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

GÂ 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.

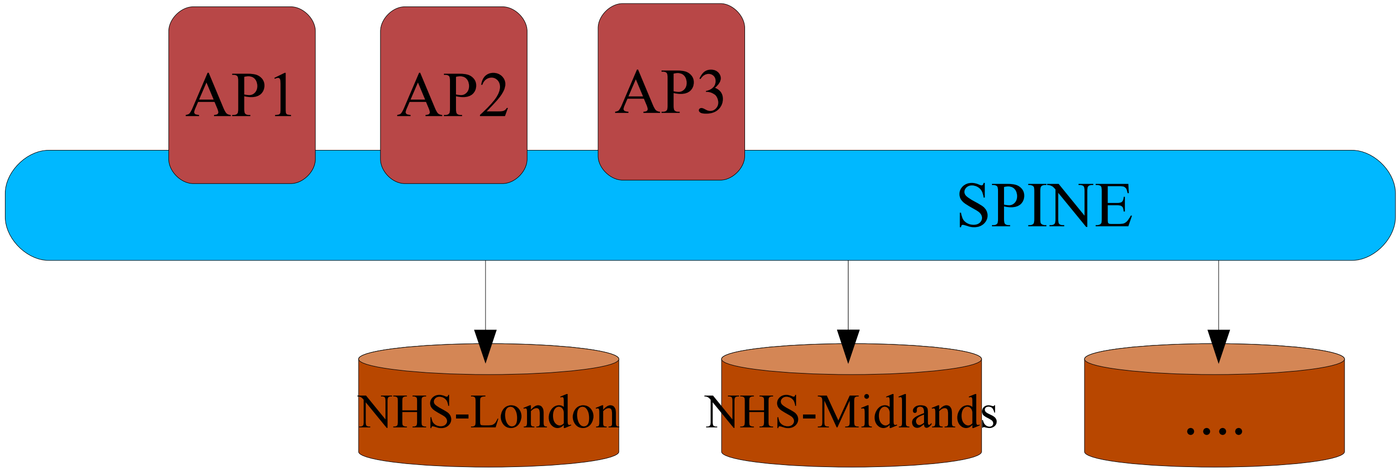
Particular emphasis is put on the modular description of security policies and their generic combination and its consequences for model-based testing.

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)

datatype α decision = allow α | deny α

types (α,β) policy = $\alpha \rightharpoonup \beta$ decision (* = $\alpha \Rightarrow \beta$ option *)

notation $\alpha \mapsto \beta = (\alpha,\beta)$ policy

Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Policy Constructors

definition $\varnothing \equiv \lambda y.\ None$ (* $\varnothing :: \alpha \Rrightarrow \beta$ *)

definition $p(x{+}{\mapsto}t) \equiv p(x \mapsto Some(allow\ t))$ (* $p :: \alpha \Rrightarrow \beta$ *)
$p(x{-}{\mapsto}t) \equiv p(x \mapsto Some(deny\ t))$ (* where $p(x \mapsto t) \equiv \lambda y.\ if\ y = x\ then\ A\ else\ p\ y$ *)

definition (*AllowAll :: "($\alpha \rightharpoonup \beta$) $\Rightarrow$ ($\alpha \Rrightarrow \beta$)" *)

$\forall_A x.\ pf(x) \equiv (\lambda x.\ case\ pf\ x\ of\ Some\ y \Rightarrow Some(allow(y))\ |\ None \Rightarrow None)$

(*DenyAll :: "($\alpha \rightharpoonup \beta$) $\Rightarrow$ ($\alpha \Rrightarrow \beta$)"*)

$\forall_D x.\ pf(x) \equiv (\lambda x.\ case\ pf\ x\ of\ Some\ y \Rightarrow Some(allow(y))\ |\ None \Rightarrow None)$

Modeling Framework: Unified Policy Framework (UPF)

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

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

definition $\text{dom} :: \alpha \rightharpoonup \beta \Rightarrow \alpha \text{ set}$
where $\text{dom } f \equiv \{x.\ f\ x \neq \text{None}\}$

definition $\text{ran} :: \alpha \rightharpoonup \beta \Rightarrow \beta \text{ set}$ ...

definition $\_ \lhd \_ :: \alpha \text{ set} \Rightarrow \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta$
where $S \lhd p \equiv (\lambda\, x.\ \text{if } x \in S \text{ then } p\ x \text{ else none})$ (* domain restriction *)

definition $\_ \rhd \_ :: \alpha \rightharpoonup \beta \Rightarrow \alpha \text{ set} \Rightarrow \alpha \rightharpoonup \beta$ ... (* range restriction *)

definition $\_ \oplus \_ :: \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta \Rightarrow \alpha \rightharpoonup \beta$ ...(* first fit override *)

Example: Firewalls

- Firewall Policies in UPF
  - Data:

    ip-address = $int \times int \times int \times int$
    ip-packet = ip-address × protocol × content × ip-address

  - Firewall - Policies:

    policy : ip-packet $\Rightarrow$ ip-packet

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

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))

