Model-based Conformance Testing of Security Properties

Achim D. Brucker and Lukas Brügger
joint work with Burkhart Wolff

Dagstuhl Seminar 13021
"Symbolic Methods in Testing"
`http://www.dagstuhl.de/13021`

06.01.2013 – 11.01.2013

Abstract

Modern systems need to comply to large and complex security policies that need to enforced at runtime. This runtime enforcement needs to happen on different levels, e.g., ranging from high level access control models to firewall rules.
We present an approach for the modular specification of security policies (e.g., access control policies, firewall policies). Based on this formal model, i.e, the specification, we discuss a model-based test case generation approach that can be used for both testing the correctness of the security infrastructure as well as the conformance of its configuration to a high-level security policy.

Outline

Motivation

Observation:

IT systems need to enforce complex security policies.

Question:

Are these rules correctly *enforced* at runtime?

Approach:

Conformance testing of

- runtime enforcement infrastructure (implementation) and
- security policy (configuration).

Security Policies

- Define rules according to which access must be regulated
- Come in many different flavors
  (RBAC, Bell-LaPadula, firewall policies)
- Complex implementation of policy-decision-points
  - Optimized for performance
  - Complex policy languages
- Configuration often hard to get right and maintain:
  - Large number of rules
  - A lot of changes over time
  - Configuration by different entities
  - Interaction with other policies and legacy systems

Conformance Testing of Security Policies

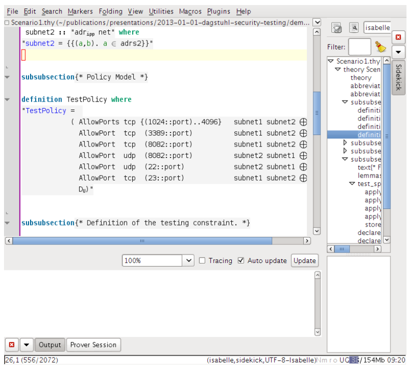
Validation that a range of diverse and partially unknown systems conform to a set of high-level security policies

- **Characteristics:** Specification-based black-box test
- **Coverage:** Security policy model
- **Scalability:** Security policies are large and complex

Components of HOL-TestGen

- **HOL (Higher-order Logic):**
  - "Functional Programming Language with Quantifiers"
  - plus definitional libraries on Sets, Lists, . . .
  - used as meta-language for Hoare Calculus for Java, Z, . . .
- **HOL-TestGen:**
  - based on the interactive theorem prover Isabelle/HOL
  - integrates formal proofs and test case generation
- **Interactive User Interface:**
  - user interface for Isabelle and HOL-TestGen
  - step-wise processing of specifications/theories
  - shows current proof states

Model-based Testing with HOL-TestGen



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/`

The HOL-TestGen Workflow

The HOL-TestGen workflow is basically four-fold:

1. *Step I:* writing a **test specification**
   Step I': analyzing or optimizing **test specification**
2. *Step II:* generating a **test theorem** (roughly: testcases)
3. *Step III:* generating **test data**
4. *Step IV:* generating a **test script**

And of course:

- building an executable test driver
- and running the test driver

Demo

A Simple Test Theory

```
theory List_test
imports Main begin
  fun    is_sorted:: "('a::ord) list  ⇒ bool"
  where " is_sorted []        = True"
         | is_sorted  (x#xs) = case xs of
                                   [] ⇒ True
                                 | y#ys ⇒(x <= y)
                                         ∧ is_sorted xs"

  test_spec "is_sorted (prog (l::('a list))"
    apply(gen_test_cases prog)
  store_test_thm "test_sorting"

  gen_test_data "test_sorting"
  gen_test_script "test_lists.sml" list" prog
end
```


Outline

The Unified Policy Framework (UPF)

- An extensible framework for policy modelling in Isabelle/HOL
- Main features:
  - Applicable to a wide range of different kinds of policies
  - Modular modelling approach (combination of subpolicies)
  - Geared towards use in test case generation
  - Large executable subset
  - Possibility to model higher-order policies
  - Integrated with modeling states and state transitions

UPF: Foundations

- Main concept:
  - Policies are modelled as partial policy decision functions
  - Formally: $\alpha \mapsto \beta = \alpha \rightharpoonup \beta$ decision
    where α decision = allow α | deny α
  - Input data α: users, operations, network packets, state
  - Output data β: return messages, state

