬check-no-overlap γ ft ∧ ¬(∃ fe1 fe2 p. fe1 ∈ set ft ∧ fe2 ∈ set ft ∧ fe1 ≠ fe2 ∧ ofe-prio fe1 = ofe-prio fe2 ∧ guha-table-semantics γ ft p (Some (ofe-action fe1))) ∧ guha-table-semantics γ ft p (Some (ofe-action fe2)))

proof −

obtain adef aa ab :: 'action where anb[simp]: aa ≠ ab adef ≠ aa adef ≠ ab using assms(1) card3-eI by blast

let ?cex = [OFEntry 1 x adef, OFEntry 0 x aa, OFEntry 0 x ab]

have ol: ¬check-no-overlap γ ?cex
unfolding check-no-overlap-def ball-simps
apply(rule bexI [where x = OFEntry 0 x aa, rotated], (simp; fail))
apply(rule bexI [where x = OFEntry 0 x ab, rotated], (simp; fail))
apply(simp add: assms)
done

have df: guha-table-semantics γ ?cex p oc ⇒ oc = Some adef for p oc
unfolding guha-table-semantics.simps
apply(eqm d(assms simp: assms))
subgoal for fe ft1 ft2
apply(cases ft1 = [])
apply(fastforce)
apply(cases ft2 = [])
apply(fastforce)
apply(subgoal-tac ft1 = [OFEntry 1 x adef] ∧ fe = OFEntry 0 x aa ∧ ft2 = [OFEntry 0 x ab])
apply(simp; fail)
apply(clarsimp simp add: List.neq-Nil-conv)
apply(rename-tac ya ys yz)
apply(case-tac ys; clarsimp simp add: List.neq-Nil-conv)
done done

show ?thesis
apply(intro exI[where x = ?cex], intro conjI, fact ol)
apply(clarify)
apply(unfold set.simps)
apply(eqm d(assms simp: assms))
apply(eqm d(assms simp: assms))
done

qed
theory LinuxRouter-OpenFlow-Translation
imports IP-Addresses.CIDR-Split
  Automatic-Refinement.Misc
  Simple-Firewall.Generic-SimpleFw
  Semantics-OpenFlow
  OpenFlow-Matches
  OpenFlow-Action
  Routing/Linux-Router
begin

hide-const Misc.uncurry
hide-fact Misc.uncurry-def

definition route2match r =
  \(\{\text{iface} = \text{ifaceAny}, \text{iface} = \text{ifaceAny},\)\
  \(\text{src} = (0,0), \text{dst} = (\text{pfxm-prefix (routing-match r)}, \text{pfxm-length (routing-match r)}),\)\
  \(\text{proto} = \text{ProtoAny}, \text{sports} = (0,\text{max-word}), \text{ports} = (0,\text{max-word})\)\)

definition toprefixmatch where
  toprefixmatch m \equiv (let pm = PrefixMatch (fst m) (snd m) in if pm = PrefixMatch 0 0 then None else Some pm)

lemma prefix-match-semantics-simple-match:
  assumes some: toprefixmatch m = Some pm
  assumes vld: valid-prefix pm
  shows prefix-match-semantics pm = simple-match-ip m
  using some
  by (cases m)
    NOT-mask-shifted-lenword[symmetric]
    split: if-splits)

definition simple-match-to-of-match-single ::
  32 simple-match \Rightarrow string list \Rightarrow of-match-field set
  where
  simple-match-to-of-match-single m ifs
  \equiv (let
    npm = (\lambda p. \text{fst} p = 0 \wedge \text{snd} p = \text{max-word});
    sb = (\lambda p. \text{if} npm p \text{ then } \text{[None]} \text{ else if } \text{fst} p \leq \text{snd} p \text{ then map } (\text{Some } (\lambda pfs. (\text{pfxm-prefix pfs}, \text{NOT} \text{pfxm-mask pfs})))
      \text{(wordinterval-CIDR-split-prefixmatch (WordInterval (fst p) (snd p))) else []})
    in \text{simple-match-to-of-match-single m ifs (proto m) sport dport .}
    iif \leftarrow (\text{if iiface m = ifaceAny then } \text{[None]} \text{ else } \text{[Some i. i \leftarrow ifs, match-iiface (iiface m) i]}),
    sport \leftarrow sb (sports m),
    dport \leftarrow sb (dports m))
lemma smtoms-eq-hlp: simple-match-to-of-match-single r a b c d = simple-match-to-of-match-single r f g h i ⊆ → (a = f ∧ b = g ∧ c = h ∧ d = i)

proof(rule iffI, goal-cases)
  case 1
  thus ?case proof(intro conjI)
    have *: ∃P x. [∀x :: of-match-field. P x; z = Some x] ⇒ P (IngressPort x) by simp
    show a = f using 1 by(cases a; cases f)
      (simp add: option2set-None simple-match-to-of-match-single-def toprefixmatch-def option2set-def;
       subst(asm) set-eq-iff; drule (1) *; simp split: option.splits uncurry-splits protocol.splits)+
  next
    have *: ∃P x. [∀x :: of-match-field. P x; z = Proto x] ⇒ P (IPv4Proto x) by simp
    show b = g using 1 by(cases b; cases g)
      (simp add: option2set-None simple-match-to-of-match-single-def toprefixmatch-def option2set-def;
       subst(asm) set-eq-iff; drule (1) *; simp split: option.splits uncurry-splits protocol.splits)+
  next
    have *: ∃P x. [∀x :: of-match-field. P x; z = Some x] ⇒ P (uncurry L4Src x) by simp
    show c = h using 1 by(cases c; cases h)
      (simp add: option2set-None simple-match-to-of-match-single-def toprefixmatch-def option2set-def;
       subst(asm) set-eq-iff; drule (1) *; simp split: option.splits uncurry-splits protocol.splits)+
  next
    have *: ∃P x. [∀x :: of-match-field. P x; z = Some x] ⇒ P (uncurry L4Dst x) by simp
    show d = i using 1 by(cases d; cases i)
      (simp add: option2set-None simple-match-to-of-match-single-def toprefixmatch-def option2set-def;
       subst(asm) set-eq-iff; drule (1) *; simp split: option.splits uncurry-splits protocol.splits)+
  qed
qed simp

lemma simple-match-to-of-match-generates-prereqs: simple-match-valid m ⇒ r ∈ set (simple-match-to-of-match m ifs) ⇒ all-prerequisites r
unfolding simple-match-to-of-match-def Let-def
proof(clarsimp, goal-cases)
  case 1 xiface xsrcp xdstp
  note o = this
  show ?case unfolding simple-match-to-of-match-single-def all-prerequisites-def
    unfolding ball-un
    proof((intro conjI; ((simp:fail)| - )), goal-cases)
      case 1
      have e: (∃f (sports m) = 0 ∧ snd (sports m) = max-word) ∨ proto m = Proto TCP ∨ proto m = Proto UDP ∨ proto m = Proto L4-Protocol.SCTP
        using o(1)
        unfolding simple-match-valid-alt Let-def
        by(clarsimp split: if-splits)
      show ?case
        using o(3) e
        by(elim disjE; simp add: option2set-def split: if-splits prod.splits uncurry-splits)
      next
      case 2
      have e: (∃f (dports m) = 0 ∧ snd (dports m) = max-word) ∨ proto m = Proto TCP ∨ proto m = Proto UDP ∨ proto m = Proto L4-Protocol.SCTP
        using o(1)
unfolding simple-match-valid-alt Let-def
by (clarsimp split: if-splits)
show ?case
using (4) e
by (elim disjE; simp add: option2set-def split: if-splits prod.splits uncurry-splits)
qed
qed

lemma and-assoc: a ∧ b ∧ c ↔ (a ∧ b) ∧ c by simp

lemmas custom-simpset = Let-def set-concat set-map map-map comp-def concat-map-maps set-maps UN-iff fun-app-def Set.image-iff

abbreviation simple-fw-prefix-to-wordinterval ≡ prefix-to-wordinterval ◦ uncurry PrefixMatch

lemma simple-match-port-alt: simple-match-port m p ↔ p ∈ wordinterval-to-set (uncurry WordInterval m) by (simp split: uncurry-splits)

lemma simple-match-src-alt: simple-match-valid r =⇒ simple-match-ip (src r) p ↔ prefix-match-semantics (PrefixMatch (fst (src r)) (snd (src r))) p by (cases (src r)) (simp add: prefix-match-semantics-ipset-from-netmask2 prefix-to-wordset-ipset-from-cidr simple-match-valid-def valid-prefix-fw-def)

lemma simple-match-dst-alt: simple-match-valid r =⇒ simple-match-ip (dst r) p ↔ prefix-match-semantics (PrefixMatch (fst (dst r)) (snd (dst r))) p by (cases (dst r)) (simp add: prefix-match-semantics-ipset-from-netmask2 prefix-to-wordset-ipset-from-cidr simple-match-valid-def valid-prefix-fw-def)

lemma x ∈ set (wordinterval-CIDR-split-prefixmatch w) =⇒ valid-prefix x using wordinterval-CIDR-split-prefixmatch-all-valid-Ball[THEN bspec, THEN conjunct1].

lemma simple-match-to-of-matchI:
assumes mv: simple-match-valid r
assumes mm: simple-matches r p
assumes ii: p-iiface p ∈ set ifs
assumes ippkt: p-l2type p = 0x800
shows eq: ∃ gr ∈ set (simple-match-to-of-match r ifs). OF-match-fields gr p = Some True
proof −
let ?npm = λp. fst p = 0 ∧ snd p = max-word
let ?sb = λp. (if ?npm p then None else Some r)
obtain si where si: case si of Some ssi ⇒ p-sport p ∈ prefix-to-wordset ssi | None ⇒ True
  case si of None ⇒ True | Some ssi ⇒ ssi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r)))
  si = None if-then-else ?npm (sports r)
proof (cases ?npm (sports r), goal-cases)
  case 1
  hence (case None of None ⇒ True | Some ssi ⇒ p-sport p ∈ prefix-to-wordset ssi) ∧
    (case None of None ⇒ True | Some ssi ⇒ ssi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r)))) by simp
  with t show ?thesis by blast
next
  case 2
from mm have p-sport p ∈ wordinterval-to-set (uncurry WordInterval (sports r))
by (simp only: simple-matches.simps simple-match-port-alt)
then obtain ssi where ssi:
let `?mf = map-option (apsnd (wordNOT ∘ mask ∘ (−) 16) ∘ prefix-match-dtor)
let `?gr = simple-match-to-of-match-single r
(if iface r = ifaceAny then None else Some (p-iface p))
(if proto r = ProtoAny then ProtoAny else Proto (p-proto p))
(?mf si) (?mf di)

proof
let `mfu = simple-match-port.simps[of fst (sports r) snd (sports r), unfolded surjective-pairing[of sports r,symmetric]]
  simple-match-port.simps[of fst (dports r) snd (dports r), unfolded surjective-pairing[of dports r,symmetric]]
let `u = `mm[unfolded simple-matches.simps mfu ord-class.atLeastAtMost-iff simple-packet-unext-def simple-packet.simps]

next

case 2

from `mm have p-dport p ∈ wordinterval-to-set (uncurry WordInterval (dports r))
  by(simp only: simple-matches.simps simple-match-port-alt)
then obtain ddi where ddi:
  ddi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))
  p-dport p ∈ prefix-to-wordset ddi
using wordinterval-CIDR-split-existential by fast

hence (case Some ddi of None ⇒ True | Some dsi ⇒ p-sport p ∈ prefix-to-wordset dsi) ∧
  (case Some ddi of None ⇒ True
   | Some dsi ⇒ dsi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r))))
by simp

with 2 show ?thesis by blast

qed

show ?thesis

proof
let `gr = simple-match-to-of-match-single r

let ?gr = simple-match-to-of-match-single r
(if iface r = ifaceAny then None else Some (p-iface p))
(if proto r = ProtoAny then ProtoAny else Proto (p-proto p))
(?mf si) (?mf di)

note mfu = simple-match-port.simps[of fst (sports r) snd (sports r), unfolded surjective-pairing[of sports r,symmetric]]
  simple-match-port.simps[of fst (dports r) snd (dports r), unfolded surjective-pairing[of dports r,symmetric]]
note `u = `mm[unfolded simple-matches.simps mfu ord-class.atLeastAtMost-iff simple-packet-unext-def simple-packet.simps]

note of-safe-unsafe-match-eq[OF simple-match-to-of-match-generates-prereqs]

from `mm have p: fst (sports r) ≤ snd (sports r) fst (dports r) ≤ snd (dports r) by force+

show eq: `?gr ∈ set (simple-match-to-of-match r tfs)
  unfolding simple-match-to-of-match-def
  unfolding custom-simpset
  unfolding snotms-eq-blp
  proof(intro bezl. (intro conjl: (rule refl?!)), goal-cases)
  case 2 thus ?case using `ple(2) di
    apply(simp add: pfzm-mask-def prefix-match-dtor-def Set.image-iff split: option.splits prod.splits uncurry-splits)
    apply(erule bezl[rotated])
    apply(simp split: prefix-match.splits)
  done
next

\begin{verbatim}
case 3 thus \text{?case using ple(1) si}
  apply(simp add: pfzm-mask-def prefix-match-dtor-def Set.image-iff
          split: option.splits prod.splits uncurry.splits)
  apply(rule bexl[rotated])
  apply(simp split: prefix-match.splits)
  done
next

case 4 thus \text{?case}
  using u ii by(clarsimp simp: set-maps split: if-splits)
next

case 1 thus \text{?case using ii u by simp-all (metis match-proto.elims(2))}
qed
\end{verbatim}

have \text{dpm: } di = Some \text{(PrefixMatch } x1 x2) \implies p-dport p \&\& \sim\sim mask (16 - x2) = x1 \text{ for } x1 x2

\begin{verbatim}
  proof -
    have \text{*: } di = Some \text{(PrefixMatch } x1 x2) \implies prefix-match-semantics \text{(the di) } (p-dport p) \implies p-dport p \&\& \sim\sim mask (16 - x2) = x1 
      by(clarsimp simp: prefix-match-semantics-def pfzm-mask-def word-bw-comms;fail)
    have \text{**: } pfx \in set \text{(wordinterval-CIDR-split-prefixmatch ra)} \implies prefix-match-semantics pfx a = (a \in prefix-to-wordset pfx) \text{ for pfx ra and a :: 16 word}
      by (fact prefix-match-semantics-wordset[OF wordinterval-CIDR-split-prefixmatch-all-valid-Ball[THEN bspec, THEN conjunct1]])
    have \text{[di = Some \text{(PrefixMatch } x1 x2); \text{p-dport p } \in \text{prefix-to-wordset} \text{(PrefixMatch } x1 x2); \text{PrefixMatch } x1 x2 \in set \text{(wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))}]}
      \implies \text{p-dport p } \&\& \sim\sim mask (16 - x2) = x1
      using di(1,2)
      using ** by auto
    thus \text{di = Some \text{(PrefixMatch } x1 x2) \implies p-dport p } \&\& \sim\sim mask (16 - x2) = x1 \text{ using di(1,2) by auto}
    qed

have \text{spm: } si = Some \text{(PrefixMatch } x1 x2) \implies p-sport p \&\& \sim\sim mask (16 - x2) = x1 \text{ for } x1 x2

\begin{verbatim}
  using si
  proof -
    have \text{*: } si = Some \text{(PrefixMatch } x1 x2) \implies prefix-match-semantics \text{(the si) } (p-sport p) \implies p-sport p \&\& \sim\sim mask (16 - x2) = x1 
      by(clarsimp simp: prefix-match-semantics-def pfzm-mask-def word-bw-comms;fail)
    have \text{**: } pfx \in set \text{(wordinterval-CIDR-split-prefixmatch ra)} \implies prefix-match-semantics pfx a = (a \in prefix-to-wordset pfx) \text{ for pfx ra and a :: 16 word}
      by (fact prefix-match-semantics-wordset[OF wordinterval-CIDR-split-prefixmatch-all-valid-Ball[THEN bspec, THEN conjunct1]])
    have \text{[si = Some \text{(PrefixMatch } x1 x2); \text{p-sport p } \in \text{prefix-to-wordset} \text{(PrefixMatch } x1 x2); \text{PrefixMatch } x1 x2 \in set \text{(wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r)))}]}
      \implies \text{p-sport p } \&\& \sim\sim mask (16 - x2) = x1
      using si(1,2)
      using ** by auto
    thus \text{si = Some \text{(PrefixMatch } x1 x2) \implies p-sport p } \&\& \sim\sim mask (16 - x2) = x1 \text{ using si(1,2) by auto}
    qed
\end{verbatim}

show \text{OF-match-fields ?gr p = Some True}

\begin{verbatim}
unfolding \text{of-safe-unsafe-match-eq[OF simple-match-to-of-match-generates-prereqs[OF mv eg]]}
by(cases si; cases di)
\end{verbatim}

\begin{verbatim}
  (simp-all
    add: simple-match-to-of-match-single-def OF-match-fields-unsafe-def spm
    option2set-def u ipkht prefix-match-dtor-def toprefixmatch-def dpm
    simple-match-dst-alt[OF mv, symmetric] simple-match-src-alt[OF mv, symmetric]
    split: prefix-match.splits)
\end{verbatim}

