LOFT — Verified Migration of Linux Firewalls to SDN

Julius Michaelis and Cornelius Diekmann

March 17, 2025

Abstract

We present LOFT — *L*inux firewall *O*pen*F*low *T*ranslator, 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

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*

next case (*IPv4Src s*) **thus** $s = (case\ field\text{-}match\ of\ IPv4Src\ s \Rightarrow s)$ **unfolding** *IPv4Src of-match-field.simps* **by** *rule*
next case (*IPv4Dst s*) **thus** $s = (case\ field\text{-}match\ of\ IPv4Dst\ s \Rightarrow s)$ **by** *simp*
next case (*IPv4Proto s*) **thus** $s = (case\ field\text{-}match\ of\ IPv4Proto\ s \Rightarrow s)$ **by** *simp*
next case (*L4Src p l*) **thus** $p = (case\ field\text{-}match\ of\ L4Src\ p\ m \Rightarrow p) \wedge l = (case\ field\text{-}match\ of\ L4Src\ p\ m \Rightarrow m)$ **by** *simp*
next case (*L4Dst p l*) **thus** $p = (case\ field\text{-}match\ of\ L4Dst\ p\ m \Rightarrow p) \wedge l = (case\ field\text{-}match\ of\ L4Dst\ p\ m \Rightarrow m)$ **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* -) *m* = (∃ *id*. *let v* = *VlanId id in v* ∈ *m* ∧ *prerequisites v m*) |

prerequisites (*IPv4Proto* -) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*) |

prerequisites (*IPv4Src* -) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*) |

prerequisites (*IPv4Dst* -) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*) |

prerequisites (*L4Src* - -) *m* = (∃ *proto* ∈ {*TCP*,*UDP*,*L4-Protocol.SCTP*}. *let v* = *IPv4Proto proto in v* ∈ *m* ∧ *prerequisites v m*) |
prerequisites (*L4Dst* - -) *m* = *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 *less-eq-of-match-field1* (*a*::*of-match-field*) (*b*::*of-match-field*) ⟷ (*case* (*a*, *b*) *of*
(*IngressPort a*, *IngressPort b*) ⇒ *a* ≤ *b* |
(*VlanId a*, *VlanId b*) ⇒ *a* ≤ *b* |
(*EtherDst a*, *EtherDst b*) ⇒ *a* ≤ *b* |
(*EtherSrc a*, *EtherSrc b*) ⇒ *a* ≤ *b* |
(*EtherType a*, *EtherType b*) ⇒ *a* ≤ *b* |
(*VlanPriority a*, *VlanPriority b*) ⇒ *a* ≤ *b* |
(*IPv4Proto a*, *IPv4Proto b*) ⇒ *a* ≤ *b* |
(*IPv4Src a*, *IPv4Src b*) ⇒ *a* ≤ *b* |
(*IPv4Dst a*, *IPv4Dst b*) ⇒ *a* ≤ *b* |
(*L4Src a1 a2*, *L4Src b1 b2*) ⇒ *if a2* = *b2 then a1* ≤ *b1 else a2* ≤ *b2* |
(*L4Dst a1 a2*, *L4Dst b1 b2*) ⇒ *if a2* = *b2 then a1* ≤ *b1 else a2* ≤ *b2* |
(*a*, *b*) ⇒ *match-sorter a* < *match-sorter b*)

instantiation *of-match-field* :: *linorder*
begin

definition
less-eq-of-match-field (*a*::*of-match-field*) (*b*::*of-match-field*) ⟷ *less-eq-of-match-field1 a b*

definition
less-of-match-field (*a*::*of-match-field*) (*b*::*of-match-field*) ⟷ *a* ≠ *b* ∧ *less-eq-of-match-field1 a b*

instance
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

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

definition *match-prereq* :: *of-match-field* ⇒ *of-match-field set* ⇒ (*32*,*'a*) *simple-packet-ext-scheme* ⇒ *bool option* **where**
match-prereq i s p = (*if prerequisites i s then Some* (*match-no-prereq i p*) *else None*)

definition *set-seq s* ≡ *if* (∀ *x* ∈ *s*. *x* ≠ *None*) *then Some* (*the* ‘ *s*) *else None*
definition *all-true s* ≡ ∀ *x* ∈ *s*. *x*
term *map-option*
definition *OF-match-fields* :: *of-match-field set* ⇒ (*32*,*'a*) *simple-packet-ext-scheme* ⇒ *bool option* **where** *OF-match-fields m p* = *map-option all-true* (*set-seq* ((λ*f*. *match-prereq f m p*) ‘ *m*))
definition *OF-match-fields-unsafe* :: *of-match-field set* ⇒ (*32*,*'a*) *simple-packet-ext-scheme* ⇒ *bool* **where** *OF-match-fields-unsafe m p* = (∀ *f* ∈ *m*. *match-no-prereq f p*)
definition *OF-match-fields-safe m* ≡ *the* ∘ *OF-match-fields m*

definition *all-prerequisites m* ≡ ∀ *f* ∈ *m*. *prerequisites f m*

lemma
all-prerequisites p ⟹
L4Src x y ∈ *p* ⟹
IPv4Proto ‘ {*TCP*, *UDP*, *L4-Protocol.SCTP*} ∩ *p* ≠ {}
unfolding *all-prerequisites-def* **by** *auto*

lemma *of-safe-unsafe-match-eq*: *all-prerequisites m* ⟹ *OF-match-fields m p* = *Some* (*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*: (λ*f*. *if prerequisites f m then Some* (*match-no-prereq f p*) *else None*) ‘ *m* = (λ*f*. *Some* (*match-no-prereq f p*)) ‘ *m*
using *1* **by** *fastforce*
have *3*: ∀ *x*∈(λ*f*. *Some* (*match-no-prereq f p*)) ‘ *m*. *x* ≠ *None* **by** *blast*
show *?case*
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** *OF-match-fields-safe m* = *OF-match-fields-unsafe m*
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* =
(*if* ∃ *f* ∈ *m*. ¬*prerequisites f m then None else*
if ∀ *f* ∈ *m*. *match-no-prereq f p 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*: *ball-Un*)

lemma *of-match-fields-safe-eq2*: **assumes** *all-prerequisites m* **shows** *OF-match-fields-safe m p* ⟷ *OF-match-fields m p* = *Some True*
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** *simp*

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*) = (*case a 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 ''oif'']

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
type-synonym ('m, 'p) field-matcher = ('m set ⇒ 'p ⇒ bool)

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. 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* γ *ft* = ($\forall$ *a* $\in$ *set ft*. $\forall$ *b* $\in$ *set ft*. $\forall$ *p* $\in$ *UNIV*. (*ofe-prio a* = *ofe-prio b* $\wedge$ γ (*ofe-fields a*) *p* $\wedge$ *a* $\neq$ *b*) $\longrightarrow$ $\neg\gamma$ (*ofe-fields b*) *p*)
definition *check-no-overlap2* γ *ft* = ($\forall$ *a* $\in$ *set ft*. $\forall$ *b* $\in$ *set ft*. (*a* $\neq$ *b* $\wedge$ *ofe-prio a* = *ofe-prio b*) $\longrightarrow$ $\neg$($\exists$ *p* $\in$ *UNIV*. γ (*ofe-fields a*) *p* $\wedge$ γ (*ofe-fields b*) *p*))
lemma *check-no-overlap-alt*: *check-no-overlap* γ *ft* = *check-no-overlap2* γ *ft*
unfolding *check-no-overlap2-def check-no-overlap-def*
by *blast*

lemma *no-overlap-not-unefined*: *check-no-overlap* γ *ft* $\Longrightarrow$ *OF-same-priority-match2* γ *ft p* $\neq$ *Undefined*
proof
assume *goal1*: *check-no-overlap* γ *ft OF-same-priority-match2* γ *ft p* = *Undefined*
let *?as* = {*f*. *f* $\in$ *set ft* $\wedge$ γ (*ofe-fields f*) *p* $\wedge$ ($\forall$ *fo* $\in$ *set ft*. *ofe-prio f* < *ofe-prio fo* $\longrightarrow$ $\neg$ γ (*ofe-fields fo*) *p*)}
have *fin*: *finite ?as* **by** *simp*
note *goal1*(*2*)[*unfolded OF-same-priority-match2-def*]
then have *2* $\leq$ *card* (*ofe-action* ‘ *?as*) **unfolding** *f-Img-ex-set*
unfolding *Let-def*
by(*cases card* (*ofe-action* ‘ *?as*), *simp*) (*rename-tac nat1*, *case-tac nat1*, *simp add*: *image-Collect*, *presburger*)
then have *2* $\leq$ *card ?as* **using** *card-image-le*[*OF fin*, *of ofe-action*] **by** *linarith*
then obtain *a b* **where** *ab*: *a* $\neq$ *b a* $\in$ *?as b* $\in$ *?as* **using** *card2-eI* **by** *blast*
then have *ab2*: *a* $\in$ *set ft* γ (*ofe-fields a*) *p* ($\forall$ *fo*$\in$*set ft*. *ofe-prio a* < *ofe-prio fo* $\longrightarrow$ $\neg$ γ (*ofe-fields fo*) *p*)
b $\in$ *set ft* γ (*ofe-fields b*) *p* ($\forall$ *fo*$\in$*set ft*. *ofe-prio b* < *ofe-prio fo* $\longrightarrow$ $\neg$ γ (*ofe-fields fo*) *p*) **by** *simp-all*
then have *ofe-prio a* = *ofe-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*, *'p*) *field-matcher* $\Rightarrow$ (*'m*, *'a*) *flowtable* $\Rightarrow$ *'p* $\Rightarrow$ *'a flowtable-behavior* **where**
OF-match-linear - [] - = NoAction |
OF-match-linear γ (*a*#*as*) *p* = (*if* γ (*ofe-fields a*) *p then Action* (*ofe-action a*) *else OF-match-linear* γ *as p*)

lemma *OF-match-linear-ne-Undefined*: *OF-match-linear* γ *ft p* $\neq$ *Undefined*
by(*induction ft*) *auto*

lemma *OF-match-linear-append*: *OF-match-linear* γ (*a* @ *b*) *p* = (*case OF-match-linear* γ *a p of NoAction* $\Rightarrow$ *OF-match-linear* γ *b p* | *x* $\Rightarrow$ *x*)
by(*induction a*) *simp-all*
lemma *OF-match-linear-match-allsameaction*: $[\![$*gr* $\in$ *set oms*; γ *gr p* = *True*$]\!]$
$\Longrightarrow$ *OF-match-linear* γ (*map* (λ*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* γ *ft p* = *NoAction* $\longleftrightarrow$ ($\forall$ *e*$\in$*set ft*. $\neg$ γ (*ofe-fields e*) *p*)
by(*induction ft*) (*simp-all split*: *if-splits*)

lemma *set-eq-rule*: ($\bigwedge$*x*. *x* $\in$ *a* $\Longrightarrow$ *x* $\in$ *b*) $\Longrightarrow$ ($\bigwedge$*x*. *x* $\in$ *b* $\Longrightarrow$ *x* $\in$ *a*) $\Longrightarrow$ *a* = *b* **by**(*rule antisym*[*OF subsetI subsetI*])

lemma *unmatching-insert-agnostic*: $\neg$ γ (*ofe-fields a*) *p* $\Longrightarrow$ *OF-same-priority-match2* γ (*a* # *ft*) *p* = *OF-same-priority-match2* γ *ft p*
proof −
let *?as* = {*f*. *f* $\in$ *set ft* $\wedge$ γ (*ofe-fields f*) *p* $\wedge$ ($\forall$ *fo* $\in$ *set ft*. *ofe-prio f* < *ofe-prio fo* $\longrightarrow$ $\neg$ γ (*ofe-fields fo*) *p*)}
let *?aas* = {*f* |*f*. *f* $\in$ *set* (*a* # *ft*) $\wedge$ γ (*ofe-fields f*) *p* $\wedge$ ($\forall$ *fo*$\in$*set* (*a* # *ft*). *ofe-prio f* < *ofe-prio fo* $\longrightarrow$ $\neg$ γ (*ofe-fields fo*) *p*)}
assume *nm*: $\neg$ γ (*ofe-fields a*) *p*

