LOFT — Verified Migration of Linux Firewalls to SDN

Julius Michaelis and Cornelius Diekmann

October 11, 2017

Abstract

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

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

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

Contents

I Code

II Documentation

1 Configuration Translation
1.1 Linux Firewall Model
1.1.1 Routing Table
1.1.2 iptables Firewall
1.2 OpenFlow Switch Model
1.2.1 Matching Flow Table entries
1.2.2 Evaluating a Flow Table
1.3 Translation Implementation
1.3.1 Chaining Firewalls
1.3.2 Translation Implementation
1.3.3 Comparison to Exodus

2 Evaluation
2.1 Mininet Examples
2.2 Performance Evaluation

3 Conclusion and Future Work
Part I
Code

theory OpenFlow-Matches
imports IP-Addresses.Prefix-Match
  Simple-Firewall.Simple-Packet
  HOL-Library.Monad-Syntax

  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)

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 |

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 ⟨proof⟩

definition less-eq-of-match-field1 (a::of-match-field) (b::of-match-field) ≡ (case (a, b) of
(IngressPort a, IngressPort b) ⇒ String.implode a ≤ String.implode 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) |

instance ⟨proof⟩

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 ⇒ 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 o 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 ≠ {} 
(proof)

lemma of-safe-unsafe-match-eq: all-prerequisites m ⇒ OF-match-fields m p = Some (OF-match-fields-unsafe m p) 
(proof)

lemma of-match-fields-safe-eq: assumes all-prerequisites m shows OF-match-fields-safe m = OF-match-fields-unsafe m 
(proof)

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)
(proof)

lemma of-match-fields-safe-eq2: assumes all-prerequisites m shows OF-match-fields-safe m p ←→ OF-match-fields m p = 
Some True 
(proof)

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 ∈ 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 = (∀ a ∈ set ft. \∀ b ∈ set ft. \forall p ∈ UNIV. (ofe-prio a = ofe-prio b ∧ γ (ofe-fields a) p ∧ a \neq b) → → γ (ofe-fields b) p)

**definition** check-no-overlap2 γ ft = (∀ a ∈ set ft. \forall b ∈ set ft. (a \neq b ∧ ofe-prio a = ofe-prio b) → → (∃ p ∈ UNIV. γ (ofe-fields a) p ∧ γ (ofe-fields b) p))

**lemma** check-no-overlap-alt: check-no-overlap γ ft = check-no-overlap2 γ ft
(proof)

**lemma** no-overlap-not-undefined: check-no-overlap γ ft => OF-same-priority-match2 γ ft p \neq Undefined
(proof)

**fun** OF-match-linear :: ('m, 'p) field-matcher => ('m, 'a) flowtable => 'p => '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
(proof)

**lemma** OF-match-linear-append: OF-match-linear γ (a @ b) p = (case OF-match-linear γ a p of NoAction ⇒
  OF-match-linear γ b p | x ⇒ x)
(proof)

**lemma** OF-match-linear-match-allsameaction: (\forall r ∈ set oms; γ gr p = True) \rightarrow \rightarrow
  OF-match-linear γ (map (\lambda x. split3 OFEntry (pri, x, act)) oms) p = Action act
(proof)

**lemma** OF-lm-noa-none-iff: OF-match-linear γ ft p = NoAction \leftrightarrow (\forall e ∈ set ft. \neg γ (ofe-fields e) p)
(proof)

**lemma** set-eq-rule: (\forall x. x ∈ a \rightarrow x ∈ b) \rightarrow (\forall x. x ∈ b \rightarrow x ∈ a) \rightarrow a = b (proof)

**lemma** unmatching-insert-agnostic: \neg γ (ofe-fields a) p \rightarrow OF-same-priority-match2 γ (a \# ft) p
(proof)

**lemma** OF-match-eq: sorted-descending (map ofe-prio ft) \rightarrow check-no-overlap γ ft \rightarrow
  OF-same-priority-match2 γ ft p = OF-match-linear γ ft p
(proof)

**lemma** overlap-sort-invar[simp]: check-no-overlap γ (sort-descending-key k ft) = check-no-overlap γ ft
(proof)

**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

6
lemma prio-match-matcher-alt: \{ 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)\} = (let matching = \{ f. f ∈ set flow-entries ∧ γ (ofe-fields f) packet \} in \{ f. f ∈ matching ∧ (∀ fo ∈ matching. ofe-prio fo ≤ ofe-prio f) \})
(\proof)

lemma prio-match-matcher-alt2: (let matching = \{ f. f ∈ set flow-entries ∧ γ (ofe-fields f) packet \} in \{ f. f ∈ matching ∧ (∀ fo ∈ set matching. ofe-prio fo ≤ ofe-prio f) \}) = set (let matching = filter (λ f. γ (ofe-fields f) packet) flow-entries in filter (λ f. ∀ fo ∈ set matching. ofe-prio fo ≤ ofe-prio f) matching)
(\proof)

definition OF-priority-match where
OF-priority-match γ flow-entries packet ≡ let m = filter (λ f. γ (ofe-fields f) packet) flow-entries;
    m′ = filter (λ f. ∀ fo ∈ set m. ofe-prio fo ≤ ofe-prio f) m in
case m′ of [] ⇒ NoAction
    | [s] ⇒ Action (ofe-action s)
    | - ⇒ Undefined

definition OF-priority-match-ana where
OF-priority-match-ana γ flow-entries packet ≡ let m = filter (λ f. γ (ofe-fields f) packet) flow-entries;
    m′ = filter (λ f. ∀ fo ∈ set m. ofe-prio fo ≤ ofe-prio f) m in
case m′ of [] ⇒ NoAction
    | [s] ⇒ Action s
    | - ⇒ Undefined

lemma filter-singleton: [x→ s x] = [y] ⇒ f y ∧ y ∈ set s (\proof)

(\proof)

fun no-overlaps where
no-overlaps [] = True |
no-overlaps γ (a#as) = (no-overlaps γ as ∧ (∀ b ∈ set as. ofe-prio a = ofe-prio b → ¬(∃ p ∈ UNIV. γ (ofe-fields a) p ∧ γ (ofe-fields b) p)))

definition no-overlap-ConsI: check-no-overlap2 γ (x#xs) ⇒ check-no-overlap2 γ xs
(\proof)

lemma no-overlapsI: check-no-overlap γ t ⇒ distinct t ⇒ no-overlaps γ t
(\proof)

lemma check-no-overlapI: no-overlaps γ t ⇒ check-no-overlap γ t
(\proof)
lemma \((\forall e. e \in set t \implies \neg \gamma (ofe-fields e) p) \implies no-overlaps \gamma t\)

