

Verified Firewall Policy Transformations for Test Case Generation

Achim D. Brucker¹ Lukas Brügger²
Paul Kearney³ Burkhardt Wolff⁴

¹SAP Research, Germany

²Information Security, ETH Zürich, Switzerland

³Security Futures Practice, BT Innovate & Design, UK

⁴Université Paris-Sud, France

ICST 2010

- 1 Motivation
- 2 Background
- 3 Firewall Testing: the Direct Approach
 - Scenario
 - Model
 - Testing
- 4 Firewall Testing: the Optimized Approach
 - The Idea
 - The Method
 - Empirical Results
- 5 Conclusion

- 1 Motivation
- 2 Background
- 3 Firewall Testing: the Direct Approach
- 4 Firewall Testing: the Optimized Approach
- 5 Conclusion

Motivation

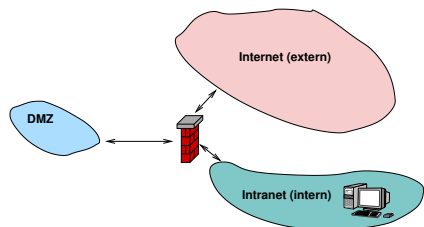
- Firewalls are cornerstones of security infrastructures
- Policies often change heavily over time
- Their configuration varies and is highly error-prone:

“NSA found that inappropriate or incorrect security configurations were responsible for 80 percent of Air Force vulnerabilities.”

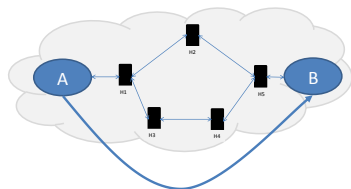
- **Our goal:** Test if a firewall configuration conforms to a specification

Motivation

Scenario 1: Single Firewall



Scenario 2: Networks



- **Our goal:** Test if several network component configurations conform to a specification

- 1 Motivation
- 2 Background**
- 3 Firewall Testing: the Direct Approach
- 4 Firewall Testing: the Optimized Approach
- 5 Conclusion

Model-based Testing with HOL-TestGen

```

126 definition
127   policy :: "(IntegerPort/i) Policy" where
128     "policy ≡ deny-all ++
129       Intranet-Internet ++
130       Intranet-mail ++
131       Internet-mail ++
132       Internet-web"
133
134 test-spec "not-in-same-net x ⟶ FUT x = policy x"
135 apply (prepare-fw-spec)
136 apply (simp add: PolicyLemmas policy-def not-in-same-net-def)
137 apply (gen-test-cases "FUT")
138 apply simp-all
139 store-test-thm "policy-test"
140 gen-test-data "policy-test"
141

```

Result Window

```

proof (prove): step 0
goal (1 subgoal):
1. not-in-same-net x ⟶ FUT x = policy x

```

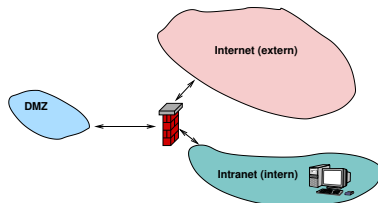
An **interactive**
model-based test tool

- built upon the theorem prover **Isabelle/HOL**
- generates test drivers
- successfully used in various case-studies
- freely available at:

<http://www.brucker.ch/projects/hol-testgen/>

- 1 Motivation
- 2 Background
- 3 Firewall Testing: the Direct Approach**
 - Scenario
 - Model
 - Testing
- 4 Firewall Testing: the Optimized Approach
- 5 Conclusion

A Typical Scenario



source	destination	protocol	port	action
Internet	dmz	smtp	25	allow
Internet	dmz	http	80	allow
dmz	intranet	smtp	25	allow
intranet	dmz	imaps	993	allow
intranet	Internet	http	80	allow
any	any	any	any	deny

- In this talk, firewalls are stateless packet filters
- HOL-TestGen can also handle stateful firewalls (not considered in this talk)

HOL-Model of a Firewall Policy

- A firewall makes a decision based on single packets.

types (α, β) packet = id \times ($\alpha::\text{adr}$) src \times ($\alpha::\text{adr}$) dest \times β content

Different address and content representations are possible.

