Úlfar Erlingsson
Roel Wieringa
Nicola Zannone (Eds.)

# Engineering Secure Software and Systems

LNCS 6542

Third International Symposium, ESSoS 2011
Madrid, Spain, February 2011
Proceedings

Springer

Lecture Notes in Computer Science 6542

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Úlfar Erlingsson  Roel Wieringa
Nicola Zannone (Eds.)

# Engineering Secure Software and Systems

Third International Symposium, ESSoS 2011
Madrid, Spain, February 9-10, 2011
Proceedings

Springer

Volume Editors

Úlfar Erlingsson
Google Inc.
1288 Pear Ave, Mountain View, CA 94043, USA
E-mail: ulfar@google.com

Roel Wieringa
University of Twente, Computer Science Department
Drienerlolaan 5, 7522 NB Enschede, The Netherlands
E-mail: r.j.wieringa@ewi.utwente.nl

Nicola Zannone
Eindhoven University of Technology
Faculty of Mathematics and Computer Science
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
E-mail: n.zannone@tue.nl

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19124-4 e-ISBN 978-3-642-19125-1
DOI 10.1007/978-3-642-19125-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011920029

CR Subject Classification (1998): C.2, E.3, D.4.6, K.6.5, J.2

LNCS Sublibrary: SL 4 – Security and Cryptology



*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

It is our pleasure to welcome you to the third edition of the International Symposium on Engineering Secure Software and Systems.

This unique event aims at bringing together researchers from software engineering and security engineering, which might help to unite and further develop the two communities in this and future editions. The parallel technical sponsorships from the ACM SIGSAC (the ACM interest group in security) and ACM SIGSOF (the ACM interest group in software engineering) is a clear sign of the importance of this interdisciplinary research area and its potential.

The difficulty of building secure software systems is no longer focused on mastering security technology such as cryptography or access control models. Other important factors include the complexity of modern networked software systems, the unpredictability of practical development life-cycles, the intertwining of and trade-off between functionality, security and other qualities, the difficulty of dealing with human factors, and so forth. Over the last few years, an entire research domain has been building up around these problems.

The conference program include two major keynotes from George Candea (École Polytechnique Fédérale de Lausanne) on automated cloud-based software reliability services and Mark Ryan (University of Birmingham) on the analysis of security properties of electronic voting systems, and an interesting blend of research and idea papers.

In response to the call for papers, 63 papers were submitted. The Program Committee selected 18 contributions as research papers (29%), presenting new research results in the realm of engineering secure software and systems. It further selected three idea papers, which gave crisp expositions of interesting, novel ideas in the early stages of development.

Many individuals and organizations contributed to the success of this event. First of all, we would like to express our appreciation to the authors of the submitted papers and to the Program Committee members and external referees, who provided timely and relevant reviews. Many thanks go to the Steering Committee for supporting this and future editions of the symposium, and to all the members of the Organizing Committee for their tremendous work and for excelling in their respective tasks. The DistriNet research group of the K.U. Leuven did an excellent job for the website and the advertising for the conference. Nicola Zannone did a great job by assembling the proceedings for Springer.

We owe gratitude to ACM SIGSAC/SIGSOFT, IEEE TCSP and LNCS for supporting us in this new scientific endeavor.

December 2010

Úlfar Erlingsson
Roel Wieringa
Manuel Clavel

# Conference Organization

## General Chair

| | |
|---|---|
| Manuel Clavel | Imdea Software/Universidad Complutense de Madrid, Spain |

## Program Co-chairs

| | |
|---|---|
| Úlfar Erlingsson | Google Inc., US/Reykjavik University, Iceland |
| Roel Wieringa | University of Twente, The Netherlands |

## Publication Chair

| | |
|---|---|
| Nicola Zannone | Eindhoven University of Technology, The Netherlands |

## Publicity Chair

| | |
|---|---|
| Pieter Philippaerts | Katholieke Universiteit Leuven, Belgium |

## Local Arrangements Chair

| | |
|---|---|
| Marina Egea | Imdea Software, Spain |

## Steering Committee

| | |
|---|---|
| Jorge Cuellar | Siemens AG, Germany |
| Wouter Joosen | Katholieke Universiteit Leuven, Belgium |
| Fabio Massacci | Università di Trento, Italy |
| Gary McGraw | Cigital, USA |
| Bashar Nuseibeh | The Open University, UK |
| Daniel Wallach | Rice University University, USA |

## Programme Committee

| | |
|---|---|
| Thomas Alspaugh | University of California at Irvine, USA |
| Jo Atlee | University of Waterloo, Canada |
| Bruno Blanchet | Ecole Normale Superieure, France |
| Hao Chen | University of California, Davis, USA |
| Frederic Cuppens | Ecole Nationale Supérieure de Télécommunication Bretagne, France |
| Prem Devanbu | University of California at Davis, USA |

| | |
|---|---|
| Eric Dubois | Centre de Recherche Public Henri Tudor, Luxembourg |
| Christof Ebert | Vector Consulting, Germany |
| Manuel Fahndrich | Microsoft Research, USA |
| Eduardo Fernandez-Medina | Universidad de Castilla-La Mancha, Spain |
| Robert France | Colorado State University, USA |
| Vinod Ganapathy | Rutgers University, USA |
| Dieter Gollman | Hamburg University of Technology, Germany |
| Siv Hilde Houmb | Telenor, Norway |
| Martin Johns | SAP Research, Germany |
| Jan Jurjens | Technische Universität Dortmund, Germany |
| Yuecel Karabulut | SAP Labs, USA |
| Seok-Won Lee | University of North Carolina Charlotte, USA |
| Lin Liu | Tsinghua University, China |
| Robert Martin | MITRE, USA |
| Vaclav Matyas | Masaryk University, Czech Republic |
| Sjouke Mauw | University of Luxembourg, Luxembourg |
| Chris Mitchell | Royal Holloway, UK |
| Akito Monden | Nara Institute of Science and Technology, Japan |
| Haralambos Mouratidis | University of East London, UK |
| Marcus Peinado | Microsoft Research, USA |
| Erik Poll | University of Nijmegen, The Netherlands |
| David Sands | Chalmers University, Sweden |
| Angela Sasse | University College London, UK |
| Venkat Venkatakrishnan | University of Illinois at Chicago, USA |

## External Reviewers

Aizatulin, Misha
Berkman, Omer
Birgisson, Arnar
Blanco, Carlos
Brucker, Achim D.
Cuppens-Boulahia, Nora
Del Tedesco, Filippo
Dobias, Jaromir
Garcia-Alafaro, Joaquin
Gerguri, Shkodran
Hirsch, Martin
Kordy, Barbara
Kur, Jiri
Magazinius, Jonas
Nikiforakis, Nick
Ochoa, Martin
Phung, Phu H.
Poolsappasit, Nayot
Radomirovic, Sasa
Rafnsson, Willard
Rosado, David G.
Russo, Alejandro
Sánchez, Luis Enrique
Schmidt, Holger
Schweitzer, Patrick
Stetsko, Andriy
Svenda, Petr
Tucek, Pavel
van Deursen, Ton
van Sinderen, Marten J.

# Table of Contents

## Session 1. Model-Based Security I



## Session 2. Tools and Mechanisms



## Session 3. Web Security

## Session 4. Model-Based Security II



## Session 5. Security Requirements Engineering



## Session 6. Authorization



## Session 7. Ideas

# Model-Based Refinement of Security Policies in Collaborative Virtual Organisations

Benjamin Aziz[1], Alvaro E. Arenas[2], and Michael Wilson[3]

[1] School of Computing, University of Portsmouth, Portsmouth, U.K.
Benjamin.Aziz@port.ac.uk
[2] Department of Information Systems, Instituto de Empresa Business School, Madrid, Spain
alvaro.arenas@ie.edu
[3] e-Science Centre, STFC Rutherford Appleton Laboratory, Oxfordshire, U.K.
michael.wilson@stfc.ac.uk

**Abstract.** Policy refinement is the process of deriving low-level policies from high-level policy specifications. A basic example is that of the refinement of policies referring to users, resources and applications at a high level, such as the level of virtual organsiations, to policies referring to user ids, resource addresses and computational commands at the low level of system and network environments. This paper tackles the refinement problem by proposing an approach using model-to-model transformation techniques for transforming XACML-based VO policies to the resource level. Moreover, the transformation results in deployable policies referring to at most a single resource, hence avoiding the problem of cross-domain intereference. The applicability of our approach is demonstrated within the domain of distributed geographic map processing.

## 1 Introduction

*Virtual Organisations* (VOs) represent a common abstraction for describing computations and storage in a Grid environment. A VO is considered as a collaborative environment in which real organisations combine to offer applications and resources to users [1]. Such resources could be storage capabilities, software services or simply computational power represented as physical machines. A *VO policy* is a statement about the expected normal behaviour of applications, resources and users in the whole VO. A VO policy is usually written in terms of VO-wide concepts; possibly all the running applications, resources and users in the VO. VO policies are crucial to the correct operation of large-scale Grid-based VOs, dealing with issues of utilisation measurement, accounting, security etc [2].

Traditionally, a VO policy is seen as the collection of the individual resource policies. This view is both easily justifiable, in the sense that the VO consists in fact of the resources themselves, and passive, in the sense that no extra effort is required to manage and enforce VO policies that attempt to give a richer statement about the overall acceptable VO behaviour. More recently, research in security policies has moved on to the adoption of VO policies that express more than what the mere individual resource policies have been set up to express, as shown in [3]. This view is more interesting in

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 1–14, 2011.

the sense that it starts from the requirements of the VO itself, prior to its population with resources, rather than starting from the policy constraints of those resources.

A problem, however, that arises with this latter view is regarding the level at which enforcement of the VO policy is applied. There are at least two solutions: The first is to enforce the VO policy directly using a VO-wide Policy Enforcement Point (PEP). This case has the limitation of having a single centralised point of failure (such as operational or semantic failures). The second solution, which we adopt in this paper, is based on refining VO policies from their VO-wide representation down to their computational-level representation at individual resources [4]. This latter approach avoids the development of a centralised VO PEP but it is constrained by the limitations of the refinement algorithm and the fact that not all VO policies can be refined. We restrict our treatment in this paper to VO security policies, i.e. policies concerned with the access control of resources by users, that can be refined. Furthermore, we allow our refinement to be guided by predefined VO-Resource hierarchies in a similar manner to [5].

The paper is structured as follows. Section 2 introduces our case study, the processing of distributed geographical maps. Next, Section 3 presents some background information about the lifecycle of VO security policies, and discusses the relation between VO policies and resource policies. Section 4 describes the refinement of VO policies to their resource-level representation. Then, Section 5 shows the implementation of our refinement approach using model-based tools. Section 6 compares our work with others. Finally, Section 7 summarises the work and highlights future work.

## 2 A Case Study: Distributed Geographical Map Processing

We consider here a case study involving distributed geographical map processing [6]. In this scenario, shown in Figure 1, a *user* requests from the VO manager the creation of a specific multimedia artefact, such as a *geographic map*, a process which involves high levels of data storage and computation intensity [7].

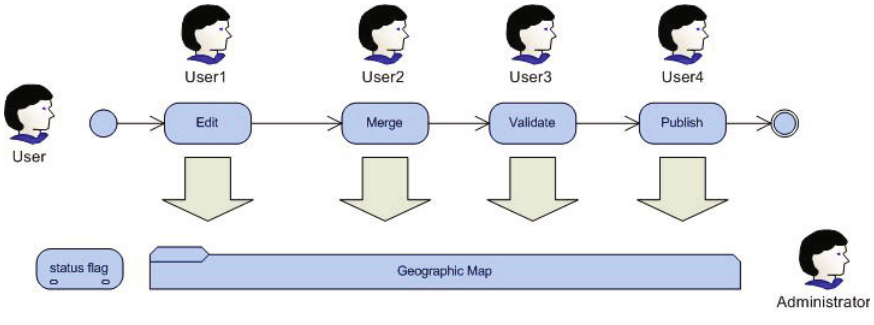



**Fig. 1.** The Distributed Geographical Map Processing Workflow

The VO manager initiates a *worflow* to execute the task leading to the production of the geographic map. This workflow consists of four main processes:

- *Editing*: During the Editing process, users carrying out the task perform different computations on the data resource, in this case the map, to edit it. These task users may belong to different administrative domains. The resource itself is protected by its own resource administrator and may have a local state indicating, for example, the current process of the workflow.
- *Merging*: Once all the computational sub-tasks on the data are finished and sub-maps are produced, these are now merged into one draft map ready for validation.
- *Validating*: This process commences when a complete draft of the map is ready, where the draft is validated to ensure that errors, incosistencies and anomalies are removed prior to the next process.
- *Publishing*: Finally, once the map has been validated, it is ready to be published.

We focus in this paper on four main concepts abstracted from this case study:

- *VO Processes*: which include the various processes in the workflow,
- *VO Users*: which include any users performing the VO processes,
- *VO Resources*: which include the main VO resources such as data, in this case the geographic map, and computational power,
- *VO Environments*: which express the state of the VO resources, in this case the flag indicating the state of the map.

The above concepts represent high level business concepts, which despite the fact that it is possible to write policies in terms of these concepts, such policies will require enforcement at the high level of the business application itself. Beside the fact that high level concepts are not always tangible (e.g. a digital map does not exist outside the individual files or data sets containing its data), such a high level solution will introduce the need for a single centralised policy enforcement point that is capable of making decisions on behalf of all the resources in the VO. Instead, in our solution, the business policies are refined in terms of their computational concepts.

## 3 Background

In this section, we provide some background on the lifecycle of VO security policies explaining what we mean by VO policies and resource policies.

### 3.1 VO Policy Management

A lifecycle for the management of VO policies may consist of the following stages.

- *Generation of VO Policies from VO Requirements*: During this stage, after the requirements of a VO have been gathered and analysed, these requirements are used to guide the creation of the VO policy. This could be done either by direct authoring or using policy generation tools (e.g. [8]). At this stage, VO policies can also be analysed for undesirable properties, such as conflicts or inconsistencies (see for example [9]). This stage is outside the scope of this paper.
- *Refinement of VO Policies to Resource Policies*: Once the VO policy has been defined and analysed, it is then refined to a corresponding resource-level policy. This refinement yields a policy that may refer to multiple resources.

– *Transformation to Deployable Resource Policies*: Policies that refer to multiple resources are not suitable for deployment, since they reveal information to administrative domains about other domains. Therefore, a transformation is required to achieve deployable policies, which refer to at most one resource.

Our focus mainly in this paper will be on the second and third stages.

### 3.2 VO Policies

There are many types of VO policies. Wasson and Humphrey [10] identify a few of these including *security policies*, *resource provisioning policies* and *QoS policies*. Our main focus here will be on security policies, which establish desirable security properties in a VO, such as acceptable access and usage behaviour, flow of information and integrity and availability of data. We define a VO policy as one that is written in terms of a *VO alphabet*. A metamodel of our VO alphabet is shown in Figure 2. The VO al-

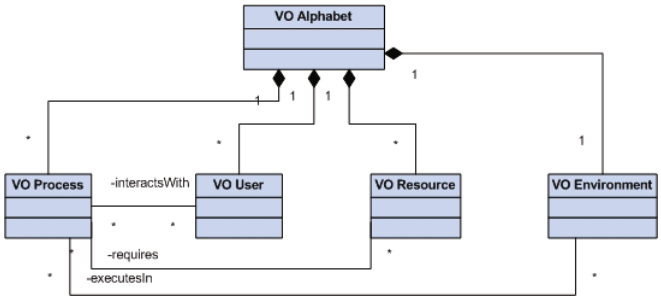



**Fig. 2.** The VO Alphabet Metamodel

phabet consists of entities such as *VO Processes*, *VO Users*, *VO Resources* and the *VO Environments*. A VO could consist of several processes composed or orchestrated in a *workflow* that determines the control flow logic for the execution of these processes. A workflow in this sense is the application itself. The VO may also have a number of users, that initiate and *interact* with its processes. A process *requires* VO resources to utilise during its *execution within* the global VO environment.

A VO alphabet is domain-specific. For example, the following is a VO policy from the domain of our case study in distributed geographic map processing.