  Principles:
  - Functional representation
  - No conflicts
  - Three-valued decision type
  - Open output type

UPF: Combining Rules and Policies

- Rules are defined by domain restrictions

  $$\{(\text{Alice}, \text{obj}_1, \text{read})\} \lhd A_U$$

  where $A_U = \lambda x.\ \lfloor \text{allow}() \rfloor$
- There are three categories of combination operators:
  - **Override** operators (e.g. first matching rule applies): $\_ \oplus \_$
  - **Parallel** combination operators: $\_ \otimes \_$
  - **Sequential** composition: $\_ O \_$
- A large number of algebraic properties hold over the operators: $(P_1 \oplus P_2) \otimes P_3 = (P_1 \otimes P_3) \oplus (P_2 \otimes P_3)$

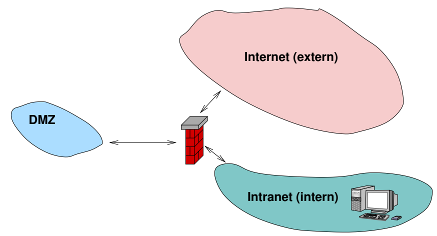
UPF: Transition Policies

- Systems that implement a security policy are often stateful
- State transitions can be modelled as partial functions
- Standard approach:
  - Model the pure policy P
  - Model the state transitions to be triggered for allow: A_{ST}
  - Model the state transitions to be triggered for deny: D_{ST}
  - Combine the three parts: $(A_{ST}, D_{ST}) \otimes_{\nabla} P$
  - To a transition policy of type: $(\iota \times \sigma) \to (\sigma \times \sigma)$

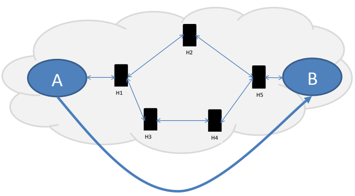
Outline

Motivation

Scenario 1: Single Firewall

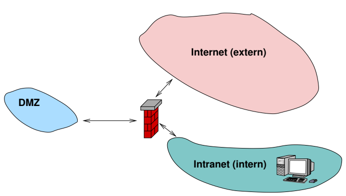


Scenario 2: Networks



- **Our goal:**
  Show correctness of
  network component configurations and implementations

A Typical Scenario



source	destination	protocol	port	action
Internet	dmz	udp	25	allow
Internet	dmz	tcp	80	allow
dmz	intranet	tcp	25	allow
intranet	dmz	tcp	993	allow
intranet	Internet	udp	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
  $= \text{id} \times (\alpha::\text{adr})\ \text{src} \times (\alpha::\text{adr})\ \text{dest} \times \beta \text{content}$

  Different address and content representations are possible.
- A policy is a mapping from packets to decisions:

  types (α,β) Policy $= (\alpha,\beta)$ packet $\mapsto$
  unit
- Policy combinators allow for defining policies:

  definition
  allow_all_from :: $(\alpha::\text{adr})$ net $\Rightarrow (\alpha,\beta)$ Policy **where**
  allow_all_from src_net = {pa. src pa $\sqsubset$ src_net} $\triangleleft$allow_all

Network Address Translation (NAT)

- Firewalls often perform network address translation
- Input to the policies remains a network packet
- Output additionally contains a description of admissible transformed packets:

$$(\alpha, \beta)\,\text{packet} \mapsto ((\alpha, \beta)\,\text{packet})\,\text{set}$$

- NAT policies are combined in parallel with stateless packet filtering policies

The Policy

source	destination	protocol	port	action
Internet	dmz	udp	25	allow
Internet	dmz	tcp	80	allow
dmz	intranet	tcp	25	allow
intranet	dmz	tcp	993	allow
intranet	Internet	udp	80	allow
any	any	any	any	deny

definition TestPolicy **where**
TestPolicy = allow_port udp 25 internet dmz $\oplus$
allow_port tcp 80 internet dmz $\oplus$
allow_port tcp 25 dmz intranet $\oplus$
allow_port tcp 993 intranet dmz $\oplus$
allow_port udp 80 intranet internet $\oplus$
D_U

Testing Stateless Firewalls

- The test specification:

  test_spec test: "P x $\Longrightarrow$ FUT x = Policy x"

- FUT: Placeholder for *Firewall Under Test*
- Predicate P restricts packets we are interested in, e.g., wellformed packets which cross some network boundary
- Generates test data like (simplified):