have *aa*: *?aas = ?as*
proof(*rule set-eq-rule*)
fix *x*
assume *x* ∈ {*f* |*f*. *f* ∈ *set* (*a* # *ft*) ∧ *γ* (*ofe-fields f*) *p* ∧ (∀ *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* ∧ (∀ *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** (∀ *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* ∧ (∀ *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* ∧ (∀ *fo*∈*set ft*. *ofe-prio f* < *ofe-prio fo* ⟶ ¬ *γ* (*ofe-fields fo*) *p*)}
hence *as*: *x* ∈ *set ft γ* (*ofe-fields x*) *p* (∀ *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** (∀ *fo*∈*set* (*a* # *ft*). *ofe-prio x* < *ofe-prio fo* ⟶ ¬ *γ* (*ofe-fields fo*) *p*) **using** *nm* **by** *simp*
ultimately show *x* ∈ {*f* |*f*. *f* ∈ *set* (*a* # *ft*) ∧ *γ* (*ofe-fields f*) *p* ∧ (∀ *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 *?case* (**is** *?kees*)
proof(*cases γ* (*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-neq list.set-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* = (*λs*.
{*f*. *f* ∈ *set s* ∧ *γ* (*ofe-fields f*) *p* ∧
(∀ *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 ccontr*, *goal-cases*)
case (*1 x*)
hence *g1*: *x* ∈ *set n γ* (*ofe-fields x*) *p*
(∀ *fo*∈*set m*. *ofe-prio x* < *ofe-prio fo* ⟶ ¬ *γ* (*ofe-fields fo*) *p*)
(∀ *fo*∈*set n*. *ofe-prio x* < *ofe-prio fo* ⟶ ¬ *γ* (*ofe-fields fo*) *p*)

```
    unfolding mn(1) by(simp-all)
   from g1(1) mn(3) have le: ofe-prio x < ofe-prio a by simp
   note le g1(3) aem True
   then show False by blast
  qed simp
  ultimately have cc: ?fc (m @ n) = ?fc m by blast
  have cm: ?fc m = {a}
  proof −
   have ∀f ∈ set m. (∀fo∈set (a # ft). ofe-prio f < ofe-prio fo ⟶ ¬ γ (ofe-fields fo) p)
    by (metis UnE less-asym mn set-append)
   hence 1: ?fc m = {f ∈ set m. γ (ofe-fields f) p} by blast
    show {f ∈ set m. γ (ofe-fields f) p ∧ (∀fo∈set (a # ft). ofe-prio f < ofe-prio fo ⟶ ¬ γ (ofe-fields fo) p)} = {a}
unfolding 1
   proof(rule set-eq-rule, goal-cases fwd bwd)
    case (bwd x)
    have a ∈ {f ∈ set m. γ (ofe-fields f) p} using True aem by simp
    thus ?case using bwd by simp
   next
    case (fwd x) show ?case proof(rule ccontr)
     assume x ∉ {a} hence ne: x ≠ a by simp
     from fwd have 1: x ∈ set m γ (ofe-fields x) p by simp-all
     have 2: ofe-prio x = ofe-prio a using 1(1) mn(2) by simp
     show False using 1 ne mover aem True 2 unfolding check-no-overlap-def by blast
    qed
   qed
  qed
  show ?kees
   unfolding mn(1)
   unfolding OF-same-priority-match2-def
   unfolding f-Img-ex-set
   unfolding cc[unfolded mn(1)]
   unfolding cm[unfolded mn(1)]
   unfolding Let-def
   by(simp only: mn(1)[symmetric] OF-match-linear.simps True if-True, simp)
  qed
qed (simp add: OF-same-priority-match2-def)

lemma overlap-sort-invar[simp]: check-no-overlap γ (sort-descending-key k ft) = check-no-overlap γ ft
 unfolding check-no-overlap-def
 unfolding sort-descending-set-inv
 ..

lemma OF-match-eq2:
  assumes check-no-overlap γ ft
  shows OF-same-priority-match2 γ ft p = OF-match-linear γ (sort-descending-key ofe-prio ft) p
proof −
 have sorted-descending (map ofe-prio (sort-descending-key ofe-prio ft)) by (simp add: sorted-descending-sort-descending-key)
 note ceq = OF-match-eq[OF this, unfolded overlap-sort-invar, OF ‹check-no-overlap γ ft›, symmetric]
 show ?thesis
  unfolding ceq
  unfolding OF-same-priority-match2-def
  unfolding sort-descending-set-inv
  ..
qed
```

lemma *prio-match-matcher-alt*: $\{f.\ f \in set\ flow\text{-}entries \wedge \gamma\ (ofe\text{-}fields\ f)\ packet \wedge$
$(\forall fo \in set\ flow\text{-}entries.\ ofe\text{-}prio\ fo > ofe\text{-}prio\ f \longrightarrow \neg\gamma\ (ofe\text{-}fields\ fo)\ packet)\}$
$=$ (
let matching $= \{f.\ f \in set\ flow\text{-}entries \wedge \gamma\ (ofe\text{-}fields\ f)\ packet\}$
in $\{f.\ f \in matching \wedge (\forall fo \in matching.\ ofe\text{-}prio\ fo \leq ofe\text{-}prio\ f)\}$
)
by(*auto simp add*: *Let-def*)
lemma *prio-match-matcher-alt2*: (
let matching $= \{f.\ f \in set\ flow\text{-}entries \wedge \gamma\ (ofe\text{-}fields\ f)\ packet\}$
in $\{f.\ f \in matching \wedge (\forall fo \in matching.\ ofe\text{-}prio\ fo \leq ofe\text{-}prio\ f)\}$
) $=$ *set* (
let matching $=$ *filter* $(\lambda f.\ \gamma\ (ofe\text{-}fields\ f)\ packet)$ *flow-entries*
in filter $(\lambda f.\ \forall fo \in set\ matching.\ ofe\text{-}prio\ fo \leq ofe\text{-}prio\ f)$ *matching*
)
by(*auto simp add*: *Let-def*)

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

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

lemma *filter-singleton*: $[x \leftarrow s.\ f\ x] = [y] \Longrightarrow f\ y \wedge y \in set\ s$ **by** (*metis filter-eq-Cons-iff in-set-conv-decomp*)

lemma *OF-spm3-get-fe*: *OF-priority-match* γ *ft p* $=$ *Action a* $\Longrightarrow \exists fe.\ ofe\text{-}action\ fe = a \wedge fe \in set\ ft \wedge$ *OF-priority-match-ana* γ *ft p* $=$ *Action fe*
unfolding *OF-priority-match-def OF-priority-match-ana-def*
by(*clarsimp split*: *flowtable-behavior.splits list.splits*) (*drule filter-singleton*; *simp*)

fun *no-overlaps* **where**
no-overlaps - $[] =$ *True* |
no-overlaps γ $(a\#as) =$ (*no-overlaps* γ *as* $\wedge$ (
$\forall b \in set\ as.\ ofe\text{-}prio\ a = ofe\text{-}prio\ b \longrightarrow \neg(\exists p \in UNIV.\ \gamma\ (ofe\text{-}fields\ a)\ p \wedge \gamma\ (ofe\text{-}fields\ b)\ p)))$

lemma *no-overlap-ConsI*: *check-no-overlap2* γ $(x\#xs) \Longrightarrow$ *check-no-overlap2* γ *xs*
unfolding *check-no-overlap2-def* **by** *simp*

lemma *no-overlapsI*: *check-no-overlap* γ *t* $\Longrightarrow$ *distinct t* $\Longrightarrow$ *no-overlaps* γ *t*
unfolding *check-no-overlap-alt*
proof(*induction t*)
case (*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* $\Longrightarrow$ *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 ($\bigwedge$*e p*. *e* $\in$ *set t* $\Longrightarrow$ $\neg\gamma$ (*ofe-fields e*) *p*) $\Longrightarrow$ *no-overlaps* γ *t*
by(*induction t*) *simp-all*
lemma *no-overlaps-append*: *no-overlaps* γ (*x* @ *y*) $\Longrightarrow$ *no-overlaps* γ *y*
by(*induction x*) *simp-all*
lemma *no-overlaps-ne1*: *no-overlaps* γ (*x* @ *a* # *y* @ *b* # *z*) $\Longrightarrow$ (($\exists$ *p*. γ (*ofe-fields a*) *p*) $\vee$ ($\exists$ *p*. γ (*ofe-fields b*) *p*)) $\Longrightarrow$ *a* $\neq$ *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** $\neg$ ($\exists$ *p*$\in$*UNIV*. γ (*ofe-fields a*) *p* $\wedge$ 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* $\Longrightarrow$ *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*$\leftarrow$*fe* . γ (*ofe-fields f*) *p*] *in* [*f*$\leftarrow$*m* . $\forall$ *fo*$\in$*set m*. *ofe-prio fo* $\leq$ *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* $\neq$ []
then obtain *b bs* **where** *bbs*: *as* = *b* # *bs* **by** (*meson neq-Nil-conv*)
note *no* = *Cons*[*unfolded Let-def filter-filter*]
have *f1*: *a* $\in$ *set ?m' b* $\in$ *set ?m'* **unfolding** *bbs local.Cons* **by** *simp-all*
hence *ofe-prio a* = *ofe-prio b* **by** (*simp add*: *antisym*)
moreover have *ms*: γ (*ofe-fields a*) *p* γ (*ofe-fields b*) *p* **using** *no*[*symmetric*] **unfolding** *bbs* **by**(*blast dest*: *Cons-eq-filterD*)+
moreover have *abis*: *a* $\in$ *set fe b* $\in$ *set fe* **using** *f1* **by** *auto*

```
  moreover have a ≠ b proof(cases ∃ x y z. fe = x @ a # y @ b # z)
   case True
   then obtain x y z where xyz: fe = x @ a # y @ b # z by blast
   from no-overlaps-ne1 ms(1) uf[unfolded xyz]
   show ?thesis by blast
  next
   case False
   then obtain x y z where xyz: fe = x @ b # y @ a # z
    using no unfolding bbs
    by (metis (no-types, lifting) Cons-eq-filterD)
   from no-overlaps-ne1 ms(1) uf[unfolded 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 fe ⟹ check-no-overlap γ fe ⟹ OF-same-priority-match2 γ fe p = OF-priority-match γ fe p
 by(rule no-overlaps-defeq) (drule (2) no-overlapsI)

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

corollary OF-eq-sort:
 assumes no: no-overlaps γ f
 shows OF-priority-match γ f p = OF-match-linear γ (sort-descending-key ofe-prio f) p
 using OF-match-eq2 check-no-overlapI no no-overlaps-defeq by fastforce

lemma OF-lm-noa-none: OF-match-linear γ ft p = NoAction ⟹ ∀ e∈set ft. ¬ γ (ofe-fields e) p
 by(induction ft) (simp-all split: if-splits)


lemma OF-spm3-noa-none:
 assumes no: no-overlaps γ ft
 shows OF-priority-match γ ft p = NoAction ⟹ ∀ e ∈ set ft. ¬γ (ofe-fields e) p
unfolding OF-eq-sort[OF no] by(drule OF-lm-noa-none) simp


lemma no-overlaps-not-unefined: no-overlaps γ ft ⟹ OF-priority-match γ ft p ≠ Undefined
 using check-no-overlapI no-overlap-not-unefined no-overlaps-defeq by fastforce

end
theory OpenFlow-Serialize
imports OpenFlow-Matches
        OpenFlow-Action
        Semantics-OpenFlow
        Simple-Firewall.Primitives-toString
```