*Example 1. User* ***Bart Simpson*** *can* ***edit*** *the* ***UK map*** *belonging to* ***National Geographic*** *only between hours* ***13:00*** *and* ***14:00*** *on* ***weekdays****, for a maximum number of* ***three*** *times in each of these periods, and when the workflow is in the* ***editing*** *phase.*

### 3.3 Resource Policies

At the resource level, a VO policy will express the desirable behaviour of the resource from the VO's perspective and it will be a refinement of the global VO policy. The policy must refer to the local resource alphabet, where the metamodel of one such alphabet

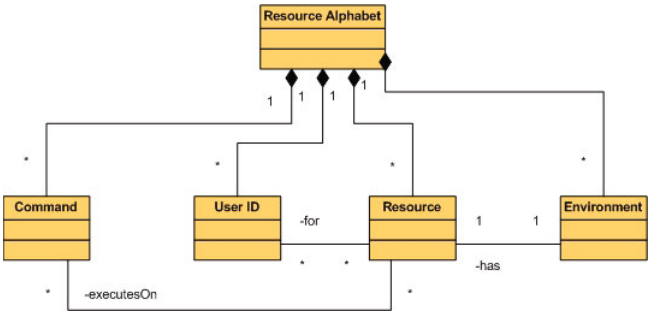



**Fig. 3.** The Resource Alphabet Metamodel

as shown in Figure 3. This metamodel comprises entities such as *resources*, execution *commands*, *user ids* and computational *environments*. Each resource would have its own set of user ids that it controls access from. Commands are the decomposition of VO level processes and they execute on resources. Finally, a resource has its own environment, which may include the current time and date at the resource.

The VO policy of Example 1 can be expressed in terms of this alphabet as follows:

*User **abc1234** can **read/write** the **topology/transportation datasets for the UK Map** belonging to **National Geographic** only between hours **13:00** and **14:00** on **weekdays**, for a maximum of **six** times in each of these periods, and when workflow is in **editing** phase.*

Where user id *abc1234* corresponds to the identity of Bart Simpson, *read/write* commands correspond to the editing process and the UK map consists of the topological and transportation data sets. Also, we assume that "...maximum of **three** times..." is for each of the read/write operations (assumed to be equal to a total of **six** such commands). For the policy to be deployable, it will need to be transformed to the following two policies, since each file could have a different owner or administrator, even within National Geographic (e.g. belongs to a different department):

*User **abc1234** can **read/write** the **transportation datasets for the UK Map** belonging to **National Geographic** only between hours **13:00** and **14:00** on **weekdays**, for a maximum of **six** times in each of these periods, and when the workflow is in **editing** phase.*

*User **abc1234** can **read/write** the **topology datasets for the UK Map** belonging to **National Geographic** only between hours **13:00** and **14:00** on **weekdays**, for a maximum of **six** times in each of these periods, and when the workflow is in **editing** phase.*

## 4 From VO to Deployable Resource Policies

We discuss in this section the refinement of VO policies to their resource level representation. In order to understand the refinement process, we explain first what we mean by the notion of *VO-Resource Hierarchies*.

### 4.1 VO-Resource Hierarchies

In order to transform policies from the VO-level to the resource-level, one needs information about how the two levels are related to each other. In Su et al. [5], the authors define a resource type hierarchy, which is a directed graph expressing the topology of a VO in terms of its intermediate and most basic elements. The hierarchy also contains information about possible actions that can be applied to resources at each of its levels. The policy refinement then becomes a matter of instantiating policies at each level of the hierarchy. Here, we follow a similar, but more direct approach by assuming the presence of two levels only as in Figure 4, which we term a *VO-Resource hierarchy*.

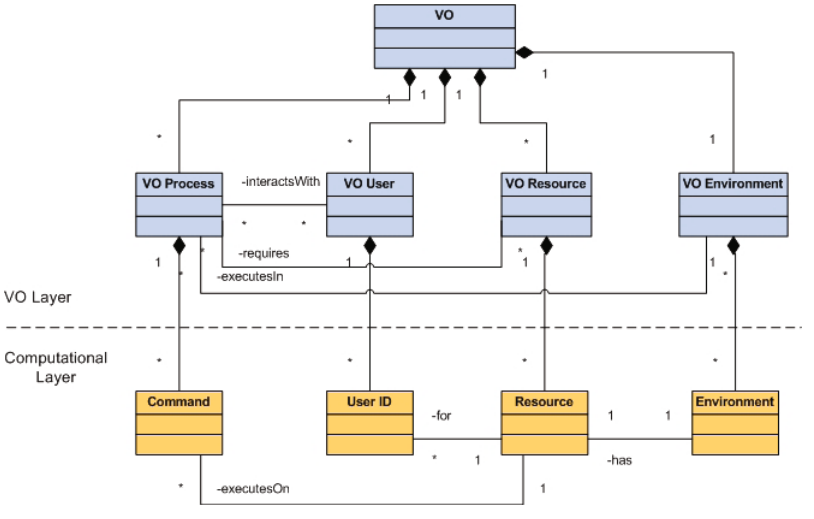



**Fig. 4.** VO-Resource Hierarchy Metamodel

The metamodel links the VO and resource alphabet metamodels of the previous sections, by defining VO processes as compositions of computational commands, VO users as compositions of user ids, VO resources as compositions of computational resources and VO environments as compositions of computational environments.

*Example 2.* Let's consider an example of this hierarchy metamodel inspired from case study of Section 2. Each of the VO processes is mapped to its set of computational level commands. For example, *Edit* may consist of the commands *read*, *write*, *copy* and *delete* of individual files. *Merge* could consist of the commands to *copy* and *paste* text in order to construct the final draft of the document. *Validate* could also consist of the commands to *read* and *write* files in the final draft in order to correct any errors, and finally, *Publish* may consist of a command to *print* the final validated version.

Similarly, a VO resource is decomposed in terms of the individual computational resources it comprises. Hence, a *geographic map* is decomposed into the individual files holding information about the *topography*, *transportation* and *administrative boundaries* in the map. VO users are mapped to computational level user ids. There could be many such ids each one of them local to the individual administrative domain or

computational resource. For example, user *Bart Simpson* maps to user ids, *abc1234*, *xyz0000* and *fgh6789*, each at the level of the individual resource. We also assume the presence of a *system administrator* who also have a user id on their local resources (root). Finally, VO-wide environment attributes are assumed to have definitions in terms of the computational level environment attributes local to each resource. *Environment* attributes could include the *date and time* of the day, the *location of the resource* if it is a mobile device, or any workflow status flags.

### 4.2 Policy Refinement

Based on the information provided in a VO-Resource hierarchy and given a VO policy, it is possible to define a *policy refinement* function, which takes VO-level policies expressed in XACML v2.0 and produces the corresponding resource-level policies also expressed in XACML v2.0. The refinement function can cater for all the elements of the XACML metamodel defined by OASIS [11]. In this model, XACML policies include rules on attributes of the subjects, resources and environments and their associated conditions as well as obligations that must be fulfilled if a policy is applicable.

*Example 3.* We consider first a simple XACML policy from Example 1:

```
<Policy PolicyId="DM VO Policy"
 <Rule Effect="Permit" RuleId="Editing Rule">
  <Target>
    <Resource>
      <AttributeValue>/geographic map/uk</AttributeValue>
    </Resource>
    <Subject>
      <AttributeValue>Bart Simpson</AttributeValue>
    </Subject>
    <Action>
      <AttributeValue>Edit</AttributeValue>
    </Action>
    <Environment>
    </Environment>
  </Target>
  <Condition>
    (resource-id-qualifier = National Geographic) AND
    (13:00:00 =< subject:request-time =< 14:00:00) AND
    (environment:current-day = weekday) AND
    (action:edit_count_var =< 3) AND
    (environment:status_flag = edit)
  </Condition>
 </Rule>
 <Obligation FulfillOn="Permit">
   (edit_count_var := edit_count_var + 1)
 </Obligation>
</Policy>
```

The subject, resource and action values assume their places in the corresponding XACML elements. There are no environment elements here. The policy considers the part "*belonging to* ***National Geographic*** *only between hours* ***13:00*** *and* ***14:00*** *on* ***weekdays****. . . and when the workflow is in the* ***editing*** *phase.*" as a condition in a conjunctive form consisting of multiple ANDed predicates. On the other hand, the policy part "*. . . and for a maximum number of* ***three*** *times in each of these periods.*" is defined in terms of a policy obligation that increments a counter for the number of times the edit process is executed and a rule condition that ensures that the value of the counter, when applying the rule, is less than or equal the value of three. The counter obligation must be fulfilled by the policy enforcement point whenever the rule is permitted.

**The Refinement Function.** Next, we define our policy refinement function that generates, from high-level VO policies, the low-level computational policies for each individual resource in a VO. We start first by defining a *VO hierarchy* tuple.

**Definition 1 (VO hierarchy).** *Define $\mathcal{H}_{VO} = (\mathcal{P}, \mathcal{U}, \mathcal{R}, \mathcal{E})$ as a VO hierarchy tuple, which is a formalisation of the metamodel of Figure 4 as follows:*

- $\mathcal{P}$ : *VO Process* $\rightarrow \wp($*Command*$)$ *is a function that refines a VO process to a multiset of the computational commands comprising it. The result is a multiset rather than a set to be able to express the number of occurrences for each command.*
- $\mathcal{U}$ : *VO User* $\rightarrow \wp($*User ID*$)$ *is a function that refines a VO user's name to the set of user ids comprising it.*
- $\mathcal{R}$ : *VO Resource* $\rightarrow \wp($*Resource*$)$ *is a function that refines a VO resource set to the set of computational resources comprising it.*
- $\mathcal{E}$ : *VO Environment* $\rightarrow \wp($*Environment*$)$ *is a function refines a global VO environment to a set of individual resource environments.*

Going back to the example of the document management workflow, the $\mathcal{H}_{VO}$ tuple can be defined as follows:

$$\mathcal{H}_{VO} = \begin{pmatrix} \mathcal{P} = \{(Edit, \{|Create, Delete, Write, Read|\}), (Publish, \{|Print|\}) \\ \qquad\qquad (Merge, \{|Copy, Paste|\}), (Validate, \{|Read, Write|\})\}, \\ \mathcal{R} = \{(\textit{UK Map}, \{\textit{Topography Data Set}, \textit{Transportation Data Set}\})\}, \\ \mathcal{U} = \{(\textit{Bart Simpson}, \{\textit{abc1234}, \textit{xyz0000}, \textit{fgh6789}\})\}, \qquad \mathcal{E} = \{\} \end{pmatrix}$$

For simplicity, we have assumed that each of the VO processes maps to single-occurrence commands. Hence, executing Merge once means executing Copy once and Paste once. Another, more general, definition could use multisets instead of sets.

In addition to the $\mathcal{H}_{VO}$ tuple, we also define the following two relations, which define how conditions/obligations are refined.

**Definition 2 (Auxiliary Relations).** *Define the condition and obligation mappings as:*

- *The $\mathcal{C}$ : Condition $\rightarrow$ Condition function maps policy conditions stated in the VO policy to a condition at the computational level for each individual resource.*
- *The $\mathcal{O}$ : Obligation $\rightarrow$ Obligation function maps a VO-level obligation stated in the VO policy to a computational-level obligation stated for individual resources.*

Hence, the $\mathcal{C}$ and $\mathcal{O}$ relations corresponding to the policy above would be as follows:

$$\begin{aligned} \mathcal{C} = \{&(\textit{resource-id-qualifier} = \textit{National Geographic}, \\ &\qquad \textit{resource-id-qualifier} = \textit{National Geographic}), \\ &(\textit{environment:current-day} = \textit{Weekday}, \textit{environment:current-day} = \textit{Weekday}), \\ &(\textit{subject:request-time} \geq 13:00:00, \textit{subject:request-time} \geq 13:00:00), \\ &(\textit{subject:request-time} \leq 14:00:00, \textit{subject:request-time} \leq 14:00:00), \\ &(\textit{edit\_count\_var} \leq 3, \textit{edit\_command\_count\_var} \leq 12), \\ &(status\_flag = edit, status\_flag = edit)\} \end{aligned}$$

$$\mathcal{O} = \{(\textit{edit\_count\_var++}, \textit{edit\_command\_count\_var++})\}$$

The fact that the editing counter has been assigned a maximum value of 12 is based on the assumption that an edit action is equal to the four commands read/write/create/delete. Hence, it is possible to perform a total of 12 read/write/create/delete commands within the maximum 3 of edits. This is slightly imprecise as it does not control the number of times each command is executed, only the total number for all the commands. However, it is safe as it permits a maximum of three correct editing processes. The more precise refinement of the condition would have required a policy enforcement point at the VO level, which would have a global view of the editing counter.

Now, we can define our policy refinement function as follows.

**Definition 3 (Policy Refinement Function).** *Define the policy refinement function as* $\mathcal{T}(Pol_{VO}, \mathcal{H}_{VO}, \mathcal{C}, \mathcal{O}) = Pol_C$*, which takes as parameters the VO policy defined in XACML,* $Pol_{VO}$*, a VO hierarchy tuple,* $\mathcal{H}_{VO} = (\mathcal{P}, \mathcal{U}, \mathcal{R}, \mathcal{E})$*, a pair of policy conditions* $\mathcal{C}$ *and policy obligations* $\mathcal{O}$ *mappings, and returns the corresponding XACML policy* $Pol_C$ *at the computational level of resources, as follows:*
$Pol_C = Pol_{VO}[$
$(< Element1 > x' < /Element1 > / < Element1 > x < /Element1 >),$
$(< Element2 >< AttributeValue > y' < /AttributeValue >< /Element2 > /$
$< Element2 >< AttributeValue > y < /AttributeValue >< /Element2 >)]$

*Where* $Element1 \in \{Condition, Obligation\}$ *and* $Element2 \in \{Resource, Action, Subject, Environment\}$*. As for the variable* $x'$*, it will have the value* $x' = \mathcal{C}(x)$ *when Element1 is "Condition" and* $x' = \mathcal{O}(x)$ *when Element1 is "Obligation". Similarly,* $y' = \mathcal{R}(y)$ *whenever Element2 is "Resource",* $y' = \mathcal{P}(y)$ *whenever Element2 is "Action",* $y' = \mathcal{U}(y)$ *whenever Element2 is "Subject" and* $y' = \mathcal{E}(y)$ *whenever Element2 is "Environment".*

Informally, the refinement applies a substitution to replace the resource, action, subject and environment attribute values in the VO policy with their values at the computational resource level taken from from the VO hierarchy relation as well as replace all the VO conditions and obligations by their corresponding computational siblings. The resulting policy now refers to resource concepts rather than the original VO concepts.

*Example 4.* Going back to the policy of Example 3, applying our refinement function, $\mathcal{T}$, yields the resource-level policy below.

```
<Policy PolicyId="Topography Data Set Resource Policy"
 <Rule Effect="Permit" RuleId="Editing Rule">
  <Target>
   <Resource>
    <AttributeValue>/geographic map/uk/topography data set
    </AttributeValue>
   </Resource>
   <Resource>
    <AttributeValue>/geographic map/uk/transportation data set
    </AttributeValue>
   </Resource>
 <Subject>
  <AttributeValue>abc1234</AttributeValue>
 </Subject>
 <Action>
  <AttributeValue>Create</AttributeValue>
 </Action>
 <Action>
```

```
  <AttributeValue>Delete</AttributeValue>
 </Action>
  <Action>
   <AttributeValue>Read</AttributeValue>
  </Action>
  <Action>
   <AttributeValue>Write</AttributeValue>
  </Action>
  <Environment>
  </Environment>
   </Target>
   <Condition>
  (resource-id-qualifier = National Geographic) AND
  (13:00:00 =< subject:request-time =< 14:00:00) AND
  (environment:current-day = weekday) AND
  (edit_command_count_var =< 12) AND
  ((status_flag = edit))
  </Condition>
 </Rule>
 <Obligation FulfillOn="Permit">
  (edit_command_count_var := edit_command_count_var + 1)
 </Obligation>
</Policy>
```

## 5 The ATL-Based Policy Refinement Engine

### 5.1 The Atlas Transformation Language (ATL)

The Atlas Transformation Language (ATL)[1] [12, 13] is a rule-based transformation language and toolkit that can be used to transform one model to another. In Model-Driven Engineering (MDE), models are considered to be first-class entities that can be manipulated using any functions, such as transformation functions. ATL provides an environment for developing model transformation functions as part of the Model-to-Model (M2M) project[2]. The general assumption of model transformations is that models themselves can be modeled. This is done through the concept of *meta-models*.

Transformations consist of rules, each dealing with specific elements of the input model. For example, in the case of our first transformation from VO policies to the resource representation, the following rule transforms an XACML *Policy* element:

```
rule Policy2Policy {
from v : VOPOLICY!Policy
to   r : RESOURCEPOLICY!Policy (
PolicyId <- v.PolicyId,
RuleCombiningAlgId <- v.RuleCombiningAlgId,
Version <- v.Version,
Rule <- v.Rule,
Target <- v.Target)}
```

The rule searches for any elements of type Policy in the input model, VOPOLICY, and transforms them to elements of type Policy in the output model, RESOURCEPOLICY. This is done by calling the relevant rules for transforming all the sub-elements of Policies, such as *PolicyId*, *RuleCombiningAlgId*, *Version*, *Rule* and *Target* elements, to their corresponding elements in the output model.