  FUT(1,((8,13,12,10),6,tcp),((172,168,2,1),80,tcp),data)
  = $\lfloor$(deny()$\rfloor$

Demo

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

Testing Firewalls | Firewall Testing: the Direct Approach

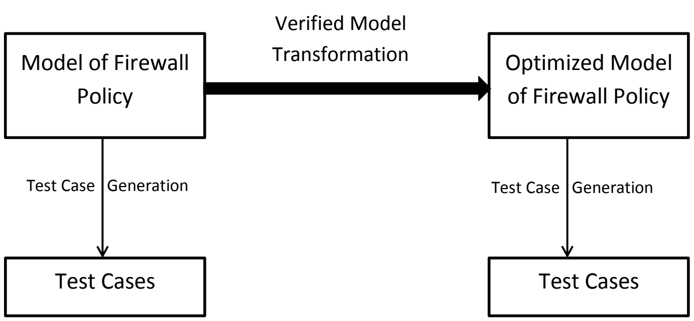
Problems with the direct approach

- 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
    $\Longrightarrow$ However, many of these case splits are redundant
  - Many combinations due to subnets
    $\Longrightarrow$ Pre-partitioning of test space according to subnets

Model Transformations for TCG (1/2)



Model Transformations for TCG (2/2)

- Idea is fundamental to model-based test case generation. E.g.:
  - $\text{if } x < -10 \text{ then if } x < 0 \text{ then } P \text{ else } Q \text{ else } Q$
  - $\text{if } x < -10 \text{ then } P \text{ else } Q$

  lead to different test cases

Model Transformations for TCG (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:
  - AllowAll dmz internet $\oplus$ DenyPort dmz internet 21 $\oplus$ D_U
  - AllowAll dmz internet $\oplus$ D_U

The Transformation

- Transformations are encoded as recursive function in HOL
- Provide only a fixed number of combinators

```
datatype (α,β) Combinators =
  DenyAll
| DenyAllFromTo αα
| AllowPortFromTo αα  β
| Conc ((α,β) Combinators) ((α,β) Combinators) (⊕)
```

- 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 ⊕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) ∈ (set 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 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:

theorem C_eq_normalize:
assumes member DenyAll p
assumes allNetsDistinct p
shows C (list2policy (normalize p)) = C p