IP-Addresses.Lib-Word-toString
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''*]

value (*map* ((<<) (*1*::*48 word*) ∘ (∗) *8*) ∘ *rev*) [*0*..<*6*]

definition *serialize-mac* (*m*::*48 word*) ≡ (*intersperse* (*CHR '':''*) ∘ *map* (*hex-string-of-word 1* ∘ (λ*h*. (*m* >> *h* ∗ *8*) && *0xff*)) ∘ *rev*) [*0*..<*6*]
lemma *serialize-mac 0xdeadbeefcafe* = *''de*:*ad*:*be*:*ef*:*ca*:*fe''* **by** *eval*

definition *serialize-action pids a* ≡ (*case a of*
Forward oif ⇒ *''output*:*''* @ *pids oif* |
ModifyField-l2dst na ⇒ *''mod-dl-dst*:*''* @ *serialize-mac na*)

definition *serialize-actions pids a* ≡ *if length a* = *0 then ''drop'' else* (*intersperse* (*CHR '',''*) ∘ *map* (*serialize-action pids*)) *a*

lemma *serialize-actions* (λ*oif*. *''42''*) (*ofe-action serialization-test-entry*) =
''mod-dl-dst:*a6*:*41*:*f1*:*85*:*e8*:*62*,*output*:*42''* **by** *eval*
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 pids (*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*) = *''tp-src=''* @ *dec-string-of-word0 i* @ (*if m* = − *1 then* [] *else ''/0x''* @ *hex-string-of-word 3 m*) |
serialize-of-match - (*L4Dst i m*) = *''tp-dst=''* @ *dec-string-of-word0 i* @ (*if m* = − *1 then* [] *else ''/0x''* @ *hex-string-of-word 3 m*)

definition *serialize-of-matches* :: (*string* ⇒ *string*) ⇒ *of-match-field set* ⇒ *string*
where
serialize-of-matches pids ≡ (@) *''hard-timeout=0,idle-timeout=0,''* ∘ *intersperse* (*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* (λ*oif*. ''*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* ∘ *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,t*
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 *ta*: *CARD*('*a*) ≥ *2*
assumes *ms*: ∃ *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 *ta* **obtain** *a1 a2* :: '*a* **where** *as*: *a1* ≠ *a2* **using** *card2-eI* **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.sel*] |
rule guha-table-semantics.intros(*1*)[*of* γ *?fe2 p* [*?fe1*] [], *unfolded append-Nil2 flow-entry-match.sel append.simps*] |
simp add: *m*)+
thus *?thesis* **using** *as* **by**(*intro exI conjI*)
qed

lemma *guha-umstaendlich*:
assumes *ae*: *a* = *ofe-action fe*
assumes *ele*: *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** *ftspl*: *ft* = *ft1* @ *fe* # *ft2* **using** *split-list* **by** *fastforce*
show *?thesis* **unfolding** *ae ftspl*

apply(*rule guha-table-semantics.intros*(*1*))
using *rest*(*1*) **apply**(*simp*)
using *rest*(*2*)[*unfolded ftspl*] **apply** *simp*
done
qed

lemma *guha-matched-rule-inversion*:
assumes *guha-table-semantics* γ *ft p* (*Some a*)
shows $\exists$ *fe* $\in$ *set ft. a* = *ofe-action fe* $\wedge$ γ (*ofe-fields fe*) *p* $\wedge$ ($\forall$ *fe'* $\in$ *set ft. ofe-prio fe'* > *ofe-prio fe* $\longrightarrow$ $\neg\gamma$ (*ofe-fields fe'*) *p*)
proof −
{
fix *d*
assume *guha-table-semantics* γ *ft p d*
hence *Some a* = *d* $\Longrightarrow$ ($\exists$ *fe* $\in$ *set ft. a* = *ofe-action fe* $\wedge$ γ (*ofe-fields fe*) *p* $\wedge$ ($\forall$ *fe'* $\in$ *set ft. ofe-prio fe'* > *ofe-prio fe* $\longrightarrow$ $\neg\gamma$ (*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 −
note *spm*[*THEN OF-spm3-get-fe*] **then obtain** *fe* **where** *a*: *ofe-action fe* = *a* **and** *fein*: *fe* $\in$ *set ft* **and** *feana*: *OF-priority-match-ana* γ *ft p* = *Action fe* **by** *blast*
show *?thesis*
apply(*rule guha-umstaendlich*)
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-conv 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-unefined*)

done

lemma *guha-deterministic1*: *guha-table-semantics* γ *ft p* (*Some x1*) $\Longrightarrow \neg$ *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*)$]\!]$
$\Longrightarrow$ *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* ($\forall$ *fe′*∈*set ft*. *ofe-prio fe1* < *ofe-prio fe′* $\longrightarrow \neg$ γ (*ofe-fields fe′*) *p*)
fe2∈*set ft a* = *ofe-action fe2* γ (*ofe-fields fe2*) *p* ($\forall$ *fe′*∈*set ft*. *ofe-prio fe2* < *ofe-prio fe′* $\longrightarrow \neg$ γ (*ofe-fields fe′*) *p*)
by *blast*
from ‹*x1* ≠ *a*› **have** *fene*: *fe1* ≠ *fe2* **using** *fes*(*2*,*6*) **by** *blast*
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*]
note *this*[*unfolded Ball-def*, *THEN spec*, *THEN mp*, *OF fes*(*1*), *THEN spec*, *THEN mp*, *OF fes*(*5*),*THEN spec*, *THEN mp*, *OF UNIV-I*, *of p*] *pe fene fes*(*3*,*7*)
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* $\longleftrightarrow$ *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-unefined*[*OF no*] **apply** *fastforce*
using *no-overlaps-not-unefined*[*OF no*] **apply** *fastforce*
done

lemma *guha-nondeterministicD*:
assumes ¬*check-no-overlap* γ *ft*
shows $\exists$ *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*)
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 $CARD('action) \geq 3$
assumes $\forall p.\ \gamma\ x\ p$
shows $\exists ft::('a, 'action)\ flow\text{-}entry\text{-}match\ list.\ \neg check\text{-}no\text{-}overlap\ \gamma\ ft\ \wedge$
$\neg(\exists fe1\ fe2\ p.\ fe1 \in set\ ft \wedge fe2 \in set\ ft \wedge fe1 \neq fe2 \wedge ofe\text{-}prio\ fe1 = ofe\text{-}prio\ fe2$
$\wedge\ guha\text{-}table\text{-}semantics\ \gamma\ ft\ p\ (Some\ (ofe\text{-}action\ fe1))$
$\wedge\ guha\text{-}table\text{-}semantics\ \gamma\ ft\ p\ (Some\ (ofe\text{-}action\ fe2)))$
proof −
obtain *adef aa ab* :: *'action* **where** *anb*[*simp*]: $aa \neq ab\ adef \neq aa\ adef \neq ab$ **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* $\Longrightarrow$ *oc* = *Some adef* **for** *p oc*
unfolding *guha-table-semantics.simps*
apply(*elim disjE*; *clarsimp 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(*elim insertE*; *clarsimp*)
apply((*drule df*)+; *unfold option.inject*; (*elim anb*[*symmetric*, *THEN notE*] | (*simp*;*fail*))*?*)+
done
qed

end
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
Pure−ex.Guess
begin
hide-const *Misc.uncurry*
hide-fact *Misc.uncurry-def*

definition *route2match r* =
(|*iiface* = *ifaceAny*, *oiface* = *ifaceAny*,
src = (*0*,*0*), *dst*=(*pfxm-prefix* (*routing-match r*),*pfxm-length* (*routing-match r*)),
proto=*ProtoAny*, *sports*=(*0*,− *1*), *ports*=(*0*,− *1*)|)

definition *toprefixmatch* **where**
toprefixmatch m ≡ (*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*)
(*clarsimp*
simp add: *toprefixmatch-def ipset-from-cidr-def pfxm-mask-def fun-eq-iff*
prefix-match-semantics-ipset-from-netmask[*OF vld*] *NOT-mask-shifted-lenword*[*symmetric*]
split: *if-splits*)

definition *simple-match-to-of-match-single* ::
(*32*, *'a*) *simple-match-scheme*
⇒ *char list option* ⇒ *protocol* ⇒ (*16 word* × *16 word*) *option* ⇒ (*16 word* × *16 word*) *option* ⇒ *of-match-field set*
where
simple-match-to-of-match-single m iif prot sport dport ≡
uncurry L4Src ‘ *option2set sport* ∪ *uncurry L4Dst* ‘ *option2set dport*
∪ *IPv4Proto* ‘ (*case prot of ProtoAny* ⇒ {} | *Proto p* ⇒ {*p*}) — protocol is an 8 word option anyway...
∪ *IngressPort* ‘ *option2set iif*
∪ *IPv4Src* ‘ *option2set* (*toprefixmatch* (*src m*)) ∪ *IPv4Dst* ‘ *option2set* (*toprefixmatch* (*dst m*))
∪ {*EtherType 0x0800*}

definition *simple-match-to-of-match* :: *32 simple-match* ⇒ *string list* ⇒ *of-match-field set list* **where**
simple-match-to-of-match m ifs ≡ (*let*
npm = (λ*p*. *fst p* = *0* ∧ *snd p* = − *1*);
sb = (λ*p*. (*if npm p then* [*None*] *else if fst p* ≤ *snd p*
then map (*Some* ∘ (λ*pfx*. (*pfxm-prefix pfx*, *Bit-Operations.not* (*pfxm-mask pfx*)))) (*wordinterval-CIDR-split-prefixmatch*
(*WordInterval* (*fst p*) (*snd p*))) *else* []))
in [*simple-match-to-of-match-single m iif* (*proto m*) *sport dport*.
iif ← (*if iiface m* = *ifaceAny then* [*None*] *else* [*Some i*. *i* ← *ifs*, *match-iface* (*iiface m*) *i*]),
sport ← *sb* (*sports m*),
dport ← *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 z 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 z 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 z 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 z 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*: (*fst* (*sports m*) = *0* ∧ *snd* (*sports m*) = − *1*) ∨ *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*: (*fst* (*dports m*) = *0* ∧ *snd* (*dports m*) = − *1*) ∨ *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*(*4*) *e*
by(*elim disjE*; *simp add*: *option2set-def split*: *if-splits prod.splits uncurry-splits*)
qed
qed