22
lemma prefix-match-00 [simp, intro!]: prefix-match-semantics (PrefixMatch 0 0) p

proof -
from mv have validpfx:
valid-prefix (uncurry PrefixMatch (src r))
valid-prefix (uncurry PrefixMatch (dst r))
\[\forall pm. \text{toprefixmatch (src r)} = \text{Some pm} \implies \text{valid-prefix pm}\]
\[\forall pm. \text{toprefixmatch (dst r)} = \text{Some pm} \implies \text{valid-prefix pm}\]

unfolding simple-match-valid-def

by (simp-all split : uncurry-splits if-splits)

from mo have mo:
OF-match-fields-unsafe gr p

unfolding of-safe-unsafe-match-eq

\[\OF simple-match-to-of-match-generates-prereqs[\OF mv eg]\]

by simp

note this [unfolded OF-match-fields-unsafe-def]

note eg [unfolded simple-match-to-of-match-def]

then guess x .. moreover from this (2) guess xa .. moreover from this (2) guess zb ..

note xx = calculation (1, 3) this

{ fix a b xc xa
  fix pp :: 16 word
  have [pp &
    \text{prefix-match-semantics xc (pp)} for xc]
  by (simp add: prefix-match-semantics-def word-bw-comms; fail)
  moreover have pp \in wordinterval-to-set (WordInterval a b) \implies a \leq pp \land pp \leq b by simp
  moreover have xc \in set (wordinterval-CIDR-split-prefixmatch (WordInterval a b)) \implies pp \in prefix-to-wordset xc \implies pp \in wordinterval-to-set (WordInterval a b)
  by (subst wordinterval-CIDR-split-prefixmatchblat)
  moreover have [xc \in set (wordinterval-CIDR-split-prefixmatch (WordInterval a b)); xa = Some (pfzm-prefix xc, \sim,pfzm-mask xc); prefix-match-semantics xc (pp)] \implies pp \in prefix-to-wordset xc
  apply (subst (asm) (1) prefix-match-semantics-wordset)
  apply (erule wordinterval-CIDR-split-prefixmatch-all-valid-Ball[THEN bspec, THEN conjunct1]; fail)
  apply assumption
  done
  ultimately have [xc \in set (wordinterval-CIDR-split-prefixmatch (WordInterval a b)); xa = Some (pfzm-prefix xc, \sim,pfzm-mask xc);]
  pp &
  \text{pfzm-mask xc = pfzm-prefix xc]
  \implies a \leq pp \land pp \leq b
  by metis
} note l4port-logic = this

show \text{thesis unfolding simple-matches.simps}
using xx(1) mo unfolding xx(4) OF-match-fields-unsafe-def
apply(simp only: if-False set-maps UN-iff)
apply(clarify)
apply(rename-tac a; subgoal-tac match-iface (iiface r) a)
apply(clarsimp simp add: option2set-def; fail)
apply(rule ccontr, simp; fail)
done
next
show match-iface (oiface r) (p-oiface p) using me.
next
show simple-match-ip (src r) (p-src p)
using mo unfolding xx(4) OF-match-fields-unsafe-def toprefixmatch-def
by(clarsimp
  simp add: simple-packet-unext-def option2set-def split: if-splits)
next
show simple-match-ip (dst r) (p-dst p)
using mo unfolding xx(4) OF-match-fields-unsafe-def toprefixmatch-def
by(clarsimp
  simp add: simple-packet-unext-def option2set-def split: if-splits)
next
show simple-match-port (sports r) (p-sport p)
using mo unfolding xx(4) OF-match-fields-unsafe-def toprefixmatch-def
by(clarsimp
  simp add: simple-packet-unext-def option2set-def split: if-splits)
next
show simple-match-port (dports r) (p-dport p)
using mo xx(2) unfolding xx(4) OF-match-fields-unsafe-def
by(cases sports r)
by(clarsimp simp add: l4port-logic simple-packet-unext-def option2set-def split: if-splits)
next
show simple-match-port (dports r) (p-dport p)
using mo xx(3) unfolding xx(4) OF-match-fields-unsafe-def
by(cases dports r)
by(clarsimp simp add: l4port-logic simple-packet-unext-def option2set-def split: if-splits)
qed
qed
primrec annotate-rlen where
annotate-rlen [] = []
annotate-rlen (a#as) = (length as, a) # annotate-rlen as
lemma annotate-rlen "asf" = [(3, CHR "a"), (2, CHR "s"), (1, CHR "d"), (0, CHR "f")]
by simp

lemma fst-annotate-rlen-le: (k, a) ∈ set (annotate-rlen l) ⇒ k < length l
by(induction l arbitrary: k; simp; force)

lemma distinct-fst-annotate-rlen: distinct (map fst (annotate-rlen l))
using fst-annotate-rlen-le by(induction l) (simp, fastforce)
lemma distinct-annotate-rlen: distinct (annotate-rlen l)
using distinct-fst-annotate-rlen unfolding distinct-map by blast
lemma in-annotate-rlen: (a,x) ∈ set (annotate-rlen l) ⇒ x ∈ set l
by(induction l) (simp-all, blast)
lemma map-snd-annotate-rlen: map snd (annotate-rlen l) = l
  by (induction l) simp-all
lemma sorted-descending (map fst (annotate-rlen l))
  by (induction l; clarsimp) (force dest: fst-annotate-rlen-le)
lemma annotate-rlen l = zip (rev [0..<length l]) l
  by (induction l; simp)

primrec annotate-rlen-code where
annotate-rlen-code [] = (0,[])
annotate-rlen-code (a#as) = (case annotate-rlen-code as of (r,aas) ⇒ (Suc r, (r, a) # aas))
lemma annotate-rlen-len: list (annotate-rlen-code r) = length r
  by (induction r) (clarsimp split: prod.splits)+
lemma annotate-rlen-code[code]: annotate-rlen s = snd (annotate-rlen-code s)
proof (induction s)
  case (Cons s ss) thus ?case using annotate-rlen-len[OF ss] by (clarsimp split: prod.split)
qed simp

lemma suc2plus-inj-on: inj-on (of-nat :: nat ⇒ ('l :: len) word) {0..unat (max-word :: 'l word)}
proof (rule inj-on)
  let ?mmw = (max-word :: 'l word)
  let ?mstp = (of-nat :: nat ⇒ 'l word)
  fix x y :: nat
  assume x ∈ {0..unat ?mmw} y ∈ {0..unat ?mmw}
  hence se: x ≤ unat ?mmw y ≤ unat ?mmw by simp-all
  assume eq: ?mstp x = ?mstp y
  note f = le-unat-oui[OF se(1)] le-unat-oui[OF se(2)]
  have x = y using eq le-unat-oui se by metis
qed

lemma distinct-of-nat-list:
distinct l ⇒ ∀ e ∈ set l. e ≤ unat (max-word :: ('l::len) word) ⇒ distinct (map (of-nat :: nat ⇒ 'l word) l)
proof (induction l)
  let ?mmw = (max-word :: 'l word)
  let ?mstp = (of-nat :: nat ⇒ 'l word)
  case (Cons a as)
  have distinct as ∀ e ∈ set as. e ≤ unat ?mmw using Cons.prems by simp-all
  note mIH = Cons.IH[OF this]
  moreover have ?mstp a ≠ ?mstp ' set as
    proof
      have representable-set: set as ⊆ {0..unat ?mmw} using ∀ e ∈ set (a ≠ as). e ≤ unat max-word by fastforce
      have a-reprbl: a ∈ {0..unat ?mmw} using ∀ e ∈ set (a ≠ as). e ≤ unat max-word by simp
      assume ?mstp a ∈ ?mstp ' set as
      with inj-on-image-mem-iff[OF suc2plus-inj-on a-reprbl representable-set]
      have a ∈ set as by simp
      with (distinct (a ≠ as)) show False by simp
    qed
  ultimately show ?case by simp
qed simp

lemma annotate-first-le-hlp:
length l < unat (max-word :: ('l :: len) word) ⇒ ∀ e ∈ set (map fst (annotate-rlen l)). e ≤ unat (max-word :: 'l word)
  by (clarsimp split: prod.split)
lemmas distinct-of-prio-hlp = distinct-of-nat-list[OF distinct-fst-annotate-rlen annotate-first-le-hlp]
lemma fst-annotate-rlen: \( \text{map } \text{fst} \ (\text{annotate-rlen} \ l) = \text{rev} \ [0..<\text{length} \ l] \)
by (induction \( l \)) (simp-all)

lemma sorted-word-upt:
\[
\begin{align*}
\text{defines} & : \ \text{won} \equiv \ (\text{of-nat} :: \ \text{nat} \Rightarrow ('l :: \ \text{len}) \ \text{word}) \\
\text{assumes} & : \ \text{length} \ l \leq \ \text{unat} \ (\text{max-word} :: ('l :: \ \text{len}) \ \text{word}) \\
\text{shows} & : \ \text{sorted-descending} \ \text{map} \ \text{won} \ (\text{rev} \ [0..<\text{Suc} \ (\text{length} \ l)])
\end{align*}
\]
using assms
by (induction \( l \) rule: rev-induct; clarsimp)

(metis (mono-tags, hide-lams) le-SucI le-unat-uoi of-nat-Suc order-refl word-le-nat-alt)

lemma sorted-annotated:
\[
\begin{align*}
\text{assumes} & : \ \text{length} \ l \leq \ \text{unat} \ (\text{max-word} :: ('l :: \ \text{len}) \ \text{word}) \\
\text{shows} & : \ \text{sorted-descending} \ \text{map} \ \text{fst} \ (\text{map} \ (\text{apfst} \ (\text{of-nat} :: \ \text{nat} \Rightarrow 'l \ \text{word}))) \ (\text{annotate-rlen} \ l))
\end{align*}
\]
proof
−
let \(?\text{won} = (\text{of-nat} :: \ \text{nat} \Rightarrow 'l \ \text{word})\)

have \(\text{sorted-descending} \ \text{map} \ ?\text{won} \ (\text{rev} \ [0..<\text{Suc} \ (\text{length} \ l)])\)
using sorted-word-upt.

hence \(\text{sorted-descending} \ \text{map} \ ?\text{won} \ (\text{map} \ \text{fst} \ (\text{annotate-rlen} \ l))\)
by (simp add: fst-annotate-rlen)

thus \(\text{sorted-descending} \ \text{map} \ \text{fst} \ (\text{map} \ (\text{apfst} \ ?\text{won}) \ (\text{annotate-rlen} \ l))\)
by simp
qed

l3 device to l2 forwarding

definition lr-of-tran-s3 ifs ard = 
\[
[(p, b, \ \text{case} \ a \ \text{of} \ \text{simple-action}\text{.Accept} \Rightarrow [\text{Forward} \ c] \ | \ \text{simple-action}\text{.Drop} \Rightarrow []). \\
(p, r, (\epsilon, a)) \leftarrow \text{ard}, \ b \leftarrow \text{simple-match-to-of-match} \ r \ \text{ifs})
\]

definition oif-ne-iif-p1 ifs \equiv \[(\text{simple-match-any}(oiface := \text{Iface} \ oif, \ iiface := \text{Iface} \ iif), \ \text{simple-action}\text{.Accept}, \ oif \leftarrow \text{ifs}, \ iif \leftarrow \text{ifs}, \ oif \neq iif)\]

definition oif-ne-iif-p2 ifs \equiv \[(\text{simple-match-any}(oiface := \text{Iface} \ i, \ iiface := \text{Iface} \ i), \ \text{simple-action}\text{.Drop}, \ i \leftarrow \text{ifs})\]

definition oif-ne-iif ifs = oif-ne-iif-p2 ifs @ oif-ne-iif-p1 ifs

definition lr-of-tran-s4 ard ifs \equiv \text{generalized-fw-join} \ text{ard} (oif-ne-iif \ ifs)

definition lr-of-tran-s1 rt = [(\text{route2match} \ r, \ \text{output-iface} \ (\text{routing-action} \ r)). \ r \leftarrow \text{rt}]

definition lr-of-tran-fbs rt fw ifs \equiv let
\(\text{gfw} = \text{map} \ \text{simple-rule-dtor} \ \text{fw}\); \ − \ \text{generalized} \ \text{simple} \ \text{fw}, \ \text{hopefully} \ \text{for} \ \text{FORWARD} \\
\text{frt} = \text{lr-of-tran-s1} \ rt; \ − \ \text{rt} \ \text{as} \ \text{fw} \\
\text{prd} = \text{generalized-fw-join} \ \text{frt} \ \text{gfw} \\
\text{in} \ \text{prd}
\)

definition pack-OF-entries ifs ard \equiv \text{map} \ (\text{split3} \ \text{OFEntry}) \ (\text{lr-of-tran-s3} \ text{ifs} \ \text{ard})

definition no-oif-match \equiv \text{list-all} \ (\lambda m. \ oiface \ (\text{match-sel} \ m) = \text{ifaceAny})