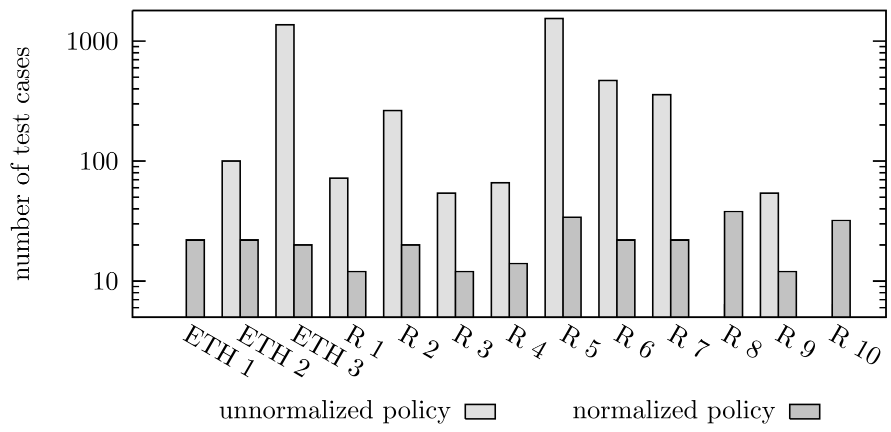
Demo

Empirical Results

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

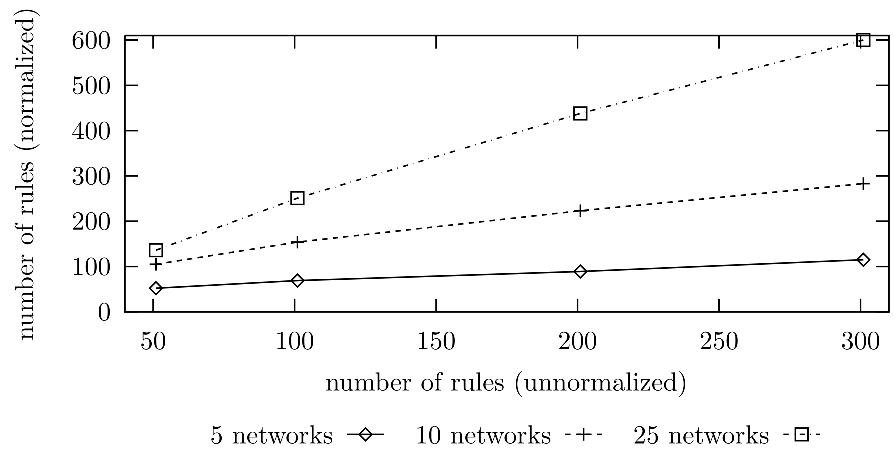
		R1	R2	R3	R4
Not Normalized	Networks	3	3	4	3
	Rules	12	9	13	13
	TC Generation Time (sec)	26382	187	59364	1388
	Test Cases	1368	264	1544	470
Normalized	Rules	14	14	24	26
	Normalization (sec)	0.6	0.4	1.1	0.8
	TC Generation Time (sec)	0.9	0.6	1.2	0.7
	Test Cases	20	20	34	22

Number of Test Cases



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

Number of Rules



Outline

NPfIT: Overview

- National Programme for IT (NPfIT) in the NHS
- Health care records of every patient (accessible over the network)
- Large number of applications that need to conform to Information Governance Principles (policy):
  - RBAC
  - Legitimate Relationships
  - Patient Consent
  - Sealed Envelopes

NPfIT: Lessons Learned

- We modeled large parts of the Information Governance Principles in UPF
  - different parts are modelled separate and using the UPF operators
  - Modelling system behaviour considerably more complex than the pure policy rules alone
- Testing requires choice of good test specification

Today's World is Distributed

Modern applications are built

- by composing (black-box) services
- are re-composing happens relatively often
- require complex security configurations

There are

- widely adopted standards (e. g., WSDL)
- powerful frameworks for building Web Services

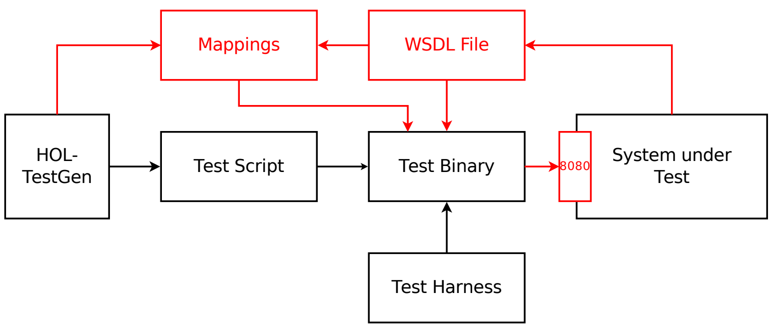
Idea:

- Let's try to apply HOL-TestGen in this scenario

Necessary steps:

- model Web Service Application API in HOL
- connect HOL-TestGen to a Web service Framework

WS Testing: Remote Setup



Provide support for the .net/mono framework:

- Add support for F# code generator to Isabelle (HOL-TestGen)
- Develop Test Harness in F#
- Use the WSDL toolchain for C# (F# not stable yet)

Outline

Conclusion

- Approach based on theorem proving
  - test specifications are written in HOL
  - functional programming, higher-order, pattern matching
- Verified Transformations of test-specifications
- Test hypothesis explicit and controllable by the user
- Proof-state explosion controllable by the user
- Verified tool inside a (well-known) theorem prover

Thank you
for your attention!

Any questions or remarks?

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

Please consider to submit a paper to
"Tests and Proofs" 2013
Deadline February, 1st

http://www.spacios.eu/TAP2013/

Bibliography I

Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff.
Verified firewall policy transformations for test-case generation.
In *Third International Conference on Software Testing, Verification, and Validation* (ICST), pages 345–354. IEEE Computer Society, 2010.

Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff.
An approach to modular and testable security models of real-world health-care applications.
In *ACM symposium on access control models and technologies (SACMAT)*, pages 133–142. ACM Press, 2011.

Lukas Brügger.
A Framework for Modelling and Testing of Security Policies.
PhD thesis, ETH Zurich, 2012.

Bibliography II

Achim D. Brucker and Burkhart Wolff.
Test-sequence generation with HOL-TestGen – with an application to firewall testing.
In Bertrand Meyer and Yuri Gurevich, editors, TAP *2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

Achim D. Brucker and Burkhart Wolff.
On theorem prover-based testing.
Formal Aspects of Computing (FAC), 2012.