[1] www.eclipse.org/atl

[2] www.eclipse.org/m2m

Two ATL transformations were developed as an implementation of our policy refinement algorithm, one for transforming VO policies to their resource representation and one for transforming the resulting policies to a deployable form.

### 5.2 The VO2RESOURCE Policy Transformation

The VO2RESOUREC transformation consists of rules for transforming XACML 2.0 elements refering to VO concepts to the same elements refering to their corresponding resource-level concepts. The transformation requires a VO-Resource hierarchy input as well as the XACML policy to be transformed. As an output, it generates the transformed policy. For example, applying the rule:

```
rule AttributeValue2AttributeValue {
from    u : String, v: VOPOLICY!AttributeValue
to      r: RESOURCEPOLICY!AttributeValue
( DataType <- v.DataType, Text <- u)}
```

to the element,

```
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI"
Text="/geographic map/uk"></AttributeValue>
```

and given the VO-Resource hiearchy information,

```
<VOResource resName="/geographic map/uk">
<compResource resString = "/geographic map/france/file_uk"></compResource>
</VOResource>
```

yields the following element:

```
<AttributeValue Text="/geographic map/uk/file_uk"
DataType="http://www.w3.org/2001/XMLSchema#anyURI"/>
```

with the outcome that it transformed the map of UK (VO resource) to the file representing that map (computational resource).

### 5.3 The RESOURCE_POLICY_DEPLOYMENT Transformation

The RESOURCE_POLICY_DEPLOYMENT transformation *deploys* the results of the VO2RESOURCE transformation. For example, applying the transformation to the resource policy of Example 4 would result in the following two "deployable" policies,

```
<PolicySet xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns="RESOURCEPOLICY"
PolicySetId="DM VO Policy(The Resource Version)">
  <Policy PolicyId="VO Policy21">
   <Rule RuleId="VO Publishing Rule" Effect="Permit">
    <Target>
    <Resource>
      <AttributeValue>/geographic map/uk/transportation data set</AttributeValue>
    </Resource>
  </Target>
   </Rule>
  </Policy>
    <Policy PolicyId="VO Policy22">
   <Rule RuleId="VO Publishing Rule" Effect="Permit">
    <Target>
    <Resource>
      <AttributeValue>/geographic map/uk/topography data set</AttributeValue>
    </Resource>
    </Target>
   </Rule>
  </Policy>
</PolicySet>
```

which refer to separate resources without the risk of interfering with each other.

### 5.4 Interfacing with the Engine

Our policy refinement engine could have several methods of interfacing, which in general supply the XACML policy and the VO-Resource hierarchy information. For example, the policy could be generated by means of an XACML editor, like the UMU editor[3] or directly using any XML editor with the XACML schema. The VO-Resource hierarchy itself could be generated from any XML editor, however, in [14], an Eclipse-based graphical user interface tool was designed to assist the user in creating the hierarchy and then generating its corresponding XML representation.

## 6 Related Work

Policy refinement is a topic that covers a wide area of research among the security policies community incorporating various approaches [5, 15–19] with various degrees of success.

The work of [5] define the notion of resource type hierarchies, which is a generalisation of our VO-Resource hierarchies extending $n$ number of layers. In our case, we have only considered a 2-layered approach, but this can easily be generalised by re-applying our refinement transformations to each new layer generated by the previous one.

In [15], the authors propose a policy refinement methodology based on system goals. Systems are modelled formally in the Event calculus [20] and then abductive reasoning techniques are used in deriving the sequences of operations the system must perform in order to achieve a specific goal. This methodology is proposed as an approached for providing tool support and partial automation of policy refinement. The main difference from our work is that [15] propose their own specification language [21], whereas here we focus on the refinement of a policy standard (i.e. XACML). A main advantage of our refinement over both [5, 15] is that it can lead to deployable policies referring to at most one resource, thus avoiding cross-domain interference.

Similar to [15], the authors in [16] use goal-oriented engineering methodologies for guiding the policy refinement process. In [17], the authors provide an approach for deriving low-level policies from high-level service-level agreement goals using a combination of data classification and test and development approaches, whereas in [19], the authors deal with the modeling and refinement for network security systems domain. Finally, in [18], the authors utilise semantic web ontologies for guiding the policy refinement process. The use of ontologies helps the integration of business/service-level concepts with network-level concepts, which is similar to our VO-Resource hierarchies.

## 7 Conclusion and Future Work

This paper has presented an approach for the refinement of XACML policies based on the model to model transformation methodology. The transformation allows for VO-level policies to be transformed to their corresponding resource-level forms, and then to be deployed to the individual resources. A refinement engine was developed as a

[3] `sourceforge.net/projects/umu-xacmleditor/`

plugin in the Eclipse environment. The main benefit of this approach is to aid XACML policy designers and administrators in automating the derivation and deployment of low-level resource policies from high-level VO policies in the context of collaborative environments, such as virtual organisations.

Our approach (and the resulting refinement engine) can be improved along many directions. The main objective will be to support obligation refinement in the tool, in particular since these form the main feature of the upcoming XACML 3.0 specification. Another future improvement would be to utilise the modularity of the Eclipse environment to combine the refinement engine with policy-derivation tools, in particular [8], which can then automate the process of deriving low-level deployable policies from business goals without the need for managing intermediate VO policies. Finally, a planned extension to our approach is to target other resource hierarchies in emerging distributed architectures, mainly virtualised service platforms and Cloud federations.

## References


1. Arenas, A.E., Wilson, M., Matthews, B.: On Trust Management in Grids. In: International Conference on Autonomic Computing and Communication Systems, Autonomics 2007. ACM, New York (2007)
2. Wasson, G.S., Humphrey, M.: Toward Explicit Policy Management for Virtual Organisations. In: 4th IEEE Int. Workshop on Policies for Distributed Systems and Networks (2003)
3. Aziz, B., Arenas, A.E., Martinelli, F., Matteucci, I., Mori, P.: Controlling Usage in Business Process Workflows through Fine-Grained Security Policies. In: Furnell, S.M., Katsikas, S.K., Lioy, A. (eds.) TrustBus 2008. LNCS, vol. 5185, pp. 100–117. Springer, Heidelberg (2008)
4. Moffett, J.D., Sloman, M.S.: Policy hierarchies for distributed system management. IEEE Journal of Selected Areas in Comms., Special Issue on Network Management 11(9) (1993)
5. Su, L., Chadwick, D.W., Basden, A., Cunningham, J.A.: Automated decomposition of access control policies. In: Sixth IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY 2005, pp. 3–13. IEEE, Los Alamitos (2005)
6. GridTrust: Deliverable D5.1(M19): Specifications of Applications and Test Cases (2007)
7. Aziz, B., Arenas, A.E., Cortese, G., Crispo, B., Causetti, S.: A Secure and Scalable Grid-based Content Management System. In: 5th International Workshop on Frontiers in Availability, Reliability and Security, FARES 2010. IEEE Computer Society, Los Alamitos (2010)
8. Landtsheer, R.D., Ponsard, C., Massonet, P.: Deriving Event-Based Usage Control Policies from Declarative Security Requirements Models. In: Second International Workshop on Security in Model Driven Architecture, Paris, France (2010)
9. Lupu, E., Sloman, M.: Conflict Analysis for Management Policies. In: Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management V: Integrated Management in a Virtual World, London, UK, pp. 430–443. Chapman & Hall, Ltd., Boca Raton (1997)
10. Wasson, G.S., Humphrey, M.: Policy and Enforcement in Virtual Organizations. In: GRID, pp. 125–133. IEEE Computer Society, Los Alamitos (2003)
11. Moses, T. (ed.): eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard (2005)
12. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
13. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Mmodel Rransformation Tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)

14. GridTrust: Deliverable D4.1: A Framework for Reasoning about Trust and Security in Grids at Requirement and Application Levels (2009)
15. Bandara, A.K., Lupu, E.C., Moffett, J., Russo, A.: A Goal-based Approach to Policy Refinement. In: Proceedings of the Fifth IEEE Int. Workshop on Policies for Distributed Systems and Networks, Washington, DC, USA, p. 229. IEEE Computer Society, Los Alamitos (2004)
16. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G., Lafuente, A.L.: Using Linear Temporal Model Checking for Goal-Oriented Policy Refinement Frameworks. In: Proceedings of the Sixth IEEE Int. Workshop on Policies for Distributed Systems and Networks, Washington, DC, USA, pp. 181–190. IEEE Computer Society, Los Alamitos (2005)
17. Udupi, Y.B., Sahai, A., Singhal, S.: A Classification-Based Approach to Policy Refinement. In: Integrated Network Management, pp. 785–788 (2007)
18. Guerrero, A., Villagrá, V.A., de Vergara, J.E.L., Sánchez-Macián, A., Berrocal, J.: Ontology-Based Policy Refinement Using SWRL Rules for Management Information Definitions in OWL. In: State, R., van der Meer, S., O'Sullivan, D., Pfeifer, T. (eds.) DSOM 2006. LNCS, vol. 4269, pp. 227–232. Springer, Heidelberg (2006)
19. Porto de Albuquerque, J., Krumm, H., Licio de Geus, P.: Policy Modeling and Refinement for Network Security Systems. In: Proceedings of the Sixth IEEE Int. Workshop on Policies for Distributed Systems and Networks, Washington, DC, USA, pp. 24–33. IEEE Computer Society, Los Alamitos (2005)
20. Kowalski, R., Sergot, M.: A Logic-Based Calculus of Events. New Gen. Comput. 4(1), 67–95 (1986)
21. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)

# Automatic Conformance Checking of Role-Based Access Control Policies via Alloy

David Power, Mark Slaymaker, and Andrew Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD
United Kingdom

**Abstract.** Access control policies are a crucial aspect of many security-critical software systems. It is generally accepted that the construction of access control policies is not a straightforward task. Further, any mistakes in the process have the potential to give rise both to security risks, due to the provision of inappropriate access, and to frustration on behalf of legitimate end-users when they are prevented from performing essential tasks. In this paper we describe a tool for constructing role-based access control (RBAC) policies, which are automatically checked for conformance with constraints described using predicate logic. These constraints may represent general healthiness conditions that should hold of all policies conforming to a general model, or capture requirements pertaining to a particular deployment.

## 1 Introduction

Access control is at the heart of secure data sharing, yet the construction of access control policies is often a time-consuming and error-prone activity. Getting it wrong can frustrate users and result in inappropriate disclosure of information. As such, *correctness* is important; to quote [1]: "the correctness and integrity of access control policies is crucial for an access control system to be effective." Certainly, the challenges are likely to become more pressing in the near future as greater consideration is given to, for example, insider threats [2], distributed, heterogeneous systems [3] and cloud-like environments [4]. Of course, due to the undecidability of the *safety problem* of [5], there are limitations as to what we might assert to be true; nevertheless, there will often be certain healthiness conditions—such as the enforcement of separation of duty constraints—that we can verify. There are also other constraints which may be appropriate in certain situations, such as the absence (or presence) of a user holding all permissions or all users having at least one role. In addition, redundant roles or permission allocations can be detected. Even if *proof* is not possible in many contexts, it is certainly the case that greater *assurance* can be provided.

In [6], the distinction between access control policies ("high-level rules according to which access must be regulated"), models ("formal representations of policies"), and mechanisms ("the low-level functions that implement the controls of the policy, which are represented by models") is made—with these distinctions

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 15–28, 2011.

between abstractions being essential in considering correctness: "by proving that the model is secure and the mechanism correctly implements the model, we can argue that the system is secure (with respect to the definition of security considered)." It is in this spirit that we describe a tool for constructing role-based access control (RBAC) policies (see, for example, [7]), which can be automatically checked for conformance with constraints (which may represent general healthiness conditions that should hold of all policies conforming to a general model, or capture requirements pertaining to a particular deployment) described using predicate logic. The tool creates instances of a model described using the Alloy modelling language [8,9]; these instances are then checked against user configurable constraints via the Alloy libraries. While the Alloy Analyzer is typically used for finding small instances of models that meet a set of criteria, it is also capable of analysing static instances of models of a much greater size via an evaluator normally used as part of the visualisation component. The focus of this paper is primarily the integration with, and use of, Alloy, rather than the design and implementation of the tool *per se*.

Our work is driven by practical concerns. Our middleware framework, *sif* (for service-oriented interoperability framework) [10,11,12], has been used to facilitate the sharing and aggregation of data in a wide variety of application contexts, including various healthcare-related appications [13]. Typically, the data that is shared is of a sensitive nature and it is essential that data owners have some degree of confidence in the policies that control access to their resources.

In Section 2 we describe the background to our work. In Section 3 we describe the policy construction tool. In Section 4 we describe an Alloy model of RBAC and give examples of constraints and reporting functions. In Section 5 we give a formal description of Alloy instances and also describe an alternative JavaScript Object Notation (JSON)[1] representation of instances. In Section 6 we give an example of the construction of a policy for access to databases within a research hospital and show the use of constraints both to detect errors and to remove redundancies. Finally, in Section 7, we describe alternative uses of the general technique used by the tool and outline areas of future work with respect to the consideration of the efficient handling of large instances.

## 2 Background

### 2.1 RBAC

The underlying principle upon which role-based access control is based is the association of permissions with the roles that users may hold within an organisation. There are four standard components in the ANSI standard for role-based access control systems [14].

– *Core RBAC* is mandatory in any RBAC system, and associates permissions with roles and roles with users.

[1] `http://www.json.org`

– Any combination of the following can be utilised in a particular system.
  1. *Role hierarchies* define what amounts to an inheritance relation between roles. As an example, role $r_1$ inherits from role $r_2$ if all privileges associated with $r_2$ are also associated with $r_1$.
  2. A *static separation of duty* (SSD) constraint is characterised by a role set, $rs$, such that $\# rs \geq 2$, and a natural number, $n$, such that $2 \leq n \leq \# rs$, and ensures that no user can be authorised for $n$ or more roles in $rs$.
  3. A *dynamic separation of duty* (DSD) constraint differs from a SSD constraint in that it is concerned with sessions. Again, given a role set, $rs$, such that $\# rs \geq 2$, and a natural number, $n$, such that $2 \leq n \leq \# rs$, a DSD constraint ensures that no user can be associated with $n$ or more roles in $rs$ in a particular session.

While other characterisations, extensions and notations for role-based access control (such as, for example, Organisation Based Access Control (OrBAC) [15]) exist, we build upon an existing formal model of the ANSI standard [16].

### 2.2 Alloy

The Alloy modelling language [8] is a lightweight object modelling notation based on first order predicate logic. The Alloy Analyzer provides visualisation and model finder facilities for Alloy models. Although models are built using object-oriented concepts, the logic of the language is entirely based on relations.

A relational model finder [17] is used to analyse alloy models within a bounded scope. In most cases a small scope will be sufficient to find a counter-example if one exists; this is known as the *Small Scope Hypothesis*. The model finder produces instances of relations which can be viewed using the visualisation component of the Alloy Analyzer.

Although a full description of the Alloy language is beyond the scope this paper (see [9] for a full description), we will briefly describe some of the more pertinent parts of the language.

The basic set operations union `(+)`, intersection `(&)` and set difference `(-)` are all supported. The relational product operator `(->)` takes two relations of arity `n` and `m` and produces a relation of arity `n+m` by combining all possible pairs of tuples. The composition operator `(.)` takes two relations of arity `n` and `m`, and produces a relation of arity `n+m-2` by combining tuples where the last atom in the left-hand tuple matches the first atom in the right-hand tuple, with the matching atoms being discarded. Homogenous relations can be used with the transitive closure `(^)` and reflexive transitive closure `(*)` operators.

Basic formulae include the constants `true` and `false`, as well as the basic logic operators conjunction `(&&)`, disjunction `(||)` and negation `(!)`. As relations are the only data type, there is no set membership operator—the subset operator `(in)` is used instead. There are also multiplicity logic operators, including `no` and `one`, stating that relations contain no tuples or exactly one tuple respectively; alternatively, the cardinality operator `(#)` can be used. Quantifiers

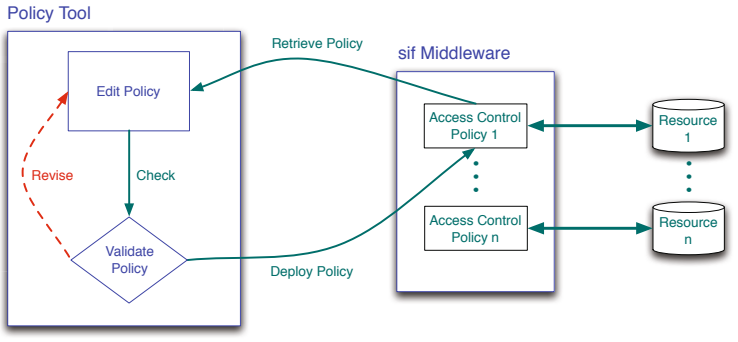



**Fig. 1.** Policy construction and verification

include `some` and `all` and have the form `quantifier declaration | formula`. Set comprehensions are of the form { `declaration | formula` }.