definition lr-of-tran rt fw ifs \equiv
\[
\text{if} \ \neg (\text{no-oif-match} \ \text{fw} \ \wedge \ \text{has-default-policy} \ \text{fw} \ \wedge \ \text{simple-fw-valid} \ \text{fw} \ \wedge \ \text{valid-prefixes} \ \text{rt} \ \wedge \ \text{has-default-route} \ \text{rt} \ \wedge \ \text{distinct} \ \text{ifs}) \\
\text{then Inl "Error in creating OpenFlow table: prerequisites not satisfied"} \\
\text{else} ( \\
\text{let} \ \text{nr} = \text{lr-of-tran-fbs} \ \text{rt} \ \text{fw} \ \text{ifs};
\)
ard = map (apfst of-nat) (annotate-rlen nrd) — give them a priority

in
if length nrd < unat (max-word :: 16 word)
then Inr (pack-OF-entries ifs ard)
else Inl "Error in creating OpenFlow table: priority number space exhausted"

definition is-iface-name i ≡ i ≠ [] ∧ ~Iface.iface-name-is-wildcard i
definition is-iface-list ifs ≡ distinct ifs ∧ list-all is-iface-name ifs

lemma max-16-word-max[simp]: (a :: 16 word) ≤ 0xffff
proof –
have ℓℓℓℓ: 0xffff = word-of-int (2 ^ 16 - 1) by fastforce
show ?thesis using max-word-max[of a] unfolding max-word-def ℓℓℓℓ by fastforce
qed

lemma replicate-FT-hlp: x ≤ 16 ∧ y ≤ 16 ⇒ replicate (16 - x) False @ replicate x True = replicate (16 - y) False @ replicate y True ⇒ x = y
proof –
let ?ns = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
assume x ≤ 16 ∧ y ≤ 16
hence x ∈ ?ns y ∈ ?ns by(simp; presburger)+
moreover assume replicate (16 - x) False @ replicate x True = replicate (16 - y) False @ replicate y True
ultimately show x = y by simp (elim disjE; simp-all)
qed

lemma mask-inj-hlp1: inj-on (mask :: nat ⇒ 16 word) {0..16}
proof(intro inj-onI, goal-cases)
case (1 x)
from 1(3)
have oe: of-bl (replicate (16 - x) False @ replicate x True) = (of-bl (replicate (16 - y) False @ replicate y True) :: 16 word)
unfolding mask-bl of-bl-rep-False .
have ℓℓℓℓ: z ≤ 16 ⇒ length (replicate (16 - z) False @ replicate z True) = 16 by auto
with 1(1,2)
have ps: replicate (16 - x) False @ replicate x True ∈ {bl. length bl = len-of TYPE(16)} replicate (16 - y) False @ replicate y True ∈ {bl. length bl = len-of TYPE(16)} by simp-all
from inj-onD[OF word-bl.Abs-inj-on, OF oe ps]
show ?case using 1(1,2) by(fastforce intro: replicate-FT-hlp)
qed

lemma distinct-simple-match-to-of-match-portlist-hlp:
fixes ps :: (16 word × 16 word)
shows distinct ifs ==> distinct
(if fst ps = 0 ∧ snd ps = max-word then [None]
else if fst ps ≤ snd ps
then map (Some o (λpfx. (pfxm-prefix pfx, ~ pfxm-mask pfx)))
(wordinterval-CIDR-split-prefixmatch (WordInterval (fst ps) (snd ps)))
else [])
proof –
assume di: distinct ifs
{ define wsi where wsi = set (wordinterval-CIDR-split-prefixmatch (WordInterval (fst ps) (snd ps)))
fix x y :: 16 prefix-match

27
obtain \( xm \ x n \ y m \ y n \) where \( y z d [ \text{simp} ] : x = \text{PrefixMatch} \ x m \ x n \ y = \text{PrefixMatch} \ y m \ y n \) by(cases \( x \); cases \( y \))
assume \( iw \) : \( x \in \ y s \) \( y w s \) \( a n d \) \( e t \) : \( \text{pfzm-prefix} \ x \) \( \text{prefix} \ y \) \( \text{pfzm-mask} \ x \) = \( \text{pfzm-prefix} \ y \) \( \text{prefix} \ y \) \( \text{pfzm-mask} \ y \)
  hence le16: \( x n \leq 16 \ y n \leq 16 \) unfolding wis-def using wordinterval-CIDR-split-prefixmatch-all-valid-Ball[unfolded Ball-def, THEN spec, THEN mp] by force+
  with et have 16 - \( x n = 16 - y n \) unfolding pfzm-mask-def by(auto intro: mask-inj-hlp1[THEN inj-onD])
  hence \( x = y \) using et le16 using diff-diff-cancel by simp
}
note \( * = \text{this} \)
show \( ?\text{thesis} \) note \( } \) subgoal by apply apply apply apply using \( * \)
subgoal by subgoal by subgoal by subgoal by subgoal by subgoal by subgoal by

proof (clarsimp simp add: smtoms-eq-hlp distinct-map wordinterval-CIDR-split-distinct)
proof (subgoal proof)

apply(clarsimp simp add: smtoms-eq-hlp)

qed

lemma distinct-simple-match-to-of-match: distinct ifs \( \rightarrow \) distinct (simple-match-to-of-match m ifs)
apply(unfold simple-match-to-of-match-def Let-def)
apply(rule distinct-3lcompr1)
subgoal by(induction ifs; clarsimp)
subgoal by(fact distinct-simple-match-to-of-match-portlist-hlp)
subgoal by(fact distinct-simple-match-to-of-match-portlist-hlp)
subgoal by(simp-all add: smtoms-eq-hlp)
done

lemma inj-inj-on: inj \( F \) \( \rightarrow \) inj-on \( F \ A \) using subset-inj-on by auto

lemma no-overlaps-lroft-hlp2: distinct (map fst amr) \( \rightarrow \) \( (\forall r. \text{distinct} \ (\text{fm} \ r)) \rightarrow \right) \text{distinct} (\text{concat} (\text{map} (\lambda p. p, r, c, a) \ (\text{map} (\lambda b. (p, b, fs a c)) \ (\text{fm} \ r)) \ (\text{amr})))
by(induction amr; force intro: injl inj-onI simp add: distinct-map split: prod.splits)

lemma distinct-lroft-s3: [distinct (map fm amr); distinct ifs] \( \rightarrow \) distinct (lr-of-tran-s3 ifs amr)
unfolding lr-of-tran-s3-def
by(erule no-overlaps-lroft-hlp2, simp add: distinct-simple-match-to-of-match)

lemma no-overlaps-lroft-hlp3: distinct (map fm amr) \( \rightarrow \) \( (a a, a b, a c) \in \text{set} \ (lr-of-tran-s3 ifs amr) \Rightarrow \ (b a, b b, b c) \in \text{set} \ (lr-of-tran-s3 ifs amr) \Rightarrow \)
ac \( \neq \) bc \( \Rightarrow \) \( \text{aa} \neq \text{ba} \)
apply(unfold lr-of-tran-s3-def)
apply(clarsimp)
apply(clarsimp split: simple-action.splits)
apply(metis map-of-eq-Some-iff old.prod.inject option.inject)
apply(metis map-of-eq-Some-iff old.prod.inject option.inject simple-action.distinct(2))
done

lemma no-overlaps-lroft-s3-hlp: \( [\text{distinct} (\text{map} \ fm \ amr); \text{OF-match-fields-unsafe} \ ab \ p; ab \ \neq ad \lor ba \neq bb; \text{OF-match-fields-unsafe} \ ad \ p;]
\quad (ac, ab, ba) \in \text{set} \ (lr-of-tran-s3 ifs amr); (ac, ad, bb) \in \text{set} \ (lr-of-tran-s3 ifs amr)]\)
\( \rightarrow \) \( \text{False} \)
proof (case \( \text{disjE}; \text{goal-cases} \)
  case 1
  have 4: \( \text{[distinct (map fm amr); (ac, ab, x1, z2) \in set amr; (ac, bb, x4, z5) \in set amr; ab} \neq bb] \)
  \( \Rightarrow \) \( \text{False for ab x1 x2 bx x4 x5} \)
  by (meson distinct-map-fstD old.prod.inject)
  have conjunctSomeProtoAnyD: Some ProtoAny = simple-proto-conjunct a (Proto b) \( \Rightarrow \) \( \text{False for ab} \)
    using conjunctProtoD by force
  have 5: 28
[\(\text{OF-match-fields-unsafe} \ am \ p; \ \text{OF-match-fields-unsafe} \ bm \ p; \ am \neq bm;\]
\[am \in \text{set \ (simple-match-to-of-match \ ab \ ifs); \ bm \in \text{set \ (simple-match-to-of-match \ bb \ ifs);} \ \neg \ ab \neq bb]\]
\[\implies \text{False \ for \ ab \ bb \ am \ bm}\]
\by{\text{clarify} | \text{unfold} \ \text{simple-match-to-of-match-def} \ \text{smtoms-eq-hlp} \ \text{Let-def} \ \text{set-concat} \ \text{set-map} \ \text{de-Morgan-conj} \ \text{not-False-eq-True}) + \ (\text{auto} \ \text{dest: conjunctSomeProtoAnyD cidrsplit-no-overlaps} \ \text{simp add: OF-match-fields-unsafe-def simple-match-to-of-match-single-def option2set-def comp-def} \ \text{split: if-splits} \ \text{cong: smtoms-eq-hlp})}

\text{from} \ 1 \ \text{show} \ \text{?case} \ \text{using} \ 4 \ 5 \ \text{by(\text{clarsimp simp add: lr-of-tran-s3-def}) blast}
\text{qed(\text{metis no-overlaps-lroft-hlp3})}

\text{lemma no-overlaps-lroft-s3-hlp: distinct \ (map \ \text{fst} \ \text{amr})} \implies \ \text{distinct \ ifs} \implies \ \text{no-overlaps OF-match-fields-unsafe \ (map \ (\text{split3 \ OFEntry}) \ (\text{lr-of-tran-s3} \ ifs \ \text{amr}))}
\\ \text{apply(\text{rule no-overlapsI[rotated]})}
\\ \text{apply(\text{subst distinct-map, rule conjI})}
\\ \text{subgoal \ by(\text{erule (1) distinct-lroft-s3})}
\\ \text{subgoal}
\\ \text{apply(\text{rule inj-inj-on})}
\\ \text{apply(\text{rule injI})}
\\ \text{apply(\text{rename-tac x y, case-tac x, case-tac y})}
\\ \text{apply(\text{simp add: split3-def fail})}
\ \text{done}
\text{subgoal}
\ \text{apply(\text{unfold check-no-overlap-def})}
\ \text{apply(\text{clarify})}
\ \text{apply(\text{unfold set-map})}
\ \text{apply(\text{clarify})}
\ \text{apply(\text{unfold split3-def prod.simps flow-entry-match.simps flow-entry-match.sel de-Morgan-conj})}
\ \text{apply(\text{clar simp only:})}
\ \text{apply(\text{erule (1) no-overlaps-lroft-s3-hlp-hlp})}
\ \text{apply simp}
\ \text{apply assumption}
\ \text{apply assumption}
\ \text{apply simp}
\text{done}
\text{done}

\text{lemma lr-of-tran-no-overlaps: assumes distinct \ ifs \ shows \ \text{Inr \ t} = \ (\text{lr-of-tran \ rt \ fw \ ifs})} \implies \ \text{no-overlaps \ OF-match-fields-unsafe \ t}
\ \text{apply(\text{unfold \ lr-of-tran-def \ Let-def \ pack-OF-entries-def})}
\ \text{apply(\text{simp split: \ if-splits})}
\ \text{apply(\text{thin-tac t = -})}
\ \text{apply(\text{drule distinct-of-prio-hlp})}
\ \text{apply(\text{rule no-overlaps-lroft-s3-hlp[rotated]})}
\ \text{subgoal \ by(\text{simp add: \ asms})}
\ \text{subgoal \ by(\text{simp add: \ a-assoc})}
\text{done}

\text{lemma \text{sorted-lr-of-tran-s3-hlp}: \forall \ x \in \text{set \ f}. \ \text{fst \ x} \leq \ a \implies \ b \in \text{set \ (lr-of-tran-s3} \ s \ f) \implies \ \text{fst \ b} \leq \ a}
\ \text{by(\text{auto simp add: lr-of-tran-s3-def})}
lemma \( lr\text{-}of\text{-}tran\text{-}s3\text{-}Cons \): \( lr\text{-}of\text{-}tran\text{-}s3 \iff ([p, b, case a of simple\text{-}action.Accept \Rightarrow [\text{Forward } c] | simple\text{-}action.Drop \Rightarrow []]) \)
\((p, r, (c, a)) \leftarrow [a], b \leftarrow \text{simple\text{-}match\text{-}to\text{-}of\text{-}match } r \iff s) \) @ \( lr\text{-}of\text{-}tran\text{-}s3 \iff s \text{ ard} \)
by(clarsimp simp: lr-of-tran-s3-def)

lemma sorted-lr-of-tran-s3: \( \text{sorted\text{-}descending } (map \ f s) \iff \text{sorted\text{-}descending } (map \ f (lr\text{-}of\text{-}tran\text{-}s3 \ s f)) \)
apply(induction \( f \))
subgoal by(simp add: lr-of-tran-s3-def)
apply(clarsimp simp: lr-of-tran-s3-Cons map-concat comp-def)
apply(unfold sorted\text{-}descending\text{-}append)
apply(simp add: sorted\text{-}descending\text{-}alt rev\text{-}map sorted\text{-}lr\text{-}of\text{-}tran-s3-hlp sorted\text{-}const)
done

lemma sorted-lr-of-tran-hlp: \( (\text{ofe}\text{-}prio \circ \text{split3 } \text{OFEntry}) = \text{fst} \) by(simp add: fun\text{-}eq\text{-}iff comp-def split3-def)

lemma \( lr\text{-}of\text{-}tran\text{-}s1\text{-}split \): \( lr\text{-}of\text{-}tran\text{-}s1 \ ) (\( a \# rt \)) = \( \text{route2match } a \), \( \text{output\text{-}iface } (\text{routing\text{-}action } a) \) \@ \( lr\text{-}of\text{-}tran\text{-}s1 \ rt \)
by(unfold lr-of-tran-s1-def list.map, rule)

lemma route2match-correct: \( \text{valid\text{-}prefix } (\text{routing\text{-}match } a) \iff \text{prefix\text{-}match\text{-}semantics } (\text{routing\text{-}match } a) \ (p\text{-}dst } p) \iff \text{simple\text{-}matches } (\text{route2match } a) \ ) (\( p \))
by(simp add: route2match-def simple\text{-}matches.simps match\text{-}iface\text{Any} match\text{-}iface\text{refl} ipset\text{-}from\text{cidr}\text{-}0
prefix\text{-}match\text{-}semantics\text{-}ipset\text{-}from\text{netmask}\text{2})

lemma s1-correct: \( \text{valid\text{-}prefixes } rt \iff \text{has\text{-}default\text{-}route } (rt::\ (\text{rt::len}) \text{prefix\text{-}routing}) \iff \exists \text{ ra. generalized\text{-}sfw } (lr\text{-}of\text{-}tran\text{-}s1 \ rt) \ p = \text{Some } (\text{rm}, ra) \land ra = \text{output\text{-}iface } (\text{routing\text{-}table\text{-}semantics } rt \ (p\text{-}dst } p) \)
apply(induction \( rt \))
apply(simp; fail)
apply(drule valid\text{-}prefixes\text{-}split)
apply(clarsimp)
apply(erule disjE)
subgoal for \( a \) \( rt \)
apply(case-tac \( a \))
apply(rename-tac routing\text{-}m metric routing\text{-}action)
apply(case-tac routing\text{-}m)
apply(simp add: valid\text{-}prefix\text{def} prefix\text{mask\text{-}def} prefix\text{-}match\text{-}semantics\text{def} generalized\text{-}sfw\text{def}
\text{lr\text{-}of\text{-}tran\text{-}s1\text{-}def} \text{route2match\text{-}def} \text{simple\text{-}matches.simps} \text{match\text{-}iface\text{Any} match\text{-}iface\text{refl} ipset\text{-}from\text{cidr}\text{-}0
\text{prefix\text{-}match\text{-}semantics\text{-}ipset\text{-}from\text{netmask}\text{2}})

done

subgoal
apply(rule conjI)
apply(simp add: generalized\text{-}sfw\text{def} lr\text{-}of\text{-}tran\text{-}s1\text{-}def route2match\text{-}correct; fail)
apply(simp add: route2match\text{-}def simple\text{-}matches.simps prefix\text{-}match\text{-}semantics\text{-}ipset\text{-}from\text{netmask}\text{2}
\text{lr\text{-}of\text{-}tran\text{-}s1\text{-}split} generalized\text{-}sfw\text{-}simps)