lemma *and-assoc*: $a \wedge b \wedge c \longleftrightarrow (a \wedge b) \wedge c$ **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* = λ*p*. *fst p* = *0* ∧ *snd p* = − *1*
let *?sb* = λ*p r*. (*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* ⟷ *?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 *1* **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*:
ssi ∈ *set* (*wordinterval-CIDR-split-prefixmatch* (*uncurry WordInterval* (*sports r*)))

```
  p-sport p ∈ prefix-to-wordset ssi
  using wordinterval-CIDR-split-existential by fast
 hence (case Some ssi of None ⇒ True | Some ssi ⇒ p-sport p ∈ prefix-to-wordset ssi) ∧
          (case Some ssi of None ⇒ True
          | Some ssi ⇒ ssi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r)))) by simp
       with 2 show ?thesis by blast
   qed
obtain di where di: case di of Some ddi ⇒ p-dport p ∈ prefix-to-wordset ddi | None ⇒ True
 case di of None ⇒ True | Some ddi ⇒ ddi ∈ set (
 wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))
 di = None ⟷ ?npm (dports r)
proof(cases ?npm (dports r), goal-cases)
 case 1
 hence (case None of None ⇒ True | Some ssi ⇒ p-dport p ∈ prefix-to-wordset ssi) ∧
          (case None of None ⇒ True
          | Some ssi ⇒ ssi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))) by simp
       with 1 show ?thesis by blast
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 ssi ⇒ p-dport p ∈ prefix-to-wordset ssi) ∧
          (case Some ddi of None ⇒ True
          | Some ssi ⇒ ssi ∈ set (wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))) by simp
       with 2 show ?thesis by blast
   qed
   show ?thesis
proof
 let ?mf = map-option (apsnd (wordNOT ∘ mask ∘ (−) 16) ∘ prefix-match-dtor)
 let ?gr = simple-match-to-of-match-single r
  (if iiface r = ifaceAny then None else Some (p-iiface 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 u have ple: fst (sports r) ≤ snd (sports r) fst (dports r) ≤ snd (dports r) by force+
 show eg: ?gr ∈ set (simple-match-to-of-match r ifs)
  unfolding simple-match-to-of-match-def
  unfolding custom-simpset
  unfolding smtoms-eq-hlp
  proof(intro bexI, (intro conjI; ((rule refl)?)), goal-cases)
   case 2 thus ?case using ple(2) di
    apply(simp add: pfxm-mask-def prefix-match-dtor-def Set.image-iff
              split: option.splits prod.splits uncurry-splits)
    apply(erule bexI[rotated])
    apply(simp split: prefix-match.splits)
   done
  next
```

```
    case 3 thus ?case using ple(1) si
     apply(simp add: pfxm-mask-def prefix-match-dtor-def Set.image-iff
                split: option.splits prod.splits uncurry-splits)
     apply(erule bexI[rotated])
     apply(simp split: prefix-match.splits)
    done
   next
    case 4 thus ?case
      using u ii by(clarsimp simp: set-maps split: if-splits)
   next
    case 1 thus ?case using ii u by simp-all (metis match-proto.elims(2))
   qed
  have dpm: di = Some (PrefixMatch x1 x2) ⟹ p-dport p && ~~ (mask (16 - x2)) = x1 for x1 x2
   proof -
     have ∗: di = Some (PrefixMatch x1 x2) ⟹ prefix-match-semantics (the di) (p-dport p) ⟹ p-dport p && ~~ (mask
(16 - x2)) = x1
       by(clarsimp simp: prefix-match-semantics-def pfxm-mask-def word-bw-comms;fail)
     have ∗∗: pfx ∈ set (wordinterval-CIDR-split-prefixmatch ra) ⟹ prefix-match-semantics pfx a = (a ∈ prefix-to-wordset
pfx) 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 ⟦di = Some (PrefixMatch x1 x2); p-dport p ∈ prefix-to-wordset (PrefixMatch x1 x2); PrefixMatch x1 x2 ∈ set
(wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (dports r)))⟧
          ⟹ p-dport p && ~~ (mask (16 - x2)) = x1
     using di(1,2)
      using ∗ ∗∗ by auto
     thus di = Some (PrefixMatch x1 x2) ⟹ p-dport p && ~~ (mask (16 - x2)) = x1 using di(1,2) by auto
   qed
  have spm: si = Some (PrefixMatch x1 x2) ⟹ p-sport p && ~~ (mask (16 - x2)) = x1 for x1 x2
   using si
   proof -
     have ∗: si = Some (PrefixMatch x1 x2) ⟹ prefix-match-semantics (the si) (p-sport p) ⟹ p-sport p && ~~ (mask
(16 - x2)) = x1
       by(clarsimp simp: prefix-match-semantics-def pfxm-mask-def word-bw-comms;fail)
     have ∗∗: pfx ∈ set (wordinterval-CIDR-split-prefixmatch ra) ⟹ prefix-match-semantics pfx a = (a ∈ prefix-to-wordset
pfx) 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 ⟦si = Some (PrefixMatch x1 x2); p-sport p ∈ prefix-to-wordset (PrefixMatch x1 x2); PrefixMatch x1 x2 ∈ set
(wordinterval-CIDR-split-prefixmatch (uncurry WordInterval (sports r)))⟧
          ⟹ p-sport p && ~~ (mask (16 - x2)) = x1
     using si(1,2)
      using ∗ ∗∗ by auto
     thus si = Some (PrefixMatch x1 x2) ⟹ p-sport p && ~~ (mask (16 - x2)) = x1 using si(1,2) by auto
   qed
  show OF-match-fields ?gr p = Some True
  unfolding of-safe-unsafe-match-eq[OF simple-match-to-of-match-generates-prereqs[OF mv eg]]
   by(cases si; cases di)
      (simp-all
    add: simple-match-to-of-match-single-def OF-match-fields-unsafe-def spm
         option2set-def u ippkt 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)
 qed
```

qed

lemma *prefix-match-00*[*simp*,*intro*!]: *prefix-match-semantics* (*PrefixMatch 0 0*) *p*
by (*simp add*: *valid-prefix-def zero-prefix-match-all*)

lemma *simple-match-to-of-matchD*:
assumes *eg*: *gr* ∈ *set* (*simple-match-to-of-match r ifs*)
assumes *mo*: *OF-match-fields gr p* = *Some True*
assumes *me*: *match-iface* (*oiface r*) (*p-oiface p*)
assumes *mv*: *simple-match-valid r*
shows *simple-matches r p*
proof −
from *mv* **have** *validpfx*:
valid-prefix (*uncurry PrefixMatch* (*src r*)) *valid-prefix* (*uncurry PrefixMatch* (*dst r*))
⋀*pm*. *toprefixmatch* (*src r*) = *Some pm* ⟹ *valid-prefix pm*
⋀*pm*. *toprefixmatch* (*dst r*) = *Some pm* ⟹ *valid-prefix pm*
unfolding *simple-match-valid-def valid-prefix-fw-def toprefixmatch-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 simple-match-to-of-match-single-def custom-simpset option2set-def*]
then guess *x* **.. moreover from** *this*(*2*) **guess** *xa* **.. moreover from** *this*(*2*) **guess** *xb* **..**
note *xx* = *calculation*(*1*,*3*) *this*

{ fix *a b xc xa*
fix *pp* :: *16 word*
have ⟦*pp* && ~~ (*pfxm-mask xc*) = *pfxm-prefix xc*⟧
⟹ *prefix-match-semantics xc* (*pp*) **for** *xc*
by(*simp add*: *prefix-match-semantics-def word-bw-comms*;*fail*)
moreover have *pp* ∈ *wordinterval-to-set* (*WordInterval a b*) ⟹ *a* ≤ *pp* ∧ *pp* ≤ *b* **by** *simp*
moreover have *xc* ∈ *set* (*wordinterval-CIDR-split-prefixmatch* (*WordInterval a b*)) ⟹ *pp* ∈ *prefix-to-wordset xc* ⟹ *pp* ∈ *wordinterval-to-set* (*WordInterval a b*)
by(*subst wordinterval-CIDR-split-prefixmatch*) *blast*
moreover have ⟦*xc* ∈ *set* (*wordinterval-CIDR-split-prefixmatch* (*WordInterval a b*)); *xa* = *Some* (*pfxm-prefix xc*, ~~ (*pfxm-mask xc*)); *prefix-match-semantics xc* (*pp*)⟧ ⟹ *pp* ∈ *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* ∈ *set* (*wordinterval-CIDR-split-prefixmatch* (*WordInterval a b*)); *xa* = *Some* (*pfxm-prefix xc*, ~~ (*pfxm-mask xc*));
pp && ~~ (*pfxm-mask xc*) = *pfxm-prefix xc*⟧
⟹ *a* ≤ *pp* ∧ *pp* ≤ *b*
by *metis*
} note *l4port-logic* = *this*

show *?thesis* **unfolding** *simple-matches.simps*
proof(*unfold and-assoc*, (*rule*)+)
show *match-iface* (*iiface r*) (*p-iiface p*)
apply(*cases iiface r* = *ifaceAny*)
apply (*simp add*: *match-ifaceAny*)
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 validpfx simple-match-src-alt*[*OF mv*] *toprefixmatch-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 validpfx simple-match-dst-alt*[*OF mv*] *toprefixmatch-def*
split: *if-splits*)
next
show *match-proto* (*proto r*) (*p-proto p*)
using *mo* **unfolding** *xx*(*4*) *OF-match-fields-unsafe-def*
using *xx*(*1*) **by**(*clarsimp*
simp add: *singleton-iff simple-packet-unext-def option2set-def prefix-match-semantics-simple-match ball-Un*
split: *if-splits protocol.splits*)
next
show *simple-match-port* (*sports r*) (*p-sport p*)
using *mo xx*(*2*) **unfolding** *xx*(*4*) *OF-match-fields-unsafe-def*
by(*cases sports r*) (*clarsimp simp add*: *l4port-logic simple-packet-unext-def option2set-def prefix-match-semantics-simple-match 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*) (*clarsimp simp add*: *l4port-logic simple-packet-unext-def option2set-def prefix-match-semantics-simple-match split*: *if-splits*)
qed
qed

primrec *annotate-rlen* **where**
annotate-rlen [] = [] |
annotate-rlen (*a*#*as*) = (*length as*, *a*) # *annotate-rlen as*
lemma *annotate-rlen ''asdf''* = [(*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: fst (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-onI)
   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-uoi[OF se(1)] le-unat-uoi[OF se(2)]
   show x = y using eq le-unat-uoi 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) (meson fst-annotate-rlen-le less-trans nat-less-le)
lemmas distinct-of-prio-hlp = distinct-of-nat-list[OF distinct-fst-annotate-rlen annotate-first-le-hlp]
```

lemma *fst-annotate-rlen*: *map fst* (*annotate-rlen l*) = *rev* [*0*..<*length l*]
by(*induction l*) (*simp-all*)

lemma *sorted-word-upt*:
defines[*simp*]: *won* ≡ (*of-nat* :: *nat* ⇒ (*'l* :: *len*) *word*)
assumes *length l* ≤ *unat* (*max-word* :: *'l word*)
shows *sorted-descending* (*map won* (*rev* [*0*..<*Suc* (*length l*)]))
using *assms*
by(*induction l rule*: *rev-induct*;*clarsimp*)
(*metis* (*mono-tags*, *opaque-lifting*) *le-SucI le-unat-uoi of-nat-Suc order-refl word-le-nat-alt*)

lemma *sorted-annotated*:
assumes *length l* ≤ *unat* (*max-word* :: (*'l* :: *len*) *word*)
shows *sorted-descending* (*map fst* (*map* (*apfst* (*of-nat* :: *nat* ⇒ *'l word*)) (*annotate-rlen l*)))
proof −
let *?won* = (*of-nat* :: *nat* ⇒ *'l word*)
have *sorted-descending* (*map ?won* (*rev* [*0*..<*Suc* (*length l*)]))
using *sorted-word-upt*[*OF assms*] **.**
hence *sorted-descending* (*map ?won* (*map fst* (*annotate-rlen l*))) **by**(*simp add*: *fst-annotate-rlen*)
thus *sorted-descending* (*map fst* (*map* (*apfst ?won*) (*annotate-rlen l*))) **by** *simp*
qed

l3 device to l2 forwarding

definition *lr-of-tran-s3 ifs ard* = (
[(*p*, *b*, *case a of simple-action.Accept* ⇒ [*Forward c*] | *simple-action.Drop* ⇒ []).
(*p*,*r*,(*c*,*a*)) ← *ard*, *b* ← *simple-match-to-of-match r ifs*])

definition *oif-ne-iif-p1 ifs* ≡ [(*simple-match-any*⦇*oiface* := *Iface oif*, *iiface* := *Iface iif*⦈, *simple-action.Accept*). *oif* ← *ifs*, *iif* ← *ifs*, *oif* ≠ *iif*]
definition *oif-ne-iif-p2 ifs* = [(*simple-match-any*⦇*oiface* := *Iface i*, *iiface* := *Iface i*⦈, *simple-action.Drop*). *i* ← *ifs*]
definition *oif-ne-iif ifs* = *oif-ne-iif-p2 ifs* @ *oif-ne-iif-p1 ifs*

definition *lr-of-tran-s4 ard ifs* ≡ *generalized-fw-join ard* (*oif-ne-iif ifs*)

definition *lr-of-tran-s1 rt* = [(*route2match r*, *output-iface* (*routing-action r*)). *r* ← *rt*]

definition *lr-of-tran-fbs rt fw ifs* ≡ *let*
gfw = *map simple-rule-dtor fw*; — generalized simple fw, hopefully for FORWARD
frt = *lr-of-tran-s1 rt*; — rt as fw
prd = *generalized-fw-join frt gfw*
in prd

definition *pack-OF-entries ifs ard* ≡ (*map* (*split3 OFEntry*) (*lr-of-tran-s3 ifs ard*))
definition *no-oif-match* ≡ *list-all* (λ*m*. *oiface* (*match-sel m*) = *ifaceAny*)

definition *lr-of-tran rt fw ifs* ≡
if ¬ (*no-oif-match fw* ∧ *has-default-policy fw* ∧ *simple-fw-valid fw* ∧ *valid-prefixes rt* ∧ *has-default-route rt* ∧ *distinct ifs*)
then Inl ''Error in creating OpenFlow table: *prerequisites not satisifed''*
else (
let nrd = *lr-of-tran-fbs rt fw ifs*;
ard = *map* (*apfst of-nat*) (*annotate-rlen nrd*) — give them a priority

in
if length nrd < unat (− 1 :: 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 *0xFFFF = (− 1 :: 16 word)* **by** *simp*
then show *?thesis* **by** (*simp only:*) *simp*
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 add: numeral-eq-Suc*)
qed

lemma *mask-inj-hlp1: inj-on (mask :: nat ⇒ 16 word) {0..16}*
proof(*intro inj-onI*, *goal-cases*)
case (*1 x y*)
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 *⋀z. 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 = LENGTH(16)} replicate (16 − y) False @ replicate y True ∈ {bl. length bl = LENGTH(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 ∘ (λpfx. (pfxm-prefix pfx, ~~ (pfxm-mask pfx))))
(wordinterval-CIDR-split-prefixmatch (WordInterval (fst ps) (snd ps)))
else [])
proof −
assume *di: distinct ifs*
{ define *wis* **where** *wis = set (wordinterval-CIDR-split-prefixmatch (WordInterval (fst ps) (snd ps)))*
fix *x y :: 16 prefix-match*
obtain *xm xn ym yn* **where** *xyd*[*simp*]: *x = PrefixMatch xm xn y = PrefixMatch ym yn* **by**(*cases x*; *cases y*)

