Model-based Security Testing of a Health-Care System Architecture:

A Case Study

Achim Brucker (SAP), Lukas Brügger (ETH), Paul Kearney (BT) and Burkhart Wolff*

*Université Paris-Sud, Laboratoire de Recherche Informatique (LRI)
Abstract

We present a generic modular policy modelling framework and instantiate it with a substantial case study for model-based testing of some key security mechanisms of the NPfIT. NPfIT, “the National Program for IT” is a very large-scale development project aiming to modernise the IT infrastructure in the English health care system (NHS). Consisting of heterogeneous and distributed code, it is an ideal target for model-based testing techniques of a very large system exhibiting critical security features. We will model the four information governance principles, comprising a role-based access control model, as well as policy rules governing the concepts of patient consent, sealed envelopes and legitimate relationship. The model is given in higher-order logic (HOL) and processed together with suitable test-specifications in the HOL-TestGen system, that generates semi-automatically test sequences according to them.

Particular emphasis is put on the modular description of security policies and their generic combination and its consequences for model-based testing.
Overview

- NPfIT
- NPfIT formalized in UPF (formalized in HOL)
- System: HOL-TestGen
- First Results and Experiences
National Program for IT (NPfIT)

- Large Case-Study together with British Telecom
- Test-Goal: NHS patient record access control mechanism
- Large Distributed, Heterogeneous System
- Legally required Access Control Policy (practically mostly enforced on the application level)
Case-Study: NPfIT

• Challenges:

  • access control rules for patient-identifiable information are complex and reflect the trade-off between patient confidentiality, usability, functional, and legislative constraints.

  • Traditional discretionary and mandatory access control and RBAC are insufficiently expressive to capture complex policies such as Legitimate Relationships, Sealed Envelopes or Patient Consent Management.

  • access rules of such a large system comprise not only elementary rules of data-access, but also access to security policies themselves enabling policy management. The latter is conventionally modeled in ABAC [6–8] and administrative RBAC [9, 10] models; A uniform modelling framework must be able to accommodate this.

  • The requirements are mandated by laws, official guidelines and ethical positions (e. g. [11, 12]) that are prone to change.
Case-Study: NPfIT

• Different “Information Gouvernance Principles” (= Policies):

  • Role-Based Access Control (RBAC): NPfIT uses administrative RBAC [9] to control who can access what system functionality. Each user is assigned one or more User Role Profile (URP). Each URP permits the user to perform several Activities.

  • Legitimate Relationship (LR): A user is only allowed to access the data of patients in whose care he is actually involved. Users are assigned to hierarchically ordered workgroups that reflect the organisational structure of a workplace.

  • Patient Consent (PC): Patients can opt out in having a Summary Care Record (SCR) at all, or to control uploads of data into the SCR. This requires additional mechanisms to manage consent.

  • Sealed Envelope (SE): The sealing concept is used to hide parts of an SCR from users. Kinds of seals: seal, seal and lock, clinician seal.
Modeling Framework: Unified Policy Framework (UPF)

• UPF (A Theory in HOL / for HOL-TestGen)

  • A Policy: A Decision Function
    (Modeling a “Policy Enforcement Point” in a System)

```haskell
datatype α decision = allow α | deny α

types (α,β) policy = α ⊸ β decision    (* = α ⇒ β option *)

notation α ⇒ β = (α,β) policy
```
Modeling Framework: Unified Policy Framework (UPF)

• **UPF (A Theory in HOL / for HOL-TestGen)**

• Policy Constructors

\[
\text{definition } \emptyset \equiv \lambda y. \text{None} \quad (* \emptyset : \alpha \Rightarrow \beta *)
\]

\[
\text{definition } p(x \mapsto t) \equiv p(x \Rightarrow \text{Some(allow } t)) \quad (* p : \alpha \Rightarrow \beta *)
\]
\[
p(x \mapsto t) \equiv p(x \Rightarrow \text{Some(deny } t)) \quad (* \text{where } p(x \mapsto t) \equiv \\
\quad \lambda y. \text{if } y = x \text{ then } A \text{ else } p y *)
\]

\[
\text{definition } (*\text{AllowAll} :: "(\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)" *)
\]
\[
\forall A x. pf(x) \equiv (\lambda x. \text{case pf x of } \text{Some } y \Rightarrow \text{Some(allow } y)) \\quad (*\text{DenyAll} :: "(\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)"*)
\]
\[
\forall D x. pf(x) \equiv (\lambda x. \text{case pf x of } \text{Some } y \Rightarrow \text{Some(allow } y)) \quad l \text{ None } \Rightarrow \text{None}
\]
Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Domain, Range and Restrictions on Policies (Z-like)