30
definition to-OF-action \( a \equiv (\text{case } a \text{ of } (p,d) \Rightarrow (\text{case } d \text{ of } \text{simple-action}.\text{Accept} \Rightarrow [\text{Forward } p] \mid \text{simple-action}.\text{Drop} \Rightarrow [])) \)
definition from-OF-action \( a = (\text{case } a \text{ of } [] \Rightarrow ("",\text{simple-action}.\text{Drop}) \mid [\text{Forward } p] \Rightarrow (p, \text{simple-action}.\text{Accept})) \)

lemma \( \text{OF-match-linear-not-noD}: \text{OF-match-linear } \gamma \text{ oms } p \neq \text{NoAction} \implies \exists \text{ ome } \in \text{ set oms } \land \gamma (\text{ofe-fields ome}) p \)
apply(induction oms)
apply(simp)
apply(simp split: if-splits)
apply blast+
done

lemma \( s3\text{-noaction-hlp} : [\text{simple-match-valid } ac; \neg \text{simple-matches ac } p; \text{match-iface } (\text{ofiface ac }) (p\text{-offace } p)] \implies \text{OF-match-linear } \text{OF-match-fields-safe } (\text{map } (\lambda x. \text{split3 } \text{OFEntry } (x1, x, \text{case } ba \text{ of } \text{simple-action}.\text{Accept} \Rightarrow [\text{Forward } ad] \mid \text{simple-action}.\text{Drop} \Rightarrow [])) (\text{simple-match-to-of-match } ac \text{ ifs}) p = \text{NoAction} \)
apply(rule ccontr)
apply(drule \text{OF-match-linear-not-noD})
apply(clarsimp)
apply(rename-tac x)
apply(subgoal-tac all-prerequisites x)
apply(drule \text{simple-match-to-of-matchD})
apply(simp add: split3-def)
apply(subst(asm) of-match-fields-safe-eq2)
apply(simp:fail)+
using \text{simple-match-to-of-match-generates-prereqs} by blast

lemma \( s3\text{-correct} : \)
assumes \( \text{vsfwm} : \text{list-all } \text{simple-match-valid } (\text{map } (\text{fst} \circ \text{snd}) \text{ ard}) \)
assumes \( \text{ippkt} : p\text{-l2type } p = 0x800 \)
assumes \( \text{iiifs} : \text{p-iiface } p \in \text{ set ifs} \)
assumes \( \text{oiifs} : \text{list-all } (\lambda m. \text{ofiface } (\text{fst } (\text{snd } m))) = \text{ifaceAny} \text{ ard} \)
shows \( \text{OF-match-linear } \text{OF-match-fields-safe } (\text{pack-OF-entries } \text{ifs } \text{ard}) p = \text{Action } ao \iff (\exists r \text{ af } . \text{generalized-sfw } (\text{map } \text{snd ard}) p = (\text{Some } (r, af)) \land (\text{if } \text{snd } af = \text{simple-action}.\text{Drop} \text{ then } ao = [] \text{ else } ao = [\text{Forward } (\text{fst } af)])) \)
unfolding \( \text{pack-OF-entries-def } \text{lr-of-tran-s3-def } \text{fun-app-def} \)
using \( \text{vsfwm } \text{oiifs} \)
apply(induction \text{ard})
subgoal by(simp add: \text{generalized-sfw-simps})
apply simp
apply(clarsimp simp add: \text{generalized-sfw-simps split: prod.splits})
apply(intro conjI)
subgoal for \( \text{ard } x1 \text{ ac } ad \text{ ba} \)
apply(clarsimp simp add: \text{OF-match-linear-append split: prod.splits})
apply(drule \text{simple-match-to-of-matchI[rotated]})
apply(rule \text{iiifs})
apply(rule \text{ippkt})
apply blast
apply(clarsimp simp add: \text{comp-def})
apply(drule \text{OF-match-linear-match-allsameaction}[where \( \gamma = \text{OF-match-fields-safe } \text{and } \text{pri } = x1 \text{ and } \text{oms } = \text{simple-match-to-of-match } ac \text{ ifs } \text{and } \text{act } = \text{case } ba \text{ of } \text{simple-action}.\text{Accept} \Rightarrow [\text{Forward } ad] \mid \text{simple-action}.\text{Drop} \Rightarrow []))
apply(unfold \text{OF-match-fields-safe-def comp-def})
apply(rule Some-to-the[symmetric]; fail)
apply(clarsimp)
apply(intro iffI)

subgoal
apply(rule exI[where x = ac])
apply(rule exI[where x = ad])
apply(rule exI[where x = ba])
apply(clarsimp simp: split3-def split: simple-action.splits flowtable-behavior.splits if-splits)
done
subgoal
apply(clarsimp)
apply(rename-tac b)
apply(case-tac b)
apply(simp-all)
done
done

subgoal for ard z1 ac ad ba
apply(simp add: OF-match-linear-append OF-match-fields-safe-def comp-def)
apply(clarify)
apply(subgoal-tac OF-match-linear OF-match-fields-safe (map (λx. split3 OFEntry (x1, x, case ba of simple-action.Accept ⇒ [Forward ad] | simple-action.Drop ⇒ [])) (simple-match-to-of-match ac ifs)) p = NoAction)
apply(simp; fail)
apply(erule (1) s3-noaction-blp)
apply(simp add: match-ifaceAny; fail)
done
done

done

context
notes valid-prefix-00[simp, intro!]
begin

lemma lr-of-tran-s1-valid: valid-prefixes rt ⇒ gsfw-valid (lr-of-tran-s1 rt)
unfolding lr-of-tran-s1-def route2match-def gsfw-valid-def list-all-iff
apply(clarsimp simp: simple-match-valid-def valid-prefixs-fw-def)
apply(intro conjI)
apply force
apply(simp add: valid-prefixes-alt-def)
done
done

lemma simple-match-valid-fbs-rlen: [valid-prefixes rt; simple-fw-valid fw; (a, aa, ab, b) ∈ set (annotate-rlen (lr-of-tran-fbs rt fw ifs))] ⇒ simple-match-valid aa
proof(goal-cases)
case 1
note 1 [unfolded lr-of-tran-fbs-def Let-def]
have gsfw-valid (map simple-rule-dtor fw) using gsfw-valid! 1 by blast
moreover have gsfw-valid (lr-of-tran-s1 rt) using 1 lr-of-tran-s1-valid by blast
ultimately have gsfw-valid (generalized-fw-join (lr-of-tran-s1 rt) (map simple-rule-dtor fw)) using gsfw-join-valid by blast
moreover have (aa, ab, b) ∈ set (lr-of-tran-fbs rt fw ifs) using 1 using in-annotate-rlen by fast
ultimately show ?thesis unfolding lr-of-tran-fbs-def Let-def gsfw-valid-def list-all-iff by fastforce
qed

lemma simple-match-valid-fbs: [valid-prefixes rt; simple-fw-valid fw] ⇒ list-all simple-match-valid (map fst (lr-of-tran-fbs rt fw ifs))
proof(goal-cases)

32
case 1

note 1 [unfolded lr-of-tran-fbs-def Let-def]

have gsfw-valid (map simple-rule-dtor fw) using gsfw-valid1 1 by blast

moreover have gsfw-valid (lr-of-tran-s1 rt) using 1 lr-of-tran-s1-valid by blast

ultimately have gsfw-valid (generalized-fw-join (lr-of-tran-s1 rt) (map simple-rule-dtor fw)) using gsfw-join-valid by blast

thus thesis unfolding lr-of-tran-fbs-def Let-def gsfw-valid-def list-all-iff by fastforce

qed

lemma lr-of-tran-prereqs: valid-prefixes rt =⇒ simple-fw-valid fw =⇒ lr-of-tran rt fw ifs = Inr oft =⇒

list-all (all-prerequisites ◦ ofe-fields) oft

unfolding lr-of-tran-def pack-OF-entries-def lr-of-tran-s3-def Let-def

apply (simp add: map-concat comp-def prod.case-distrib split3-def split: if-splits)

apply (clarsimp simp add: list-all-iff)

apply (clarsimp)

apply (drule simple-match-valid-fbs-rlen [rotated])

apply (simp add: list-all-iff)

apply (clarsimp)

apply (rule simple-match-to-of-match-generates-prereqs; assumption)

done

lemma OF-unsafe-safe-match3-eq:
list-all (all-prerequisites ◦ ofe-fields) oft =⇒


unfolding OF-priority-match-def [abs-def]

proof (goal-cases)
   case 1
   from 1 have ∀ packet. [f ← oft . OF-match-fields-unsafe (ofe-fields f) packet] = [f ← oft . OF-match-fields-safe (ofe-fields f) packet]

   apply (clarsimp simp add: list-all-iff OF-match-fields-safe-eq)

   using OF-match-fields-safe-eq by (metis (mono-tags, lifting) filter-cong)

   thus ?case by metis

qed

lemma OF-unsafe-safe-match-linear-eq:
list-all (all-prerequisites ◦ ofe-fields) oft =⇒


unfolding fun-eq-iff

by (induction oft) (clarsimp simp add: list-all-iff of-match-fields-safe-eq)+

lemma simple-action-ne [simp]:
  b ≠ simple-action. Accept ⇔ b = simple-action. Drop
  b ≠ simple-action. Drop ⇔ b = simple-action. Accept

using simple-action.exhaust by blast+

lemma map-snd-apfst: map snd (map (apfst x) l) = map snd l

unfolding map-map comp-def snd-apfst ..

lemma match-ifaceAny-eq: ofiface m = iface.Any =⇒ simple-matches m p = simple-matches m (p|p-oiface := any)

by (cases m) (simp add: simple-matches.simps match-iface.Any)

lemma no-oif-matchD: no-oif-match fw =⇒ simple-fw fw p = simple-fw fw (p|p-oiface := any)

by (induction fw)
   (auto simp add: no-oif-match-def simple-fw-alt dest: match-iface.Any-eq)
lemma lr-of-tran-fbs-acceptD:
assumes s1: valid-prefixes rt has-default-route rt
assumes s2: no-oif-match fw
shows generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Accept) \implies
simple-linux-router-nol12 rt fw p = Some (p[p-oiface := oif])

proof (goal-cases)
  case 1
  note 1 [unfolded lr-of-tran-fbs-def Let-def, THEN generalized-fw-joinD]
  then guess r1 .. then guess r2 .. note r12 = this
  note s1-correct (OF s1, of p)
  then guess rm .. then guess ra .. note rmra = this
  from r12 rmra have oifra: oif = ra by simp
  from r12 have sfw: simple-fw fw p = Decision FinalAllow using simple-fw-iff-generalized-fw-accept by blast
  note ifupdateirrel = no-oif-matchD (OF s2, where any = output-iface (routing-table-semantics rt (p-dst p)) and p = p, symmetric)
  show ?case unfolding simple-linux-router-nol12-def by (simp add: Let-def ifupdateirrel sfw oifra rmra split: Option.bind-splits option.splits)
qed

lemma lr-of-tran-fbs-acceptI:
assumes s1: valid-prefixes rt has-default-route rt
assumes s2: no-oif-match fw has-default-policy fw
shows simple-linux-router-nol12 rt fw p = Some (r[t-oiface := oif]) \implies
\exists r. generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Accept)

proof (goal-cases)
  from s2 have nud: \forall p. simple-fw fw p \neq Undecided by (metis has-default-policy state.distinct(1))
  note ifupdateirrel = no-oif-matchD (OF s2(1), symmetric)
  case 1
  from 1 have simple-fw fw p = Decision FinalAllow by (simp add: simple-linux-router-nol12-def Let-def nud ifupdateirrel split: Option.bind-splits state.splits final-decision.splits)
  then obtain r where r: generalized-sfw (map simple-rule-dtor fw) p = Some (r, simple-action.Accept) using simple-fw-iff-generalized-fw-accept by blast
  have oif-def: oif = output-iface (routing-table-semantics rt (p-dst p)) using 1 by (cases p) (simp add: simple-linux-router-nol12-def Let-def nud ifupdateirrel split: Option.bind-splits state.splits final-decision.splits)
  note s1-correct (OF s1, of p) then guess rm .. then guess ra .. note rmra = this
  show ?case unfolding lr-of-tran-fbs-def Let-def
    apply (rule exI)
    apply (rule generalized-fw-joinI)
    unfolding oif-def using rmra apply simp
    apply (rule r)
  done
qed

lemma lr-of-tran-fbs-dropD:
assumes s1: valid-prefixes rt has-default-route rt
assumes s2: no-oif-match fw
shows generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Drop) \implies
simple-linux-router-nol12 rt fw p = None

proof (goal-cases)
  note ifupdateirrel = no-oif-matchD (OF s2(1), symmetric)
  case 1
  from 1 [unfolded lr-of-tran-fbs-def Let-def, THEN generalized-fw-joinD]
  obtain rr fr where generalized-sfw (lr-of-tran-s1 rt) p = Some (rr, oif) \land
generalized-sfw (map simple-rule-dtor fw) p = Some (fr, simple-action.Drop) ∧ Some r = simple-match-and rt fr
by presburger

hence fd: ∃ u. simple-fw fw (p(p-oiface := u)) = Decision FinalDeny unfolding ifupdateirrel
using simple-fw-iff-generalized-fw-drop by blast

show ?thesis
by(clarsimp simp: simple-linux-router-nol12-def Let-def split: Option.bind-splits)

qed

lemma lr-of-tran-fbs-dropI:
assumes s1: valid-prefixes rt has-default-route rt
assumes s2: no-oif-match fw has-default-policy fw
shows simple-linux-router-nol12 rt fw p = None ⟷
∃ r oif. generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Drop)

proof (goal-cases)
from s2 have nud: ∃ p. simple-fw fw p ≠ Undecided by (metis has-default-policy state.different(1))

note ifupdateirrel = no-oif-matchD[OF s2(1), symmetric]