```
    assume iw: x ∈ wis y ∈ wis and et: (pfxm-prefix x, ~~ (pfxm-mask x)) = (pfxm-prefix y, ~~ (pfxm-mask y))
     hence le16: xn ≤ 16 yn ≤ 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 − xn = 16 − yn unfolding pfxm-mask-def by(auto intro: mask-inj-hlp1[THEN inj-onD])
    hence x = y using et le16 using diff-diff-cancel by simp
  } note ∗ = this
  show ?thesis
    apply(clarsimp simp add: smtoms-eq-hlp distinct-map wordinterval-CIDR-split-distinct)
    apply(subst comp-inj-on-iff[symmetric]; intro inj-onI)
  using ∗ by simp-all
qed

lemma distinct-simple-match-to-of-match: distinct ifs ⟹ distinct (simple-match-to-of-match m ifs)
  apply(unfold simple-match-to-of-match-def Let-def)
  apply(rule distinct-3lcomprI)
  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 ⟹ inj-on F A using subset-inj-on by auto

lemma no-overlaps-lroft-hlp2: distinct (map fst amr) ⟹ (⋀r. distinct (fm r)) ⟹
    distinct (concat (map (λ(p, r, c, a). map (λb. (p, b, fs a c)) (fm r)) amr))
  by(induction amr; force intro: injI inj-onI simp add: distinct-map split: prod.splits)

lemma distinct-lroft-s3: ⟦distinct (map fst amr); distinct ifs⟧ ⟹ 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 fst amr) ⟹
(aa, ab, ac) ∈ set (lr-of-tran-s3 ifs amr) ⟹ (ba, bb, bc) ∈ set (lr-of-tran-s3 ifs amr) ⟹
ac ≠ bc ⟹ aa ≠ 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-hlp:
  ⟦distinct (map fst amr); OF-match-fields-unsafe ab p; ab ≠ ad ∨ ba ≠ bb; OF-match-fields-unsafe ad p;
       (ac, ab, ba) ∈ set (lr-of-tran-s3 ifs amr); (ac, ad, bb) ∈ set (lr-of-tran-s3 ifs amr)⟧
      ⟹ False
proof(elim disjE, goal-cases)
  case 1
  have 4: ⟦distinct (map fst amr); (ac, ab, x1, x2) ∈ set amr; (ac, bb, x4, x5) ∈ set amr; ab ≠ bb⟧
      ⟹ False for ab x1 x2 bb x4 x5
      by (meson distinct-map-fstD old.prod.inject)
  have conjunctSomeProtoAnyD: Some ProtoAny = simple-proto-conjunct a (Proto b) ⟹ False for a b
    using conjunctProtoD by force
  have 5:
      ⟦OF-match-fields-unsafe am p; OF-match-fields-unsafe bm p; am ≠ bm;
```

```
      am ∈ set (simple-match-to-of-match ab ifs); bm ∈ set (simple-match-to-of-match bb ifs); ¬ ab ≠ bb⟧
     ⟹ False for ab bb am bm
    by(clarify | unfold
      simple-match-to-of-match-def smtoms-eq-hlp Let-def set-concat set-map de-Morgan-conj not-False-eq-True)+
     (auto dest: conjunctSomeProtoAnyD cidrsplit-no-overlaps
         simp add: OF-match-fields-unsafe-def simple-match-to-of-match-single-def option2set-def comp-def
         split: if-splits
         cong: smtoms-eq-hlp)
  from 1 show ?case
  using 4 5 by(clarsimp simp add: lr-of-tran-s3-def) blast
qed(metis no-overlaps-lroft-hlp3)


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

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

lemma sorted-lr-of-tran-s3-hlp: ∀ x∈set f. fst x ≤ a ⟹ b ∈ set (lr-of-tran-s3 s f) ⟹ fst b ≤ a
 by(auto simp add: lr-of-tran-s3-def)

lemma lr-of-tran-s3-Cons: lr-of-tran-s3 ifs (a#ard) = (
```

[(*p*, *b*, *case a of simple-action.Accept* ⇒ [*Forward c*] | *simple-action.Drop* ⇒ []).
(*p*,*r*,(*c*,*a*)) ← [*a*], *b* ← *simple-match-to-of-match r ifs*]) @ *lr-of-tran-s3 ifs ard*
by(*clarsimp simp*: *lr-of-tran-s3-def*)

lemma *sorted-lr-of-tran-s3*: *sorted-descending* (*map fst f*) ⟹ *sorted-descending* (*map fst* (*lr-of-tran-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-descending-append*)
apply(*simp add*: *sorted-descending-alt rev-map sorted-lr-of-tran-s3-hlp sorted-const*)
done

lemma *sorted-lr-of-tran-hlp*: (*ofe-prio* ∘ *split3 OFEntry*) = *fst* **by**(*simp add*: *fun-eq-iff comp-def split3-def*)

lemma *lr-of-tran-sorted-descending*: *Inr r* = *lr-of-tran rt fw ifs* ⟹ *sorted-descending* (*map ofe-prio r*)
apply(*unfold lr-of-tran-def Let-def*)
apply(*simp split*: *if-splits*)
apply(*thin-tac r* = -)
apply(*unfold sorted-lr-of-tran-hlp pack-OF-entries-def split3-def*[*abs-def*] *fun-app-def map-map comp-def prod.case-distrib*)
apply(*simp add*: *fst-def*[*symmetric*])
apply(*rule sorted-lr-of-tran-s3*)
apply(*drule sorted-annotated*[*OF less-or-eq-imp-le*, *OF disjI1*])
apply(*simp add*: *o-assoc*)
done

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

lemma *route2match-correct*: *valid-prefix* (*routing-match a*) ⟹ *prefix-match-semantics* (*routing-match a*) (*p-dst p*) ⟷ *simple-matches* (*route2match a*) (*p*)
by(*simp add*: *route2match-def simple-matches.simps match-ifaceAny match-iface-refl ipset-from-cidr-0 prefix-match-semantics-ipset-from-netmask2*)

lemma *s1-correct*: *valid-prefixes rt* ⟹ *has-default-route* (*rt*::('*i*::*len*) *prefix-routing*) ⟹
∃ *rm ra*. *generalized-sfw* (*lr-of-tran-s1 rt*) *p* = *Some* (*rm*,*ra*) ∧ *ra* = *output-iface* (*routing-table-semantics rt* (*p-dst p*))
apply(*induction rt*)
apply(*simp*;*fail*)
apply(*drule valid-prefixes-split*)
apply(*clarsimp*)
apply(*erule disjE*)
subgoal for *a rt*
apply(*case-tac a*)
apply(*rename-tac routing-m metric routing-action*)
apply(*case-tac routing-m*)
apply(*simp add*: *valid-prefix-def pfxm-mask-def prefix-match-semantics-def generalized-sfw-def*
lr-of-tran-s1-def route2match-def simple-matches.simps match-ifaceAny match-iface-refl ipset-from-cidr-0
max-word-mask[**where** *'a* = *'i*, *symmetric*, *simplified*])
done
subgoal
apply(*rule conjI*)
apply(*simp add*: *generalized-sfw-def lr-of-tran-s1-def route2match-correct*;*fail*)
apply(*simp add*: *route2match-def simple-matches.simps prefix-match-semantics-ipset-from-netmask2*
lr-of-tran-s1-split generalized-sfw-simps)
done

done

definition *to-OF-action a* ≡ (*case a of* (*p*,*d*) ⇒ (*case d of simple-action.Accept* ⇒ [*Forward p*] | *simple-action.Drop* ⇒ []))
definition *from-OF-action a* = (*case a of* [] ⇒ (''',*simple-action.Drop*) | [*Forward p*] ⇒ (*p*, *simple-action.Accept*))

lemma *OF-match-linear-not-noD*: *OF-match-linear γ oms p* ≠ *NoAction* ⟹ ∃ *ome*. *ome* ∈ *set oms* ∧ *γ* (*ofe-fields ome*) *p*
apply(*induction oms*)
apply(*simp*)
apply(*simp split*: *if-splits*)
apply *blast*+
done

lemma *s3-noaction-hlp*: ⟦*simple-match-valid ac*; ¬*simple-matches ac p*; *match-iface* (*oiface ac*) (*p-oiface p*)⟧ ⟹
OF-match-linear OF-match-fields-safe (*map* (λ*x*. *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(*rule ccontr*)
apply(*drule OF-match-linear-not-noD*)
apply(*clarsimp*)
apply(*rename-tac x*)
apply(*subgoal-tac all-prerequisites x*)
apply(*drule simple-match-to-of-matchD*)
apply(*simp add*: *split3-def*)
apply(*subst*(*asm*) *of-match-fields-safe-eq2*)
apply(*simp*;*fail*)+
using *simple-match-to-of-match-generates-prereqs* **by** *blast*

lemma *aux*:
‹*v* = *Some x* ⟹ *the v* = *x*›
by *simp*

lemma *s3-correct*:
assumes *vsfwm*: *list-all simple-match-valid* (*map* (*fst* ∘ *snd*) *ard*)
assumes *ippkt*: *p-l2type p* = *0x800*
assumes *iiifs*: *p-iiface p* ∈ *set ifs*
assumes *oiifs*: *list-all* (λ*m*. *oiface* (*fst* (*snd m*)) = *ifaceAny*) *ard*
shows *OF-match-linear OF-match-fields-safe* (*pack-OF-entries ifs ard*) *p* = *Action ao* ⟷ (∃ *r af*. *generalized-sfw* (*map snd ard*) *p* = (*Some* (*r*,*af*)) ∧ (*if snd af* = *simple-action.Drop then ao* = [] *else ao* = [*Forward* (*fst af*)]))
unfolding *pack-OF-entries-def lr-of-tran-s3-def fun-app-def*
using *vsfwm oiifs*
apply(*induction ard*)
subgoal by(*simp add*: *generalized-sfw-simps*)
apply *simp*
apply(*clarsimp simp add*: *generalized-sfw-simps split*: *prod.splits*)
apply(*intro conjI*)
subgoal for *ard x1 ac ad ba*
apply(*clarsimp simp add*: *OF-match-linear-append split*: *prod.splits*)
apply(*drule simple-match-to-of-matchI*[*rotated*])
apply(*rule iiifs*)
apply(*rule ippkt*)
apply *blast*
apply(*clarsimp simp add*: *comp-def*)
apply(*drule*
OF-match-linear-match-allsameaction[**where**
γ=*OF-match-fields-safe* **and** *pri* = *x1* **and**

```
      oms = simple-match-to-of-match ac ifs and
      act = case ba of simple-action.Accept ⇒ [Forward ad] | simple-action.Drop ⇒ []])
   apply(unfold OF-match-fields-safe-def comp-def)
   apply(erule aux)
  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 x1 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-hlp)
  apply(simp add: match-ifaceAny;fail)
 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-prefix-fw-def)
  apply(intro conjI)
   apply force
  apply(simp add: valid-prefixes-alt-def)
 done
end