Models are built up using signatures, which have the following general form: `sig A extends B { fields } { constraints }`. Here, a new type `A` is created that has all the fields and constraints of `B` plus those in its declaration. Named facts can be added thus: `fact name { constraint }` (facts are assumed to always hold). Finally, a function, `name`, with inputs `in` and outputs `expression` can be defined by `fun name(in) : out { expression }`, where `out` is the type of the result describe by `expression`.

## 3 RBAC Policy Tool

The RBAC policy tool is a Java application that allows the editing, validation and uploading of RBAC policies to deployments of the aforementioned sif middleware.

The permission to update policies is itself subject to a server-level policy which cannot be directly edited. The overall 'workflow' is shown in Figure 1; the user interface of the tool is shown in Figure 2.

The interface is divided into a number of panes, each of which captures a single aspect of the RBAC policy. The first few panes facilitate the collection of basic information about users, roles, actions, resources and permissions—with each permission being a combination of a single action and a single resource.

The user-role pane (shown in Figure 2) allows roles to be allocated to users; the role-permission pane allocates permissions to roles. Role hierarchies are built using the role-role pane, where one role is set to be senior to another: a senior role inherits all of the permissions of the junior role in addition to its own.

The SMER[2] pane forms part of the constraints on the policy: it allows the selection of a number of roles and a limit on the number of those roles a user

[2] The term *Static Mutually Exclusive Roles* (SMER) indicates a *Static Separation of Duty* (SSD) in the manner of [16].

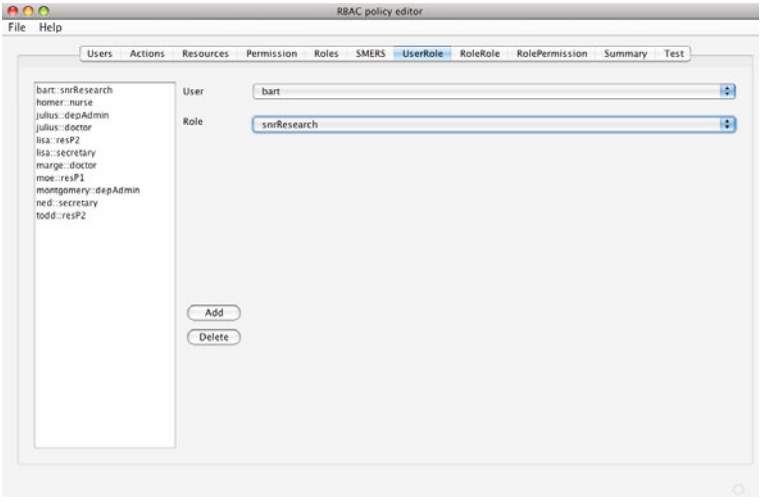



**Fig. 2.** RBAC policy tool

may hold.[3] The testing pane (described in more detail in Section 6) checks the constraints on the policy. We now consider what constitutes a valid policy—which we do via our Alloy model.

## 4 An Alloy Representation of RBAC

This section utilises the Alloy modelling language [9] to formally define a model of RBAC. We present a model for core RBAC and hierarchal RBAC, as well as static separation of duty constraints. The model is based on that of [16], which gives a formal description of RBAC using the Z formal description language [18].

First, we define `User`, `Role`, `Action` and `Resource`, with `Action` and `Resource` being used to define the contents of a `PRMSBase`—which represents a permission.

```
sig User, Role, Action, Resource {}

sig PRMSBase {
  action : Action,
  resource : Resource
}
```

The `SMER` signature consists of an integer, `limit`, and a set of roles, `roles`. The `SMER` signature also has a constraint, which states that the value of `limit` ranges between 2 and the size of the `roles` set. By convention, there is an implicit `(&&)` between the two lines of the constraint.

[3] The current version of the tool does not support sessions or dynamic separation of duty constraints.

```
sig SMER {
  limit : Int,
  roles : set Role
} {
  2 <= limit
  limit <= (# roles)
}
```

The fact `uniquePRMSBase` ensures that each `PRMSBase` is unique, that is to say, that no two different (signified by `disj`) elements of `PRMSBase` have the same action/resource pair. This simplifies the subsequent definitions in the RBAC model. It should be noted that the use of `(.)` for field selection is consistent with the underlying representation of fields in which `action` is a binary relation between `PRMSBase` and `Action`.

```
fact uniquePRMSBase{
  all disj pb1, pb2 : PRMSBase |
    pb1.action != pb2.action || pb1.resource != pb2.resource
}
```

The `Core` signature represents an RBAC system without a role hierarchy but with static separation of duty constraints. It contains the sets `USERS`, `ROLES` and `PRMS`, which represent the particular users, roles and permissions the policy relates to. It also contains the relations `UA` and `PA`, which represent the user-role and role-permission mappings respectively. The set `SC` is a set of SMER constraints, the roles of which must be a subset of `ROLES`. The restriction on the roles a user may hold due to `SC` is described by the last line of the constraint. The composition `UA.PA` can be used to relate users to their granted permissions.

```
sig Core {
  UA : User -> Role,
  PA : Role -> PRMSBase,
  SC : set SMER,
  USERS :  set User,
  ROLES : set Role,
  PRMS : set PRMSBase
} {
  UA in USERS -> ROLES
  PA in ROLES -> PRMS
  all s : SC | s.roles in ROLES
  all s : SC | all u : USERS | #(s.roles & u.UA) < s.limit
}
```

The signature `Hierarchy` extends `Core` by inheriting all of its fields and constraints and adding one extra field, `RH` (the role hierarchy). This relation must be acyclic, and this is ensured by `no (^RH & iden)`, where `iden` is the identity relation. In this model, it is assumed that inherited roles are also restricted by the SMER constraints; for that reason, the `u.UA` is changed to `u.UA.*RH` in what is

an additional constraint on the roles a user may hold. Similarly, the composition `UA.*RH.PA` can be used to relate users to their permissions.

```
sig Hierarchy extends Core {
  RH: Role -> Role
} {
  RH in ROLES -> ROLES
  no (^RH & iden)
  all s : SC | all u : USERS |
    #(s.roles & u.UA.*RH) < s.limit
}
```

The model that has been described up to this point represents the compulsory parts of any RBAC policy produced by the tool. We now describe a number of optional constraints that can be used to validate policies. The applicability of each constraint will depend, of course, upon the context of the deployed policy.

The first example is a constraint on any individual user having all permissions, which is represented as a fact called `NobodyCanDoEverything` affecting all elements of `Hierarchy`. This fact could have been included in the signature of `Hierarchy`, but is written as a separate fact both to promote modularity and (consequently) to allow it to be tested independently.

```
fact NobodyCanDoEverything {
  all h : Hierarchy |
    all u : h.USERS | u.(h.UA).*(h.RH).(h.PA) != h.PRMS
}
```

If it is found that this fact does not hold, then the policy tool calls a function called `fun_NobodyCanDoEverything`, which reports all the `Hierarchy` and `User` pairs that breach the constraint. By creating facts and functions using this naming convention, the tool can automatically pick up new constraints and report any breaches without prior knowledge of their existence.

```
fun fun_NobodyCanDoEverything() : Hierarchy -> User {
  { h : Hierarchy , u : User |
    u in h.USERS && u.(h.UA).*(h.RH).(h.PA) = h.PRMS }
}
```

Similar facts are `NobodyHasAllRoles` and `EverybodyCanDoSomething`.

While giving a user all permissions could be considered an error, there are also constraints relating to redundancy in the model. One such example is `NoRedundantPermissions`, which restricts a role from being assigned a permission that it already holds due to inheritance.

```
fact NoRedundantPermissions {
  all h : Hierarchy |
    all r : h.ROLES | no (r.(h.PA) & r.^(h.RH).(h.PA))
}
```

```
fun fun_NoRedundantPermissions() :
  Hierarchy -> Role -> PRMSBase {
  { h : Hierarchy , r : Role , p : PRMSBase |
    r in h.ROLES && p in h.PRMS &&
    p in (r.(h.PA) & r.^(h.RH).(h.PA)) }
}
```

The facts `AllRolesHaveAPermission` and `AllPermissionsReachable` may also be considered as redundancy constraints.

## 5 Instances

The Alloy Analyzer finds instances of models that meet a set of criteria, with each instance containing values for the relations defined in the model. These instances are typically visualised graphically; however, more detailed analysis can be performed using the evaluator. Using the evaluator, it is possible to use the predicates and functions defined in the Alloy model to construct expressions and formulae; these are then evaluated using the values of the relations in the current instance. An instance consists of two types of relations: *signature relations* and *field relations*. Signature relations are unary and contain the atoms of a specific signature; field relations have an arity of 2 or more and represent the fields of a signature, with the arity of a field relation being 1 more than that of the field it represents (as the first element of each tuple specifies the particular atom to which it relates).

There is an XML format for instances that contain elements for signature and field relations, as well as other information such as the command that was used to create the instance, values of quantified variables and details of the model. The RBAC tool does not create instances in this format directly as it contains significant typing information and requires the explicit calculation of signature and field relations. Rather, a JSON format is used in which the fields of each object are described in isolation. Shown below is a JSON fragment that defines the roles `Doctor` and `DepAdmin`, together with a SMER constraint that prevents a user from holding both roles.

```
{ id : Doctor , type : Role },
{ id : DepAdmin , type : Role },
{ id : DoctorOrDepAdmin , type : SMER ,
  fields : {
    limit : [[2]] ,
    roles : [[Doctor],[DepAdmin]]}
}
```

This is treated as being equivalent to the following Alloy statements.

```
one sig Doctor extends Role {}
one sig DepAdmin extends Role {}
```

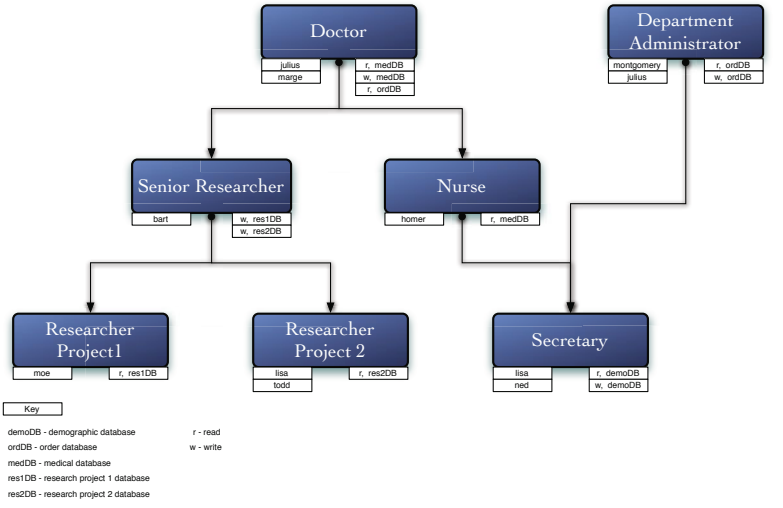



**Fig. 3.** Initial hierarchy

```
one sig DoctorOrDepAdmin extends SMER {} {
  limit = 2
  roles = Doctor + DepAdmin
}
```

To perform constraint checking, the Alloy model is first parsed using a modified version of the parser used by the Alloy Analyzer. By combining the type information from the Alloy model with the JSON file, it is then possible to calculate the signature and field relations and create a standard XML instance file. The XML instance file is then loaded using standard Alloy API calls to create a solution object that can be used to perform evaluations.

Using the parsed Alloy model, the tool can list and evaluate all of the facts. It is, of course, possible that the created instance is not valid, in which case some of the facts may be false—possibly including facts associated with the signature declarations. It should be noted that the Alloy Analyzer would never create such an instance; however, that does not stop the libraries from evaluating them.

If a failed fact has an associated function in the model, the policy tool will evaluate it and display the results. Hopefully this will indicate the source of the problem. Should further analysis be required, it is possible to export the instance as a standard Alloy model populated with `one sig` declarations as above.

## 6 Example

In this section we present an example of an RBAC system that is required to regulate access to a number of databases in a research hospital. This (much simplified) example is based on a real scenario associated with Diabetes data,

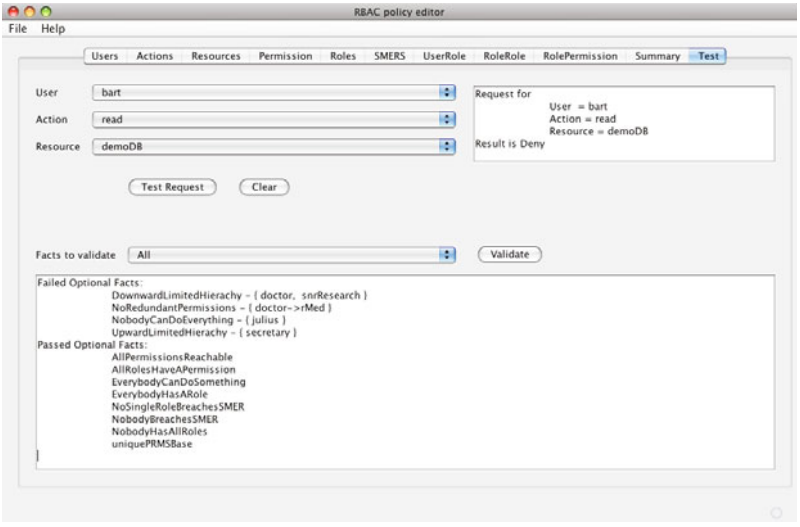



**Fig. 4.** RBAC validation

which was supported as part of the GIMI (Generic Infrastructure for Medical Informatics) project [13].

There are five databases in total (demographic, order, medical, research 1 and research 2)—with each database being represented as a resource in the model. Read access and write access are represented as actions. There are a total of seven roles which form the role hierarchy illustrated in Figure 3.

The JSON representation of the policy is given below.

```
{ id : examplerbac , type : Hierarchy,
  fields : {
    USERS : [[lisa],[homer],[montgomery],[moe],[marge],[bart],[todd],
             [ned],[julius]],
    ROLES : [[depAdmin],[doctor],[snrResearch],[resP2],[resP1],[nurse],
             [secretary]],
    PRMS : [[rDemo],[wDemo],[rOrd],[wOrd],[rMed],[wMed],[rRes1],[wRes1],
            [wRes2],[rRes2]],
    UA : [[bart,snrResearch],[julius,doctor],[marge,doctor],[moe,resP1],
          [lisa,resP2],[todd,resP2],[homer,nurse],[lisa,secretary],
          [ned,secretary],[montgomery,depAdmin],[julius,depAdmin]],
    RH : [[depAdmin,secretary],[doctor,nurse],[doctor,snrResearch],
          [snrResearch,resP1],[snrResearch,resP2],[nurse,secretary]],
    PA : [[depAdmin,rOrd],[secretary,rDemo],[doctor,rMed],[doctor,rOrd],
          [doctor,wMed],[snrResearch,wRes2],[resP2,rRes2],[depAdmin,wOrd],
          [snrResearch,wRes1],[secretary,wDemo],[resP1,rRes1],[nurse,rMed]],
    SC : []}
}
```

The results of the analysis are shown in the lower panel in Figure 4. Four facts failed, two of which (`NobodyCanDoEverything` and `NoRedundantPermissions`) were described in Section 4. It can be seen that Julius has all permissions due to the fact that he holds both the doctor and departmental administrator roles. Subsequently, this has been discovered to be an error and Julius should only

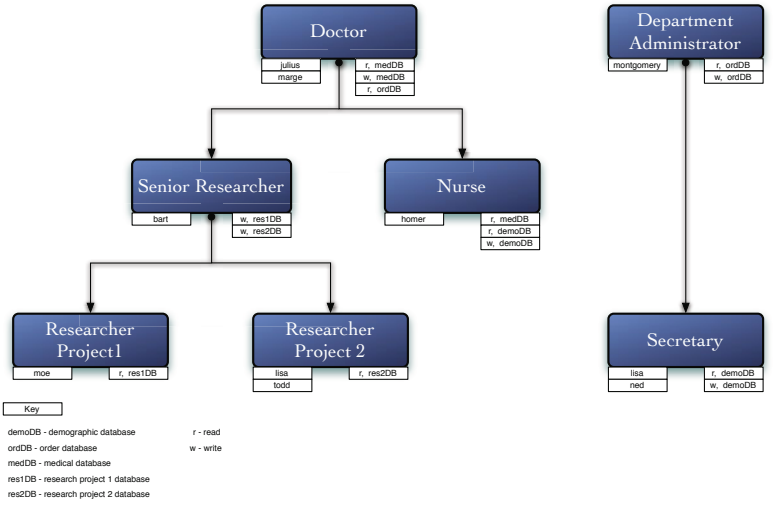



**Fig. 5.** Alternative hierarchy

have the role of doctor; by adding the `DoctorOrDepAdmin` SMER constraint of Section 5 we could prevent such a mistake happening again. The doctor role has the permission to read the medical database both directly and through the nurse role—it would be safe to remove this permission from the doctor role if one wished to minimise the size and complexity of the policy.