1 case:

from 1 have simple-fw fw p = Decision FinalDeny by(simp add: simple-linux-router-nol12-def Let-def nud ifupdateirrel
split: Option.bind-splits stateplits final-decision.splits)

then obtain r where r: generalized-sfw (map simple-rule-dtor fw) p = Some (r, simple-action.Drop) using
simple-fw-iff-generalized-fw-drop by blast

note s1-correct[OF s1, of p] then guess rm .. then guess ra .. note rmra = this

show ?case unfolding lr-of-tran-fbs-drop Let-def
apply(rule exI)
apply(rule exI[where x = ra])
apply(rule generalized-fw-joinI)

using rmra apply simp

apply(rule r)

qed

lemma no-oif-match-fbs:
no-oif-match fw ⇒ list-all (λ m. oiface (fst (snd m)) = iface.Any) (map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt fw ifs)))

proof (goal-cases)

1 case:

have c: ∃ mr ar mf af a. [(mr, ar) ∈ set (lr-of-tran-s1 rt); (mf, af) ∈ simple-rule-dtor ‘ set fw; simple-match-and mr mf
= Some a] ⟷ oiface a = iface.Any

proof (goal-cases)

1 case:

have oiface mr = iface.Any using 1(1) unfolding lr-of-tran-s1-def route2match-def by(clarsimp simp add: Set.image-iff)

moreover have oiface mf = iface.Any using 1(2) (no-oif-match fw) unfolding no-oif-match-def simple-rule-dtor-def[abs-def]

by(clarsimp simp: list-all-iff split: simple-rule.splits) fastforce

ultimately show ?case using 1(3) by(cases a; cases mr; cases mf) (simp add: iface-conjunct iface.Any split: option.splits)

qed

have l: list-all (λ m. oiface (fst m) = iface.Any) (lr-of-tran-fbs rt fw ifs)

unfolding lr-of-tran-fbs-def Let-def list-all-iff
apply(clarify)
apply(subst(asm) generalized-sfw-join-set)
apply(clarsimp)

using c by blast

thus ?case
proof (goal-cases)
case 1 
  have \( \lambda m. \text{iface} (f \text{st} (\text{snd} m)) = \text{ifaceAny} \) \( \lambda m. \text{iface} (f \text{st} m) = \text{ifaceAny} \) \( \text{snd} \) unfolding comp-def ..
  show ?case unfolding \* list-all-map \{ symmetric \} map-snd-apfst map-snd-annotate-rlen using la . 
  qed 

qed

lemma lr-of-tran-correct:

fixes p :: \((32', 'a)\) simple-packet-ext-scheme

assumes nerr: lr-of-tran rt fw ifs = Inr oft 
and ipptk: p-l2type p = 0x800 
and ifeld: p-iface p \( \in \) set ifs

shows OF-priority-match OF-match-fields-safe oft p = Action \[ \text{Forward} \text{of} \] \( \leftrightarrow \) simple-linux-router-nol12 rt fw p = \( \text{Some} \) \( (p(p\text{-iface} := \text{of}')) \)

OF-priority-match OF-match-fields-safe oft p \( \neq \) NoAction OF-priority-match OF-match-fields-safe oft p \( \neq \) Undefined

OF-priority-match OF-match-fields-safe oft p = Action ls \( \rightarrow \) length ls \( \leq \) 1

\( \exists \text{ls} . \) length ls \( \leq \) 1 \( \land \) OF-priority-match OF-match-fields-safe oft p = Action ls

proof

have s1: valid-prefixes rt has-default-route rt 
and s2: has-default-policy fw simple-fw-valid fw no-oif-match fw 
and dfs: distinct ifs

using nerr unfolding lr-of-tran-def by (simp-all split: if-splits)

have no-oif-match fw using nerr unfolding lr-of-tran-def by (simp split: if-splits)

note s2 = s2 this 

have unsafe-safe-eq: 


apply subst OF-unsafe-safe-match3-eq (rule lr-of-tran-prereqs s1 s2 nerr refl)+

apply subst OF-unsafe-safe-linear-eq (rule lr-of-tran-prereqs s1 s2 nerr refl)+

done


using OF-eq (OF lr-of-tran-no-overlaps lr-of-tran-sorted-descending, OF dfs nerr)\{ symmetric \} nerr\{ symmetric \] unfolding fun-eq-iff unsafe-safe-eq by metis

let ?ard = map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt fw ifs))

have oft-def: oft = pack-OF-entries ifs \?ard using nerr unfolding lr-of-tran-def Let-def by (simp split: if-splits)

have vld: list-all simple-match-valid (map (fst o snd) \?ard)

unfolding fun-app-def map-map \{ symmetric \} snd-apfst map-snd-apfst map-snd-annotate-rlen using simple-match-linear-fbs OF s111 s222 unfolding

have \*: list-all (\( \lambda m. \text{iface} (f \text{st} (\text{snd} m)) = \text{ifaceAny} \)) ?ard using no-oif-match-fbs OF s23 .

have not-unde: \( \land p. \text{simple-fw fw p \neq Undefined} \) by (metis has-default-policy s21 state.simps(3))

have w1-1: \( \lambda oif. \) OF-match-linear OF-match-fields-safe oft p = Action \[ \text{Forward} \text{of} \] \( \Rightarrow \) simple-linux-router-nol12 rt fw p = \( \text{Some} \) \( (p(p\text{-iface} := \text{of}')) \)

\( \land \) oif = output-iface (routing-table-semantics rt (p-dst p))

proof [intro conj, goal-cases]

case \( \{ 1 \text{ oif} \} \)

note s3-correct \( (OF \text{vld ipptk ifeld(1) *}, \text{THEN} \text{iffO1}, \text{unfolded oft-def}\{\text{symmetric}, \text{OF \ 1} \}) \)

then obtain r where generalized-sfw (map snd (map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt fw ifs)))) \( = \) \( \text{Some} \) \( (r, (oif, \text{simple-action.Accept})) \)

by (clarsimp split: if-splits)

then obtain r where generalized-sfw (lr-of-tran-fbs rt fw ifs) \( = \) \( \text{Some} \) \( (r, (oif, \text{simple-action.Accept})) \)

unfolding map-comp-def snd-apfst map-snd-annotate-rlen by blast

36
thus \( \text{case using } \text{lr-of-tran-fbs-acceptD}(OF \ s1 \ s2(3)) \text{ by metis} \)

thus \( \text{oif} = \text{output-iface (routing-table-semantics rt } (p \text{-dst } p)) \)

by(cases \( p \)) (clarsimp simp: simple-linux-router-nol12-def Let-def not-undec split: Option.bind-splits state.splits

final-decision.splits)

qed

have \( w1-2: \forall \text{oif}. \text{simple-linux-router-nol12 } rt \text{ fw } p = \text{Some } (p|p\text{-oiface := oif}) \implies \text{OF-match-linear OF-match-fields-safe} \)

oif \( p = \text{Action } [\text{Forward oif}] \)

proof (goal-cases)

  case (1 \( oif \))

    note \( \text{lr-of-tran-fbs-acceptI}(OF \ s1 \ s2(3) \ s2(1) \text{ this} \ _{\text{ifs}}) \) then guess \( r \) .. note \( r = \text{this} \)

    hence \( \text{generalized-sfw (map snd (map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt } \text{ fw } \text{ifs}))) } p = \text{Some } (r, (oif, simple-action.Accept)) \)

    unfolding map snd apfst map snd annotate rlen .

    moreover note \( s3\text{-correct[}(OF \ vld \text{ ippkt } \text{ifold}(1)) \ast, \text{THEN iffD2, unfolded oft-def[symmetric], of } [\text{Forward oif}] \)

    ultimately show \( \text{?case by simp} \)

qed

show \( w1: \forall \text{oif}. (\text{OF-priority-match OF-match-fields-safe} \text{ oif } p = \text{Action } [\text{Forward oif}]) = (\text{simple-linux-router-nol12 } rt \text{ fw } p = \text{Some } (p|p\text{-oiface := oif}))) \)

  unfolding lin using \( w1-1 \ w1-2 \) by blast

show \( w2: (\text{OF-priority-match OF-match-fields-safe} \text{ oif } p = \text{Action } []) = (\text{simple-linux-router-nol12 } rt \text{ fw } p = \text{None}) \)

unfolding lin

proof (rule iffI, goal-cases)

  case 1

    note \( s3\text{-correct[}(OF \ vld \text{ ippkt } \text{ifold}(1)) \ast, \text{THEN iffD1, unfolded oft-def[symmetric], OF } 1 \)

then obtain \( r \) oif \( \text{where roif: generalized-sfw (lr-of-tran-fbs rt } \text{ fw } \text{ifs} p = \text{Some } (r, oif, simple-action.Drop) \)

  unfolding map snd apfst map snd annotate rlen by(clarsimp split: if-splits)

note \( \text{lr-of-tran-fbs-dropD}(OF \ s1 s2(3) \text{ this}) \)

thus \( \text{?case} . \)

next

  case 2

  note \( \text{lr-of-tran-fbs-dropI}(OF \ s1 s2(3) \ s2(1) \text{ this} \ _{\text{ifs}}) \) then

  obtain \( r \) oif \( \text{where generalized-sfw (lr-of-tran-fbs rt } \text{ fw } \text{ifs} p = \text{Some } (r, oif, simple-action.Drop) \) by blast

  hence \( \text{generalized-sfw (map snd (map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt } \text{ fw } \text{ifs}))) } p = \text{Some } (r, oif, \text{simple-action.Drop}) \)

  unfolding map snd apfst map snd annotate rlen .

  moreover note \( s3\text{-correct[}(OF \ vld \text{ ippkt } \text{ifold}(1)) \ast, \text{THEN iffD2, unfolded oft-def[symmetric], of } [] \)

  ultimately show \( \text{?case by force} \)

qed

have \( \text{br-determ: } \forall a. \text{simple-linux-router-nol12 } rt \text{ fw } p = \text{Some } a \implies a = p|p\text{-oiface := output-iface (routing-table-semantics rt } (p\text{-dst } p)) \)

by(clarsimp simp: simple-linux-router-nol12-def Let-def not-undec split: Option.bind-splits state.splits final-decision.splits)

show notation \( \text{OF-priority-match OF-match-fields-safe} \text{ oif } p \neq \text{NoAction} \)

apply(cases simple-linux-router-nol12 rt fw p)

using \( w2 \) apply(simp)

using \( w1[\text{of output-iface (routing-table-semantics rt } (p\text{-dst } p)) ] \) apply(simp)

apply(drule br-determ)

apply(simp)

done

show notb: \( \forall s. \text{OF-priority-match OF-match-fields-safe} \text{ oif } p = \text{Action } s \implies \text{length } s \leq 1 \)

apply(cases simple-linux-router-nol12 rt fw p)

using \( w2 \) apply(simp)

using \( w1[\text{of output-iface (routing-table-semantics rt } (p\text{-dst } p)) ] \) apply(simp)

37
apply (drule lr-determ)
apply (clarsimp)
done

show \exists ls. length ls \leq 1 \land OF-priority-match OF-match-fields-safe oft p = Action ls
apply (cases OF-priority-match OF-match-fields-safe oft p)
using notmult apply blast
using notno apply blast
using notub apply blast
done

done

qed

end

theory OF-conv-test

imports
Iptables-Semantics.Parser
Simple-Firewall.SimpleFw-toString
Routing.IpRoute-Parser
../../LinuxRouter-OpenFlow-Translation
../../OpenFlow-Serialize

begin

parse-iptables-save SQRL-fw = iptables-save

term SQRL-fw

thm SQRL-fw-def

thm SQRL-fw-FORWARD-default-policy-def

value [code] map (\lambda (c, rs). (c, map (quote-rewrite o common-primitive-rule-toString) rs)) SQRL-fw

definition unfolded = unfold-ruleset-FORWARD SQRL-fw-FORWARD-default-policy (map-of-string-ipv4 SQRL-fw)

lemma map (quote-rewrite o common-primitive-rule-toString) unfolded =
["p icmp -j ACCEPT",
"-i s l -lan -p tcp -m tcp --spts [1024:65535] -m tcp --dpts [80] -j ACCEPT",
"-i s l -wan -p tcp -m tcp --spts [80] -m tcp --dpts [1024:65535] -j ACCEPT",
"-j DROP"] by eval

lemma length unfolded = 4 by eval

value [code] map (quote-rewrite o common-primitive-rule-toString) (upper-closure unfolded)
lemma length (upper-closure unfolded) = 4 by eval

value [code] upper-closure (packet-assume-new unfolded)

lemma length (lower-closure unfolded) = 4 by eval

lemma check-simple-fw-preconditions (upper-closure unfolded) = True by eval
lemma \forall m \in get-match'set (upper-closure (packet-assume-new unfolded)), normalized-nnf-match m by eval
lemma \forall m \in get-match'set (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded))), normalized-nnf-match m by eval