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-validI 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*)
case *1*
note *1*[*unfolded lr-of-tran-fbs-def Let-def*]
have *gsfw-valid* (*map simple-rule-dtor fw*) **using** *gsfw-validI 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(*simp add*: *list-all-iff*)
apply(*clarsimp*)
apply(*drule simple-match-valid-fbs-rlen*[*rotated*])
apply(*simp add*: *list-all-iff*;*fail*)
apply(*simp add*: *list-all-iff*;*fail*)
apply(*rule simple-match-to-of-match-generates-prereqs*; *assumption*)
done

lemma *OF-unsafe-safe-match3-eq*:
list-all (*all-prerequisites* ∘ *ofe-fields*) *oft* ⟹
OF-priority-match OF-match-fields-unsafe oft = *OF-priority-match OF-match-fields-safe 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* ⟹
OF-match-linear OF-match-fields-unsafe oft = *OF-match-linear OF-match-fields-safe 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*: *oiface m* = *ifaceAny* ⟹ *simple-matches m p* = *simple-matches m* (*p*(|*p-oiface* := *any*|))
by(*cases m*) (*simp add*: *simple-matches.simps match-ifaceAny*)
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-ifaceAny-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*) ⟹
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* (*p*(|*p-oiface* := *oif*|)) ⟹
∃ *r*. *generalized-sfw* (*lr-of-tran-fbs rt fw ifs*) *p* = *Some* (*r*, *oif*, *simple-action.Accept*)
proof(*goal-cases*)
from *s2* **have** *nud*: ⋀*p*. *simple-fw fw p* ≠ *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*) ⟹
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*) ∧
generalized-sfw (*map simple-rule-dtor fw*) *p* = *Some* (*fr*, *simple-action.Drop*) ∧ *Some r* = *simple-match-and rr 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 fd 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.distinct*(*1*))
note *ifupdateirrel* = *no-oif-matchD*[*OF s2*(*1*), *symmetric*]
case *1*
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 state.splits 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-def Let-def*
apply(*rule exI*)
apply(*rule exI*[**where** *x* = *ra*])
apply(*rule generalized-fw-joinI*)
using *rmra* **apply** *simp*
apply(*rule r*)
done
qed

lemma *no-oif-match-fbs*:
no-oif-match fw ⟹ *list-all* (λ*m*. *oiface* (*fst* (*snd m*)) = *ifaceAny*) (*map* (*apfst of-nat*) (*annotate-rlen* (*lr-of-tran-fbs rt fw ifs*)))
proof(*goal-cases*)
case *1*
have *c*: ⋀*mr ar mf af f 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* = *ifaceAny*
proof(*goal-cases*)
case (*1 mr ar mf af f a*)
have *oiface mr* = *ifaceAny* **using** *1*(*1*) **unfolding** *lr-of-tran-s1-def route2match-def* **by**(*clarsimp simp add*: *Set.image-iff*)
moreover have *oiface mf* = *ifaceAny* **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-ifaceAny split*: *option.splits*)
qed
have *la*: *list-all* (λ*m*. *oiface* (*fst m*) = *ifaceAny*) (*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 ∗: (λm. oiface (fst (snd m)) = ifaceAny) = (λm. oiface (fst m) = ifaceAny) ∘ 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 ippkt: p-l2type p = 0x800
  and ifvld: p-iiface p ∈ set ifs
 shows OF-priority-match OF-match-fields-safe oft p = Action [Forward oif] ⟷ simple-linux-router-nol12 rt fw p = (Some
(p(|p-oiface := oif|)))
       OF-priority-match OF-match-fields-safe oft p = Action [] ⟷ simple-linux-router-nol12 rt fw p = None