(proof)

lemma no-overlaps-append: no-overlaps \gamma (x @ y) \implies no-overlaps \gamma y

(proof)

lemma no-overlaps-ne1: no-overlaps \gamma (x @ a @ b @ z) \implies ((\exists p. \gamma (ofe-fields a) p) \lor (\exists p. \gamma (ofe-fields b) p)) \implies a \neq b

(proof)

lemma no-overlaps-defeq: no-overlaps \gamma fe \implies OF-same-priority-match2 \gamma fe p = OF-priority-match \gamma fe p

(proof)

lemma distinct fe \implies check-no-overlap \gamma fe \implies OF-same-priority-match2 \gamma fe p = OF-priority-match \gamma fe p

(proof)

theorem OF-eq:

assumes no: no-overlaps \gamma f

and so: sorted-descending (map ofe-prio f)

shows OF-match-linear \gamma f p = OF-priority-match \gamma f p

(proof)

corollary OF-eq-sort:

assumes no: no-overlaps \gamma f

shows OF-priority-match \gamma f p = OF-match-linear \gamma (sort-descending-key ofe-prio f) p

(proof)

lemma OF-lm-noa-none: OF-match-linear \gamma ft p = NoAction \implies \forall e \in set ft. \neg \gamma (ofe-fields e) p

(proof)

lemma OF-spm3-noa-none:

assumes no: no-overlaps \gamma ft

shows OF-priority-match \gamma ft p = NoAction \implies \forall e \in set ft. \neg \gamma (ofe-fields e) p

(proof)

lemma no-overlaps-not-undefined: no-overlaps \gamma ft p \implies OF-priority-match \gamma ft p \neq Undefined

(proof)

end

theory OpenFlow-Serialize

imports OpenFlow-Matches

OpenFlow-Action

Semantics-OpenFlow

Simple-Firewall,Primitives-toString

IP-Addresses.Lib-Word-toString

HOL-Library.Code-Char

begin

definition serialization-test-entry \equiv 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 (op << (1::48 word) o op + 8) o rev) [0..<6]

definition serialize-mac (m::48 word) ≡ (intersperse (CHR "::") o map (hex-string-of-word 1 o (λh. (m >> h * 8) & & 0xff)) o rev) [0..<6]

lemma serialize-mac 0xdeadbeefafe = "de:ad:be:ef:ca:fe" (proof)

definition serialize-action pids a ≡ (case a of
    Forward oif ⇒ "output:" @ pids oif |
    ModifyField-l2dst na ⇒ "mod-l2-dst:" @ serialize-mac na)

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

lemma serialize-actions (λoif. "42") (ofe-action serialization-test-entry) = "mod-l2-dst:a0:41:f1:85:e8:62,output:42" (proof)

lemma serialize-actions anything [] = "drop" (proof)

definition prefix-to-string pfz ≡ ipv4-cidr-toString (pfzm-prefix pfz, pfzm-length pfz)

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 = max-word then [] else "/0x" @ hex-string-of-word 3 m) |
serialize-of-match - (L4Dst i m) = "tp-dst=" @ dec-string-of-word0 i @ (if m = max-word 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 ≡ ap @ "hard-timeout=0,idle-timeout=0," o intersperse (CHR ",") o map (serialize-of-match pids) o 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)))) (proof)

export-code serialize-of-matches checking SML

lemma serialize-of-matches (λoif. "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" (proof)

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 =

⟨ proof ⟩

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 ma: ∃ 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 ⟩

lemma guha-amstaendlich:

  assumes ae: a = ofe-action fe
  assumes ele: fe ∈ set ft
  assumes rest: γ (ofe-fields fe) p
  shows ∀ fe′ ∈ set ft. ofe-prio fe′ > ofe-prio fe ⟹ ¬γ (ofe-fields fe′) p

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

⟨ proof ⟩

lemma guha-matched-rule-inversion:

  assumes guha-table-semantics γ ft p (Some a)
  shows ∃ fe ∈ set ft. a = ofe-action fe ∧ γ (ofe-fields fe) p ∧ (∀ fe′ ∈ set ft. ofe-prio fe′ > ofe-prio fe ⟹ ¬γ (ofe-fields fe′) p)

⟨ proof ⟩

lemma guha-equal-Action:

  assumes no: no-overlaps γ ft
  assumesspm: OF-priority-match γ ft p = Action a
  shows guha-table-semantics γ ft p (Some a)

⟨ proof ⟩

lemma guha-equal-NoAction:

  assumes no: no-overlaps γ ft
  assumesspm: OF-priority-match γ ft p = NoAction
  shows guha-table-semantics γ ft p None

⟨ proof ⟩

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))
⟨proof⟩

lemma guha-deterministic1: guha-table-semantics γ ft p (Some x1) ⇒¬ guha-table-semantics γ ft p None
⟨proof⟩

lemma guha-deterministic2: \[\text{no-overlaps γ ft}; \text{guha-table-semantics γ ft p (Some x1)}; \text{guha-table-semantics γ ft p (Some a)}\] ⇒ x1 = a
⟨proof⟩

lemma guha-equal:
assumes no: no-overlaps γ ft
shows OF-priority-match γ ft p = option-to-ftb d ⇔ guha-table-semantics γ ft p d
⟨proof⟩

lemma guha-nondeterministicD:
assumes ¬ check-no-overlap γ ft
shows ∃ fe1 fe2 p. fe1 ∈ set ft ∧ fe2 ∈ set ft
∧ guha-table-semantics γ ft p (Some (ofe-action fe1))
∧ guha-table-semantics γ ft p (Some (ofe-action fe2))
⟨proof⟩

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 will never run into undefined behavior.

lemma
assumes CARD(‘action) ≥ 3
assumes ∀ p. γ x p
shows ∃ ft::(‘a, ‘action) flow-entry-match list. ¬ check-no-overlap γ ft ∧
¬(∃ fe1 fe2 p. fe1 ∈ set ft ∧ fe2 ∈ set ft ∧ fe1 ≠ fe2 ∧ ofe-prio fe1 = ofe-prio fe2
∧ guha-table-semantics γ ft p (Some (ofe-action fe1))
∧ guha-table-semantics γ ft p (Some (ofe-action fe2)))
⟨proof⟩
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
begin
hide-const Misc.uncurry
hide-fact Misc.uncurry-def

definition route2match r =
  \(\text{\text{iiface} = iiface\text{Any}, \text{oiface} = oiface\text{Any}},\)
\[ \text{let pm} = \text{PrefixMatch} (\text{fst } m \text{ and } \text{snd } m) \text{ in if pm = PrefixMatch 0 0 then None else Some pm} \]