38
lemma check-simple-fu-preconditions (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded)))) by eval

lemma length (to-simple-firewall (upper-closure (packet-assume-new unfolded))) = 4 by eval

lemma (lower-closure (optimize-matches abstract-for-simple-firewall (lower-closure (packet-assume-new unfolded))))) = lower-closure unfolded
  lower-closure unfolded = upper-closure unfolded
  (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded))))) = upper-closure unfolded by eval+

value[code] (getParts (to-simple-firewall (lower-closure (optimize-matches abstract-for-simple-firewall (lower-closure (packet-assume-new unfolded))))))

definition SQRL-fw-simple ≡ remdups-rev (to-simple-firewall (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded)))))

value[code] SQRL-fw-simple

lemma simple-fw-valid SQRL-fw-simple by eval

parse-ip-route SQRL-rtbl-main = ip−route

value SQRL-rtbl-main

lemma SQRL-rtbl-main = [[routing-match = PrefixMatch 0xA000100 24, metric = 0, routing-action = {output-iface = "s1−lan", next-hop = None}]],
  [[routing-match = PrefixMatch 0xA000200 24, metric = 0, routing-action = {output-iface = "s1−wan", next-hop = None}]],
  [[routing-match = PrefixMatch 0 0, metric = 0, routing-action = {output-iface = "s1−wan", next-hop = Some 0xA000201}]] by eval

value dotdecimal-of-ipv4addr 0xA0D2500

lemma SQRL-rtbl-main-sorted ≡ rev (sort-key (λ. pfxm-length (routing-match r)) SQRL-rtbl-main)

value SQRL-rtbl-main-sorted

definition SQRL-ifs ≡ [ [iface-name = "s1−lan", iface-mac = 0x10001], [iface-name = "s1−wan", iface-mac = 0x10002] ]

value SQRL-ifs

definition SQRL-macs ≡ [ "s1−lan", ipv4addr-of-dotdecimal (10.0.1.2), 0x1],
  "s1−lan", ipv4addr-of-dotdecimal (10.0.1.3), 0x2),
  "s1−wan", ipv4addr-of-dotdecimal (10.0.2.1), 0x3)

value SQRL-ifs

definition SQRL-ports ≡ [ "s1−lan", "1"]
lemma let fu = SQRL-fu-simple in no-if-match fu ∧ has-default-policy fu ∧ simple-fu-valid fu by eval

lemma let rt = SQRL-rtbl-main-sorted in valid-prefxes rt ∧ has-default-route rt by eval

lemma let ifs = (map iface-name SQRL-ifs) in distinct ifs by eval

definition of ≡
case (br-of-tran SQRL-rtbl-main-sorted SQRL-fu-simple (map iface-name SQRL-ifs))
of (Hr openflow-rules) ⇒ map (serialize-of-entry (the o map-of SQRL-ports)) openflow-rules

lemma of =

''"\text{40}""
value[code] of

end
theory RFC2544
imports
   Iptables-Semantics.Parser
   Routing.IpRoute-Parser
   ../../../LinuxRouter-OpenFlow-Translation
   ../../../OpenFlow-Serialize
begin

parse-iptables-save SQRL-fw=iptables-save

term SQRL-fw
thm SQRL-fw-def
thm SQRL-fw-FORWARD-default-policy-def

value[code] map (λ(c,rs). (c, map (quote-rewrite ◦ common-primitive-rule-toString) rs)) SQRL-fw

definition unfolded = unfold-ruleset-FORWARD SQRL-fw-FORWARD-default-policy (map-of-string-ipv4 SQRL-fw)

lemma length unfolded = 26 by eval

value[code] unfolded
value[code] (upper-closure unfolded)
value[code] map (quote-rewrite ◦ common-primitive-rule-toString) (upper-closure unfolded)

lemma length (upper-closure unfolded) = 26 by eval

value[code] upper-closure (packet-assume-new unfolded)

lemma length (lower-closure unfolded) = 26 by eval

lemma check-simple-fw-preconditions (upper-closure unfolded) by eval
lemma ∀ m ∈ get-match’set (upper-closure (packet-assume-new unfolded)). normalized-nnf-match m by eval
lemma ∀ m ∈ get-match’set (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded))). normalized-nnf-match m by eval
lemma check-simple-fw-preconditions (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (package-assume-new unfolded)))) by eval
lemma length (to-simple-firewall (upper-closure (packet-assume-new unfolded))) = 26 by eval

lemma (lower-closure (optimize-matches abstract-for-simple-firewall (lower-closure (packet-assume-new unfolded)))) = lower-closure unfolded
   lower-closure unfolded = upper-closure unfolded
   (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded)))) = upper-closure unfolded by eval+

value[code] (getParts (to-simple-firewall (lower-closure (optimize-matches abstract-for-simple-firewall (lower-closure
(packet-assume-new unfolded)))))

41
```

definition SQRL-fu-simple ≡ remdups-rev (to-simple-firewall (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new-unfolded)))))
value [code] SQRL-fu-simple
lemma simple-fu-valid SQRL-fu-simple by eval

parse-ip-route SQRL-rtbl-main = ip-route

value SQRL-rtbl-main

lemma SQRL-rtbl-main = [\{routing-match = PrefixMatch 0xC6120100 24, metric = 0, routing-action = \{output-iface = "ip1", next-hop = None\}\},
\{routing-match = PrefixMatch 0xC6130100 24, metric = 0, routing-action = \{output-iface = "ip1", next-hop = None\}\},
\{routing-match = PrefixMatch 0 0, metric = 0, routing-action = \{output-iface = "ip1", next-hop = Some 0x26130102\}\}] by eval

lemma SQRL-rtbl-main = [
rr-ctor (198,18,1,0) 24 "ip1" None 0,
rr-ctor (198,19,1,0) 24 "ip1" None 0,
rr-ctor (0,0,0,0) 0 "ip1" (Some (198,19,1,2)) 0
] by eval

definition SQRL-ports ≡ [
("ip1", "1"),
("ip1", "2")
]

definition ofi ≡
case (br-of-tran SQRL-rtbl-main SQRL-fu-simple (map fst SQRL-ports)) of
(map (_hr openflow-rules) ⇒ map (serialize-of-entry (the map-of SQRL-ports)) openflow-rules)

lemma ofi =
["priority=27,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-dst=198.18.1.0/24,action=drop",
"priority=26,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-dst=198.19.1.0/24,action=drop",
"priority=25,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.1.1/32,nw-dst=192.18.101.1/32,action=drop",
"priority=24,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.2.3/32,nw-dst=192.18.102.2/32,action=drop",
"priority=23,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.3.3/32,nw-dst=192.18.103.3/32,action=drop",
"priority=22,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.4.4/32,nw-dst=192.18.104.4/32,action=drop",
"priority=21,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.5.5/32,nw-dst=192.18.105.5/32,action=drop",
"priority=20,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.6.6/32,nw-dst=192.18.106.6/32,action=drop",
"priority=19,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.7.7/32,nw-dst=192.18.107.7/32,action=drop",
"priority=18,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.8.8/32,nw-dst=192.18.108.8/32,action=drop",
"priority=17,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.9.9/32,nw-dst=192.18.109.9/32,action=drop",
"priority=16,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.10.10/32,nw-dst=192.18.110.10/32,action=drop",
"priority=15,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.11.11/32,nw-dst=192.18.111.11/32,action=drop",
"priority=14,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.12.12/32,nw-dst=192.18.112.12/32,action=drop",
"priority=13,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.19.1.3/32,nw-dst=192.19.65.1/32,action=drop",
"priority=12,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.13.13/32,nw-dst=192.18.113.13/32,action=drop",
"priority=11,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.14.14/32,nw-dst=192.18.114.14/32,action=drop",
"priority=10,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.15.15/32,nw-dst=192.18.115.15/32,action=drop",
"priority=9,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.16.16/32,nw-dst=192.18.116.16/32,action=drop",
"priority=8,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.17.17/32,nw-dst=192.18.117.17/32,action=drop",
"priority=7,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.18.18/32,nw-dst=192.18.118.18/32,action=drop",
"priority=6,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.19.19/32,nw-dst=192.18.119.19/32,action=drop",
"priority=5,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.20.20/32,nw-dst=192.18.120.20/32,action=drop",
"priority=4,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.21.21/32,nw-dst=192.18.121.21/32,action=drop",
"priority=3,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.22.22/32,nw-dst=192.18.122.22/32,action=drop"
]
```

42
value[0] length off
Part II
Documentation

1 Configuration Translation

All the results we present in this section are formalized and verified in Isabelle/HOL [11]. This means that their formal correctness can be trusted a level close to absolute certainty. The definitions and lemmas stated here are merely a repetition of lemmas stated in other theory files. This means that they have been directly set to this document from Isabelle and no typos or hidden assumptions are possible. Additionally, it allows us to omit various helper lemmas that do not help the understanding. However, it causes some notation inaccuracy, as type and function definitions are stated as lemmas or schematic goals.

theory OpenFlow-Documentation

1.1 Linux Firewall Model

We want to write a program that translates the configuration of a linux firewall to that of an OpenFlow switch. We furthermore want to verify that translation. For this purpose, we need a clear definition of the behavior of the two device types – we need their models and semantics. In case of a linux firewall, this is problematic because a linux firewall is a highly complex device that is ultimately capable of general purpose computation. Creating a comprehensive semantics that encompasses all possible configuration types of a linux firewall is thus highly non-trivial and not useful for the purpose of analysis. We decided to approach the problem from the other side: we created a model that includes only the most basic features. (This implies neglecting IPv6.) Fortunately, many of the highly complex features are rarely essential and even our basic model is still of some use.

We first divided the firewall into subsystems. Given a routing table \( rt \), the firewall rules \( fw \), the routing decision for a packet \( p \) can be obtained by \( \text{routing-table-semantics } rt \) (\( p \)-dst \( p \)), the firewall decision by \( \text{simple-fw } fw \) \( p \). We draft the first description of our linux router model:

1. The destination MAC address of an arriving packet is checked: Does it match the MAC address of the ingress port? If it does, we continue, otherwise, the packet is discarded.
2. The routing decision \( rd \equiv \text{routing-table-semantics } rt \) \( p \) is obtained.
3. The packet’s output interface is updated based on \( rd \).
4. The firewall is queried for a decision: \( \text{simple-fw } fw \) \( p \). If the decision is to Drop, the packet is discarded.
5. The next hop is computed: If \( rd \) provides a next hop, that is used. Otherwise, the destination address of the packet is used.
6. The MAC address of the next hop is looked up; the packet is updated with it and sent.

We decided that this description is best formalized as an abortable program in the option monad:

\[
\text{lemma simple-linux-router } rt \ fw \ mlf \ ifl \ p \equiv \text{do } \{ \\
- \leftarrow \text{iface-packet-check } ifl \ p; \\
\text{let } rd \equiv \text{routing-table-semantics } rt (p\text{-dst } p); \\
\text{let } p = p\langle p\text{-oiface} := \text{output-iface } rd\rangle; \\
\text{let } fd \equiv \text{firewall decision} = \text{simple-fw } fw \ p; \\
- \leftarrow (\text{case } fd \text{ of Decision FinalAllow} \Rightarrow \text{Some } () | \text{Decision FinalDeny} \Rightarrow \text{None}); \\
\text{let } nh = (\text{case } \text{next-hop } rd \text{ of None } \Rightarrow p\text{-dst } p \mid \text{Some } a \Rightarrow a); \\
\text{ma } \leftarrow \text{mlf } nh; \\
\text{Some } (p\langle p\text{-l2dst} := ma\rangle) \}
\]

\text{unfolding \text{fromMaybe-def [symmetric] by (fact simple-linux-router-def)}

where \text{mlf} is a function that looks up the MAC address for an IP address.

There are already a few important aspects that have not been modelled, but they are not core essential for the functionality of a firewall. Namely, there is no local traffic from/to the firewall. This is problematic since this model can not generate ARP replies — thus, an equivalent OpenFlow device will not do so, either. Furthermore, this model is problematic because it requires access to a function that looks up a MAC address, something that may not be known at the time of time running a translation to an OpenFlow configuration.

\footnote{Note that we assume a packet model with input and output interfaces. The origin of this is explained in Section 1.1.2}
It is possible to circumvent these problems by inserting static ARP table entries in the directly connected devices and looking up their MAC addresses a priori. A test-wise implementation of the translation based on this model showed acceptable results. However, we deemed the a priori lookup of the MAC addresses to be rather inelegant and built a second model.

**definition** simple-linux-router-altered rt fw ifl p ≡ do
let rd = routing-table-semantics rt (p-dst p);
let p = p(p-oiface := output-iface rd);
- ⊢ if p-oiface p = p-iface p then None else Some ();
let fd = simple-fw fw p;
- ⊢ (case fd of Decision FinalAllow ⇒ Some () | Decision FinalDeny ⇒ None);
Some p

In this model, all access to the MAC layer has been eliminated. This is done by the approximation that the firewall will be asked to route a packet (i.e. be addressed on the MAC layer) iff the destination IP address of the packet causes it to be routed out on a different interface. Because this model does not insert destination MAC addresses, the destination MAC address has to be already correct when the packet is sent. This can only be achieved by changing the subnet of all connected device, moving them into one common subnet².

While a test-wise implementation based on this model also showed acceptable results, the model is still problematic. The check p-oiface p = p-iface p and the firewall require access to the output interface to determine the MAC address of the packet. This is done by the approximation that the packet has the correct destination MAC never received packet has the correct destination MAC.

A routing table is then a list of these entries:

```haskell
let rd = routing-table-semantics rt (p-dst p);
let p = p(p-oiface := output-iface rd);
- ⊢ (case fd of Decision FinalAllow ⇒ Some () | Decision FinalDeny ⇒ None);
Some p
```

We thus simplified the model even further:

```haskell
let rd = routing-table-semantics rt (p-dst p);
let p = p(p-oiface := output-iface rd);
- ⊢ (case fd of Decision FinalAllow ⇒ Some () | Decision FinalDeny ⇒ None);
Some p
```

We continue with this definition as a basis for our translation. Even this strongly altered version and the original Linux firewall still behave the same in a substantial amount of cases:

²There are cases where this is not possible — A limitation of our system.

**theorem**

```haskell
ifact-face-packet-check ifl ppi ̸= None;
mlf (case next-hop (routing-table-semantics rt (p-dst ppi))
of None ⇒ p-dst ppi | Some a ⇒ a) ̸= None) ⇒
  ∃ x. map-option (λp. p[p-l2dst := x])
  (simple-linux-router-nol12 rt fw ppi) = simple-linux-router
  rt fw mlf ifl ppi
```

The conditions are to be read as “The check whether a received packet has the correct destination MAC never returns False” and “The next hop MAC address for all packets can be looked up”. Obviously, these conditions do not hold for all packets. We will show an example where this makes a difference in Section 2.1.

### 1.1.1 Routing Table

The routing system in Linux features multiple tables and a system that can use the iptables firewall and an additional match language to select a routing table. Based on our directive, we only focused on the single most used `main` routing table.

We define a routing table entry to be a record (named tuple) of a prefix match, a metric and the routing action, which in turn is a record of an output interface and an optional next-hop address.

```haskell
schematic-goal (?rtbl-entry :: (a::len) routing-rule) = (rtbl-entry)
  routing-match = PrefixMatch pfx len, metric = met,
  routing-action = (output-iface = oif-string, next-hop = (h :: 'a word option) | |) ..
```

A routing table is then a list of these entries:

```haskell
lemma (rtbl :: ('a :: len) prefix-routing) = (rtbl :: 'a routing-rule list) by rule
```

Not all members of the type `prefix-routing` are sane routing tables. There are three different validity criteria that we require so that our definitions are adequate.

- The prefixes have to be 0 in bits exceeding their length.
- There has to be a default rule, i.e. one with prefix length 0. With the condition above, that implies that all its prefix bits are zero and it thus matches any address.
- The entries have to be sorted by prefix length and metric.
The first two are set into code in the following way:

```plaintext
lemma valid-prefix (PrefixMatch pfx len) ≡ pfx && (2 ^ (32 - len) - 1) = (0 :: 32 word)
```

unfolding valid-prefix-def pfzm-mask-def mask-def by (simp add: word-bw-comms(1))

```plaintext
lemma has-default-route rt ←→ (∃ r ∈ set rt. pfzm-length (routing-match r) = 0)
```

by (fact has-default-route-alt)

The third is not needed in any of the further proofs, so we omit it.

The semantics of a routing table is to simply traverse the list until a matching entry is found.

```plaintext
schematic-goal routing-table-semantics (rt-entry # rt) dst-addr = (if prefix-match-semantics (routing-match rt-entry) dst-addr then routing-action rt-entry else routing-table-semantics rt dst-addr) by (fact routing-table-semantics.simps)
```

If no matching entry is found, the behavior is undefined.

### 1.1.2 iptables Firewall

The firewall subsystem in a linux router is not any less complex than any of the other systems. Fortunately, this complexity has been dealt with in [6, 5] already and we can directly use the result.