       OF-priority-match OF-match-fields-safe oft p ≠ NoAction OF-priority-match OF-match-fields-safe oft p ≠ Undefined
       OF-priority-match OF-match-fields-safe oft p = Action ls ⟶ length ls ≤ 1
       ∃ ls. length ls ≤ 1 ∧ 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 difs: 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:
    OF-priority-match OF-match-fields-unsafe oft = OF-priority-match OF-match-fields-safe oft
    OF-match-linear OF-match-fields-unsafe oft = OF-match-linear OF-match-fields-safe oft
    apply(subst OF-unsafe-safe-match3-eq; (rule lr-of-tran-prereqs s1 s2 nerr refl)+)
    apply(subst OF-unsafe-safe-match-linear-eq; (rule lr-of-tran-prereqs s1 s2 nerr refl)+)
  done
  have lin: OF-priority-match OF-match-fields-safe oft = OF-match-linear OF-match-fields-safe oft
   using OF-eq[OF lr-of-tran-no-overlaps lr-of-tran-sorted-descending, OF difs 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 ∘ snd) ?ard)
            unfolding fun-app-def map-map[symmetric] snd-apfst map-snd-apfst map-snd-annotate-rlen using
simple-match-valid-fbs[OF s1(1) s2(2)] .
  have ∗: list-all (λm. oiface (fst (snd m)) = ifaceAny) ?ard using no-oif-match-fbs[OF s2(3)] .
  have not-undec: ⋀p. simple-fw fw p ≠ Undecided by (metis has-default-policy s2(1) state.simps(3))
  have w1-1: ⋀oif. OF-match-linear OF-match-fields-safe oft p = Action [Forward oif] ⟹ simple-linux-router-nol12 rt fw
p = Some (p(|p-oiface := oif|))
   ∧ oif = output-iface (routing-table-semantics rt (p-dst p))
  proof(intro conjI, goal-cases)
    case (1 oif)
    note s3-correct[OF vld ippkt ifvld(1) ∗, THEN iffD1, unfolded oft-def[symmetric], OF 1]
    hence ∃ r. generalized-sfw (map snd (map (apfst of-nat) (annotate-rlen (lr-of-tran-fbs rt fw ifs)))) p = Some (r, (oif,
simple-action.Accept))
      by(clarsimp split: if-splits)
```

then obtain *r* **where** *generalized-sfw* (*lr-of-tran-fbs rt fw ifs*) *p* = *Some* (*r*, (*oif*, *simple-action.Accept*))
unfolding *map-map comp-def snd-apfst map-snd-annotate-rlen* **by** *blast*
thus *?case* **using** *lr-of-tran-fbs-acceptD*[*OF s1 s2*(*3*)] **by** *metis*
thus *oif* = *output-iface* (*routing-table-semantics rt* (*p-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*: ⋀*oif*. *simple-linux-router-nol12 rt fw p* = *Some* (*p*⦇*p-oiface* := *oif*⦈) ⟹ *OF-match-linear OF-match-fields-safe oft p* = *Action* [*Forward oif*]
proof(*goal-cases*)
case (*1 oif*)
note *lr-of-tran-fbs-acceptI*[*OF s1 s2*(*3*) *s2*(*1*) *this*, *of ifs*] **then guess** *r* **.. note** *r* = *this*
hence *generalized-sfw* (*map snd* (*map* (*apfst of-nat*) (*annotate-rlen* (*lr-of-tran-fbs rt fw ifs*)))) *p* = *Some* (*r*, (*oif*, *simple-action.Accept*))
unfolding *map-snd-apfst map-snd-annotate-rlen* **.**
moreover note *s3-correct*[*OF vld ippkt ifvld*(*1*) ∗, *THEN iffD2*, *unfolded oft-def*[*symmetric*], *of* [*Forward oif*]]
ultimately show *?case* **by** *simp*
qed
show *w1*: ⋀*oif*. (*OF-priority-match OF-match-fields-safe oft p* = *Action* [*Forward oif*]) = (*simple-linux-router-nol12 rt fw p* = *Some* (*p*⦇*p-oiface* := *oif*⦈))
unfolding *lin* **using** *w1-1 w1-2* **by** *blast*
show *w2*: (*OF-priority-match OF-match-fields-safe oft p* = *Action* []) = (*simple-linux-router-nol12 rt fw p* = *None*)
unfolding *lin*
proof(*rule iffI*, *goal-cases*)
case *1*
note *s3-correct*[*OF vld ippkt ifvld*(*1*) ∗, *THEN iffD1*, *unfolded oft-def*[*symmetric*], *OF 1*]
then obtain *r oif* **where** *roif*: *generalized-sfw* (*lr-of-tran-fbs rt fw ifs*) *p* = *Some* (*r*, *oif*, *simple-action.Drop*)
unfolding *map-snd-apfst map-snd-annotate-rlen* **by**(*clarsimp split*: *if-splits*)
note *lr-of-tran-fbs-dropD*[*OF s1 s2*(*3*) *this*]
thus *?case* **.**
next
case *2*
note *lr-of-tran-fbs-dropI*[*OF s1 s2*(*3*) *s2*(*1*) *this*, *of ifs*] **then**
obtain *r oif* **where** *generalized-sfw* (*lr-of-tran-fbs rt fw ifs*) *p* = *Some* (*r*, *oif*, *simple-action.Drop*) **by** *blast*
hence *generalized-sfw* (*map snd* (*map* (*apfst of-nat*) (*annotate-rlen* (*lr-of-tran-fbs rt fw ifs*)))) *p* = *Some* (*r*, *oif*, *simple-action.Drop*)
unfolding *map-snd-apfst map-snd-annotate-rlen* **.**
moreover note *s3-correct*[*OF vld ippkt ifvld*(*1*) ∗, *THEN iffD2*, *unfolded oft-def*[*symmetric*], *of* []]
ultimately show *?case* **by** *force*
qed
have *lr-determ*: ⋀*a*. *simple-linux-router-nol12 rt fw p* = *Some a* ⟹ *a* = *p*⦇*p-oiface* := *output-iface* (*routing-table-semantics rt* (*p-dst p*))⦈
by(*clarsimp simp*: *simple-linux-router-nol12-def Let-def not-undec split*: *Option.bind-splits state.splits final-decision.splits*)
show *notno*: *OF-priority-match OF-match-fields-safe oft p* ≠ *NoAction*
apply(*cases simple-linux-router-nol12 rt fw p*)
using *w2* **apply**(*simp*)
using *w1*[*of output-iface* (*routing-table-semantics rt* (*p-dst p*))] **apply**(*simp*)
apply(*drule lr-determ*)
apply(*simp*)
done
show *notub*: *OF-priority-match OF-match-fields-safe oft p* ≠ *Undefined* **unfolding** *lin* **using** *OF-match-linear-ne-Undefined*
.
show *notmult*: ⋀*ls*. *OF-priority-match OF-match-fields-safe oft p* = *Action ls* ⟶ *length ls* ≤ *1*
apply(*cases simple-linux-router-nol12 rt fw p*)

using *w2* **apply**(*simp*)
using *w1*[*of output-iface* (*routing-table-semantics rt* (*p-dst p*))] **apply**(*simp*)
apply(*drule lr-determ*)
apply(*clarsimp*)
done
show ∃ *ls*. *length ls* ≤ *1* ∧ *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
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* (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 *map* (*quote-rewrite* ∘ *common-primitive-rule-toString*) *unfolded* =
[*''*−*p icmp* −*j ACCEPT''*,
''−*i s1*−*lan* −*p tcp* −*m tcp* −−*spts* [*1024*:*65535*] −*m tcp* −−*dpts* [*80*] −*j ACCEPT''*,
''−*i s1*−*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* ∘ *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 ∀ *m* ∈ *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*
lemma *check-simple-fw-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* = [
rr-ctor (*10*,*0*,*1*,*0*) *24* *''s1−lan''* *None 0*,
rr-ctor (*10*,*0*,*2*,*0*) *24* *''s1−wan''* *None 0*,
rr-ctor (*0*,*0*,*0*,*0*) *0* *''s1−wan''* (*Some* (*10*,*0*,*2*,*1*)) *0*
]
by *eval*

definition *SQRL-rtbl-main-sorted* ≡ *rev* (*sort-key* (λ*r*. *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-wan'', (ipv4addr-of-dotdecimal (10,0,1,1), 0x3)),
(''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*))
(''s1-wan'', (ipv4addr-of-dotdecimal (10,0,2,4), 0x6dabd0152059))
]

definition *SQRL-ports* ≡ [

(''s1−lan'', ''1''),
(''s1−wan'', ''2'')
]

lemma *let fw = SQRL-fw-simple in no-oif-match fw ∧ has-default-policy fw ∧ simple-fw-valid fw* **by** *eval*
lemma *let rt = SQRL-rtbl-main-sorted in valid-prefixes rt ∧ has-default-route rt* **by** *eval*
lemma *let ifs = (map iface-name SQRL-ifs) in distinct ifs* **by** *eval*

definition *ofi* ≡
case (*lr-of-tran SQRL-rtbl-main-sorted SQRL-fw-simple* (*map iface-name SQRL-ifs*))
of (*Inr openflow-rules*) ⇒ *map* (*serialize-of-entry* (*the* ∘ *map-of SQRL-ports*)) *openflow-rules*
lemma *ofi* =
[''priority=11,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-proto=1,nw-dst=10.0.2.0/24,action=output:2'',
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=1024/0xfc00,tp-dst=
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=2048/0xf800,tp-dst=
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=4096/0xf000,tp-dst=
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=8192/0xe000,tp-dst=
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=16384/0xc000,tp-dst
''priority=10,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=32768/0x8000,tp-dst
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=1024/0xfc0
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=2048/0xf80
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=4096/0xf00
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=8192/0xe00
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=16384/0xc0
''priority=9,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.2.0/24,tp-src=80,tp-dst=32768/0x80
''priority=8,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-dst=10.0.2.0/24,action=drop'',
''priority=7,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-proto=1,nw-dst=10.0.1.0/24,action=output:1'',
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=1024/0xfc00,tp-dst=8
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=2048/0xf800,tp-dst=8
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=4096/0xf000,tp-dst=8
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=8192/0xe000,tp-dst=8
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=16384/0xc000,tp-dst=
''priority=6,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=32768/0x8000,tp-dst=
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=1024/0xfc0
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=2048/0xf80
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=4096/0xf00
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=8192/0xe00
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=16384/0xc0
''priority=5,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,nw-dst=10.0.1.0/24,tp-src=80,tp-dst=32768/0x80
''priority=4,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-dst=10.0.1.0/24,action=drop'',
''priority=3,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-proto=1,action=output:2'',
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=1024/0xfc00,tp-dst=80,action=output:2'',
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=2048/0xf800,tp-dst=80,action=output:2'',
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=4096/0xf000,tp-dst=80,action=output:2'',
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=8192/0xe000,tp-dst=80,action=output:2'',
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=16384/0xc000,tp-dst=80,action=output:2''
''priority=2,hard-timeout=0,idle-timeout=0,in-port=1,dl-type=0x800,nw-proto=6,tp-src=32768/0x8000,tp-dst=80,action=output:2''
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=1024/0xfc00,action=output:2'',
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=2048/0xf800,action=output:2'',
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=4096/0xf000,action=output:2'',
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=8192/0xe000,action=output:2'',
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=16384/0xc000,action=output:2''
''priority=1,hard-timeout=0,idle-timeout=0,in-port=2,dl-type=0x800,nw-proto=6,tp-src=80,tp-dst=32768/0x8000,action=output:2''

''priority=0,hard-timeout=0,idle-timeout=0,dl-type=0x800,action=drop''] **by** *eval*

value[*code*] *ofi*

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* (*packet-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*))))))

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 0xC6120100 24*, *metric* = *0*, *routing-action* = ⦇*output-iface* = *''ip1''*, *next-hop* = *None*⦈⦈,
 ⦇*routing-match* = *PrefixMatch 0xC6130100 24*, *metric* = *0*, *routing-action* = ⦇*output-iface* = *''op1''*, *next-hop* = *None*⦈⦈,
 ⦇*routing-match* = *PrefixMatch 0 0*, *metric* = *0*, *routing-action* = ⦇*output-iface* = *''op1''*, *next-hop* = *Some 0xC6130102*⦈⦈]
by *eval*
lemma *SQRL-rtbl-main* = [
 rr-ctor (*198*,*18*,*1*,*0*) *24* *''ip1''* *None* *0*,
 rr-ctor (*198*,*19*,*1*,*0*) *24* *''op1''* *None* *0*,
 rr-ctor (*0*,*0*,*0*,*0*) *0* *''op1''* (*Some* (*198*,*19*,*1*,*2*)) *0*
 ]
by *eval*

definition *SQRL-ports* ≡ [
 (*''ip1''*, *''1''*),
 (*''op1''*, *''2''*)
]

definition *ofi* ≡
 case (*lr-of-tran SQRL-rtbl-main SQRL-fw-simple* (*map fst SQRL-ports*))
 of (*Inr 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=192.18.2.2/32,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=192.19.1.2/32,nw-dst=192.19.65.1/32,action=output:2'',
 ''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'',
''priority=2,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.23.23/32,nw-dst=192.18.123.23/32,action=drop'',
''priority=1,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-src=192.18.24.24/32,nw-dst=192.18.124.24/32,action=drop'',
''priority=0,hard-timeout=0,idle-timeout=0,dl-type=0x800,action=drop''] **by** *eval*
value[*code*] *length ofi*

end

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 *routing-table-semantics rt* (*p-dst p*), the firewall decision by *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 $rd \equiv$ *routing-table-semantics rt p* is obtained.
3. The packet's output interface is updated based on *rd*[1].
4. The firewall is queried for a decision: *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:

lemma *simple-linux-router rt fw mlf ifl p* ≡ *do* {
 - ← *iface-packet-check ifl p*;
 let rd — (routing decision) = *routing-table-semantics rt* (*p-dst p*);
 let p = *p*(|*p-oiface* := *output-iface rd*|);
 let fd — (firewall decision) = *simple-fw fw p*;
 - ← (*case fd of Decision FinalAllow* ⇒ *Some* () | *Decision FinalDeny* ⇒ *None*);
 let nh = (*case next-hop rd of None* ⇒ *p-dst p* | *Some a* ⇒ *a*);
 ma ← *mlf nh*;
 Some (*p*(|*p-l2dst* := *ma*|))
}
unfolding *fromMaybe-def*[*symmetric*] **by**(*fact simple-linux-router-def*)

where *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.

[1] 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-iiface 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 subnet[2].

While a test-wise implementation based on this model also showed acceptable results, the model is still problematic. The check *p-oiface p = p-iiface p* and the firewall require access to the output interface. The details of why this cannot be provided are be elaborated in Section 1.3. The intuitive explanation is that an OpenFlow match can not have a field for the output interface. We thus simplified the model even further:

lemma *simple-linux-router-nol12 rt fw p ≡ do {*
let rd = routing-table-semantics rt (p-dst p);
let p = p(|p-oiface := output-iface rd|);
let fd = simple-fw fw p;
- ← (case fd of Decision FinalAllow ⇒ Some () | Decision FinalDeny ⇒ None);
Some p
} **by**(*fact simple-linux-router-nol12-def*)

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:

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

theorem
⟦*iface-packet-check ifl pii ≠ None;*
mlf (case next-hop (routing-table-semantics rt (p-dst pii)) of None ⇒ p-dst pii | Some a ⇒ a) ≠ None⟧ ⟹
∃ *x. map-option* (λ*p. p*(|*p-l2dst* := *x*|)) (*simple-linux-router-nol12 rt fw pii*) = *simple-linux-router rt fw mlf ifl pii*
by(*fact rtr-nomac-eq*[*unfolded fromMaybe-def*])

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.

schematic-goal (*?rtbl-entry* :: ('*a*::*len*) *routing-rule*) = (| *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:

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:

lemma *valid-prefix* (*PrefixMatch pfx len*) ≡ *pfx* && (*2* ^ (*32* − *len*) − *1*) = (*0* :: *32 word*)
by (*simp add*: *valid-prefix-def pfxm-mask-def mask-eq-decr-exp and.commute*)
lemma *has-default-route rt* ⟷ (∃ *r* ∈ *set rt*. *pfxm-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.

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 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. $p_d \in \{1..1023\}$

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 for 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 (*?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*)
} **by**(*fact of-match-field-typeset*)

Two things are worth additional mention: L3 and L4 "addressess". The *IPv4Src* and *IPv4Dst* matches are specified as "can be subnet masked" in [2], whereras [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[3], executing a set of given *of-match-field*s 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 2048* 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 2048*), the matches that had their precondition fulfilled by it (e.g., *IPv4Src* and *IPv4Src* in this example) are evalutated. 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 134744072 32*)} instead of just those destined for 8.8.8.8 or causing an error. We found this to be unsatisfactory.

[3] See also: [8, §2.3]

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

lemma
prerequisites (*VlanPriority pri*) *m* = (∃ *id*. *let v* = *VlanId id in v* ∈ *m* ∧ *prerequisites v m*)
prerequisites (*IPv4Proto pr*) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*)
prerequisites (*IPv4Src a*) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*)
prerequisites (*IPv4Dst a*) *m* = (*let v* = *EtherType 0x0800 in v* ∈ *m* ∧ *prerequisites v m*)
prerequisites (*L4Src p msk*) *m* = (∃ *proto* ∈ {*TCP*, *UDP*, *L4-Protocol.SCTP*}. *let v* = *IPv4Proto proto in v* ∈ *m* ∧ *prerequisites v m*)
prerequisites (*L4Dst p msk*) *m* = *prerequisites* (*L4Src undefined undefined*) *m*
by(*fact prerequisites.simps*)+

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

lemma *OF-match-fields m p* =
(*if* ∃ *f* ∈ *m*. ¬*prerequisites f m then None else*
if ∀ *f* ∈ *m*. *match-no-prereq f p then Some True else Some False*)
by(*fact OF-match-fields-alt*)

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

flow entry from a table can be selected. To prevent to much entanglement with the previous section, we assume an arbitrary match function γ. This function γ takes the match condition m from a flow entry *OFEntry priority m action* and decides whether a packet matches those.

The flow table is simply a list of flow table entries *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 γ), in Isabelle/HOL[5]:

lemma γ (*ofe-fields fe*) p = *True* $\Longrightarrow$
$\forall fe' \in set\ (ft1\ @\ ft2).\ ofe\text{-}prio\ fe' > ofe\text{-}prio\ fe \longrightarrow \gamma\ (ofe\text{-}fields\ fe')\ p = False \Longrightarrow$
guha-table-semantics γ (*ft1* @ *fe* # *ft2*) p (*Some* (*ofe-action fe*))
$\forall fe \in set\ ft.\ \gamma\ (ofe\text{-}fields\ fe)\ p = False \Longrightarrow$
guha-table-semantics γ *ft p None* **by**(*fact guha-matched guha-unmatched*)+

Guha *et al.* have deliberately made their semantics nondeterministic, to match the fact that the switch "may choose any ordering". This can lead to undesired results:

lemma $CARD('action) \geq 2 \Longrightarrow \exists ff.\ \gamma\ ff\ p \Longrightarrow \exists ft\ (a1 :: 'action)\ (a2 :: 'action).\ a1 \neq a2 \wedge$ *guha-table-semantics* γ *ft p* (*Some a1*) $\wedge$ *guha-table-semantics* γ *ft p* (*Some a2*)
by(*fact guha-table-semantics-ex2res*)

[4]This behavior has been deprecated.

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

This means that, given at least two distinct actions exist and our matcher γ 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
OF-priority-match γ *flow-entries packet* = (
let m = *filter* $(\lambda f.\ \gamma\ (ofe\text{-}fields\ f)\ packet)$ *flow-entries*;
m' = *filter* $(\lambda f.\ \forall fo \in set\ m.\ ofe\text{-}prio\ fo \leq ofe\text{-}prio\ f)$ *m in*
case m' *of* [] $\Rightarrow$ *NoAction*
| [*s*] $\Rightarrow$ *Action* (*ofe-action s*)
| - $\Rightarrow$ *Undefined*)
unfolding *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 *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 *Undefined* for indicating undefined behavior is returned.

[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* = ($\forall$ *a* $\in$ *set ft*. $\forall$ *b* $\in$ *set ft*. (*a* $\neq$ *b* $\wedge$ *ofe-prio a* = *ofe-prio b*) $\longrightarrow$ $\neg$($\exists$ *p*. γ (*ofe-fields a*) *p* $\wedge$ 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 abscence of *Undefined*[7]:

lemma ⟦*check-no-overlap* γ *ft*; *distinct ft*⟧ $\Longrightarrow$
OF-priority-match γ *ft p* $\neq$ *Undefined* **by** (*simp add*: *no-overlapsI no-overlaps-not-unefined*)

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*⟧ $\Longrightarrow$
OF-priority-match γ *ft p* = *option-to-ftb d* $\longleftrightarrow$ *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
OF-match-linear γ [] *p* = *NoAction*
OF-match-linear γ (*a*#*as*) *p* = (*if* γ (*ofe-fields a*) *p then Action* (*ofe-action a*) *else OF-match-linear* γ *as p*)
by(*fact OF-match-linear.simps*)+

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

lemma
⟦*no-overlaps* γ *f* ;*sorted-descending* (*map ofe-prio f*)⟧ $\Longrightarrow$
OF-match-linear γ *f p* = *OF-priority-match* γ *f p*
by(*fact OF-eq*)

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
generalized-sfw [] *p* = *None*
generalized-sfw (*a* # *as*) *p* = (*if* (*case a of* (*m*,-) ⇒ *simple-matches m p*) *then Some a else generalized-sfw as p*)
by(*fact generalized-sfw-simps*)+

Based on that, we asked: if fw_1 makes the decision a (where a is the second element of the result tuple from *generalized-sfw*) and fw_2 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:

[7] It is slightly stronger than necessary, overlapping rules might be shadowed and thus never influence the behavior.

[8] Note 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* ≡ [(*u*,*a*,*b*). (*m1*,*a*) ← *l1*, (*m2*,*b*) ← *l2*, *u* ← (*case simple-match-and m1 m2 of None* ⇒ [] | *Some s* ⇒ [*s*])]
by(*fact generalized-fw-join-def*[*unfolded option2list-def*])+

This definition validates the following lemma:

lemma *generalized-sfw* (*generalized-fw-join* fw_1 fw_2) *p* = *Some* (*u*, d_1,d_2) ⟷ (∃ r_1 r_2. *generalized-sfw* fw_1 *p* = *Some* (r_1,d_1) ∧ *generalized-sfw* fw_2 *p* = *Some* (r_2,d_2) ∧ *Some u* = *simple-match-and* r_1 r_2)
by (*auto dest*: *generalized-fw-joinD sym simp add*: *generalized-fw-joinI*)

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* ≡ (*if a* = *simple-action.Accept* ∧ *b* = *simple-action.Accept then simple-action.Accept else simple-action.Drop*)
definition *simple-rule-conj* ≡ (*uncurry SimpleRule* ∘ *apsnd* (*uncurry simple-action-conj*))
theorem *simple-fw* rs_1 *p* = *Decision FinalAllow* ∧ *simple-fw* rs_2 *p* = *Decision FinalAllow* ⟷
simple-fw (*map simple-rule-conj* (*generalized-fw-join* (*map simple-rule-dtor* rs_1) (*map simple-rule-dtor* rs_2))) *p* = *Decision FinalAllow*
unfolding *simple-rule-conj-def simple-action-conj-def*[*abs-def*] **using** *simple-fw-join*
by(*force simp add*: *comp-def apsnd-def map-prod-def case-prod-unfold uncurry-def*[*abs-def*])

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 preconditions 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-rule*s 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 *rr* 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*) ∘ *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* × *32 simple-match* × *char list* × *simple-action*) *list* to an OpenFlow table. While converting the *char list* × *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.[9]

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

lemma *simple-match-to-of-match*:
assumes
simple-match-valid r
p-iiface p ∈ *set ifs*
match-iface (*oiface r*) (*p-oiface p*)
p-l2type p = *0x800*
shows
simple-matches r p ⟷ (∃ *gr* ∈ *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:

[9] 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 assing separate priorities) and we did not have a verified implementation of an algorithm that does so.