  \[
  \text{definition } A \equiv \{x. \exists y. x = \text{allow } y\}, \quad D \equiv \{x. \exists y. x = \text{deny } y\}
  \]

  \[
  \text{definition } \text{dom} :: \alpha \rightarrow \beta \Rightarrow \alpha \text{ set}
  \]
  \[
  \text{where } \text{dom } f \equiv \{x. f x \neq \text{None}\}
  \]

  \[
  \text{definition } \text{ran} :: \alpha \rightarrow \beta \Rightarrow \beta \text{ set} \quad ...
  \]

  \[
  \text{definition } _{\triangleleft} _{\triangleleft} :: \alpha \text{ set } \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta
  \]
  \[
  \text{where } S \triangleleft p \equiv (\lambda x. \text{ if } x \in S \text{ then } p x \text{ else } \text{none}) \quad \text{ (* domain restriction *)}
  \]

  \[
  \text{definition } _{\triangleright} _{\triangleright} :: \alpha \rightarrow \beta \Rightarrow \alpha \text{ set } \Rightarrow \alpha \rightarrow \beta \quad \text{ (* range restriction *)}
  \]

  \[
  \text{definition } _{\oplus} _{\oplus} :: \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta \quad \text{(* first fit override *)}
  \]
Example: Firewalls

- Firewall Policies in UPF
  - Data:
    
    \[
    \text{ip-address} = \text{int} \times \text{int} \times \text{int} \times \text{int} \\
    \text{ip-packet} = \text{ip-address} \times \text{protocol} \times \text{content} \times \text{ip-address}
    \]
  
  - Firewall - Policies:
    
    \[
    \text{policy} : \text{ip-packet} \Rightarrow \text{ip-packet}
    \]

... this covers also Network Adress Translations (NAT's)
Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

\[
\text{definition} \quad \text{me-ftp} :: \text{ip-packet} \Rightarrow \text{ip-packet} \\
\text{where} \quad \text{me-ftp} \equiv \emptyset ((192,22,14,76),\text{ftp},d,(192,22,14,76)) \\
+\rightarrow (192,22,14,76),\text{ftp},d,(192,22,14,76))
\]
Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies
    
    \[
    \text{definition } \text{me-ftp} :: \text{ip-packet} \Rightarrow \text{ip-packet} \\
    \text{where } \text{me-ftp} \equiv \emptyset ((192,22,14,76),\text{ftp},\text{d},(192,22,14,76) \\
    +\rightarrow (192,22,14,76),\text{ftp},\text{d},(192,22,14,76))
    \]
  
  - Combined Policies:
    
    \[
    \text{definition } \text{me-none-else} :: \text{ip-packet} \Rightarrow \text{ip-packet} \\
    \text{where } \text{me-none-else} \equiv \text{me-ftp} \oplus \forall_D x. x
    \]
Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies
    
    definition me-ftp :: ip-packet ⇒ ip-packet
    where me-ftp = ∅ ((192,22,14,76),ftp,d,(192,22,14,76)
     +⇒(192,22,14,76),ftp,d,(192,22,14,76))
  
  - Combined Policies:
    
    definition me-none-else:: ip-packet ⇒ ip-packet
    where me-none-else = me-ftp ⊕ ∀D x. x
Example: RBAC

- **RBAC Policies in UPF**
  - Domain: $UR = \text{users} \times \text{role}$
    $RP = \text{role} \times \text{permission}$
  - 2-Policies:
    - UserTab :: $UR \Rightarrow \text{unit,}$
      PermTab:: $\text{permission} \Rightarrow \text{role} \Rightarrow \text{unit}$
Example: RBAC

- **RBAC Policies in UPF**
  
  - Domain: \( UR = \text{users} \times \text{role} \)
    
    \( RP = \text{role} \times \text{permission} \)
  
  - 2-Policies:
    
    \( \text{UserTab} :: UR \Rightarrow \text{unit} , \)
    
    \( \text{PermTab} :: \text{permission} \Rightarrow \text{role} \Rightarrow \text{unit} \)

```
datatype users = ...
datatype roles  = ...
datatype permissions  = ...

definition rbac ... RBAC (perm) = UserTab \ o_{VD} PermTab(perm)
```
Example: RBAC

- **RBAC Policies in UPF**
  - Domain: \( UR = \text{users} \times \text{role} \)
    \[ \text{RP} = \text{role} \times \text{permission} \]
  - 2-Policies:
    \[
    \begin{align*}
    \text{UserTab} :: UR & \Rightarrow \text{unit}, \\
    \text{PermTab} :: \text{permission} & \Rightarrow \text{role} \Rightarrow \text{unit}
    \end{align*}
    \]

    \[
    \begin{align*}
    \text{datatype users} & = \ldots \\
    \text{datatype roles} & = \ldots \\
    \text{datatype permissions} & = \ldots \\
    \text{definition rbac} & \ldots \text{RBAC (perm)} = \text{UserTab} \ o_{\text{VD}} \text{PermTab}(perm)
    \end{align*}
    \]
Example: RBAC