\text{lemma prefix-match-semantics-simple-match:}
\begin{align*}
\text{assumes some: } & \text{toprefixmatch } m = \text{Some pm} \\
\text{assumes vld: } & \text{valid-prefix pm} \\
\text{shows prefix-match-semantics pm = simple-match-ip m} \\
\end{align*}

\text{definition simple-match-to-of-match-single ::}
\[
(32, 0) \text{ simple-match-scheme} \Rightarrow \text{char list option} \Rightarrow \text{protocol} \Rightarrow (16 \text{ word } \times 16 \text{ word}) \text{ option} \Rightarrow \text{of-match-field set} \\
\text{where}
\text{simple-match-to-of-match-single } m \text{ iff sport dport} \equiv
\begin{align*}
\text{uncurry L4Src ' option2set sport} \cup \text{uncurry L4Dst ' option2set dport} \\
\cup IP4Proto ' (\text{case proto of ProtoAny} \Rightarrow \{} | \text{Proto p} \Rightarrow \{p\}) \text{ (* protocol is an 8 word option anyway... *)} \\
\cup \text{IngressPort ' option2set if} \\
\cup \text{IP4Src ' option2set (toprefixmatch (src m))} \cup \text{IP4Dst ' option2set (toprefixmatch (dst m))} \\
\cup \{\text{EtherType 0x0800}\}
\end{align*}
\]

\text{definition simple-match-to-of-match :: 32 simple-match \Rightarrow string list \Rightarrow of-match-field set list where}
\[
\text{simple-match-to-of-match } m \text{ ifs} \equiv \text{(let }
\text{npm} = (\lambda p. \text{fst } p = 0 \land \text{snd } p = \text{max-word}); \\
\text{sb} = (\lambda p. \text{if pm } p \text{ then [None] else if } \text{fst } p \leq \text{snd } p \text{ then map (Some o (\lambda pfx. (pfxm-prefix pfx, NOT pfxm-mask pfx))) (wordinterval-CIDR-split-prefixmatch (WordInterval (fst } p \text{) } (\text{snd } p))} \text{ else []}) \\
\text{in [simple-match-to-of-match-single } m \text{ iff (proto } m \text{) sport dport.} \\
\text{if } \text{ifface } m = \text{ifaceAny then [None] else [Some } i \text{ i } \text{iff}, match-iface (\text{ifface } m \text{ i}), \\
\text{sport } \text{sb (sports } m), \\
\text{dport } \text{sb (dports } m))]
\]

\text{lemma smtoms-eq-hlp: simple-match-to-of-match-single } r a b c d = \text{simple-match-to-of-match-single } r f g h i \leftrightarrow (a = f \land b = g \land c = h \land d = i)

\langle proof \rangle

\text{lemma simple-match-to-of-match-generates-prereqs: simple-match-valid } m \Rightarrow r \in \text{set (simple-match-to-of-match } m \text{ ifs)} \Rightarrow \text{all-prerequisites } r 

\langle proof \rangle

\text{lemma and-assoc: } a \land b \land c \leftrightarrow (a \land b) \land c \langle proof \rangle

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

\text{abbreviation simple-fw-prefix-to-wordinterval \equiv \text{prefix-to-wordinterval } \circ \text{uncurry PrefixMatch}}

\text{lemma simple-match-port-alt: simple-match-port } m p \leftrightarrow p \in \text{wordinterval-to-set (uncurry WordInterval } m) \langle proof \rangle

\text{lemma simple-match-src-alt: simple-match-valid } r \Rightarrow \text{simple-match-ip (src } r \text{) } p \leftrightarrow \text{prefix-match-semantics (PrefixMatch (fst (src } r \text{)) (snd (src } r\text{))) } p
proof

lemma simple-match-dst-alt: simple-match-valid r \implies
  simple-match-ip (dst r) p \iff prefix-match-semantics (PrefixMatch (fst (dst r)) (snd (dst r))) p
  
proof

lemma \( x \in \text{set} \text{ (wordinterval-CIDR-split-prefixmatch } w ) \implies \text{valid-prefix } x \)

proof

lemma simple-match-to-of-matchI:
  assumes mv: simple-match-valid r
  assumes mm: simple-matches r p
  assumes ii: p-iface p \in \text{set } ifs
  assumes ipkt: p-l2type p = 0x800
  shows eq: \exists gr \in \text{set } (\text{simple-match-to-of-match } r \text{ ifs}). \text{OF-match-fields } gr p = \text{Some True}

proof

lemma prefix-match-00\text{[simp.intro]}: prefix-match-semantics (PrefixMatch 0 0) p

proof

lemma simple-match-to-of-matchD:
  assumes eg: gr \in \text{set } (\text{simple-match-to-of-match } r \text{ ifs})
  assumes mo: \text{OF-match-fields } gr p = \text{Some True}
  assumes me: match-iface (oiface r) (p-oiface p)
  assumes mv: simple-match-valid r
  shows simple-matches r p

proof

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')] 

proof

lemma fst-annotate-rlen-le: (k, a) \in \text{set } (\text{annotate-rlen } l) \implies k < \text{length } l
  
proof

lemma distinct-fst-annotate-rlen: distinct (map fst (annotate-rlen l))
  
proof

lemma distinct-annotate-rlen: distinct (annotate-rlen l)
  
proof

lemma in-annotate-rlen: (a,x) \in \text{set } (\text{annotate-rlen } l) \implies x \in \text{set } l
  
proof

lemma map-snd-annotate-rlen: map snd (annotate-rlen l) = l
  
proof

lemma sorted-descending (map fst (annotate-rlen l))
  
proof

lemma annotate-rlen l = zip (rev [0..<\text{length } l]) l
  
proof

primrec annotate-rlen-code where
  annotate-rlen-code [] = (0,[])
  annotate-rlen-code (a#as) = (case annotate-rlen-code as of (r,aas) \Rightarrow (Suc r, (r, a) # aas))

lemma annotate-rlen-len: fst (annotate-rlen-code r) = length r
  
proof

lemma annotate-rlen-code[code]: annotate-rlen s = snd (annotate-rlen-code s)
lemma suc2plus-inj-on: inj-on (of-nat :: nat ⇒ ('l :: len) word) {0..unat (max-word :: 'l word)}
(proof)