- A policy either allows or denies a packet:

datatype α decision = allow α | deny α

- A policy is a mapping from packets to decisions:

types (α, β) Policy = (α, β) packet \rightarrow $((\alpha, \beta)$ packet) decision

- A library of policy combinators allows to define policies on a natural level:

definition

allow_all_from :: $(\alpha::\text{adr})$ net \Rightarrow (α, β) Policy **where**

allow_all_from src_net \equiv allow_all | ' {pa. src pa \sqsubset src_net}

The Policy

source	destination	protocol	port	action
Internet	dmz	smtp	25	allow
Internet	dmz	http	80	allow
dmz	intranet	smtp	25	allow
intranet	dmz	imaps	993	allow
intranet	Internet	http	80	allow
any	any	any	any	deny

definition Policy \equiv

deny_all ++

allow_port intranet internet 80 ++

allow_port intranet dmz 993 ++

allow_port dmz intranet 25 ++

allow_port internet dmz 80 ++

allow_port internet dmz 25

Testing Stateless Firewalls

- The test specification:

test_spec test: “ $P\ x \implies \text{FUT}\ x = \text{Policy}\ x$ ”

- FUT: Placeholder for *Firewall Under Test*
- P: a predicate specifying which kind of packets we are interested in.
E.g.: wellformed packets which cross some network boundary.
- Generates test data like:

$\text{FUT}\ (12, ((7,13,12,10),6), ((172,168,2,1),80), \text{content}) =$
 $\text{Some}(\text{deny}\ (12, ((7,13,12,10),6), ((172,168,2,1),80), \text{content}))$

Problems with the direct approach

- The direct approach **does not scale**:

	R1	R2	R3	R4
Networks	3	3	4	3
Rules	12	9	13	13
TC Generation Time (sec)	26382	187	59364	1388
Test Cases	1368	264	1544	470

Problems with the direct approach

- The direct approach **does not scale**:

	R1	R2	R3	R4
Networks	3	3	4	3
Rules	12	9	13	13
TC Generation Time (sec)	26382	187	59364	1388
Test Cases	1368	264	1544	470

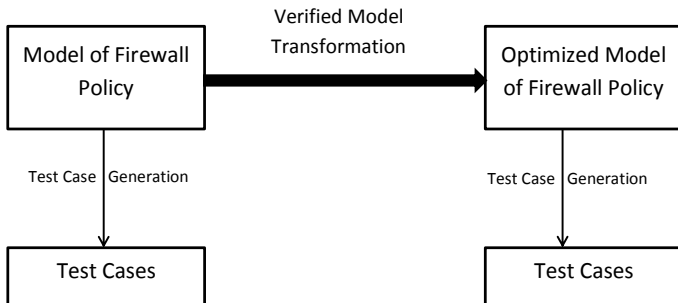
- **Reason:** Large cascades of case distinctions over input and output
- However, many of these case splits are redundant

- 1 Motivation
- 2 Background
- 3 Firewall Testing: the Direct Approach
- 4 Firewall Testing: the Optimized Approach**
 - The Idea
 - The Method
 - Empirical Results
- 5 Conclusion

Idea

- Input to test case generation is a **representation** of the model
- Test case generation depends on that specific representation
- Use a representation of (semantically equivalent) model, which is “easier” to test
- Idea is to remove redundant case-splits beforehand; they can be detected syntactically on a sequence of policy rules
- Make transformations between representations automatic and **verify** them in Isabelle

Model Transformations for Test Case Generation (1/2)



Model Transformations for Test Case Generation (2/2)

- Idea is fundamental to model-based test case generation. E.g.:

if $x < -10$ then if $x < 0$ then P else Q else Q

if $x < -10$ then P else Q

lead to different test cases

Model Transformations for Test Case Generation (2/2)

- Idea is fundamental to model-based test case generation. E.g.:

if $x < -10$ then if $x < 0$ then P else Q else Q

if $x < -10$ then P else Q

lead to different test cases

- Similarly, the following two policies produce a different set of test cases:

```
deny_all ++ deny_port dmz internet 21
++ allow_all_from_to dmz internet
```

```
deny_all ++ allow_all_from_to dmz internet
```

The Method

- Represent transformations as recursive function directly in HOL
- Provide only a fixed number of combinators

datatype (α, β) Combinators =

DenyAll

| DenyAllFromTo α α

| AllowPortFromTo α α β

| Conc $((\alpha, \beta)$ Combinators) $((\alpha, \beta)$ Combinators) (\oplus)

- and map them to the standard combinators:

fun C **where**

C DenyAll = deny_all

| C (DenyAllFromTo x y) = deny_all_from_to x y

| C (AllowPortFromTo x y p) = allow_port x y p

| C (x \oplus y) = C x ++ C y

A Typical Transformation

Remove all rules allowing a port between two networks, if a former rule already denies all the rules between these two networks.

fun removeShadowRules2::

where

```

removeShadowRules2 ((AllowPortFromTo x y p)#z) =
  if (DenyAllFromTo x y) mem z
  then removeShadowRules2 z
  else (AllowPortFromTo x y p)#(removeShadowRules2 z)
| removeShadowRules2 (x#y) = x#(removeShadowRules2 y)
| removeShadowRules2 [] = []