The two limited hierarchy facts refer to the structure of the role hierarchy. Further, both rely on the concept of an immediate successor, which can be calculated from the `RH` relation. In an upward limited hierarchy, a role cannot be an immediate successor to two or more roles; in a downward limited hierarchy, two roles cannot be the immediate successor of the same role. The fact related to upward limited hierarchies is written below, where `let` statements are used to declare local constants and `no` is used as a quantifier.

```
fun ImmediateSuccessor(RH : Role -> Role) : Role -> Role  {
  let succ = *RH |
  { disj r1 , r2 : Role | r1 -> r2 in succ &&
    (no r3 : Role - (r1 + r2) |
      r1 -> r3 in succ && r3 -> r2 in succ) }
}

fact UpwardLimitedHierachy {
  all h : Hierarchy |
    let imm = ImmediateSuccessor[h.RH] |
      no disj r1 , r2 , r3 :  Role | (r2 + r3) -> r1 in imm
}
```

From Figure 4, it can be seen that the secretary role is immediate successor to both the nurse and departmental administrator roles and hence breaches

the above fact. Similarly, both the doctor and senior researcher roles have two immediate successors and breach the `DownwardLimitedHierarchy` fact. If we wished to have an upward limited hierarchy, then we could refactor the policy so that the nurse role was no longer senior to the secretary role. To maintain the equivalence of permissions, the nurse role would need to gain the permissions of reading and writing the demographic database. A refactored policy along these lines is shown in Figure 5.

## 7 Discussion

We have described a tool for the creation and testing of RBAC policies. By using the Alloy modelling language, it has been possible to formally define a range of validity constraints which can be checked automatically within the tool. While the tool we have described is used to create and verify RBAC policies, the general technique could be used for verifying the integrity of any access control policy language for which an Alloy model can be written. Other policy languages for which we have created such models include the Amazon Web Services (AWS) access policy language [19] and the eXtensible Access Control Markup Language (XACML) [20]. The tool could also be extended to support the comparison of policies, which would be particularly useful when refactoring: it would then be possible to verify that the two policies of Section 6 were indeed equivalent.

Other authors who have considered the design and analysis of role-based policies include Crampton [21] and Hu and Ahn [22]. More broadly, there is a great deal of ongoing work with respect to the formal design and analysis of access control policies, with that of [23], [24] and [25] being of particular note. Our work is driven by practical concerns, with a key focus being the integration of formal analysis into the processes associated with the construction and deployment of access control policies. While cognisant of the fact that there are limits on establishing provable correctness, we endeavour to provide greater *assurance* to policy writers and data owners as to the *appropriateness* of policies.

While the model finder used by Alloy is capable of dealing with the case when the relations are all fixed, it still is restricted by internal data structures which put a limit on the total number of atoms of approximately $2^{31/n}$ (where $n$ is the largest arity of any relation). The `UA`, `PA` and `RH` field relations are all of arity 3, which imposes a practical limit of approximately 1000 total atoms[4]. One solution to this would be to use a simpler relational engine (such as RelView [26] or CrocoPat[5]) that can cope with problems of larger scope. Another area of immediate interest is the modelling of dynamic state, such as session data. Dynamic state may also be subject to constraints, such as dynamic separation of duty. Such constraints would be best supported by coupling constraint checking and dynamic data storage into a single component, so that it is possible to ensure that policy constraints are not breached by changes in the dynamic state.

[4] This is theoretical limit, as we are yet to encounter a policy which requires 1000 atoms.
[5] `http://www.sosy-lab.org/~dbeyer/CrocoPat`

**Acknowledgements.** This work was undertaken as part of the JISC-funded SOFA (Service-Oriented Federated Authorization) project.

## References

1. Zhang, N., Ryan, M., Guelev, D.: Evaluating access control policies through model checking. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 446–460. Springer, Heidelberg (2005)
2. Crampton, J., Huth, M.: Towards an access-control framework for countering insider threats. In: Bishop, M., Gollman, D., Hunker, J., Probst, C. (eds.) Insider Threats in Cyber Security and Beyond. Springer, Heidelberg (2010)
3. Bertino, E., Crampton, J.: Security for distributed systems: Foundations of access control. In: Qian, Y., Tipper, D., Krishnamurthy, P., Joshi, J. (eds.) Information Assurance: Survivability and Security in Networked Systems, pp. 39–80. Morgan Kaufmann, San Francisco (2007)
4. Gouglidis, A., Mavridis, I.: On the definition of access control requirements for grid and cloud computing systems. In: Networks for Grid Applications. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 25, part 2, pp. 19–26 (2010)
5. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. Communications of the ACM 19(8), 461–471 (1976)
6. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Authorization and access control. In: Petkovíc, M., Jonker, W. (eds.) Security, Privacy, and Trust in Modern Data Management, pp. 39–53. Springer, Heidelberg (2007)
7. Ferraiolo, D.F., Kuhn, D.R., Chandramouli, R.: Role-based access control. Artech House Publishers, Boston (2003)
8. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering Methodologies 11(2), 256–290 (2002)
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
10. Simpson, A.C., Power, D.J., Russell, D., Slaymaker, M.A., Kouadri-Mostefaoui, G., Ma, X., Wilson, G.: A healthcare-driven framework for facilitating the secure sharing of data across organisational boundaries. Studies in Health Technology and Informatics 138, 3–12 (2008)
11. Slaymaker, M.A., Power, D.J., Russell, D., Wilson, G., Simpson, A.C.: Accessing and aggregating legacy data sources for healthcare research, delivery and training. In: Proceedings of the 2008 ACM Symposium on Applied Computing (SAC 2008), pp. 1317–1324 (2008)
12. Slaymaker, M.A., Power, D.J., Russell, D., Simpson, A.C.: On the facilitation of fine-grained access to distributed healthcare data. In: Jonker, W., Petković, M. (eds.) SDM 2008. LNCS, vol. 5159, pp. 169–184. Springer, Heidelberg (2008)
13. Simpson, A.C., Power, D.J., Russell, D., Slaymaker, M.A., Bailey, V., Tromans, C.E., Brady, J.M., Tarassenko, L.: GIMI: the past, the present, and the future. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 368, 3891–3905 (2010)
14. Ferraiolo, D.F., Sandhu, R.S., Gavrilla, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and Systems Security 4(3), 224–274 (2001)

15. El Kalam, A.A., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G.: Organization Based Access Control. In: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2003), Como, Italie (2003)
16. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On formalising and normalising role-based access control systems. The Computer Journal 52(3), 303–325 (2009)
17. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
18. Woodcock, J.C.P., Davies, J.W.M.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)
19. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On the modelling and analysis of amazon web services access policies. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, p. 394. Springer, Heidelberg (2010)
20. Slaymaker, M.A., Power, D.J., Simpson, A.C.: Formalising and validating RBAC-to-XACML translation using lightweight formal methods. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 349–362. Springer, Heidelberg (2010)
21. Crampton, J.: Specifying and enforcing constraints in role-based access control. In: Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), pp. 43–50 (2003)
22. Hu, H., Ahn, G.: Enabling verification and conformance testing for access control model. In: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT 2008), pp. 195–204 (2008)
23. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. International Journal on Software Tools for Technology Transfer 10(6), 503–520 (2007)
24. Zhang, N., Guelev, D.P., Ryan, M.: Synthesising verified access control systems through model checking. Journal of Computer Security 16(1), 1–61 (2007)
25. Becker, M.Y.: Specification and analysis of dynamic authorisation policies. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009), pp. 203–217 (2009)
26. Behnke, R., Berghammer, R., Meyer, E., Schneider, P.: RELVIEW - A system for calculating with relations and relational programming. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 318–321. Springer, Heidelberg (1998)

# Security Validation of Business Processes via Model-Checking*

Wihem Arsac[1], Luca Compagna[1],
Giancarlo Pellegrino[1], and Serena Elisa Ponta[1,2]

[1] SAP Research Sophia-Antipolis, 805 Avenue Dr M. Donat, 06250 Mougins, France
{wihem.arsac,luca.compagna,giancarlo.pellegrino,serena.ponta}@sap.com
[2] U. of Genova, Viale Causa 13, 16145 Genova, Italy
serena.ponta@dist.unige.it

**Abstract.** More and more industrial activities are captured through Business Processes (BPs). To evaluate whether a BP under-design enjoys certain security desiderata is hardly manageable by business analysts without tool support, as the BP runtime environment is highly dynamic (e.g., task delegation). Automated reasoning techniques such as model checking can provide the required level of assurance but suffer of well-known obstacles for the adoption in industrial systems, e.g. they require a strong logical and mathematical background. In this paper, we present a novel security validation approach for BPs that employs state-of-the-art model checking techniques for evaluating security-relevant aspects of BPs in dynamic environments and offers accessible user interfaces and apprehensive feedback for business analysts so to be suitable for industry.

## 1 Introduction

Business Processes (BPs) are at the core of many organizations and industrial sectors. While this increases business agility, flexibility and efficiency, it also requires BPs to be carefully designed and executed so to be sure that important Key Performance Indicators such as compliance with respect to regulations and directives, fraud prevention, end-user acceptance and confidence are kept high within organizations. From this, emerge critical security desiderata that have to be properly tackled within BP scenarios. Established standards for Business Process Management (BPM), in primis the Business Process Modeling Notation (BPMN),[1] recognize this necessity. Industrial BPM systems aim to enforce these assignments, but are of little help in guaranteeing business process analysts that the BP they defined fulfills the security desiderata. This, combined with the high dynamicity of the environments in which BPs are run (e.g., delegation of task assignment), makes these security desiderata assessment a very hard task for the business analyst. Though testing and tuning the BP under-design improves its

* This work was partially supported by the FP7-ICT Projects AVANTSSAR (no. 216471, www.avantssar.eu) and SPaCIoS (no. 257876, www.spacios.eu)

[1] http://www.omg.org/spec/BPMN/2.0

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 29–42, 2011.
Springer-Verlag Berlin Heidelberg 2011

final quality, they hardly provide the level of assurance required. Model checking [1] can provide such a higher level of assurance, as it allows for validating all the potential execution paths of the BP under-design against the expected security desiderata. Clear obstacles for the adoption of model checking techniques to validate security desiderata in industry systems include *(i)* the generation of the formal model on which to run the analysis as well as *(ii)* how to feed back the model checking results to a business analyst that is neither a model checking practitioner nor a security expert. In this paper we present a model-checking approach for the automatic validation of security-relevant aspects of BPs and we discuss how it can be integrated within industrial BPM systems and indeed used by business analysts via accessible user interfaces and apprehensive feedback. Thus our approach makes model checking techniques usable by business analysts and allows their integration in real industrial environments. As proof of concept we have implemented our approach as an Eclipse plug-in that has been integrated within the SAP NetWeaver Business Process Management system (NW BPM, [2]). Furthermore we tackle relevant security desiderata including data-related properties as they are of importance in the proper execution of the workflow. A business process example from the banking domain is used to assess the contributions of our work, i.e. *(i)* the viability of our security validation approach, *(ii)* a push-button technology featuring accessible user interfaces, *(iii)* the way we reduce the gap between high level industrially-suited languages for BPs and formal languages via an automatic translation, *(iv)* the way we validate security desiderata and in particular data-related properties under dynamic resource allocation. Considering strong market drivers such as compliance and customer acceptance, the advantage of our approach and tool clearly lies in increasing the quality, robustness and reliability of BP under-design, thus mitigating the risk of deploying non-compliant BP.

The paper is organized as follows. We start to set the motivation of our work by presenting a BP example from the banking domain and critical security desiderata in Section 2. In Section 3 we overview our approach and in Section 4 we show how to formally define BPs and security desiderata. Section 5 assesses our results on a real case study. In Section 6 we discuss the related work and we conclude in Section 7 outlining future research directions.

## 2 A Motivating Example

Let us consider a scenario where a BP analyst is asked to model a Loan Origination Business Process (LOBP) for a bank. To achieve such a business goal, one has to design all the sequences of tasks executed by business resources so as to perform the loan process. We assume the bank is running NW BPM to handle BPs and the SAP NetWeaver Identity Management (NW IdM) to operate a role-based access-control (RBAC, [3]) enhanced with delegation [4] on BPs. Thus the features of the scenario we consider rely on the expressiveness and behavior of the real system used to model it. The business analyst uses NW BPM to design the LOBP depicted in Figure 1. The LOBP aims to evaluate and possibly grant a customer request for a loan amount. This can be operated by the flow of tasks

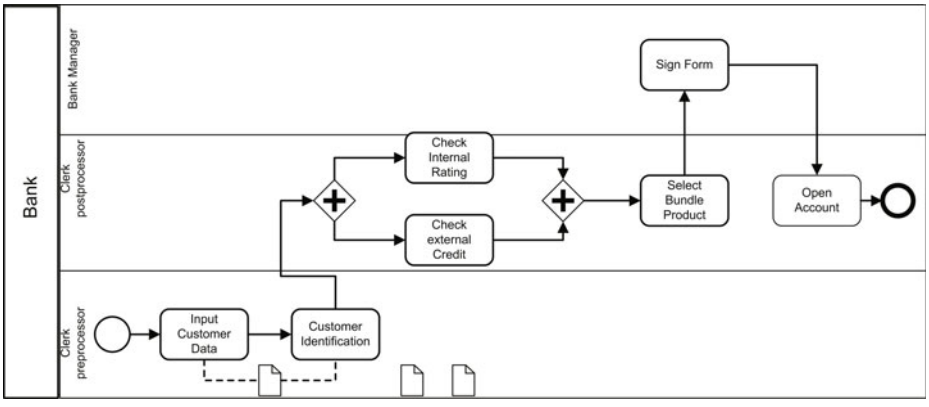



**Fig. 1.** The Loan process within the Bank

outlined hereafter. After receiving and identifying the customer, the bank carries out a careful evaluation of the customer's rating through internal mechanisms and by asking assurance to external agencies called credit bureaus. Subsequently, a bundled product is selected and, if both the customer and the bank agree, a contract is signed. The flow of human tasks that have to be manually realized by bank's users to achieve the LOBP's business goal are visible within the bank's pool. In there, three BPMN lanes capture the three main bank roles that are supposed to be active in the execution of the LOBP: the *pre-processors* who receive and identify the customer's request and update/create the customer data in the banking system (`Input Customer Data` and `Customer Identification` tasks); the *post-processor* who analyze the credit worthiness of the customer internally and externally via interaction with a credit bureau (`Check Internal Rating` and `Check External Credit Worthiness` tasks), identify the most appropriate bundle product (`Select Bundle Product` task), submit a loan proposal to the customer, and, provided that the contract is signed, they are the ones opening the customer bank account in the bank system (`Open Account in System` task); *the managers* who are responsible for the local agency of the bank and intervene to provide their approval to the loan (`Sign Form` task).

The assignment of roles to a user provides him/her with the authorizations needed to fulfill specific activities within an organization. The access control managing the execution of the process that we consider is based on RBAC enhanced with delegation. The bank's NW BPM is connected to the bank's NW IdM where roles, users, user-to-role assignments or delegations rules are defined for the bank organization. The information within NW IdM is exploited in two main ways by NW BPM. First, at design time, it is possible to associate to each human task of the BPMN model, a set of potential owners and a set of excluded owners where a owner is a principal, i.e. either a user or a role. Then, at runtime, a user can claim a task ready-to-be-executed if *(c1)* the potential owners of the task comprise either that user or a role to which he/she is assigned, and *(c2)* the excluded owners of the task do not comprise that user nor a role to

which he/she is assigned. The user that has claimed the task is normally the one that will execute it. If for some reasons he/she cannot execute the task, that user can delegate the task to a delegatee provided that the excluded owners of the task do not comprise the delegatee nor a role to which the delegatee is assigned. (This is normally referred to as delegation of execution [4].) It is thus clear that the environment in which the BP runs is highly dynamic. Notice that a more complex model of delegation could be considered, however we rely on the one in place in NW BPM, the real system on which we assess our approach. Though not visible in Figure 1, we can imagine that the business analyst has defined `preprocessor`, `postprocessor` and `manager` as potential owner of the lane pre-processing clerk, post-processing clerk and manager, respectively.