Example: Firewalls

- Firewall Policies in UPF
  - Elementary Policies

    definition me-ftp :: ip-packet $\Rightarrow$ ip-packet
    where me-ftp $\equiv$ $\varnothing$ ((192,22,14,76),ftp,d,(192,22,14,76)
    $+\mapsto$(192,22,14,76),ftp,d,(192,22,14,76))

  - Combined Policies:

    definition me-none-else:: ip-packet $\Rightarrow$ ip-packet

    where me-none-else $\equiv$ 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 ⊕ $\forall_D$ x. x

Example: RBAC

- RBAC Policies in UPF
  - Domain: UR = users × role
    RP = role × permission
  - 2-Policies:
    UserTab :: UR ⇀ unit,
    PermTab:: permission ⇀ role ⇀ unit

Example: RBAC

- RBAC Policies in UPF
  - Domain: UR = users × role
    RP = role × permission
  - 2-Policies:
    UserTab :: UR ⇛ unit,
    PermTab:: permission ⇒ role ⇛ unit

datatype users = …
datatype roles = …
datatype permissions = …

definition rbac … RBAC (perm) = UserTab $o_{\vee D}$ PermTab(perm)

Example: RBAC

- RBAC Policies in UPF
  - Domain: UR = users × role
    RP = role × permission
  - 2-Policies:
    UserTab :: UR ⇛ unit,
    PermTab:: permission ⇛ role ⇛ unit

datatype users = …
datatype roles = …
datatype permissions = …

definition rbac … RBAC (perm) = UserTab $o_{\vee D}$ PermTab(perm)

Example: RBAC

- RBAC Policies in UPF
  - Domain: $UR = users \times role$
    $RP = role \times permission$
  - 2-Policies:

    UserTab :: UR ⇛ unit,
    PermTab:: permission ⇒ role ⇛ unit

    datatype users = …
    datatype roles = …
    datatype permissions = …

    definition rbac … RBAC (perm) = UserTab $o_{\vee D}$ PermTab(perm)

    where $o_{\vee D}$ is one of the 4 policy sequential compositions

More on UPF

- Transition Policies
  - Transition Policies: Policies involving state

$$\alpha \times \sigma \Rrightarrow \beta \times \sigma \qquad \text{(input } \alpha\text{, output } \beta\text{)}$$

  - Higher-order Policies (Policies transforming policies)

$$\alpha \times (\gamma \Rrightarrow \delta) \Rrightarrow \beta \times (\gamma \Rrightarrow \delta)$$

  - 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:

```
definition prod_orA ::"['α↦'β, 'γ ↦'δ] ⇒ ('α×'γ ↦ 'βx'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))
                       | Some(deny d2)  ⇒ Some(allow(d1,d2))
                       | None ⇒ None)
             | Some(deny d1)⇒(case p2 y of
                         Some(allow d2) ⇒ Some(allow(d1,d2))
                       | Some(deny d2)  ⇒ Some(deny (d1,d2))
                       | None ⇒ None)
             | None ⇒ None))
```

Principal Use of UPF for NPfIT

- Parallel Composition of 4 Policies + Functional:

$$
\begin{aligned}
&(\text{norm\_beh}, \text{excep\_beh})\ \nabla \\
&\quad (\text{legitimate\_relation}\ \otimes_{\vee A} \\
&\quad \text{patients\_consent}\ \otimes_{\vee A} \\
&\quad \text{sealed\_envelopes}\ \otimes_{\vee A} \\
&\quad \text{rbac})
\end{aligned}
$$

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 ι σ. case p (ι,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:

definition policy2MON :: (ι×σ ⇀ o×σ) ⇒ ι ⇒ (o decision,σ)MON_{SE}

where policy2MON p =
(λ ι σ. case p (ι,σ) of

Some(allow(o,σ ')) ⇒ Some(allow o, σ')

| Some(deny(o, σ')) ⇒ Some(deny o, σ')

| None ⇒ None)

Modeling Framework: Unified Policy Framework (UPF)

- State-Exception Monads(f.Test-Sequences in HOL)

  - State-Exception Monads:

    type $(o,\sigma)MON_{SE} = \sigma \rightharpoonup (o, \sigma)$

    definition bind :: $(o,\sigma)MON_{SE} \Rightarrow (o \Rightarrow (o,\sigma)MON_{SE}) \Rightarrow (o,\sigma)MON_{SE}$ (“ _ ; _ ← _”)
    where …

    definition unit :: $(o \Rightarrow bool) \Rightarrow (o,\sigma)MON_{SE})$ (“return _ ”)
    where …

  - Computation Sequences, Valid Computation Sequences, Valid mbind-Sequences, Valid mbind-Sequences with pre-condition:

    $PUT(i_1) ; o_1 \leftarrow PUT(i_2); \ldots ; on \leftarrow PUT(i_n) ; result(post\ o_1 \ldots o_n)$

    $\sigma_0 \models PUT(i_1) ; o_1 \leftarrow PUT(i_2); \ldots ; on \leftarrow PUT(i_n) ; result(post\ o_1 \ldots o_n)$