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)

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)
(proof)

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]
(proof)

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)]))
(proof)

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)

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 (λm. oiface (match-sel m) = ifaceAny)

14
definition \( \text{lr-of-tran } rt \text{ fw ifs} \equiv \)
\( \) if \( \neg \) (no-oif-match fw \( \land \) has-default-policy fw \( \land \) simple-fw-valid fw \( \land \) valid-prefixes rt \( \land \) has-default-route rt \( \land \) distinct ifs)
\( \) then Inl "Error in creating OpenFlow table: prerequisites not satisfied"
\( \) else (let 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 (max-word :: 16 word)
then Inr (pack-OF-entries ifs ard)
else Inl "Error in creating OpenFlow table: priority number space exhausted")

definition is-iface-name i \equiv i \neq [] \land \neg \text{Iface}. if-name-is-wildcard i

definition is-iface-list ifs \equiv distinct ifs \land list-all is-iface-name ifs

lemma max-16-word-max[simp]: (a :: 16 word) \leq 0xffff
(proof)

lemma replicate-FT-hlp: x \leq 16 \land y \leq 16 \implies replicate (16 - x) False @ replicate x True = replicate (16 - y) False @ replicate y True \implies x = y
(proof)

lemma mask-inj-hlp1: inj-on (mask :: nat \Rightarrow 16 word) {0..16}
(proof)

lemma distinct-simple-match-to-of-match-portlist-hlp:
\( \) fixes ps :: (16 word \times 16 word)
\( \) shows distinct ifs \implies
distinct
\( \) (if fst ps = 0 \land snd ps = max-word then [None]
\( \) else if fst ps \leq snd ps
\( \) then map (Some \circ (\lambda pfz. (pfzm-prefix pfz. \sim pfzm-mask pfz)))
\( \) (wordinterval-CIDR-split-prefixmatch (WordInterval (fst ps) (snd ps)))
\( \) else [])
(proof)

lemma distinct-simple-match-to-of-match: distinct ifs \implies distinct (simple-match-to-of-match m ifs)
(proof)

lemma inj-inj-on: inj F \implies inj-on F A (proof)

lemma no-overlaps-lroft-hlp2: distinct (map fst amr) \implies (\forall r. distinct (fm r)) \implies
\( \) distinct (concat (map (\lambda (p, r, c, a). map (\lambdab. (p, b, fs a c)) (fm r)) amr))
(proof)

lemma distinct-lroft-s3: [distinct (map fst amr); distinct ifs] \implies distinct (lr-of-tran-s3 ifs amr)
(proof)

lemma no-overlaps-lroft-hlp3: distinct (map fst amr) \implies
\( \) (aa, ab, ac) \in set (lr-of-tran-s3 ifs amr) \implies (ba, bb, bc) \in set (lr-of-tran-s3 ifs amr) \implies
\( \) ac \neq bc \implies aa \neq ba
(proof)
lemma no-overlaps-lroft-s3-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)

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))
(proof)

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
(proof)

lemma sorted-lr-of-tran-s3-hlp: ∀ x∈set f. fst x ≤ a ⇒ b ∈ set (lr-of-tran-s3 s f) ⇒ fst b ≤ a
(proof)

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
(proof)

lemma sorted-lr-of-tran-s3: sorted-descending (map fst f) ⇒ sorted-descending (map fst (lr-of-tran-s3 s f))
(proof)

lemma sorted-lr-of-tran-hlp: (ofe-prio o split3 OFEntry) = fst (proof)

lemma lr-of-tran-sorted-descending: Inr r = lr-of-tran rt fw ifs ⇒ sorted-descending (map ofe-prio r)
(proof)

lemma lr-of-tran-s1-split: lr-of-tran-s1 (a ≠ rt) = (route2match a, output-iface (routing-action a)) ≠ lr-of-tran-s1 rt
(proof)

lemma route2match-correct: valid-prefix (routing-match a) ⇒ prefix-match-semantics (routing-match a) (p-dst p) ⇔
simple-matches (route2match a) (p)
(proof)

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))
(proof)

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 ⇒ ∃ one. one ∈ set oms ∧ γ (ofe-fields ome) p
(proof)

lemma s3-noaction-hlp: [simple-match-valid ac; ¬simple-matches ac p; match-face (oiface ac) (p-oiface p)] ⇒
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
(proof)

lemma s3-correct:
assumes vsfwm: list-all simple-match-valid (map (fst ∘ snd) ard)
assumes ip pkt: p-l2type p = 0x800
assumes iiifs: p-iiface p ∈ set ifs
assumes oiifs: list-all (λm. 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)]))

⟨proof⟩

context
notes valid-prefix-00 [simp, intro!]
begin
lemma lr-of-tran-s1-valid: valid-prefixes rt =⇒ gsfw-valid (lr-of-tran-s1 rt)
⟨proof⟩
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⟩

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⟩

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
⟨proof⟩

lemma OF-unsafe-safe-match3-eq:
list-all (all-prerequisites ∘ ofe-fields) oft =⇒
⟨proof⟩

lemma OF-unsafe-safe-match-linear-eq:
list-all (all-prerequisites ∘ ofe-fields) oft =⇒
⟨proof⟩

lemma simple-action-ne [simp]:
b ≠ simple-action. Accept ⦿ b = simple-action. Drop
b ≠ simple-action. Drop ⦿ b = simple-action. Accept
⟨proof⟩

lemma map-snd-apfst: map snd (map (apfst x) l) = map snd l
⟨proof⟩

lemma match-ifaceAny-eq: oiface m = ifaceAny =⇒ simple-matches m p = simple-matches m (p[p-oiface := any])
⟨proof⟩

lemma no-oif-matchD: no-oif-match fw =⇒ simple-fw fw p = simple-fw fw (p[p-oiface := any])
⟨proof⟩

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

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 \exists r. generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Accept)
(proof)

lemma lr-of-tran-fbs-dropD:
  assumes s1: valid-prefixes rt has-default-route rt
  assumes s2: no-oif-match fw
  shows \exists r oif. generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Drop)
(proof)

lemma lr-of-tran-fbs-dropI:
  assumes s1: valid-prefixes rt has-default-route rt
  assumes s2: no-oif-match fw
  shows \exists r oif. generalized-sfw (lr-of-tran-fbs rt fw ifs) p = Some (r, oif, simple-action.Drop)
(proof)