Another aspect of BPs is the data which are internally accessed to complete the tasks. For instance, in the LOBP, customer data profiles are created and stored in the bank system. The customer data is a data object, made of several fields, gathering all personal and financial information of a customer, including, e.g., the `name`, `gender`, and `ethnic group`. The input of a task normally needs to access certain data fields so as the completion of a task may output data fields. These relations are specified by the business analyst for each task within the BPMN model. It is worth noticing that only a few of these relations are graphically represented on the model itself. Figure 1 captures the `Input Customer Data` task writing in the `Customer_Data` object data fields that are then consumed by the `Customer Identification` task.

Let us imagine now that the business analyst receives some additional requirements. In particular, the bank requires the LOBP to be compliant with regulations (e.g., Basel II, `http://www.basel-ii-risk.com/Basel-II`), EU directives (e.g., directive 95/46/EC for the protection of individuals' personal data) related to the protection of individuals with regard to the processing of personal data and on the free movement of such data, and the bank internal security policy. Not surprisingly this gives raise to a number of security desiderata for the LOBP that are not so easy to be checked and evaluated by the business analyst. Hereafter a few important security principles the bank wants to enforce in its premises and thus on the LOBP:

(S1) need-to-know: users should access only those sensitive data strictly necessary to accomplish their tasks. For a critical task, data can be defined that should not be known by the principal executing the task. E.g., the user selecting the loan bundle product should perform it objectively and thus should not have access to personal customer information such as the `name` or the `gender`.

(S2) dual control: it aims to mitigate the risk of fraud by dividing the responsibility in executing processes. Separation of duty (SoD) is one means to implement this principle. E.g., it is desirable that the bank user selecting the loan offer is not the same signing the contract.

(S3) data confidentiality: the access to sensitive data should be restricted to certain users. E.g., pre-processing clerks should not access the sensitive fields of the data objects `loanContract` and `account`, such as the loan rate, the contract duration, or the monthly rate.

It is indeed difficult for the business analyst to be sure that security desiderata as the ones expressed above are fulfilled by the BP under-design given the high dynamicity of the environment in which the BP is executed. In other words, the dynamicity of the resource allocation makes the design of the access control on data and task assignment a complex and error-prone activity. In fact, at design time it is very hard to foresee all the behaviors that the access control mechanism, e.g., the RBAC model provided by NW IdM, may originate. The scenario is made even more complex by delegation which may offer unexpected ways to circumvent security desiderata (e.g., when delegating a task, its related data object access is also delegated so that the delegatee may gain access to data fields he/she was not supposed to). We propose an approach providing tool support to business analysts that mitigates the design of BPs non-compliant with security desiderata. In doing this, we focus on security aspects combining both data-flow (such as need-to-know or data confidentiality) and resource allocation (such as separation of duty).

## 3 An Outline of Our Approach

Our approach, outlined in Figure 2, integrates a security validation procedure within standard BP modeling environments enabling the validation of security desiderata. First of all, the business analyst uses the modeling environment to define the *BP model* (e.g., the LOBP) intended here as the workflow, the data objects as input and output to tasks, task assignment to users and roles, etc. The `Security Desiderata Specification` module offers then the business analyst with accessible user interfaces, i.e. with an easy way to specify industrially relevant *Security Desiderata* (e.g., need-to-know, data confidentiality, dual control) that the BP model is required to satisfy. The rest of the security validation procedure automatically checks whether the BP model enjoys these desiderata or not. In particular, a *Formal Model* capturing both the BP model and the security desiderata is generated (see Section 4)

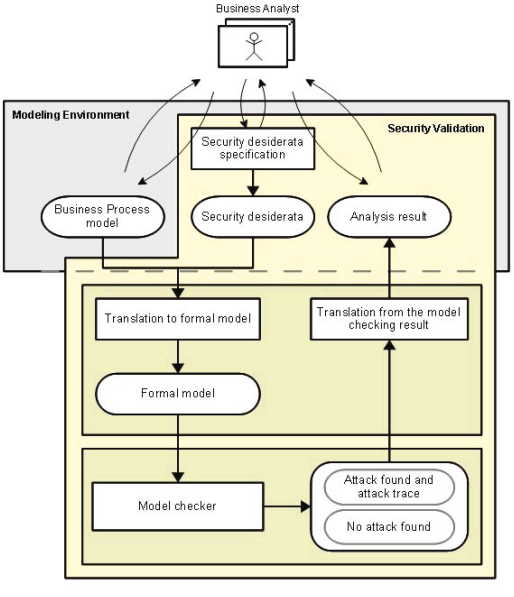



**Fig. 2.** Our Security Validator within a BPM modeling environment

and then analyzed via a `Model Checker` that completely explores all the potential execution paths of the BP model. The results of the analysis, either *no attack found* or *attack found* coupled with the *attack trace*, are retrofitted to the business analyst upon a translation of the difficult-to-read model checking output to an easy-to-understand *Analysis Result* highlighted in a graphical way. This transformation, carried out by the `Translation from the Model Checking result` module, allows the business analyst to quickly understand the attack reported (if any) and to take proper counter-measures to fix it in the BP model. Once implemented within a BPM modeling system our approach offers a push-button security validation procedure that employs state-of-the-art automated reasoning techniques for evaluating security-relevant aspects of BPs in complex dynamic environments and still features accessible user interfaces and apprehensive feedback to be usable for industrial people and business analysts.

## 4 Formalization

We now show how security relevant aspects of BPs and security desiderata can be formally specified in a model suitable for formal analysis. In particular we use the AVANTSSAR Specification Language (ASLan, [5]) to define a model checking problem where a transition system will be evaluated w.r.t. the security desiderata, i.e. security properties that the process must achieve. States of the transition system are sets of facts, i.e. atomic propositions, that have to satisfy a set of Horn clauses and transitions are captured as rewrite rules. An ASLan specification comprises an initial state, Horn clauses, rewrite rules, and security properties to be defined as attack states. Horn clauses offer a declarative way to specify dependencies among facts, while rewrite rules allow for the specification of transition rules. Let $f, g_1, \ldots, g_n$ be facts, Horn clauses of the form

$$\texttt{hc}\ \ hornClauseName\ := f\ :-\ g_1, \ldots, \mathrm{g}_n$$

assert that in each state where the body holds (i.e., $g_1, \ldots, g_n$ hold), then also the head of the Horn clause holds (i.e., $f$ holds). Conditional rewrite rules have the form

$$\texttt{step}\ rewriteRuleName := \ PFs.NFs\ \&\ Conds\ =>\ RHS$$

where $PFs$ and $RHS$ are sets of positive facts, $NFs$ is a set of negated facts, and $Conds$ represents a set of possibly negated atomic conditions, e.g. $\texttt{equal}(t_1, t_2)$ that holds if term $t_1$ is equal to term $t_2$. Given a state where $PFs$, $NFs$, and $Conds$ hold, i.e. all the positive facts and conditions hold and the negative facts and conditions do not, the application of the rewrite rule consists in replacing $PFs$ with $RHS$ in the next state. Thus the main difference between Horn clauses and rewrite rules is that the head of an Horn clause is inferred in the very same state where the body holds, while the facts in $RHS$ of a rewrite rules are inferred in the next state if $PFs$, $NFs$, and $Conds$ hold in the current state and the rule is applied. Given a set of Horn clauses $H$, a state $S$ and a rewrite rule $R$ whose

**Table 1.** Facts and their informal meaning

| Fact | Meaning |
|---|---|
| $\texttt{userToRole}(a, r)$ | user $a$ is assigned to role $r$ |
| $\texttt{poto}(p, t)$ | principal $p$ has the permission to perform task $t$ |
| $\texttt{exo}(p, t)$ | principal $p$ cannot execute task $t$ |
| $\texttt{canExecute}(a, r, t)$ | user $a$ can execute task $t$ by means of role $r$ |
| $\texttt{granted}(a, r, t)$ | user $a$ claimed the right to perform task $t$ via role $r$ |
| $\texttt{ready}(t)$ | task $t$ is ready to be executed |
| $\texttt{done}(t)$ | task $t$ has been executed |
| $\texttt{executed}(a, t)$ | user $a$ executed task $t$ |
| $\texttt{taskToData}(t, in, out)$ | task $t$ accesses data $in$ in input and data $out$ in output |
| $\texttt{aknows}(a, d)$ | user $a$ knows data $d$ |
| $\texttt{contains}(s, d)$ | set $s$ contains data $d$ |

$PFs$, $NFs$, and $Conds$ hold in $S$, the transition relation of the transition system leads to a state $S'$ obtained by substituting $PFs$ with $RHS$ and adding the heads of the Horn clauses in $H$ whose body holds (parallel execution of rules can be permitted to increase the efficiency of the approach). More details on the syntax and semantics of ASLan can be found in [5]. A simple way to describe properties of a transition system is by defining a subset of so-called bad states or attack states, i.e. sets of facts in which the desired properties are violated. The attack states specification in ASLan is syntactically similar to a rewrite rule, only that there is no $RHS$. As a result if an attack state is reached, then the set of rules leading to the attack state define the violation to the security properties. Notice that Horn clauses, rewrite rules, and attack states can be defined with variables of given types. In this way the facts in there used, e.g. in the Horn clauses head and body, can be parameterized to express families of Horn clauses, rewrite rules, and attack states respectively.

The ASLan facts we use to specify BPs in ASLan and their security desiderata are summarized with their informal meaning in Table 1. We illustrate the translation by using the LOBP and security desiderata presented in Section 2. In the ASLan specification, the process flow is expressed by means of rewrite rules managing the facts $\texttt{ready}(t)$ and $\texttt{done}(t)$. For example, the sequential execution of task `custIdentification` after task `inputCustData` can be specified as follows

```
step w_custIdentification :=
     done(inputCustData) => ready(custIdentification)
```

All the other sequential tasks are treated in a similar way. As it appears from Figure 1, the LOBP is characterized by a parallel flow of execution, i.e. an AND-split gateway that allows activities `checkIntRating` and `checkExtCreditWorthiness` to be executed simultaneously, and an AND-join that converges the flow. This behavior can be specified by using rewrite rules to express the AND-split and AND-join, i.e. a rule which adds two process-flow-related facts that will be used

to state when the tasks in the parallel flow are ready, and a rule that adds a process-flow-related fact when the tasks of all the incoming paths are completed and will be used to make the task following the parallel flow ready, i.e. `selectBundledProduct`. Other BPMN elements, such as XOR-split and XOR-join, have been modeled as well but for the sake of brevity they are not presented here as they are not used in the scenario considered.

Once the control flow has stated that a task is ready to be executed, the actual execution still requires a user willing to and even more important authorized to perform the task according to the access control policy. As described in Section 2, the core of the access control policy relies on the concepts of potential and excluded owners defined for users or roles. The potential and excluded owner relations as well as the authorization of users to claim the execution of a task (i.e. $\texttt{canExecute}(a, r, t)$) are expressed in ASLan by means of Horn clauses. As an example, the fact that the task `inputCustData` has the role `preprocessor` as a potential owner can be expressed by the Horn clause:

$$\texttt{hc poto\_inputCustData} := \texttt{poto}(\texttt{preprocessor}, \texttt{inputCustData})$$

Notice that $\texttt{poto}(\texttt{preprocessor}, \texttt{inputCustData})$ always holds as the body of the Horn clause is empty, i.e. the potential owner relation does not depend on any logical expression. Excluded owner relations can be expressed analogously. To ensure that users assigned to an excluded role for a task $t$ cannot acquire the permission to execute $t$ via other roles, the excluded owner relation involving roles is flattened to users by the following Horn clause

$$\texttt{hc user\_forbidden\_task}(a, r, t) := \texttt{exo}(a, t) \ :- \ \texttt{userToRole}(a, r), \texttt{exo}(r, t)$$

The authorization of users to claim the execution of a task is captured by Horn Clauses that express the RBAC authorization model and the direct assignment of users to tasks as follows:

$$\texttt{hc rbacAc}(a, r, t) := \texttt{canExecute}(a, r, t) : -\texttt{userToRole}(a, r), \texttt{poto}(r, t)$$

$$\texttt{hc directAc}(a, r, t) := \texttt{canExecute}(a, r, t) : -\texttt{userToRole}(a, r), \texttt{poto}(a, t)$$

The claim of a task can be expressed in ASLan by the following rewrite rule

$$\texttt{step authorizeTaskExec}(a, r, t) := \\ \texttt{canExecute}(a, r, t).\texttt{ready}(t).\texttt{not}(\texttt{exo}(a, t)) => \texttt{granted}(a, r, t)$$

where $\texttt{granted}(a, r, t)$ expresses the fact that agent $a$, having the possibility to perform task $t$ ($\texttt{canExecute}(a, r, t)$), claimed the execution of $t$ and is then in charge of performing it. As already presented in Section 2, users can delegate a task they claimed to a delegatee provided that the excluded owners of the task do not comprise the delegatee nor a role to which the delegatee is assigned. Delegation of execution can be expressed in ASLan by means of a rewrite rule stating that user $a1$ receives from $a$, who claimed task $t$, the duty to perform $t$, as follows:

$$\texttt{step delegationOfExec}(a, a1, r, r1, t, n) := \texttt{granted}(a, r, t).\texttt{userToRole}(a1, r1).\texttt{not}(\texttt{exo}(a1, t))\&\texttt{not}(\texttt{equal}(a, a1)) => \texttt{granted}(a1, r, t).\texttt{userToRole}(a1, r1)$$

Then, the execution of tasks by an authorized user is expressed by the following rewrite rule

$$\texttt{step taskExec}(a, r, t, in, out) := \texttt{granted}(a, r, t).\texttt{taskToData}(t, in, out) => \texttt{executed}(a, t).\texttt{done}(t).\texttt{taskToData}(t, in, out).\texttt{aknows}(a, in).\texttt{aknows}(a, out) \quad (1)$$

stating that the task $t$ is executed by a user $a$, i.e. $\texttt{executed}(a, t)$, if $a$ obtained the right to execute task $t$ acting in role $r$, i.e. $\texttt{granted}(a, r, t)$. Notice that as a result user $a$ accessed, and thus knows, the sets of data input and output of the task, i.e. $\texttt{aknows}(a, in)$, $\texttt{aknows}(a, out)$. As already said, performing a task is often related to accessing the data objects involved in the BP. In particular each task takes a set of data in input and returns a set of data containing the task results in output, e.g. the customer's rating is a result of the task `checkIntRating`. More in detail the data in input and output are sets of fields of possibly different business objects. These data fields are a static information that can be expressed in ASLan by facts which are included in the initial state and hold during the whole process execution. As an example, the fact that task `inputCustData` outputs, among others, the data field `name` can be expressed by the fact $\texttt{contains}(\texttt{out\_inputcustdata}, \texttt{pair}(\texttt{customerdata}, \texttt{name}))$ where $\texttt{pair}(\texttt{customerdata}, \texttt{name})$ associates the data field, `name`, to the data object containing it, `customerdata`, and `out_inputcustdata` is the set of data output of `inputCustData`. The association between tasks and its inputs and outputs is done via facts included in the initial state. E.g., $\texttt{taskToData}(\texttt{inputCustData}, \texttt{in\_inputcustdata}, \texttt{out\_inputcustdata})$ associates the task `inputCustData` to its input and output sets respectively. Notice that, since executing tasks means accessing data (as shown in (1)), the delegation of a task execution can let the delegatee access data objects he/she was not supposed to.

The security desiderata presented in Section 2 can be expressed in ASLan as attack states. As an example, the need-to-know principle (S1) according to which the user selecting the loan offer should not have access to personal customer information such as the `name` or `gender` can be expressed in ASLan by the following attack state for all $Field \in \{\texttt{name}, \texttt{gender}\}$

$$\texttt{attack\_state needToKnow}(a, set) := \texttt{executed}(a, \texttt{selectBundledProduct}).\ \texttt{aknows}(a, set).\ \texttt{contains}(set, \texttt{pair}(\texttt{customerdata}, Field)) \quad (2)$$

As another example (S2), i.e. to check if an agent can perform both the tasks `selectBundledProduct` and `signForm`, can be specified via an attack state containing facts $\texttt{executed}(a, \texttt{selectBundledProduct})$ and $\texttt{executed}(a, \texttt{signForm})$. Finally, the security desiderata (S3), i.e. pre-processing clerks should not access the loan rate, neither the duration of the contract, nor the amount monthly paid to ensure data confidentiality, can be expressed in ASLan by attack states containing $\texttt{userToRole}(a, \texttt{preprocessor})$, $\texttt{aknows}(a, set)$, and $\texttt{contains}(set,$

`pair`($Object, Field$)) where $Object$ and $Field$ have to be replaced with the data objects and fields specified in (S3).