```
lemma lr-of-tran rt fw ifs ≡
if ¬ (no-oif-match fw ∧ has-default-policy fw ∧ simple-fw-valid fw ∧ valid-prefixes rt ∧ has-default-route rt ∧
distinct ifs)
  then Inl ″Error in creating OpenFlow table: prerequisites not satisifed″
  else (
let
  nfw = map simple-rule-dtor fw;
  frt = map (λr. (route2match r, output-iface (routing-action r))) rt;
  nrd = generalized-fw-join frt nfw;
  ard = (map (apfst of-nat) ∘ annotate-rlen) nrd
  in
  if length nrd < unat (− 1 :: 16 word)
  then Inr (pack-OF-entries ifs ard)
  else Inl ″Error in creating OpenFlow table: priority number space exhausted″
)
unfolding Let-def lr-of-tran-def lr-of-tran-fbs-def lr-of-tran-s1-def comp-def route2match-def by force
```

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

- 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 *oiface* r = *ifaceAny*, since

  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 by 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 satisifed:

lemma
¬ *no-oif-match fw* ∨
¬ *has-default-policy fw* ∨
¬ *simple-fw-valid fw* ∨
¬ *valid-prefixes rt* ∨
¬ *has-default-route rt* ∨
¬ *distinct ifs* ⟹

theorem
fixes
p :: (32, 'a) simple-packet-ext-scheme
assumes
p-iiface p ∈ set ifs **and** p-l2type p = 0x800
lr-of-tran rt fw ifs = Inr oft
shows
OF-priority-match OF-match-fields-safe oft p = Action [Forward oif] ⟷ simple-linux-router-nol12 rt fw p = (Some (p(|p-oiface := oif|)))
OF-priority-match OF-match-fields-safe oft p = Action [] ⟷ simple-linux-router-nol12 rt fw p = None
OF-priority-match OF-match-fields-safe oft p ≠ NoAction OF-priority-match OF-match-fields-safe oft p ≠ Undefined
OF-priority-match OF-match-fields-safe oft p = Action ls ⟶ length ls ≤ 1
∃ls. length ls ≤ 1 ∧ OF-priority-match OF-match-fields-safe oft p = Action ls
using assms lr-of-tran-correct **by** simp-all

Figure 2: Central theorem on *lr-of-tran*

∃ *err*. *lr-of-tran rt fw ifs* = *Inl err* **unfolding** *lr-of-tran-def* **by**(*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 does not produce OpenFlow rulesets, but FlowLog [9] controller programs.
- 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 attemts to leviate 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; `s1-lan` received port number 1.) While the configuration does not fulfil any special function (especially, no traffic from the interface `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.

```
1 :FORWARD DROP [0:0]
2 -A FORWARD -d 10.0.2.0/24 -i s1-lan -p tcp -
      m tcp --sport 32768:65535 --dport 80 -j
      ACCEPT
```

(a) FORWARD chain

```
1 10.0.2.0/24 dev s1-wan  proto kernel  scope
      link  src 10.0.2.4
2 10.0.1.0/24 dev s1-lan  proto kernel  scope
      link  src 10.0.1.1
3 default via 10.0.2.1 dev s1-wan
```

(b) Routing table (sorted)

```
1 priority=4,hard_timeout=0,idle_timeout=0,in_port=1,dl_type=0x800,nw_proto=6,nw_dst=10.0.2.0/24,
      tp_src=32768/0x8000,tp_dst=80,action=output:2
2 priority=3,hard_timeout=0,idle_timeout=0,dl_type=0x800,nw_dst=10.0.2.0/24,action=drop
3 priority=2,hard_timeout=0,idle_timeout=0,dl_type=0x800,nw_dst=10.0.1.0/24,action=drop
4 priority=1,hard_timeout=0,idle_timeout=0,in_port=1,dl_type=0x800,nw_proto=6,nw_dst=10.0.2.0/24,
      tp_src=32768/0x8000,tp_dst=80,action=output:2
5 priority=0,hard_timeout=0,idle_timeout=0,dl_type=0x800,action=drop
```

(c) Resulting OpenFlow rules

Figure 3: Example Network 1 – Configuration

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.

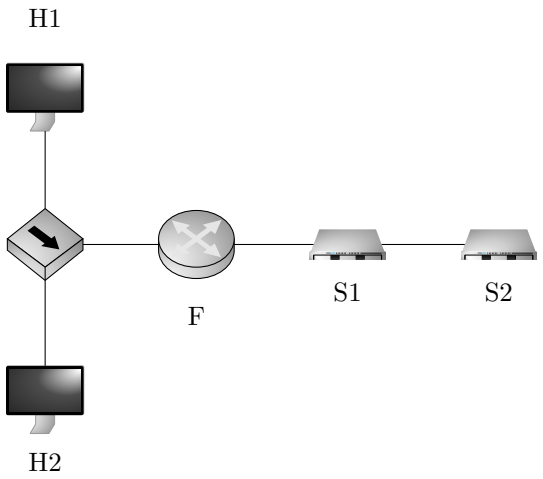
The topology has been chosen for a number of reasons: we wanted one device which is not inside a common subnet with F and thus requires no reconfiguration for the translation. Moreover, we wanted two devices in a network that can communicate with each other while being overheard by F. For this purpose, 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 rules[11], 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=1024/0xfc00`, `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 actually destined for it. Obviously, this network violates

[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

```
:FORWARD DROP [0:0]
-A FORWARD -p icmp -j ACCEPT
-A FORWARD -i s1-lan -p tcp -m tcp --sport
    1024:65535 --dport 80 -j ACCEPT
-A FORWARD -d 10.0.1.0/24 -i s1-wan -p tcp -m tcp
     --sport 80 --dport 1024:65535 -j ACCEPT
```

(b) `FORWARD` chain

Figure 4: Example Network 2

this assumption. We can trigger this in many ways, for example by sending an ICMP ping from H1 to H2. This will cause the generated rule `priority=7, icmp, nw_dst=10.0.1.0/24 actions=output:1` (where port 1 is the port facing H1 and H2) to be activated twice. This is obviously not desired behavior. Dealing with this is, as mentioned, future work.

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[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.

[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].

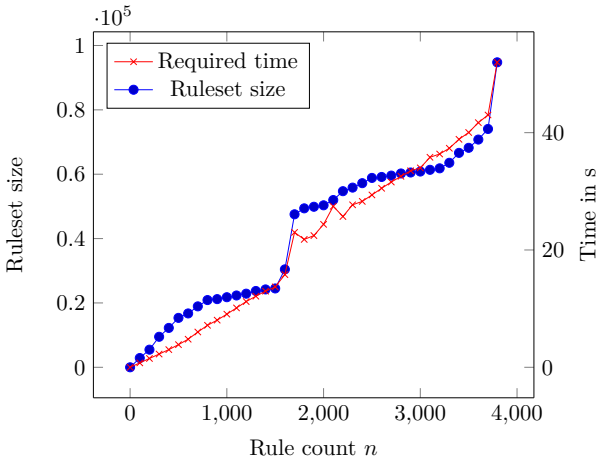


Figure 5: Benchmark

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.
- 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 coincede, 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 addressed in a firewall, based on the information from the routing table.[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.

[13] As of now this has already been implemented, but is not yet fully ready.

References

[1] Open vSwitch. http://openvswitch.org/.

[2] OpenFlow Switch Specification v1.0.0, December 2009.

[3] OpenFlow Switch Specification v1.5.1, March 2015.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[5] C. Diekmann and L. Hupel. Iptables Semantics. *Archive of Formal Proofs*, Sept. 2016. http://isa-afp.org/entries/Iptables_Semantics.shtml, Formal proof development.

[6] C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle. Verified iptables Firewall Analysis. In *Proceedings of IFIP Networking 2016 (NETWORKING 16)*, May 2016.

[7] A. Guha, M. Reitblatt, and N. Foster. Machine-verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 483–494, New York, NY, USA, 2013. ACM.

[8] J. Michaelis and C. Diekmann. Middlebox models in network verification research. In *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Winter Semester 2015/2016*, volume 17, 2016.

[9] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, 2014.

[10] T. Nelson, A. D. Ferguson, D. Yu, R. Fonseca, and S. Krishnamurthi. Exodus: toward automatic migration of enterprise network configurations to SDNs. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 13. ACM, 2015.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2015.