- **RBAC Policies in UPF**
  
  - Domain: \( UR = \text{users} \times \text{role} \)
    \[
    RP = \text{role} \times \text{permission}
    \]
  
  - 2-Policies:
    \[
    \text{UserTab} :: UR \Rightarrow \text{unit}, \\
    \text{PermTab} :: \text{permission} \Rightarrow \text{role} \Rightarrow \text{unit}
    \]

    ```
    \[
    \text{definition rbac} \ldots \text{RBAC} (\text{perm}) = \text{UserTab} \ o_{VD} \text{PermTab}(\text{perm})
    \]
    ```

    **where** \( o_{VD} \) is one of the 4 policy sequential compositions
More on UPF

• Transition Policies
  – Transition Policies: Policies involving state
    \[ \alpha \times \sigma \Rightarrow \beta \times \sigma \]  
    (input \( \alpha \), output \( \beta \))
  – Higher-order Policies (Policies transforming policies)
    \[ \alpha \times (\gamma \Rightarrow \delta) \Rightarrow \beta \times (\gamma \Rightarrow \delta) \]
  – Thus, ARBAC policies (policies describing who and how (1-order) policies may be modified) can be modelled in UPF
More on UPF

• Parallel Composition of Policies:
  
  – Idea: Considering policies as “transitions” in an automaton and putting them “in parallel” similar to automata composition.
  
  – Essentially 4 possibilities:

```plaintext
definition prod_orA :: "['α' → 'β', 'γ' → 'δ'] ⇒ ('α×'γ → 'β×'δ)" (_ )
where "p1 ⊗_A p2 = (λ(x,y). (case p1 x of
  Some(allow d1) ⇒ (case p2 y of
    Some(allow d2) ⇒ Some(allow(d1,d2))
    I Some(deny d2) ⇒ Some(allow(d1,d2))
    I None ⇒ None)
  I Some(deny d1) ⇒(case p2 y of
    Some(allow d2) ⇒ Some(allow(d1,d2))
    I Some(deny d2) ⇒ Some(deny (d1,d2))
    I None ⇒ None)
  I None ⇒ None))
```

Principal Use of UPF for NPfIT

- Parallel Composition of 4 Policies + Functional:

\[
(norm\_beh, excep\_beh) \land
(n legit im ate_ relation \otimes_{\lor} A
patients\_consent \otimes_{\lor} A
sealed\_envelopes \otimes_{\lor} A
rbac)
\]
NPfIT in UPF

- **Test - Specifications:**
  - Embedding of Transition Policies in State-Exception Monads:

    ```
    definition policy2MON :: (ι×σ ⇒ o×σ) ⇒ ι ⇒ σ ↦ (o ⇝ σ)
    where policy2MON p =
      (λ i σ. case p (i,σ) of
        Some(allow(o, σ')) ⇒ Some(allow o, σ')
        | Some(deny(o, σ')) ⇒ Some(deny o, σ')
        | None ⇒ None)
    ```
NPfIT in UPF

- Test - Specifications:
  - Embedding of Transition Policies in State-Exception Monads:

\[
\text{definition policy2MON :: } (\iota \times \sigma \Rightarrow o \times \sigma) \Rightarrow \iota \Rightarrow (o \text{ decision}, \sigma)\text{MON}_{SE}
\]

\[
\text{where policy2MON } p = \\
(\lambda \iota \sigma. \text{case } p (\iota, \sigma) \text{ of}
\]

- Some(allow(o, \sigma')) \Rightarrow Some(allow o, \sigma')
- I Some(deny(o, \sigma')) \Rightarrow Some(deny o, \sigma')
- I None \Rightarrow None)
Modeling Framework: Unified Policy Framework (UPF)

- **State-Exception Monads (f.Test-Sequences in HOL)**
  - **State-Exception Monads:**
    
    \[
    \text{type} \quad (o, \sigma) \text{MON}_\text{SE} = \sigma \rightarrow (o, \sigma)
    \]

    \[
    \text{definition} \quad \text{bind} :: (o, \sigma) \text{MON}_\text{SE} \Rightarrow (o \Rightarrow (o, \sigma) \text{MON}_\text{SE}) \Rightarrow (o, \sigma) \text{MON}_\text{SE} \quad \text{("_ ; _ \leftarrow ")}
    \]
    
    \[
    \text{where} \quad \ldots
    \]

    \[
    \text{definition} \quad \text{unit} :: (o \Rightarrow \text{bool}) \Rightarrow (o, \sigma) \text{MON}_\text{SE} \quad \text{("return _ ")}
    \]
    
    \[
    \text{where} \quad \ldots
    \]