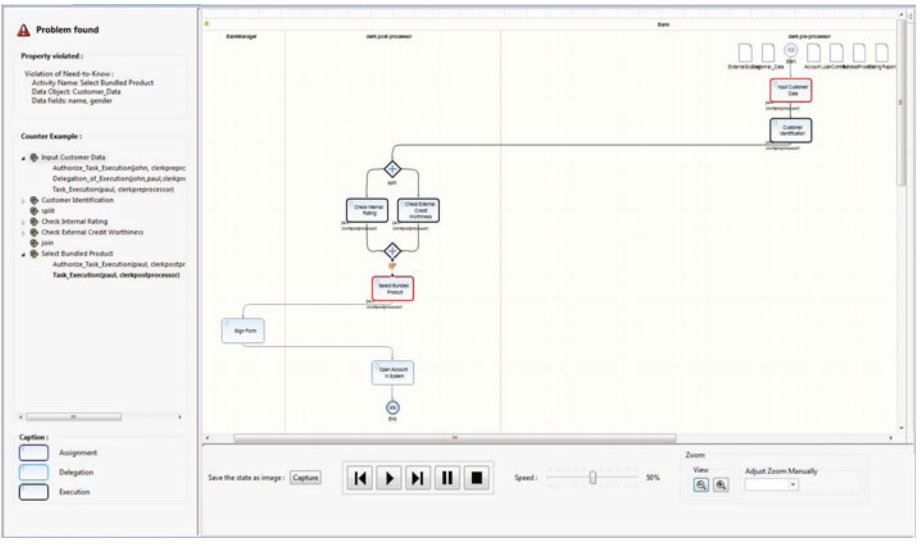
## 5 Assessment

In order to assess our work, we have implemented our approach and formalization within the NW BPM industrial environment which provides, besides others, a Process Composer module that enables process architects and developers (referred in our paper as business analysts) to design and deploy executable BP models. We have enhanced the Process Composer with a Security Validation plug-in that implements our approach and that business analysts can smoothly run to validate security desiderata for the BP under-design. We consider the business analyst has designed the LOBP in NW BPM as discussed in Section 2 and that he/she is left with the problem of evaluating that the LOBP enjoys the security desiderata (S1), (S2), and (S3) required by the bank. By running the Security Validation plug-in, the business analyst accesses a wizard where he/she can easily create the security desiderata to be validated, i.e. an implementation of the `Security Desiderata Specification` module. It would be clearly unfeasible to expect the business analyst to specify security desiderata as attack states in ASLan, e.g. (2). We have thus invested effort to devise user-friendly GUIs through which the business analyst can express his/her security desiderata without being neither a formal methods practitioner nor a security expert. For instance, in order to validate the need-to-know security desiderata (S1) the business analyst can simply provide the key information, i.e. the task and the data fields that the user performing the task should not have access to, and press a button. The security-relevant aspects of the BP under-design are gathered from the BPMN model in NW BPM and from the other relevant sources (e.g., the NW IdM deployed at the bank where bank users, roles, and user-to-role assignments are specified) and are automatically translated together with the security desiderata into a formal model (see Section 4). Model-checking functionalities are provided by the AVANTSSAR Platform (AVP) as a service[2]. In particular our Security Validator plug-in invokes the SAT-based Model-Checker [6] service. If there exists an execution path of the BP under-design that violates one of the security desiderata, such a violation is discovered by the model checker and returned to our plug-in. The output from the model checker service is not very human-readable and it is thus transformed back into a graphical representation that the business analyst can easily digest and play with. This is critical to allow the business analyst to understand the root cause of the violation (if any) and to take proper counter-measures to fix the issue in the model. By running our Security Validator to check the LOBP for (S1), (S2), and (S3), the SATMC service returns the counter-example listed in Figure 3a. The counter-example is automatically presented via the `Analysis result` GUI of Figure 3b to the business analyst that can see which desiderata has been violated—in this case (S1)—and play the attack trace step-by-step in a movie-like fashion. The attack trace shows

[2] `http://satmc.ai.dist.unige.it/avantssar/`

```
GOAL needToKnow(paul,out_inputcustdata)
[w_inputCustData]
[authorizeTaskExec(john,preprocessor,inputCustData)]
[delegationOfExec(john,paul,preprocessor,postprocessor,inputCustData)]
[taskExec(paul,preprocessor,inputcustomerdata,...,out_inputcustdata)]
[...]
[w_selectBundledProduct]
[authorizeTaskExec(paul,postprocessor,selectBundledProduct)]
[taskExec(paul,postprocessor,selectBundledProduct,...)]
```

(a) Analysis result: SATMC counter-example




(b) Graphical representation of the analysis result

**Fig. 3.** Need-to-know property counter-example

to the business analyst that user *paul* can execute the task `inputCustData` accessing in this way to the `name` and `gender` of the customer and can later on execute the task `selectBundledProduct`, thereby violating the need-to-know desiderata. Moreover, by playing the attack trace step-by-step it is easy for the business analyst to see that *paul* executes the task `selectBundledProduct` through his role `postprocessor` and is able to perform the task `inputCustData` by being delegated by the pre-processing clerk *john*. Figure 3 clearly shows the benefits of the graphical representation of the attack with respect to the raw output of the model checker which is not suitable for business analysts. It is worth noticing that there is a clear correspondence between the two representations. In particular the graphical attack trace shows the execution flow, i.e. steps of the form `w_`*task* in Figure 3a, by highlighting the activities currently executed and gives key information about the action performed by using different color to highlight the activities, e.g., dark blue for task claim, black for task execution, and blue for delegation. Furthermore some other information are placed with the BP model to provide, e.g., the user claiming the task, the

user executing the task, and the user being delegated. To avoid the violation in Figure 3, the business analyst can think to set role `postprocessor` as an excluded owner for task `inputCustData`. Though this prevents the attack trace just presented, the security desiderata are not fulfilled and a new run of the Security Validation plug-in spots a violation where the pre-processing clerk *john* executes task `inputCustData` and is then delegated by *paul* to perform task `selectBundledProduct`. This new violation stresses the fact that the execution of tasks can be delegated to users of any other role. Thus the business analyst has to define excluded owners for the two tasks involved in the violation such that no roles other than the ones defined as potential owner can be delegated to execute them. New runs of the Security Validator plug-in spot attack traces to properties (S2) and (S3). The business analyst must then analyze the attack traces and modify the process model, e.g. by adding excluded owners. This process is iterated until either no attack traces are detected or there is not a way to fix the problem in the BP model, i.e. the business analyst acknowledges the risk and hence can put in place monitoring systems at run-time in order to prevent frauds. The work was conducted jointly with practitioners of BPM, thus profiting from their expertise in designing user-friendly GUIs for business analysts. The experiments were performed on a desktop computer with an Intel Core 2 Duo processor with 2.66GHz clock and 4Gb of RAM memory. The plug-in takes at most 26 secs to detect the violations to (S1), (S2), and (S3). We are currently considering larger specifications so as to evaluate the scalability of the approach. The experiments performed so far show that, as model checking techniques suffer of the state explosion problem, the time required for the analysis increases with the depth of the attack trace, i.e. the number of steps required to reach the attack state, from 0.62 sec for a trace of depth 1 to 875.02 sec for depth 12, on a business process characterized by 33 tasks with a potential owner each, an AND-split, 5 XOR-splits, and 4 roles. To mitigate the time increase we are currently working on some optimization in the translation of the model so that by reducing the time required for the analysis would ensure the scalability of the approach. Preliminary results are promising both on the toy example presented in this paper as well as on a complex BP for aviation featuring over 60 tasks.

## 6 Related Work

The use of model checking for the automatic analysis of business processes has been put forward and investigated in [7]. The paper proposes the use of the NuSMV model checker to formalize and verify business processes with RBAC policies and delegation. However it does not take into account the data-flow and the need to make the approach accessible to business analysts which are not familiar with formal methods. Another approach based on model checking to the analysis of business processes is presented in [8]. It considers workflows and security policies modeled in a security enhanced BPMN notation and a formal semantics based on Coloured Petri nets (CP-nets). It then presents an automatic translation from the process model into the Promela specification language and

the usage of SPIN to verify SoD properties. However no provision is made for the assignment of an agent to multiple roles, delegation, and data-flow. Another approach which uses CP-nets in this context is [9]. It presents a formal technique to model and analyze RBAC using CP-nets which can be composed with context-specific aspects of the application of interest. However the approach does not take into account data and thus data related properties. An approach based on model checking for the analysis of access control policies is presented in [10]. It supports the specification of complex policies where permissions are the ability of agents to access data to read or write, however the scope is not dedicated to business processes as it does not take into account the process workflow. In [11] the authors consider workflow and authorization policies for privacy purposes. In particular they propose the use of Color-X diagrams to represent the process which is then translated in Prolog to perform an analysis of privacy relevant properties over data, e.g. need-to-know principle. However, their approach is restricted to privacy and does not take into consideration critical properties, such as dual control. A data-aware language for expressing compliance requirements is presented in [12]. It extends the BPMN-Q query language to express compliance rules, e.g. how the content of data should influence the workflow. The rules are expressed as queries and then formalized by mapping them into past linear temporal logic formulae which are then model checked against the process models to decide about compliance. In particular BPMN-Q allows users to express compliance requirements in a visual way very similar to the way processes are modeled and whenever a compliance rule is violated, execution paths causing violations are visualized to the user. However the paper does not take into account the access control policy in place and thus security properties related to the access to data by unauthorized users or dual control. An extension of UML for secure systems development, i.e. UMLsec, and a tool for the analysis of software configurations that ensures compliance with security policies are presented in [13,14] respectively. In particular the approach allows to check extended UML models of business applications and their security permissions for vulnerabilities to security rules (such as separation of duty). The approach and motivations are similar to ours, however we here focus on aiding the design of security relevant aspects of business processes modeled by using BPMN which is broadly employed in industrial business process management systems. Nonetheless, it is worth to further investigate this approach in the future.

## 7 Conclusion and Future Work

Is this critical BP task executable in an objective way within my organization? Are these sensitive BP data accessible to the manager director only? These are recurrent difficult-to-answer questions a business analyst may face during the design of a BP. In this paper we have detailed a novel security validation approach for BPs that employs state-of-the-art model checking techniques for evaluating security-relevant aspects of BPs in dynamic environments, and still features accessible user interfaces and apprehensive feedback for business analysts.

At the core of our approach lies a general way to formally capture security-relevant aspects of BPs, not only those that have been well-studied so far (e.g., dual control), but also new important ones from the area of data-flow such as need-to-know and data confidentiality. As proof of concept we have implemented our approach as an Eclipse plug-in within NW BPM and assessed it against a BP example from the banking area. All in all, our approach increases the quality, robustness and reliability of BP under-design, mitigating the risk of deploying non-compliant BPs.

In the early future, we want to capture in our formal specification other aspects of BPs including service invocation and consumption as well as data objects shared among several BP instances. Some optimizations in the translation to the formal model are under development in order to scale the approach on industrial-scale BPs. Preliminary results run on a BP from the aviation domain are promising despite the over 60 tasks executed in that BP. This makes more concrete a potential productization of our approach within industrial tools such as NW BPM. We are currently discussing with SAP business units in that respect.

## References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking (2000)
2. Karch, S., Heilig, L.: SAP NetWeaver, 1. aufl edn. Galileo Press, Bonn (2004)
3. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer 29(2), 38–47 (1996)
4. Giorgini, P., Massacci, F., Mylopoulos, J.: Modeling security requirements through ownership, permission and delegation. In: RE, pp. 167–176. IEEE Press, Los Alamitos (2005)
5. AVANTSSAR: Deliverable 2.1: Requirements for modelling and ASLan v.1 (2008), `http://www.avantssar.eu`
6. Armando, A., Carbone, R., Compagna, L.: LTL Model Checking for Security Protocols. In: JANCL, Special Issue on Logic and Information Security (2009)
7. Schaad, A., Lotz, V., Sohr, K.: A model-checking approach to analysing organisational controls in a loan origination process. In: SACMAT, pp. 139–149. ACM, New York (2006)
8. Wolter, C., Miseldine, P., Meinel, C.: Verification of business process entailment constraints using SPIN. In: Massacci, F., Redwine Jr., S.T., Zannone, N. (eds.) ESSoS 2009. LNCS, vol. 5429, pp. 1–15. Springer, Heidelberg (2009)
9. Rakkay, H., Boucheneb, H.: Security analysis of role based access control models using colored petri nets and cpntools, pp. 149–176 (2009)
10. Zhang, N., Ryan, M., Guelev, D.P.: Evaluating access control policies through model checking. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 446–460. Springer, Heidelberg (2005)
11. Teepe, W., van de Riet, R., Olivier, M.: Workflow analyzed for security and privacy in using databases. J. Comput. Secur. 11(3), 353–363 (2003)
12. Awad, A., Weidlich, M., Weske, M.: Specification, verification and explanation of violation for data aware compliance rules. In: ICSOC-Service Wave (2009)
13. Jan, J.: Secure Systems Development with UML. Springer Academic Publishers, Heidelberg (2005)
14. Höhn, S., Jürjens, J.: Rubacon: automated support for model-based compliance engineering. In: ICSE, pp. 875–878 (2008)

# On-Device Control Flow Verification for Java Programs

Arnaud Fontaine, Samuel Hym, and Isabelle Simplot-Ryl

Univ Lille Nord de France, INRIA Lille – Nord Europe, CNRS UMR 8022 LIFL

**Abstract.** While mobile devices have become ubiquitous and generally multi-application capable, their operating systems provide few high level mechanisms to protect services offered by application vendors against potentially hostile applications coexisting on the device. In this paper, we tackle the issue of controlling application interactions including collusion in Java-based systems running on open, constrained devices such as smart cards or mobile phones. We present a model specially designed to be embedded in constrained devices to verify on-device at loading-time that interactions between applications abide by the security policies of each involved application without resulting in run-time computation overheads; this model deals with application (un)installations and policy changes in an incremental fashion. We sketch the application of our approach and its security enhancements on a multi-application use case for GlobalPlatform/Java Card smart cards.

## 1 Introduction

Ubiquitous devices such as mobile phones and smart cards tend to be multi-application devices and to support post–issuance installation of applications. Applications are also evolving to take advantages of these new features: they are shifting from standalone designs to a collaborative model where they provide services to other applications, as it is for instance the case in Android. This new shape of ubiquitous devices causes major *security* concerns for end-users but also for applications vendors.

In a multi-application environment, the system generally provides some basic mechanisms to application developers for controlling access to services they provide to other applications. In open contexts, *i.e.* when some applications can be added (or changed) to the system after its issuance to the end-user, *access control* mechanisms are however very limited and insufficient to fulfill requirements of international security agreements such as the Common Criteria Recognition Agreement[1]. Actually, formal methods are required to reach the highest certification levels of these agreements.

Formal methods currently applied to verify some security properties (confidentiality, integrity, application interactions, safety, *etc.*) have been developed to analyze a finite set of applications known before issuance of the system. These

[1] Common Criteria (`http://www.commoncriteriaportal.org`)

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 43–57, 2011.

methods are not appropriate in "open" contexts as they have not been designed to incrementally deal with changes (addition/removal of applications) and thus to refrain from useless consumption of resources of small constrained devices.

In this paper, we focus on the design of on-device verification of absence of undesired control flow between applications in small open multi-application Java-based systems. The model is designed to avoid collusion between applications that aims at exploiting services provided by other applications. The main originality of the model we propose in Section 2 is to allow an incremental verification process at installation-time. Corresponding algorithms exposed in Section 3 have been designed to have a small memory footprint, low memory requirements and limited computation overheads in order to be fully embedded in constrained devices such as smart cards. In Section 3 we also illustrate the concrete achievements of our approach on a concrete multi-application use case for GlobalPlatform/Java Card smart cards. We finally discuss about related work in Section 4 and conclude in Section 5.

## 2 Model for Controlling Service Calls between Applications

Before we present our model, we first introduce the technological context of our work. Since we want our model to be suitable to any constrained Java-based systems, we decide to target the most constrained ones: Java-based smart cards.

### 2.1 Technological Context

Smart cards are strongly constrained in terms of memory and computation resources. Standard Java virtual machines and run-time environments are not suitable for such systems because of mechanisms such as garbage collector or threads that are too greedy for resources, but also because of the very large API that contains unneeded features (AWT, Swing, reflection, *etc.*).