```

More Transformations

- Other transformations include:
 - ▶ Remove all the rules after a DenyAll
 - ▶ Sort the rules along the subnet hierarchy
 - ▶ Add additional rules (i.e. split a global rule into smaller ones)
 - ▶ Remove duplicate rules
 - ▶ Remove rules with an empty domain
 - ▶ Separate the policy into several policies
- Each of them is **proven formally** to keep the semantics under certain preconditions

Computing a Normal Form for Policy Models

- Transformations can be combined to compute a **normal form**
- The full normalization procedure consists of nine such transformations
- The result is a list of policies, in which:
 - ▶ each element completely specifies the behavior of some network segment
 - ▶ no element contains redundant rules
- Thus, the normalization does:
 - ▶ pre-partition the test space
 - ▶ remove redundancies

Correctness of the Normalization

- **Correctness** of the normalization must hold for arbitrary input policies, satisfying certain preconditions
- As HOL-TestGen is built upon the theorem prover Isabelle/HOL, we can **prove formally** the correctness of such normalizations: E.g.:

theorem C_eq_normalize:

assumes member DenyAll p

assumes allNetsDistinct p

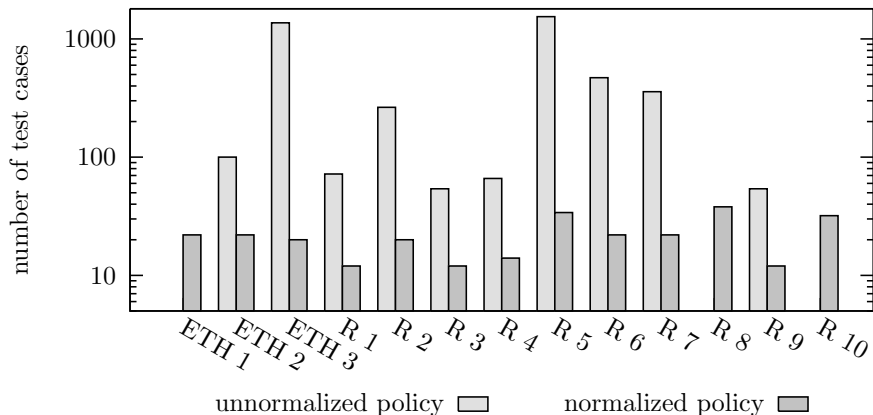
shows C (list2policy (normalize p)) = C p

Empirical Results

- The normalization of policies decreases the number of test cases and the required time by several orders of magnitude.

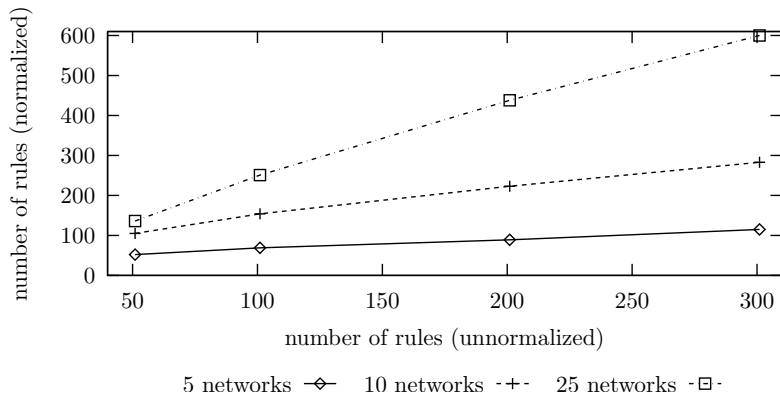
		R1	R2	R3	R4
Not Normalized	Networks	3	3	4	3
	Rules	12	9	13	13
	TC Generation Time (sec)	26382	187	59364	1388
	Test Cases	1368	264	1544	470
Normalized	Rules	14	14	24	26
	Normalization (sec)	0.6	0.4	1.1	0.8
	TC Generation Time (sec)	0.9	0.6	1.2	0.7
	Test Cases	20	20	34	22

Number of Test Cases



The normalization of policies decreases the number of test cases by several orders of magnitude.

Number of Rules



- The number of rules of a policy after normalization increases with both the number of rules in the unnormalized policy and the number of networks.
- It can be smaller or greater than before

- 1 Motivation
- 2 Background
- 3 Firewall Testing: the Direct Approach
- 4 Firewall Testing: the Optimized Approach
- 5 Conclusion**

Conclusion

- Nice scenario of an integrated approach for test and proof.
- Theorem-prover-based testing can increase the overall system security by ensuring that network devices implement the security policy correctly.
- Large gains in performance allows to process realistic policies.
- All transformations are **verified**.
- Normalization is domain-specific, could potentially be generalized.
- More investigations into the **quality** of test data required.

<http://www.brucker.ch/projects/hol-testgen/>