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

lemma lr-of-tran-correct:
  fixes p :: (32, 'a) simple-packet-ext-scheme
  assumes nerr: lr-of-tran rt fw ifs = Inr oft
  and ipkt: p-l2type p = 0x800
  and jf: p-oiface p \in set ifs
  shows OF-priority-match OF-match-fields-safe oft p = Action [Forward oif] \iff
    simple-linux-router-nol12 rt fw p = (Some (p\{p-oiface := oif\}))
  OF-priority-match OF-match-fields-safe oft p = Action [] \iff
    simple-linux-router-nol12 rt fw p = None
  OF-priority-match OF-match-fields-safe oft p \neq NoAction
  OF-priority-match OF-match-fields-safe oft p \neq Undefined
  OF-priority-match OF-match-fields-safe oft p = Action ls \iff length ls \leq 1
  \exists ls. length ls = 1 \land OF-priority-match OF-match-fields-safe oft p = Action ls
(proof)

end

theory OF-conv-test

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

18
begin

**parse-iptables-save** `SQRL-fw=iptables–save`

**term** `SQRL-fw`

**thm** `SQRL-fw-def`

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

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

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

**lemma** map (quote-rewrite \circ common-primitive-rule-toString) unfolded =

\[
\begin{align*}
&\text{"\"p icmp \textendash} j ACCEPT", \\
&\text{"\"i s1 \textendash} lan \textendash} p tcp \textendash} m tcp \textendash} spts \{1024:65535\} \textendash} m tcp \textendash} dpts \{80\} \textendash} j ACCEPT", \\
&\text{"\"i s1 \textendash} wan \textendash} p tcp \textendash} m tcp \textendash} spts \{80\} \textendash} m tcp \textendash} dpts \{1024:65535\} \textendash} j ACCEPT", \\
&\text{"\"j DROP""} \langle proof \rangle
\end{align*}
\]

**lemma** length unfolded = 4 \langle proof \rangle

**value** `[code] map (quote-rewrite \circ common-primitive-rule-toString) (upper-closure unfolded)`

**lemma** length (upper-closure unfolded) = 4 \langle proof \rangle

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

**lemma** length (lower-closure unfolded) = 4 \langle proof \rangle

**lemma** check-simple-fw-preconditions (upper-closure unfolded) = True \langle proof \rangle

**lemma** \(\forall m \in \text{get-match'set} (upper-closure (packet-assume-new unfolded)). \text{normalized-nnf-match} m \langle proof \rangle\)

**lemma** \(\forall m \in \text{get-match'set} (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded))). \text{normalized-nnf-match} m \langle proof \rangle\)

**lemma** check-simple-fw-preconditions (upper-closure (optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded)))) \langle proof \rangle

**lemma** length (to-simple-firewall (upper-closure (packet-assume-new unfolded))) = 4 \langle proof \rangle

**lemma** (lower-closure (optimize-matches abstract-for-simple-firewall (lower-closure (packet-assume-new unfolded)))) = lower-closure unfolded

\[
\text{lower-closure unfolded} = \text{upper-closure unfolded}
\]

\[
\text{lower-closure} (\text{optimize-matches abstract-for-simple-firewall (upper-closure (packet-assume-new unfolded)))} = \text{upper-closure unfolded} \langle proof \rangle
\]

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

**definition** `SQRL-fw-simple \equiv \text{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 \langle proof \rangle
parse-ip-route SQRL-rtbl-main = ip-route
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\})\} \} \} (proof)
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]
\} \} (proof)
definition SQRL-rtbl-main-sorted ≡ rev \{sort-key (λr. pfzmn-length (routing-match r))\} SQRL-rtbl-main
value SQRL-rtbl-main-sorted
definition SQRL-ifs ≡ |
\{iface-name = "s1-lan", iface-mac = 0x10001\},
\{iface-name = "s1-wan", iface-mac = 0x10002\}
\} \} value SQRL-ifs
definition SQRL-macs ≡ |
\{(*"s1-lan", (ipv4addr-of-dotdecimal (10,0,1,1), 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), 0xead0152059)),\}*
\} \} definition SQRL-macs
definition SQRL-ports ≡ |
\{("s1-lan", "1"),\},
\{("s1-wan", "2"),\}
\} \} definition SQRL-ports
definition ofi ≡ case (\{br-of-tran SQRL-rtbl-main-sorted\} SQRL-fw-simple \{map iface-name SQRL-ifs\}) of \{Inr openflow-rules\} ⇒ map (serialize-of-entry \{the o map-of SQRL-ports\\}) openflow-rules
lemma ofi =
\{"priority=11,hard-timeout=0,idle-timeout=0,dl-type=0x8060,nw-proto=1,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=10.0.2.0/24,nw-dst=10.0.0.0/8,action=output:2",\}"priority=9.0.0.0/8,action=output:2",\}"priority=8.0.0.0/8,action=output:2",\}"priority=7.0.0.0/8,action=output:2",\}
lemma let fw = SQRL-fw-simple in no-oif-match fw ∨ has-default-policy fw ∨ simple-fu-valid fw (proof)
lemma let rt = SQRL-rtbl-main-sorted in valid-prefixes rt ∨ has-default-route rt (proof)
lemma let ifs = (map iface-name SQRL-ifs) in distinct ifs (proof)
lemma let ifs = (map iface-name SQRL-ifs) in distinct ifs (proof)
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 o common-primitive-rule-toString) rs)) SQRL-fw

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

lemma length unfolded = 26 ⟨proof⟩

value[code] unfolded
value[code] (upper-closure unfolded)
value[code] map (quote-rewrite o common-primitive-rule-toString) (upper-closure unfolded)
lemma length (upper-closure unfolded) = 26 ⟨proof⟩

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

lemma length (lower-closure unfolded) = 26 ⟨proof⟩

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

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 (lower-closure (packet-assume-new unfolded)))) = upper-closure unfolded ⟨proof⟩

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-fire-rule SQRL-fw-simple ⟨proof⟩

parse-ip-route SQRL-rtbl-main = ip-route
value SQRL-rtbl-main
lemma SQRL-rtbl-main = [(routing-match = PrefixMatch 0xC6130100 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]])] ⟨proof⟩
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 ] ⟨proof⟩
definition SQRL-ports ≡ |
  ("ip1", "1")|
  ("opt", "2") |
|
definition of ≡ |
  case (lr-of-tran SQRL-rtbl-main SQRL-fu-simple (map fst SQRL-ports)) |
  of (lrn openflow-rules) ⇒ map (serialize-of-entry (the o map-of SQRL-ports)) openflow-rules |
lemma of = |
  "priority=27,hard-timeout=0,idle-timeout=0,dl-type=0x800,nw-dst=192.18.1.0/24,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=192.19.1.1/32,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"
(\text{proof})
value[code] length of
Part II
Documentation

1 Configuration Translation

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

theory OpenFlow-Documentation

1.1 Linux Firewall Model

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

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

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

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

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

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.