    $\sigma_0 \models o_S \leftarrow mbind\ i_S\ PUT ;\ result(post\ o_S)$

    $pre\ i_S \Rightarrow \sigma_0 \models o_S \leftarrow mbind\ i_S\ PUT;\ result(post\ o_S)$

NPfIT in UPF

- Example for NPfIT:

  (General Pattern, formalizing an informal requirement) :

$$\text{pre } i_S \Longrightarrow \sigma_0 \models o_S \leftarrow \text{mbind PUT } (i_S);\ \text{result}(\text{post } o_S)$$

NPfIT in UPF

- Example for NPfIT:

  (General Pattern, formalizing an informal requirement) :

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

$$\Longrightarrow \quad \sigma_0 \models os \leftarrow \text{mbind } i_S \text{ PUT}; \text{return } (os = X)$$

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

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

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

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- 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

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- Test-Data-Selection
  - constraint solver gen_test_data
  - finds x satisfying $C_i(x)$
- Test-Driver Generation
  - automatically compiled, drives external program

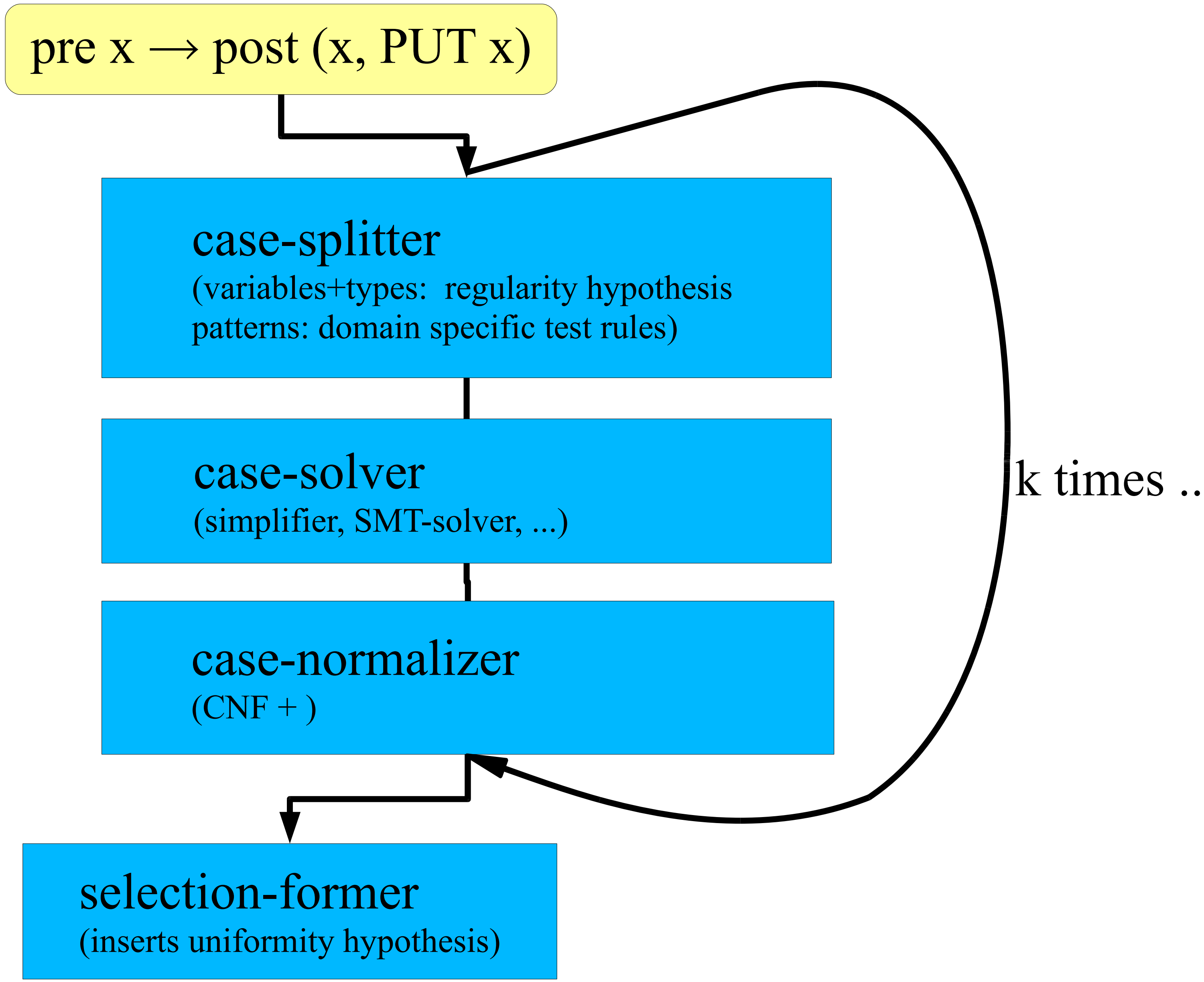
HOL-TestGen Workflow

- 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

$$C_1(x) \Longrightarrow \ldots C_n(x) \Longrightarrow \text{post } x \text{ (PUT } x)$$

- 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



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