- **Computation Sequences, Valid Computation Sequences, Valid mbind-Sequences, Valid mbind-Sequences with pre-condition:**
  
  PUT(i_1) ; o_1 \leftarrow PUT(i_2); \ldots ; \text{on} \leftarrow PUT(i_n) ; \text{result(post o}_1 \ldots o_n)

  \sigma_0 \models PUT(i_1) ; o_1 \leftarrow PUT(i_2); \ldots ; \text{on} \leftarrow PUT(i_n) ; \text{result(post o}_1 \ldots o_n)

  \sigma_0 \models os \leftarrow \text{mbind is PUT} \quad \text{result(post os)}

  \text{pre} \quad is \quad \Rightarrow \quad \sigma_0 \models os \leftarrow \text{mbind is PUT} \quad \text{result(post os)}
NPfIT in UPF

• Example for NPfIT:
  (General Pattern, formalizing an informal requirement):

  $\text{pre } i_S \implies \sigma_0 \models o_S \leftarrow \text{mbind } \text{PUT} \,(i_S); \ \text{result}(\text{post } o_S)$
NPfIT in UPF

• Example for NPfIT:

(General Pattern, formalizing an informal requirement):

\[
\llbracket \text{users } i_S \subseteq \{\text{urp1\_alice, urp2\_alice, urp\_john, urp\_bob}\}; \\
\sigma_0 \models \text{os } \leftarrow \text{mbind } i_S \text{ RBAC\_Mon; return (os } = X) \rrbracket
\]

\[\Rightarrow \sigma_0 \models \text{os } \leftarrow \text{mbind } i_S \text{ PUT; return (os } = X)\]
Our System: **HOL-TestGen** is ...

- ... based on HOL (Higher-order Logic):
  - “Functional Programming Language with Quantifiers”
  - plus definitional libraries on Sets, Lists, . . .
  - can be used meta-language for HoareCalculi, Z, CSP. . .
- ... implemented on top of Isabelle
  - an interactive prover implementing HOL
  - the test-engineer must decide over, abstraction level, split rules, breadth and depth of data structure exploration . . .
  - providing automated and interactive constraint-resolution techniques
  - interface: ProofGeneral
- ... by thy way, a verified test-tool
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
- Test-Cases: partitions of I/O relation of the form

\[ C_1(x) \quad \Rightarrow \quad \ldots \quad C_n(x) \quad \Rightarrow \quad \text{post } x \ (\text{PUT } x) \]
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form
    \[ C_1(x) \implies \ldots C_n(x) \implies \text{post } x \text{ (PUT } x) \]
- Test-Data-Selection
  - constraint Solver gen_test_data
  - finds x satisfying \( C_i(x) \)
HOL-TestGen Workflow

- **Modelisation**
  - writing background theory of problem domain

- **Test-Case-Generation from Test-Specification**
  - automated procedure gen_test_case ...
  - Test-Cases: partitions of I/O relation of the form
    \[ C_1(x) \implies \ldots C_n(x) \implies \text{post } x \text{ (PUT } x \text{)} \]

- **Test-Data-Selection**
  - constraint solver gen_test_data
  - finds x satisfying \( C_i(x) \)

- **Test-Driver Generation**
  - automatically compiled, drives external program
HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain
- Test-Case-Generation from Test-Specification
  - automated procedure gen_test_case ... 
  - Test-Cases: partitions of I/O relation of the form
    \[ C_1(x) \implies \ldots C_n(x) \implies \text{post x (PUT x)} \]
- Test-Data-Selection
  - constraint solver gen_test_data
  - finds x satisfying \( C_i(x) \)
- Test-Driver Generation
  - automatically compiled, drives external program
- Test Execution, Test-Documentation
TestGen: Symbolic Computations

pre $x \rightarrow$ post $(x, \text{PUT } x)$

case-splitter
(variables+types: regularity hypothesis
patterns: domain specific test rules)

case-solver
(simplifier, SMT-solver, ...)

case-normalizer
(CNF +)

selection-former
(inserts uniformity hypothesis)

$\text{k times ...}$
Conclusion

• HOL-TestGen used for NPfIT was success wrt:
  • superior modeling techniques
  • substantial conservative libraries
  • standardized interfaces to tactic and automatic proof
  • code generation
  • a programming interface and genericity in design

... offering lot of machinery not worth to reinvent.
Conclusion

- HOL-TestGen used for NPfIT was not successful as a project:
  - we did not manage to find partners in the NPfIT Consortium that were actually using our test data...
  - public and private awareness of security problems apparently VERY LOW
  - exploration of data space not (yet) very deep