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 \textit{a priori}. A test-wise implementation of the translation based on this model showed acceptable results. However, we deemed the \textit{a priori} lookup of the MAC addresses to be rather inelegant and built a second model.

**Definition** \texttt{simple-linux-router-altered rt fw ifl p} \equiv do

\begin{align*}
\text{let rd} &= \text{routing-table-semantics rt (p-dst p)}; \\
\text{let p} &= \text{p(p-oiface := output-iface rd)}; \\
\text{-} & \text{if p-oiface p = p-iface p then None else Some ()}; \\
\text{let fd} &= \text{simple-fw fw p}; \\
\text{-} & \text{(case fd of Decision FinalAllow \Rightarrow Some () | Decision FinalDeny \Rightarrow None)}; \\
\text{Some p}
\end{align*}

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) if 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.

While a test-wise implementation based on this model also showed acceptable results, the model is still problematic. The check \texttt{p-oiface p = p-iface 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 device cannot have a field for the output interface. We thus simplified the model even further:

**Lemma** \texttt{simple-linux-router-nol12 rt fw pi f p} \equiv do

\begin{align*}
\text{let rd} &= \text{routing-table-semantics rt (p-dst p)}; \\
\text{let p} &= \text{p(p-oiface := output-iface rd)}; \\
\text{-} & \text{(case fd of Decision FinalAllow \Rightarrow Some () | Decision FinalDeny \Rightarrow None)}; \\
\text{Some p}
\end{align*}

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**

\begin{align*}
\text{iface-packet-check ifl pi f p} \neq \text{None}; \\
\text{mlf (case next-hop (routing-table-semantics rt (p-dst p)) of None \Rightarrow p-dst p | Some a \Rightarrow a) \neq \text{None}} \Rightarrow \exists x. \text{map-option (λp. f(p-l2dst := x))} \\
\text{(simple-linux-router-nol12 rt fw pi f p) = simple-linux-router rt fw mlf ifl pi f p (proof)}
\end{align*}

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 \texttt{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.

\begin{align*}
\text{schematic-goal} \ (\text{?rtbl-entry :: (a::len) routing-rule}) = (\text{rtbl-match} = \text{PrefixMatch pfx len, metric = met, routing-action} = (\text{output-iface} = \text{of-str} \text{ing}, next-hop = (h :: (a word option))}) \ (proof)
\end{align*}

A routing table is then a list of these entries:

**Lemma** \texttt{rtbl :: (a :: len) prefix-routing} = \texttt{(rtbl :: a routing-rule list) (proof)}

Not all members of the type \texttt{prefix-routing} are same 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:

\[
\text{lemma} \text{ valid-prefix} (\text{PrefixMatch } pfx \text{ len}) \equiv pfx \&\& (2^{(32-\text{len})} - 1) = (0 :: 32 \text{ word})
\]

\[
\text{proof}
\]

\[
\text{lemma} \text{ has-default-route } rt \leftrightarrow (\exists r \in \text{ set } rt. \text{ pfzm-length } (\text{routing-match } r) = 0)
\]

\[
\text{proof}
\]

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.

\[
\text{schematic-goal} \text{ routing-table-semantics } (rt-entry \neq rt) \text{ dst-addr} = (\text{if } \text{prefix-match-semantics } (\text{routing-match } rt-entry) \text{ dst-addr then } \text{routing-action } rt-entry \text{ else } \text{routing-table-semantics } rt \text{ dst-addr} ) \text{ (proof)}
\]

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

1.1.2 iptables Firewall

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

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

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

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

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

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

1.2 OpenFlow Switch Model

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

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

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

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

- A switch consists of a single flow table.
- A flow table entry consists of a priority, a match condition and an action list.
- The only possible action (we require) is to forward the packet on a port.
We do not model controller interaction.

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

1.2.1 Matching Flow Table entries

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

\[
\text{schematic-goal} \quad \text{field-match} \quad \text{of-match-field} \in \{ \\
\text{IngressPort} \quad (\text{?s::string}), \\
\text{EtherSrc} \quad (\text{?as::48 word}), \text{EtherDst} \quad (\text{?ad::48 word}), \\
\text{EtherType} \quad (\text{?t::16 word}), \\
\text{VlanId} \quad (\text{?v::16 word}), \text{VlanPriority} \quad (\text{?p::16 word}), \\
\text{IPV4Src} \quad (\text{?ps::32 prefix-match}), \text{IPV4Dst} \quad (\text{?pms::32 prefix-match}), \\
\text{IPV4Proto} \quad (\text{?p::8 word}), \\
\text{L4Src} \quad (\text{?ps::16 word}), \text{L4Dst} \quad (\text{?ps::16 word}) \} \quad \text{⟨proof⟩}
\]

Two things are worth additional mention: L3 and L4 “addressess”. The \text{IPV4Src} and \text{IPV4Dst} matches are specified as “can be subnet masked” in [2], whereas [3] states clearly that arbitrary bitmasks can be used. We took the conservative approach here. Our alteration of \text{L4Src} and \text{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 \text{of-match-field} is not enough to classify a packet. To match packets, we thus use entire sets of match fields. As Guha et al. [7] noted, executing a set of given \text{of-match-fields} on a packet requires careful consideration. For example, it is not meaningful to use \text{IPV4Dst} if the given packet is not actually an IP packet, i.e. \text{IPV4Dst} has the prerequisite of \text{EtherType 0x800} being among the match fields. Guha et al. decided to use the fact that the preconditions can be arranged on a directed acyclic graph (or rather: an acyclic forest). They evaluated match conditions in a manner following that graph: first, all field matches without preconditions are evaluated. Upon evaluating a field match (e.g., \text{EtherType 0x800}), the matches that had their pre-condition fulfilled by it (e.g., \text{IPV4Src} and \text{IPV4Src} in this example) are evaluated. This mirrors the faulty behavior of some implementations (see [7]). Adopting that behavior into our model would mean that any packet matches against the field match set \{ \text{IPV4Dst (PrefixMatch 0x800808 32)} \} instead of just those destined for 8.8.8.8 or causing an error. We found this to be unsatisfactory.