In short, one of the results is that a complex iptables configuration can be simplified to be represented by a single list of matches that only support the following match conditions:

- (String) prefix matches on the input and output interfaces.
- A `prefix-match` on the source and destination IP address.
- An exact match on the layer 4 protocol.
- Interval matches on the source or destination port, e.g. `p_d ∈ {1..0x3FF}

The model/type of the packet is adjusted to fit that: it is a record of the fields matched on. This also means that input and output interface are coded to the packet. Given that this information is usually stored alongside the packet content, this can be deemed a reasonable model. In case the output interface is not needed (e.g., when evaluating an OpenFlow table), it can simply be left blank.

Obviously, a simplification into the above match type cannot always produce an equivalent firewall, and the set of accepted packets has to be over- or underapproximated. The reader interested in the details of this is strongly referred to [6]; we are simply going to continue with the result: `simple-fw`.

One property of the simplification is worth noting here: The simplified firewall does not know state and the simplification approximates stateful matches by stateless ones. Thus, the overapproximation of a stateful firewall ruleset that begins with accepting packets of established connections usually begins with a rule that accepts all packets. Dealing with this by writing a meaningful simplification of stateful firewalls is future work.

### 1.2 OpenFlow Switch Model

In this section, we present our model of an OpenFlow switch. The requirements for this model are derived from the fact that it models devices that are the target of a configuration translation. This has two implications:

- All configurations that are representable in our model should produce the correct behavior wrt. their semantics. The problem is that correct here means that the behavior is the same that any real device would produce. Since we cannot possibly account for all device types, we instead focus on those that conform to the OpenFlow specifications. To account for the multiple different versions of the specification (e.g. [2, 3]), we tried making our model a subset of both the oldest stable version 1.0 [2] and the newest available specification version 1.5.1 [3].

- Conversely, our model does not need to represent all possible behavior of an OpenFlow switch, just the behavior that can be invoked by the result of our translation. This is especially useful regarding controller interaction, but also for MPLS or VLANs, which we did not model in Section 1.1.

More concretely, we set the following rough outline for our model.

- A switch consists of a single flow table.
- A flow table entry consists of a priority, a match condition and an action list.
The only possible action (we require) is to forward the packet on a port.

We do not model controller interaction.

Additionally, we decided that we wanted to be able to ensure the validity of the flow table in all qualities, i.e. we want to model the conditions ‘no overlapping flow entries appear’, ‘all match conditions have their necessary preconditions’. The details of this are explained in the following sections.

1.2.1 Matching Flow Table entries

Table 3 of Section 3.1 of [2] gives a list of required packet fields that can be used to match packets. This directly translates into the type for a match expression on a single field:

schematic-goal (field-match :: of-match-field) ∈ {  
IngressPort (?s::string),  
EtherSrc (?as::48 word), EtherDst (?ad::48 word),  
EtherType (?t::16 word),  
VlanId (?vi::16 word), VlanPriority (?vp::16 word),  
IPv4Src (?ps::32 prefix-match),  
IPv4Dst (?pd::32 prefix-match),  
IPv4Proto (?ip::8 word),  
L4Src (?ps :: 16 word) (?ms :: 16 word),  
L4Dst (?pd :: 16 word) (?md :: 16 word)  
} by(fact of-match-field-typeset)

Two things are worth additional mention: L3 and L4 "addresses". The IPv4Src and IPv4Dst matches are specified as “can be subnet masked” in [2], whereas [3] states clearly that arbitrary bitmasks can be used. We took the conservative approach here. Our alteration of L4Src and L4Dst is more grave. While [2] does not state anything about layer 4 ports and masks, [3] specifically forbids using masks on them. Nevertheless, OpenVSwitch [1] and some other implementations support them. We will explain in detail why we must include bitmasks on layer 4 ports to obtain a meaningful translation in Section 1.3.

One of-match-field is not enough to classify a packet. To match packets, we thus use entire sets of match fields. As Guha et al. [7] noted, executing a set of given of-match-fields on a packet requires careful consideration. For example, it is not meaningful to use IPv4Dst if the given packet is not actually an IP packet, i.e. IPv4Dst has the prerequisite of EtherType 0x800 being among the match fields. Guha et al. decided to use the fact that the preconditions can be arranged on a directed acyclic graph (or rather: an acyclic forest). They evaluated match conditions in a manner following that graph: first, all field matches without preconditions are evaluated. Upon evaluating a field match (e.g., EtherType 0x800), the matches that had their precondition fulfilled by it (e.g., IPv4Src and IPv4Src in this example) are evaluated. This mirrors the faulty behavior of some implementations (see [7]). Adopting that behavior into our model would mean that any packet matches against the field match set \{IPv4Dst (PrefixMatch 0x8080808 32)\} instead of just those destined for 8.8.8.8 or causing an error. We found this to be unsatisfactory.

To solve this problem, we made three definitions. The first, match-no-prereq matches an of-match-field against a packet without considering prequisites. The second, prerequisites, checks for a given of-match-field whether its prequisites are in a set of given match fields. Especially:

- **Lemma**
  - prerequisite \( (\text{VlanPriority \( p \))} \) \( m = (\exists \text{id. let } v = \text{VlanId id in } v \in m \wedge \text{prerequisites } v m) \)
  - prerequisite \( (\text{IPv4Proto \( p \))} \) \( m = (\exists v = \text{EtherType 0x8000 in } v \in m \wedge \text{prerequisites } v m) \)
  - prerequisite \( (\text{IPv4Src \( p \))} \) \( m = (\exists v = \text{EtherType 0x8000 in } v \in m \wedge \text{prerequisites } v m) \)
  - prerequisite \( (\text{IPv4Dst \( p \))} \) \( m = (\exists v = \text{EtherType 0x8000 in } v \in m \wedge \text{prerequisites } v m) \)

Then, to actually match a set of of-match-field against a packet, we use the option type:

- **Lemma**
  - \( \text{OF-match-fields } m p = \)
    - \( (\text{if } \exists f \in m. \neg \text{prerequisites } f \text{ then } \text{None else}) \)
    - \( (\text{if } \forall f \in m. \text{match-no-prereq } f \text{ then } \text{Some True else Some False}) \)

1.2.2 Evaluating a Flow Table

In the previous section, we explained how we match the set of match fields belonging to a single flow entry against a packet. This section explains how the correct

---

3See also: [8, section 2.3]
flow entry from a table can be selected. To prevent to much entanglement with the previous section, we assume an arbitrary match function \( \gamma \). This function \( \gamma \) takes the match condition \( m \) from a flow entry \( \text{OFEntry priority} \ a \ \text{action} \) and decides whether a packet matches those.

The flow table is simply a list of flow table entries \( \text{flow-entry-match} \). Deciding the right flow entry to use for a given packet is explained in the OpenFlow specification [2], Section 3.4:

Packets are matched against flow entries based on prioritization. An entry that specifies an exact match (i.e., has no wildcards) is always the highest priority\(^4\).

All wildcard entries have a priority associated with them. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering.

We use the term “overlapping” for the flow entries that can cause a packet to match multiple flow entries with the same priority. Guha et al. [7] have dealt with overlapping. However, the semantics for a flow table they presented [7, Figure 5] is slightly different from what they actually used in their theory files. We have tried to reproduce the original inductive definition (while keeping our abstraction \( \gamma \), in Isabelle/HOL\(^5\):

\[ \text{lemma } \gamma (\text{ofe-fields } fe) \ p = True \Rightarrow \forall fe' \in \text{set} (ft1 @ ft2), \text{ofe-prio } fe' \geq \text{ofe-prio } fe \Rightarrow \gamma \ (\text{ofe-fields } fe') \ p = False \Rightarrow \text{guha-table-semantics } \gamma (ft1 @ fe \# ft2) \ p \ (\text{Some (ofe-action } fe)) \]

\[ \forall fe \in \text{set } ft. \gamma (\text{ofe-fields } fe) \ p = False \Rightarrow \text{guha-table-semantics } \gamma ft p \text{ None by (fact guha-matched guha-unmatched)+} \]

Guha et al. have deliberately made their semantics non-deterministic, to match the fact that the switch “may choose any ordering”. This can lead to undesired results:

\[ \text{lemma } \text{CARD(action)} \geq 2 \Rightarrow \exists ff. \gamma ff p \Rightarrow \exists ft (a1 :: action) (a2 :: action). a1 \neq a2 \land \text{guha-table-semantics } \gamma ft p \ (\text{Some a1}) \land \text{guha-table-semantics } \gamma ft p \ (\text{Some a2}) \text{ by (fact guha-table-semantics-ex2res)} \]

This means that, given at least two distinct actions exist and our matcher \( \gamma \) is not false for all possible match conditions, we can say that a flow table and two actions exist such that both actions are executed. This can be misleading, as the switch might choose an ordering on some flow table and never execute some of the (overlapped) actions.

Instead, we decided to follow Section 5.3 of the specification [3], which states:

If there are multiple matching flow entries, the selected flow entry is explicitly undefined.

This still leaves some room for interpretation, but it clearly states that overlapping flow entries are undefined behavior, and undefined behavior should not be invoked. Thus, we came up with a semantics that clearly indicates when undefined behavior has been invoked:

**lemma**

\[ \text{OF-priority-match } \gamma \text{ flow-entries packet } = ( \]

\[ \text{let } m = \text{filter } (\lambda ff. \gamma (\text{ofe-fields } ft) \text{ packet}) \text{ flow-entries; } \]

\[ m' = \text{filter } (\lambda ff. \forall fo \in \text{ set } m. \text{ofe-prio } fo \leq \text{ofe-prio } f) \text{ m in } \]

\[ \text{case } m' \text{ of } [] \Rightarrow \text{NoAction} \]

\[ | [s] \Rightarrow \text{Action (ofe-action } s) \]

\[ | - \Rightarrow \text{Undefined} \]

**unfolding** \( \text{OF-priority-match-def} \).

The definition works the following way\(^6\):

1. The flow table is filtered for those entries that match, the result is called \( m \).
2. \( m \) is filtered again, leaving only those entries for which no entries with lower priority could be found, i.e. the matching flow table entries with minimal priority. The result is called \( m' \).
3. A case distinction on \( m' \) is made. If only one matching entry was found, its action is returned for execution. If \( m \) is empty, the flow table semantics returns \( \text{NoAction} \) to indicate that the flow table does not decide an action for the packet. If, not zero or one entry is found, but more, the special value \( \text{Undefined} \) for indicating undefined behavior is returned.

\(^4\)This behavior has been deprecated.

\(^5\)The original is written in Coq [4] and we can not use it directly.

\(^6\)Note that the order of the flow table entries is irrelevant. We could have made this definition on sets but chose not to for consistency.
The use of Undefined immediately raises the question in which condition it cannot occur. We give the following definition:

**Lemma** check-no-overlap γ ft = (∀a ∈ set ft. ∀b ∈ set ft. (a ≠ b ∧ ofe-prio a = ofe-prio b) → ¬(∃p. γ (ofe-fields a) p ∧ γ (ofe-fields b) p)) unfolding check-no-overlap-alt check-no-overlap2-def by force

Together with distinctness of the flow table, this provides the absence of Undefined⁷:

**Lemma** [check-no-overlap γ ft; distinct ft] ⇒ OF-priority-match γ ft p ≠ Undefined by (simp add: no-overlapsI no-overlaps-not-undefined)

Given the absence of overlapping or duplicate flow entries, we can show two interesting equivalences. The first is the equality to the semantics defined by Guha et al.:

**Lemma** [check-no-overlap γ ft; distinct ft] ⇒ OF-priority-match γ ft p = option-to-ftb d ↔ guha-table-semantics γ ft p d by (simp add: guha-equal no-overlapsI)

where option-to-ftb maps between the return type of OF-priority-match and an option type as one would expect.

The second equality for OF-priority-match is one that helps reasoning about flow tables. We define a simple recursive traversal for flow tables:

**Lemma**

\[
\begin{align*}
\text{OF-match-linear } & γ \; [\; ] \; p = \text{NoAction} \\
\text{OF-match-linear } & γ \; (a\#as) \; p = \text{if } γ \; (\text{ofe-fields } a) \; p \; \text{then Action (ofe-action } a \text{) else OF-match-linear } γ \; a \; p \\
\text{by} & (\text{fact OF-match-linear.simps})+
\end{align*}
\]

For this definition to be equivalent, we need the flow table to be sorted:

**Lemma**

\[
\begin{align*}
\text{[no-overlaps } & γ \; f \; ; \text{sorted-descending } (\text{map ofe-prio } f)\] ⇒ \text{OF-match-linear } γ \; f \; p = \text{OF-priority-match } γ \; f \; p \\
\text{by} & (\text{fact OF-eq})
\end{align*}
\]

As the last step, we implemented a serialization function for flow entries; it has to remain unverified. The serialization function deals with one little inaccuracy: We have modelled the IngressPort match to use the interface name, but OpenFlow requires numerical interface IDs instead. We deemed that pulling this translation step into the main translation would only make the correctness lemma of the translation more complicated while not increasing the confidence in the correctness significantly. We thus made replacing interface names by their ID part of the serialization.

Having collected all important definitions and models, we can move on to the conversion.

### 1.3 Translation Implementation

This section explains how the functions that are executed sequentially in a Linux firewall can be compressed into a single OpenFlow table. Creating this flow table in a single step would be immensely complicated. We thus divided the task into several steps using the following key insights:

- All steps that are executed in the Linux router can be formulated as a firewall, more specifically, a generalization of simple-fw that allows arbitrary actions instead of just accept and drop.
- A function that computes the conjunction of two simple-fw matches is already present. Extending this to a function that computes the join of two firewalls is relatively simple. This is explained in Section 1.3.1

#### 1.3.1 Chaining Firewalls

This section explains how to compute the join of two firewalls.

The basis of this is a generalization of simple-fw. Instead of only allowing Accept or Drop as actions, it allows arbitrary actions. The type of the function that evaluates this generalized simple firewall is generalized-sfw. The definition is straightforward:

**Lemma**

\[
\begin{align*}
generalized-sfw \; [] \; p & = \text{None} \\
generalized-sfw (a \; \# \; as) \; p & = \text{if } \text{case } a \; \text{of } (m,-) \Rightarrow \text{simple-matches } m \; p \; \text{then Some } a \; \text{else generalized-sfw as p} \\
\text{by} & (\text{fact generalized-sfw.simps})+
\end{align*}
\]

Based on that, we asked: if fw1 makes the decision a (where a is the second element of the result tuple from generalized-sfw) and fw2 makes the decision b, how can we compute the firewall that makes the decision \((a, b)^8\)?

One possible answer is given by the following definition:

²Note that tuples are right-associative in Isabelle/HOL, i.e., \((a, b, c)\) is a pair of \(a\) and the pair \((b, c)\)
lemma generalized-fw-join l1 l2 \equiv \begin{cases} (u, a, b). (m1, a) \leftarrow l1, \\
(m2, b) \leftarrow l2, u \leftarrow \text{case simple-match-and m1 m2 of None} \\
= \begin{cases} | s | \end{cases} & \text{if simple-match-and r1 r2} \\
\text{Decision FinalAllow} & \text{Other} \end{cases}

by (fact generalized-fw-join-def unfolded option2list-def))

This definition validates the following lemma:

lemma generalized-sfw (generalized-fw-join fwr1 fwr2) p = 
\begin{cases} Some (u, d_1, d_2) \leftarrow (\exists r_1, r_2. generalized-sfw fwr1 p = Some (r_1, d_1) \land generalized-sfw fwr2 p = Some (r_2, d_2) \land Some u = \text{simple-match-and r1 r2}) \newline
\text{bysome dest: generalized-fw-joinD generalized-fw-joinI intro: Some-to-the symmetric) \end{cases}

Thus, generalized-fw-join has a number of applications.