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

\begin{equation}
\text{lemma} \quad \text{prerequisites} \quad \text{VlanPriority pri} \quad m = (\exists \text{id}. \text{let v = VlanId id in v \in m} \land \text{prerequisites} \text{ v m}) \\
\text{prerequisites} \quad \text{IPV4Proto} \text{ prot} \quad m = (\text{let v = EtherType 0x0800 in v \in m} \land \text{prerequisites} \text{ v m}) \\
\text{prerequisites} \quad \text{IPV4Src a} \quad m = (\text{let v = EtherType 0x0800 in v \in m} \land \text{prerequisites} \text{ v m}) \\
\text{prerequisites} \quad \text{IPV4Dst a} \quad m = (\text{let v = EtherType 0x0800 in v \in m} \land \text{prerequisites} \text{ v m}) \\
\text{prerequisites} \quad \text{L4Src msk} \quad m = (\exists \text{proto} \in \{\text{TCP,UDP,L4-Protocol,SCTP}\}. \text{let v = IPV4Proto proto in v \in m} \land \text{prerequisites} \text{ v m}) \\
\text{prerequisites} \quad \text{L4Dst msk} \quad m = \text{prerequisites} \quad \text{L4Src undefined undefined} \quad m
\end{equation}

\begin{equation}
\text{proof}
\end{equation}

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

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

\begin{equation}
\text{proof}
\end{equation}

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 too much entanglement with the previous section, we assume an arbitrary match function \( \gamma \). This function \( \gamma \)
takes the match condition \( m \) from a flow entry \( \text{OFEntry} \) and decides whether a packet matches those.

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

Packets are matched against flow entries based on prioritization. An entry that specifies an exact match (i.e., has no wildcards) is always the highest priority\(^4\). All wildcard entries have a priority associated with them. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering.

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

\[ \text{lemma} \quad \gamma \big( \text{ofe-fields \( fe \)} \big) \; p = \text{True} \implies \forall \, \text{\( fe \)} \in \text{set \( \{ ft1 \; @ \; ft2 \} \)}, \; \text{ofe-prio \( fe \)} > \text{ofe-prio \( fe \)} \implies \gamma \big( \text{ofe-fields \( fe \)} \big) \; p = \text{False} \implies \gamma \big( \text{guha-table-semantics} \big) \; \big( \text{ft1 } @ \text{ ft2} \big) \; p \big( \text{Some} \big( \text{ofe-action \( fe \)} \big) \big)
\]

\[ \forall \, \text{\( fe \)} \in \text{set \( ft \)}, \; \gamma \big( \text{ofe-fields \( fe \)} \big) \; p = \text{False} \implies \gamma \big( \text{guha-table-semantics} \big) \; \text{ft} \; p \; \text{None} \quad \langle \text{proof} \rangle
\]

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

\[ \text{lemma} \quad \text{CARD('action')} \geq 2 \implies \exists \; \text{\( ff \)}, \; \gamma \; \text{\( ff \)} \; p \implies \exists \; \text{\( ft \) \( (a1 :: \text{ ofe-action}) \)}, \; \text{\( a1 \neq a2 \; \& \; \text{guha-table-semantics} \); \( \gamma \; \text{ft} \; p \big( \text{Some} \big( \text{a1} \big) \big) \; \& \; \text{guha-table-semantics} \); \( \gamma \; \text{ft} \; p \big( \text{Some} \big( \text{a2} \big) \big) \rangle \quad \langle \text{proof} \rangle
\]

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

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

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

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

\[ \text{lemma} \quad \text{OF-priority-match} \; \gamma \; \text{flow-entries} \quad \text{packet} = \{
\]
\[ \quad \text{let} \; m = \text{filter} \big( \lambda \; \text{\( fe \)}, \; \gamma \; \text{\( fe \)} \; p \big) \; \text{flow-entries};
\]
\[ \quad m' = \text{filter} \big( \lambda \; \text{\( fo \)} \in \text{set \( m \)}, \; \text{ofe-prio \( fo \)} \leq \text{ofe-prio \( f \)} \big)
\]
\[ \quad \text{m in}
\]
\[ \quad \text{case} \; m' \; \text{of}
\]
\[ \quad 
\]
\[ \quad | \; | \; \Rightarrow \; \text{NoAction}
\]
\[ \quad | \; s \; \Rightarrow \; \text{Action} \; (\text{ofe-action} \; s)
\]
\[ \quad | \; - \; \Rightarrow \; \text{Undefined}
\]
\[ \langle \text{proof} \rangle
\]

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

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

The use of \( \text{Undefined} \) immediately raises the question in which condition it cannot occur. We give the following definition:

\[^{4}\text{This behavior has been deprecated.}\]

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

---

\[^{6}\text{Note that the order of the flow table entries is irrelevant.}\]

We could have made this definition on sets but chose not to for consistency.
lemma check-no-overlap \(\forall a \in \text{set ft}. \forall b \in \text{set ft}. (a \neq b \land \text{ofe-prio } a = \text{ofe-prio } b) \rightarrow \neg(\exists p. \gamma (\text{ofe-fields } a) p \land \gamma (\text{ofe-fields } b) p)\) (proof)

Together with distinctness of the flow table, this provides the absence of \texttt{Undefined}:

lemma [check-no-overlap \(\gamma \text{ ft}\); distinct ft] \(\rightarrow \) OF-priority-match \(\gamma \text{ ft } p \neq \text{Undefined} \) (proof)

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 \(\gamma \text{ ft}\); distinct ft] \(\rightarrow \) OF-priority-match \(\gamma \text{ ft } p = \text{option-to-ftb } d \leftrightarrow \text{guha-table-semantics } \gamma \text{ ft } p \text{ d} \) (proof)

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

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

lemma
\[
\text{OF-match-linear } \gamma \[] p = \text{NoAction} \\
\text{OF-match-linear } \gamma \{a\#as\} p = (\text{if } \gamma (\text{ofe-fields } a) p \text{ then Action (ofe-action } a) \text{ else OF-match-linear } \gamma \text{ p } d)
\]