For example, it could be used to compute a firewall ruleset that represents two firewalls that are executed in sequence.

definition simple-action-conj a b \equiv \begin{cases} \text{if } a = \text{simple-action} \&\& b = \text{simple-action} \text{. Accept then simple-action . Accept else simple-action . Drop} \end{cases}

definition simple-rule-conj \equiv \begin{cases} \text{uncurry SimpleRule } \circ \text{apsnd (uncurry simple-action-conj)} \end{cases}

theorem simple-fw rs1 p = Decision FinalAllow \land simple-fw rs2 p = Decision FinalAllow \iff \begin{cases} \text{map simple-rule-conj (generalized-fw-join (map simple-rule-dtor rs1) (map simple-rule-dtor rs2))} \end{cases} \text{p = Decision FinalAllow}


Using the join, it should be possible to compute any n-ary logical operation on firewalls. We will use it for something somewhat different in the next section.

1.3.2 Translation Implementation

This section shows the actual definition of the translation function, in Figure 1. Before beginning the translation, the definition checks whether the necessary prerequisites are valid. This first two steps are to convert fw and rt to lists that can be evaluated by generalized-sfw. For fw, this is done by map simple-rule-dtor, which just deconstructs simple-rules into tuples of match and action. For rt, we made a firewall ruleset with rules that use prefix matches on the destination IP address. The next step is to join the two rulesets. The result of the join is a ruleset with rules r that only match if both, the corresponding firewall rule fwr and the corresponding routing rule rt matches. The data accompanying r is the port from rr

and the firewall decision from fwr. Next, descending priorities are added to the rules using map (apfst word-of-nat) \circ annotate-rlen. If the number of rules is too large to fit into the 2^{16} priority classes, an error is returned. Otherwise, the function pack-OF-entries is used to convert the (16 word \times 32 simple-match \times char list \times simple-action) list to an OpenFlow table. While converting the char list \times simple-action tuple is straightforward, converting the simple-match to an equivalent list of of-match-field set is non-trivial. This is done by the function simple-match-to-of-match.

The main difficulties for simple-match-to-of-match lie in making sure that the prerequisites are satisfied and in the fact that a simple-match operates on slightly stronger match expressions.

- A simple-match allows a (string) prefix match on the input and output interfaces. Given a list of existing interfaces on the router ifs, the function has to insert flow entries for each interface matching the prefix.

- A simple-match can match ports by an interval. Now it becomes obvious why Section 1.2.1 added bitmasks to L4Src and L4Dst. Using the algorithm to split word intervals into intervals that can be represented by prefix matches from [6], we can efficiently represent the original interval by a few (32 in the worst case) prefix matches and insert flow entries for each of them.\footnote{It might be possible to represent the interval match more efficiently than a split into prefixes. However, that would produce overlapping matches (which is not a problem if we assign separate priorities) and we did not have a verified implementation of an algorithm that does so.}

The following lemma characterizes simple-match-to-of-match:

lemma simple-match-to-of-match:

assumes simple-match-valid r
p-iface p \in set ifs
match-iface (oiface r) (p-oiface p)
p-l2type p = 0x800
shows simple-matches r p \iff (\exists gr \in set (simple-match-to-of-match r ifs)). OF-match-fields gr p = Some True
using assms simple-match-to-of-matchD simple-match-to-of-matchI by blast

The assumptions are to be read as follows:
The match \( r \) has to be valid, i.e. it has to use valid-prefix matches, and it cannot use anything other than 0-65535 for the port matches unless its protocol match ensures TCP, UDP or L4-Protocol.SCTP.

- simple-match-to-of-match cannot produce rules for packets that have input interfaces that are not named in the interface list.

- The output interface of \( p \) has to match the output interface match of \( r \). This is a weakened formulation of ofinterface \( r = \text{ifaceAny} \), since

\[
\text{match-iface ifaceAny i}
\]

- We require this because OpenFlow field matches cannot be used to match on the output port — they are supposed to match a packet and decide an output port.

- The simple-match type was designed for IP(v4) packets, we limit ourselves to them.

The conclusion then states that the simple-match \( r \) matches iff an element of the result of simple-match-to-of-match matches. The third assumption is part of the explanation why we did not use simple-linux-router-altered: simple-match-to-of-match cannot deal with output interface matches. Thus, before passing a generalized simple firewall to pack-OF-entries, we would have to set the output ports to ifaceAny. A system replace output interface matches with destination IP addresses has already been formalized and will be published in a future version of [5]. For now, we limit ourselves to firewalls that do not do output port matching, i.e., we require no-oif-match \( fw \).

Given discussed properties, we present the central theorem for our translation in Figure 2. The first two assumptions are limitations on the traffic we make a statement about. Obviously, we will never see any packets with an input interface that is not in the interface list. Furthermore, we do not state anything about non-IPv4 traffic. (The traffic will remain unmatched in the flow table, but we have not verified that.) The last assumption is that the translation does not return a run-time error. The translation will return a run-time error if the rules can not be assigned priorities from a 16 bit integer, or when one of the following conditions on the input data is not satisfied:

\[
\text{lemma} \quad \text{lr-of-tran} \text{ rt} \text{ fw} \text{ ifs} \equiv
\]

if \( \neg \) (no-oif-match \( fw \) \( \land \) has-default-policy \( fw \) \( \land \) simple-fw-valid \( fw \) \( \land \) valid-prefixes \( rt \) \( \land \) has-default-route \( rt \) \( \land \) distinct \( ifs \) )

then Inl "Error in creating OpenFlow table: prerequisites not satisfied"
else (let

\[
\text{nfw} = \text{map simple-rule-dtor} \text{ fw};;
\text{frt} = \text{map} (\lambda \text{r}. (\text{route2match} \text{ r}, \text{output-iface} (\text{routing-action} \text{ r}))) \text{ rt};;
\text{nrd} = \text{generalized-fw-join} \text{ frt} \text{ nfw};;
\text{ard} = (\text{map} (\text{apfst of-nat}) \circ \text{annotate-rlen}) \text{ nrd} in
\]

if length \( \text{nrd} \) \( < \) unat (max-word :: 16 word)
then Inr (pack-OF-entries \( ifs \) \( ard \))
else Inl "Error in creating OpenFlow table: priority number space exhausted"


Figure 1: Function for translating a ‘i simple-rule list, a ‘i routing-rule list, and a list of interfaces to a flow table.
Theorem

Fixes
\( p :: (32, 'a) \text{ simple-packet-ext-scheme} \)

Assumes
\( p-iiface p \in \text{ set ifs and } p-l2type p = 0x800 \)
\( \text{lr-of-tran rt fw ifs} = \text{Inr oft} \)

Shows
\( \text{OF-priority-match OF-match-fields-safe oft } p = \text{Action [Forward oif]} \iff \text{simple-linux-router-nol12 } rt \text{ fw } p = \text{Some (p[i-p-ofiface := oif])} \)
\( \text{OF-priority-match OF-match-fields-safe oft } p = \text{NoAction} \iff \text{simple-linux-router-nol12 } rt \text{ fw } p = \text{None} \)
\( \text{OF-priority-match OF-match-fields-safe oft } p \neq \text{NoAction} \iff \text{OF-priority-match OF-match-fields-safe oft } p \neq \text{Undefined} \)
\( \exists \text{ls. length } \text{ls} \leq 1 \AND \text{OF-priority-match OF-match-fields-safe oft } p = \text{Action } \text{ls} \)

Using
\( \text{assms lr-of-tran-correct by simp-all} \)

Figure 2: Central theorem on \( lr-of-tran \)

\( \exists \text{err. } lr-of-tran rt fw ifs = \text{Inl err} \) unfolding \( lr-of-tran-def \)
by \( (\text{simp split: if-splits}) \)

1.3.3 Comparison to Exodus

We are not the first researchers to attempt automated static migration to SDN. The (only) other attempt we are aware of is Exodus by Nelson et al. [10].

There are some fundamental differences between Exodus and our work:

- Exodus focuses on Cisco IOS instead of Linux.
- Exodus is not limited to using a single flow table.
- Exodus requires continuous controller interaction for some of its functions.
- Exodus attempts to support as much functionality as possible and has implemented support for dynamic routing, VLANs and NAT.
- Nelson et al. reject the idea that the translation could or should be proven correct.

2 Evaluation

In Section 1, we have made lots of definitions and created lots of models. How far these models are in accordance with the real world has been up to the vigilance of the reader. This section attempts to alleviate this burden by providing some examples.

2.1 Mininet Examples

The first example is designed to be minimal while still showing the most important properties of our conversion. For this purpose, we used a Linux firewall F, that we want to convert. We gave it two interfaces, and connected one client each. Its original configuration and the ruleset resulting from the translation is shown in Figure 3. (The list of interfaces can be extracted from the routing table; \text{s1-lan} received port number 1.) While the configuration does not fulfill any special function (especially, no traffic from the interface \text{s1-wan} is permitted), it is small enough to let us have a detailed look. More specifically, we can see how the only firewall rule (Line 2) got combined with the first rule of the routing table to form Line 1 of the OpenFlow rules. This also shows why the bitmasks on the layer 4 ports are necessary. If we only allowed exact matches, we would have \( 2^{15} \) rules instead of just one. Line 2 of the OpenFlow ruleset has been formed by combining the default drop policy with Line 1 of the routing table.
In a similar fashion, Line 2 of the routing rules has also been combined with the two firewall rules. However, as 10.0.2.0/24 from the firewall and 10.0.1.0/24 from the routing table have no common elements, no rule results from combining Line 2 and Line 2. In a similar fashion, the rest of the OpenFlow ruleset can be explained.

We feel that it is also worth noting again that it is necessary to change the IP configuration of the two devices attached to F. Assuming they are currently configured with, e.g., 10.0.1.100/24 and 10.0.2.1/24, the subnet would have to be changed from 24 to 22 or lower to ensure that a common subnet is formed and the MAC layer can function properly.

Next, we show a somewhat more evolved example. Its topology is depicted in Figure 4a. As before, we called the device to be replaced F. It is supposed to implement the simple policy that the clients H1 and H2 are allowed to communicate with the outside world via HTTP, ICMP is generally allowed, any other traffic is to be dropped (we neglected DNS for this example). We used the iptables configuration that is shown in Figure 4b. The routing table is the same as in the first example network.

The topology has been chosen for a number of reasons: we wanted one device which is not inside a common subnet with F and thus requires no reconfiguration for the translation. Moreover, we wanted two devices in a network that can communicate with each other while being overheard by F. For this purpose, we added two clients H1 and H2 instead of just one. We connected them with a broadcasting device.10 Executing our conversion function results in 36 rules11, we decided not to include them here. Comparing to the first example network, the size of the ruleset seems relatively high. This can be explained by the port matches: 1024-65535 has to be expressed by 6 different matches, `tp_src=1024/0xf000`, `tp_src=2048/0xf800`, ..., `tp_src=32768/0x8000` (or `tp_dst` respectively). When installing these rules, we also have to move all of H1, H2 and S1 into a common subnet. We chose 10.0.0.0/16 and updated the IP configuration of the three hosts accordingly. As discussed, the configuration of S2 did not have to be updated, as it does not share any subnet with F. We then tested reachability for TCP 22 and 80 and ICMP. The connectivity between all pairs of hosts (H1,H2,S1 and S2) remained the same compared to before the conversion. This shows that the concept can be made to work.

However, the example also reveals a flaw: When substituting the more complete model of a linux firewall with the simple one in Section 1.1, we assumed that the check whether the correct MAC address is set and the packets are destined for the modelled device would never fail — we assumed that all traffic arriving at a device is ac-

---

10 For the lack of a hub in mininet, we emulated one with an OpenFlow switch.
11 If we had implemented some spoofing protection by adding `! -s 10.0.1.0/24` to the respective rule, the number of rules would have been increased to 312. This is because a cross product of two prefix splits would occur.
(a) Topology

(b) FORWARD chain

Figure 4: Example Network 2

Figure 5: Benchmark

2.2 Performance Evaluation

Unfortunately, we do not have any real-world data that does not use output port matches as required in Section 1.3. There is thus no way to run the translation on the real-world firewall rulesets we have available and obtain a meaningful result. Nevertheless, we can use a real-world ruleset to evaluate the performance of our translation. For this purpose, we picked the largest firewall from the firewall collection from [6]. A significant amount of time is necessary to convert its FORWARD chain including 4946 rules\(^\text{12}\) to the required simplified firewall form. Additionally to the simplified firewall, we acquired the routing table (26 entries) from the same machine. We then evaluated the time necessary to complete the translation and the size of the resulting ruleset when using only the first \(n\) simple firewall rules and the full routing table. The result is shown in Figure 5. Given the time necessary to complete the conversion of the iptables firewall to a simple firewall, it is reasonable to say that the translation function is efficient enough. At first glance, size of the resulting ruleset seems high. This can be explained by two facts:

- The firewall contains a large number of rules with port matches that allow the ports 1-65535, which requires 16 OpenFlow rules.

\(^{12}\)In the pre-parsed and already normalized version we used for this benchmark, it took 45s. The full required time lies closer to 11min as stated in [6].
Some combinations of matches from the firewall and the routing table cannot be ruled out, since the firewall match might only contain an output port and the rule can thus only apply for the packets matching a few routing table entries. However, the translation is not aware of that and can thus not remove the combination of the firewall rule and other routing table entries.

In some rules, the conditions above coincide, resulting in \( 416 = 16 \cdot 26 \) rules. To avoid the high number of rules resulting from the port matches, rules that forbids packets with source or destination port 0 could be added to the start of the firewall and the 1-65535 could be removed; dealing with the firewall / routing table problem is part of the future work on output interfaces.

### 3 Conclusion and Future Work

We believe that we have shown that it is possible to translate at least basic configurations of a Linux firewall into OpenFlow rulesets while preserving the most important aspects of the behavior. We recognize that our system has limited practical applicability. One possible example would be a router or firewall inside a company network whose state tables have been polluted by special attack traffic. Our translation could provide an OpenFlow based stateless replacement. However, given the current prerequisites the implementation has on the configuration, this application is relatively unlikely.

For the configuration translation, we have contributed formal models of a Linux firewall and of an OpenFlow switch. Furthermore, the function that joins two firewalls and the function that translates a simplified match from [6] to a list of equivalent OpenFlow field match sets are contributions that we think are likely to be of further use.

We want to explicitly formulate the following two goals for our future work:

- We want to deal with output interface matches. The idea is to formulate and verify a destination interface / destination IP address rewriting that can exchange output interfaces and destination IP addresses in a firewall, based on the information from the routing table.\(^\text{13}\)

- We want to develop a system that can provide a stricter approximation of stateful matches so our translation will be applicable in more cases.

### References