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

lemma [no-overlaps \(f\); sorted-descending (map ofe-prio f)] \(\rightarrow \) OF-match-linear \(\gamma \text{ f p } = \text{OF-priority-match } \gamma \text{ f p}\) (proof)

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 \texttt{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 \texttt{simple-fw} that allows arbitrary actions instead of just accept and drop.
- A function that computes the conjunction of two \texttt{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 \texttt{simple-fw}. Instead of only allowing \texttt{Accept} or \texttt{Drop} as actions, it allows arbitrary actions. The type of the function that evaluates this generalized simple firewall is \texttt{generalized-sfw}. The definition is straightforward:

lemma
\[
\text{generalized-sfw } \[] p = \text{None} \\
\text{generalized-sfw } (a \# as) p = (\text{if } \text{case } a \text{ of } (m,\text{-}) \Rightarrow \text{simple-matches } m p \text{ then Some } a \text{ else generalized-sfw } as p)
\]

Based on that, we asked: if \texttt{fw}1 makes the decision \texttt{a} (where \texttt{a} is the second element of the result tuple from \texttt{generalized-sfw}) and \texttt{fw}2 makes the decision \texttt{b}, how can we compute the firewall that makes the decision \(a, b\)? One possible answer is given by the following definition:

lemma \text{generalized-fw-join } l1 l2 \equiv [(u,a,b), (m1,a) \leftarrow l1, (m2,b) \leftarrow l2, u \leftarrow (\text{case } \text{simple-match-and } m1 m2 \text{ of } \text{None} \Rightarrow [\[] \text{ Some } s \Rightarrow [s]])

\footnote{Note that tuples are right-associative in Isabelle/HOL, i.e., \((a, b, c)\) is a pair of \(a\) and the pair \((b, c)\).}
This definition validates the following lemma:

**Lemma** generalized-sfw (generalized-fw-join \(fw_1, fw_2\)) \(p\) = Some \((u, d_1, d_2) \iff (\exists r_1, r_2, generalized-sfw fw_1 p = Some (r_1, d_1) \land generalized-sfw fw_2 p = Some (r_2, d_2) \land Some u = \text{simple-match-and } r_1, r_2)\)

(\(proof\))

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 \(a \land b\) \(\equiv\) (\(\text{if } a = \text{simple-action.} \text{Accept} \land b = \text{simple-action.} \text{Accept} \text{ then } \text{simple-action.} \text{Accept else } \text{simple-action.} \text{Drop}\))

**Definition** simple-rule-conj \(a \land b\) \(\equiv\) (\(\text{uncurry SimpleRule} \circ \text{apsnd} \text{ (uncurry simple-action-conj)}\))

**Theorem** simple-fw \(rs_1\) \(p\) = Decision FinalAllow \land simple-fw \(rs_2\) \(p\) = Decision FinalAllow \iff simple-fw (map simple-rule-conj (generalized-fw-join (map simple-rule-dtor \(rs_1\)) (map simple-rule-dtor \(rs_2\)))) \(p\) = Decision FinalAllow

(\(proof\))

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-rules into tuples of match and action. For \(rt\), we made a firewall ruleset with rules that use prefix matches on the destination IP address. The next step is to join the two rulesets. The result of the join is a ruleset with rules \(r\) that only match if both, the corresponding firewall rule \(fur\) and the corresponding routing rule \(rr\) matches. The data accompanying \(r\) is the port from \(rr\) and the firewall decision from \(fur\). Next, descending priorities are added to the rules using map (apfst word-of-nat) \(\circ\) annotate-rlen. If the number of rules is too large to fit into the \(2^{16}\) priority classes, an error is returned. Otherwise, the function pack-OF-entries is used to convert the \((16 \text{ word} \times 32 \text{ simple-match} \times \text{char list} \times \text{simple-action})\) list to an OpenFlow table. While converting the \((16 \text{ word} \times \text{simple-action})\) 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 \(r\)
\(p\)-iface \(p\) \(\in\) set \(ifs\)
match-iface (of iface \(r\)) (p-oiface \(p\))
\(p\)-l2type \(p\) = 0x800

**Shows**

simple-matches \(r\) \(p\) \(\iff\) \(\exists gr \in\) set (simple-match-to-of-match \(r\) \(ifs\)). OF-match-fields \(gr\) \(p\) = Some True

(\(proof\))

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

\(^9\)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 assign 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 satisfied"
else (let
  nfw = map simple-rule-dtor fw;
  frt = map (λ. (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 (max-word :: 16 word)
  then Inr (pack-OF-entries ifs ard)
  else Inl "Error in creating OpenFlow table: priority number space exhausted"
)
⟨proof⟩

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

- **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 offace 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 the flow table, but we have not verified that.) The last assumption is that the translation does not return a run-time error. The translation will return a run-time error if the rules can not be assigned priorities from a 16 bit integer, or when one of the following conditions on the input data is not satisfied:

lemma
¬ no-oif-match fw ∨
¬ has-default-policy fw ∨
¬ simple-fw-valid fw ∨
¬ valid-prefixes rt ∨
¬ has-default-route rt ∨
¬ distinct ifs ≡⇒
∃ err. lr-of-tran rt fw ifs = Inl err ⟨proof⟩
1.3.3 Comparison to Exodus

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

There are some fundamental differences between Exodus and our work:

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

2 Evaluation

In Section 1, we have made lots of definitions and created lots of models. How far these models are in accordance with the real world has been up to the vigilance of the reader. This section attempts to leaviate 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 fulfill 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. 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, \textit{tp\_src=1024/0xfc00}, \textit{tp\_src=2048/0xf800}, \ldots, \textit{tp\_src=32768/0x8000}\[12\] (or \textit{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 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 \textit{priority=7, icmp,} \textit{-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.

\[10\]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 \textit{-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.
Figure 4: Example Network 2

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\(^\text{12}\) to the required simplified firewall form. Additionally to the simplified firewall, we acquired the routing table (26 entries) from the same machine. We then evaluated the time necessary to complete the translation and the size of the resulting ruleset when using only the first \(n\) simple firewall rules and the full routing table. The result is shown in Figure 5. Given the time necessary to complete the conversion of the iptables firewall to a simple firewall, it is reasonable to say that the translation function is efficient enough.

\(^{12}\)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].
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 coincide, resulting in 416 (= 16 · 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 rule sets 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.

References