Basically, a Java Card run-time environment is a multi-application environment where applications can be (un)installed after the device has been issued to its end-user. Java Card "applications", called *applets*, are identified by a unique *Application IDentifier (AID)* assigned by the *International Standards Organization (ISO)* as defined in the ISO-7816 standard. Java Card applets can receive commands from outside the card through a specific unit called APDU, but they cannot directly interact with each other. Each *application*, that is an applet and all its related classes, can instead provide and register some particular objects in the system that offer *shared services* to other applications. Technically speaking, applications developed for Java Card are instances of sub-classes of the `Applet` class, and every shareable object is an instance of a class that implements the `Shareable` interface from the Java Card API. The set of methods implemented in all objects shared by an application correspond to its set of shared services. Retrieval of shared objects and invocation of their methods is not controlled by the Java Card environment; each application is entirely responsible for the access control to its shared objects relying on AIDs of requesters.

GlobalPlatform[2] is generally used to enhance interoperability and security of Java Card[3] systems, especially those dealing with sensitive information such as SIM cards or credit cards. Indeed, GlobalPlatform proposes standardized and secure features for post-issuance (un)installation of applications through encrypted communication channels as well as application isolation thanks to the *security domains* architecture. Roughly, a security domain is an area whose access is controlled by *public key infrastructure* mechanism, where some applications can be installed and which is given some privileges by the card issuer. As Java Card applets, each security domain is uniquely identified by an AID.

### 2.2 Systems and Security Policies

We define a high level abstraction of an open multi-application Java-based system. We rely on GlobalPlatform's terminology to describe the entities of our model, but the model can be instantiated on any constrained Java-based systems with a different semantic given to *domains*, *applications* and *shared services*.

For the purpose of controlling access to services shared by applications, each application must provide, for each of its shared services, the exhaustive list of domains from which this service might be used. The exhaustiveness of this list is here to be understood even up to relays: if an application *Bad* wants to call a service $S$ but resides in some domain from which calling $S$ is prohibited, it might use a third-party application *Relay* that resides in some domain from which $S$ can be called to relay its calls to $S$. Basically, we assume that domains define a partition of applications such that applications installed within the same domain can call each other's shared services, whereas the use of services between applications belonging to different domains must be explicitly allowed.

Since methods can be overridden in Java, we always consider fully qualified names for methods and often write them explicitly as $A.C.m$ where $A$ is an application containing a class $C$ providing a method of name $m$. Note that the method $A.C.m$ might be defined and implemented in a super-class of $A.C$ — which might itself be in some other application— and simply inherited. We write $\mathcal{M}$ for the set of methods of the system, and $\mathcal{M}_A$ for the one of $A$.

The set of classes is equipped with an inheritance relation $\leq$: $C_1 \leq C_2$ means that the class $C_1$ inherits from $C_2$ or is $C_2$; we use $C_1 < C_2$ when $C_1$ is a strict sub-class of $C_2$. We also use this notation for methods: when a class $C_1$ of an application $A_1$ inherits from a class $C_2$ in an application $A_2$ and redefines a method $m$, we write $A_1.C_1.m < A_2.C_2.m$; we use $\leq$ whenever the method is either inherited or redefined[4].

Before getting to the heart of our subject with the definition of security policies, let us set a few extra notations. For a function $f$, we denote by $f[x \mapsto e]$ the function $f'$ such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = e$. For a set of sets $C$ and

[2] GlobalPlatform specifications (`http://www.globalplatform.org/`)

[3] Java Card specifications (`http://java.sun.com/javacard/`)

[4] So the method $A.C.m$ of $\mathcal{M}_A$ is actually defined in the class $C$ of the application $A$ when there does not exist any $A'.C'.m$ such that $A.C.m \leq A'.C'.m$ unless $A.C.m < A'.C'.m$ or $A.C = A'.C'$.

a function $f : A \longrightarrow C$, we write $f^\bullet$ the total function such that $f^\bullet(x) = f(x)$ when $f(x)$ is defined and $\emptyset$ when it is not. For any two functions $f_1 : A \longrightarrow C$ and $f_2 : B \longrightarrow C$, we write $f_1 \sqcup f_2$ for the function from $A \cup B$ to $C$ such that $f_1 \sqcup f_2(x) = f_1^\bullet(x) \cup f_2^\bullet(x)$.

A security policy is a set of security rules in which every method is given exactly one security rule and where security rules define from which domains a given method can be called.

**Definition 1 (Security policy).** *Let $M$ be a set of methods and $D$ be a set of domains. A security policy for the set of methods is $p = (M, D, rules)$, where $rules$ is a mapping $rules : M \longrightarrow \wp(D) \cup \{\top\}$, where $\top$ stands for "*any domain*".*

**Definition 2 (On-device system).** *An on-device system $\mathcal{S}$ is defined by $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ where $\mathcal{D}$ is a finite set of domains, $\mathcal{A}$ is a finite set of applications, $\mathcal{U}$ is the subset of $\mathcal{A}$ of applications with* unsolved *dependencies, $\delta$ is the deployment function from $\mathcal{A}$ to $\mathcal{D}$ that defines which domain each application belongs to, and $\mathcal{P} = (\mathcal{M}, \mathcal{D}, rules)$ is the security policy of the system.*

Applications with *unsolved* dependencies cannot run because they need to use some services that are not available for the moment. The applications of $\mathcal{A} \setminus \mathcal{U}$ are said *selectable*. Note that the function $\delta$ can naturally be extended from applications to methods.

The value $\top$ is needed to allow evolutions of the system after deployment: the difference between $\top$ and $D$ is that $D$ is the set of all the *known* domains. When the system evolves, some new domains might be introduced . $\top$ conveys that meaning of all domains, even yet-unknown ones.

### 2.3 Semantics of the Security Policy

We build on those definitions of security policies to formally define the security property we want to assert on systems. Basically, in secure systems, an application in some domain $d$ will be enabled to invoke indirectly (*i.e.* to trigger an invoke from a method in some other domain) or directly only services explicitly expecting calls from $d$. The semantics must take into account dynamic dispatch, where the method of any sub-class of the static type can be actually called.

**Definition 3 (The context-insensitive call graph).** *The* context-insensitive call graph *of a system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ is a finite graph $CG(\mathcal{S}) = (\mathcal{M}, E)$ with $E \subseteq \mathcal{M} \times \mathcal{M}$ the set of $(A_1.C_1.m_1, A_2.C_2.m_2)$ such that there exists a bytecode* `invoke`[5] *$C_3.m_3$ in the bytecode of $A_1.C_1.m_1$ where $A_2.C_2.m_2 \leq A_3.C_3.m_3$, i.e. $A_1.C_1.m_1$ might call $A_2.C_2.m_2$.*

**Definition 4 (Secure system).** *A system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ is* secure *if and only if for all $(A_1.C_1.m_1, A_2.C_2.m_2) \in \mathcal{M} \times \mathcal{M}$ of the context-insensitive call graph $CG(\mathcal{S}) = (\mathcal{M}, E)$, there exists a path from $A_1.C_1.m_1$ to $A_2.C_2.m_2$ in $CG(\mathcal{S})$ only if $\delta(A_1.C_1.m_1) \in rules(A_2.C_2.m_2)$.*

[5] Where `invoke` stands for any invocation bytecode.

### 2.4 Generic Security Policies

When implementing an application $A$, the programmer has to decide which methods are critical and must therefore get a security policy rule. But the programmer may not be aware of the precise configuration on which the application will be deployed, in particular we do not want the set of domains to be fixed once and for all. So we allow programmers to define, for an application $A$, a set of security levels $\mathcal{L}_A$, that represents relevant security access levels for the application like *public*, *applications of the same domain*, *applications of commercial partners*, *etc.* The policy $P_A = (\mathcal{M}_A, \mathcal{L}_A, rules_A)$ defines the access rules to the methods of the $A$, *i.e.* the set of security levels from which this method can be called.

To impose as little constraints as possible on the set of levels $\mathcal{L}_A$, the only requirement is the availability of an instantiating function mapping those levels to actual domains when the application is deployed:

$$inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})} : \mathcal{L}_A \longrightarrow \wp(\mathcal{D}).$$

The model always allows communications inside a security domain, so we require the images by $inst_{A,d,\mathcal{S}}$ to include $d$ (so the co-domain of the mapping is $\wp(\mathcal{D}) \setminus \wp(\mathcal{D} \setminus \{d\})$). Then the policy of $A$ deployed on $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ is $P_A^i = (\mathcal{M}_A, \mathcal{D}, inst_{A,d,\mathcal{S}} \circ rules_A)$ with the extension of the mapping:

$$\begin{aligned} inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})} : \wp(\mathcal{L}_A) \cup \{\top\} &\longrightarrow \wp(\mathcal{D}) \cup \{\top\} \\ L \in \wp(\mathcal{L}_A) &\longmapsto \textstyle\bigcup_{l \in L} inst_{A,d,(\mathcal{D},\mathcal{A},\mathcal{U},\delta,\mathcal{P})}(l) \\ \top &\longmapsto \top \end{aligned}$$

*Remark 1.* The rules of the policy are a mapping from all the methods to the security levels. Only the security policy of methods defined in shared classes is enough to infer the security policies of other methods thanks to the computation of the closure of the least relation $rules_A$ such that $rules_A(m) \subseteq rules_A(m')$ whenever $m$ invokes $m'$.

### 2.5 Application to GlobalPlatform/Java Card Systems

For GlobalPlatform/Java Card systems, instantiation of our control flow model is straightforward: domains are naturally mapped to GlobalPlatform security domains, applications are mapped to Java Card applets and their related classes, and shared services are the methods defined in objects that implement the `Shareable` interface from the Java Card API. This mapping satisfies assumptions and hypotheses of the model: admittance in a domain is controlled by the underlying system, all applications can potentially interact with each other whatever their domain but only through methods of shared objects.

The case of the control flow policy instantiation function *inst*, and the related security levels, is trickier since it depends on what the target device is devoted

to. Two solutions are conceivable: to implement this function inside or outside the device. For the purpose of our use case (Section 3.4), we choose the outside implementation because it does not constrain us to define a set of security levels without a strongly relevant context.

A technical but important point is to properly define how to encode and to attach control flow policies to applications. Since an instantiated control flow policy of a method consists of a set of domains authorized to invoke it, it is natural to use a set of AIDs of security domains as an instantiated control flow policy. A null AID is used to map the *any domain* value ($\top$) from the model.

A more efficient encoding can be reached on-device to keep policies in memory. Indeed, keeping sets of AIDs once applications have been installed is useless since the maximal number of domains available on one device will always be small because of memory constraints. A single bit is assigned to each domain, so the control flow policy of a method is reduced to a $n$-bit value where $n$ is the number of known domains: a bit set to zero denotes that the corresponding domain is not authorized to call the method, and conversely for a bit set to one. A special value must in addition be reserved for the "any domain" value of the model. At loading-time, each time a policy involves a domain never seen before, a free position is simply assigned to this domain. Such an encoding also permits to test very efficiently inclusion using bit-wise operations between sets of domains, which reduces overhead computations at loading-time.

## 3 On-Device Algorithms

We devise now efficient on-device algorithms to ensure, at loading-time, that the system stays secure along its evolutions.

### 3.1 Addition of a New Application

When a new application $A$ with the instantiated policy $P_A^i = (\mathcal{M}_A, \mathcal{D}, inst_{A,d,\mathcal{S}} \circ rules_A)$ is loaded on the device in some domain $d$, the loader has to verify that it respects the policy of the system and, if the application successfully passes the verification, to update the system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$.

To accept the application $A$, we verify it class by class, method by method. Thus, for each method $m$, we have to check that 1) $m$ invokes only methods that grant access to applications in domain $d$ 2) methods that ask access to $m$ reside in a domain from which the rules for $m$ grant access.

We denote by $\mathcal{I}(m)$ the set of methods that $m$ directly invokes, namely the $m'$ such that an instruction `invoke` $m'$ appears in the code of $m$. To perform the verification, we have to be able to take into account, for a given method, the set of methods it invokes directly or indirectly. So all the rules and the algorithms we propose hereafter pay special attention to preserve the transitivity of the property they ensure.

**Definition 5 (On-device method verification).** *A method $m$ of a class $C$ of an application $A$ with a policy $P_A = (\mathcal{M}_A, \mathcal{L}_A, rules_A)$ passes the verification to be loaded on the system $\mathcal{S} = (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, \mathcal{P})$ in the domain $d$ if:*

$$\forall A'.C'.m' \in \mathcal{I}(A.C.m), \{d\} \cup inst_{A,d,\mathcal{S}} \circ rules_A(A.C.m) \subseteq rules(A'.C'.m') \quad (1)$$

$$\forall A'.C'.m', A.C.m \in \mathcal{I}(A'.C'.m') \Rightarrow \{\delta(A'.C'.m')\} \cup rules(A'.C'.m') \subseteq inst_{A,d,\mathcal{S}} \circ rules_A(A.C.m) \quad (2)$$

$$\forall A'.C', A.C.m \leq A'.C'.m \Rightarrow rules(A'.C'.m) \subseteq inst_{A,d,\mathcal{S}} \circ rules_A(A.C.m) \quad (3)$$

Those rules are dictated by the requirement of consistency of the system:

- (1) ensures that all the methods invoked by the incoming method grant access to the domain in which the incoming application is being installed and to all the domains from which the incoming method will be expecting calls,
- (2) ensures that all the methods calling the incoming method are in domains from which this is explicitly permitted,
- (3) ensures that, when the incoming method overrides an inherited method, it still grants access to all the domains the overridden method did: this check handles in a simple way dynamic dispatch while allowing the system to be further extended with applications unknown *a priori*.

As mentioned above, the transitivity in those rules is ensured because $\subseteq$ is obviously transitive. Those rules are translated into the Algorithm 3.1. Since applications (and classes, for that matter) may depend on each other, the algorithm must keep some information about unsatisfied dependencies ($wait(A)$ is the set of all the applications $A$ is waiting for to become selectable) and collect all the constraints imposed on methods until they are actually loaded and the constraints can be checked ($unsolved(A.C.m)$ is the set of all the domains from which already-loaded methods call $A.C.m$). We also use another temporary function $ruleswait$, similar to $rules$, to store all the rules for the methods in applications waiting for some dependency.

Rule 1 is checked by the code in Lines 7 to 13: as stated before, the verification is performed only on directly invoked methods; if the invoked method is already loaded the consistency can be checked right away; if not, the check is postponed by using *unsolved*. Rule 2 is easier to check, on Line 5: all the already-loaded methods invoking the incoming one have registered their requirements to *unsolved*. Finally Rule 3 is tested on Line 1. In the event of failure of this check when the method is loading, we decide to resolve the conflict between already installed applications and the new one by rejecting the incoming code.

The Algorithm 3.2 wraps the complete verification of an incoming application. The main task to perform is to mark newly selectable applications as such, when they were waiting for the newly-loaded one, on Lines 7 to 10. The final operation of cleaning-up temporary data structures is not detailed here.

1: **if** $A.C \leq A'.C'$ **and** $\exists A'.C'.m$ **and** $(rules \sqcup ruleswait)(A'.C'.m) \not\subseteq rules^i_A(A.C.m)$ **then**
2: **return** FAIL
3: **else if** $A.C \leq A'.C'$ **and** $wait(A') \neq \emptyset$ **then**
4: $wait \leftarrow wait[A \mapsto wait(A) \cup \{A'\}]$
5: **if** $unsolved(A.C.m) \not\subseteq rules^i_A(A.C.m)$ **then**
6: **return** FAIL
7: **for all** `invoke` $A'.C'.m'$ instruction **do**
8: **if** $A'.C'$ is already loaded **then**
9: **if** $\{d\} \cup rules^i_A(A.C.m) \not\subseteq (rules \sqcup ruleswait)(A'.C'.m')$ **then**
10: **return** FAIL
11: **else**
12: $unsolved \leftarrow unsolved[A'.C'.m' \mapsto unsolved(A'.C'.m') \cup \{d\} \cup rules^i_A(A.C.m)]$
13: $wait \leftarrow wait[A \mapsto wait(A) \cup \{A'\}]$

**Algorithm 3.1.** Verification of the method $A.C.m$ loaded in the domain $d$.

Once the methods of $A$ are verified, the application can be securely installed on the system. Thus the system has to be updated by the installation of the new application with the following function:

$$install((A, P_A), d, \mathcal{S}) = (\mathcal{D}, \mathcal{A} \cup \{A\}, \mathcal{U}', \delta[A \mapsto d], \mathcal{P}')$$

where $\mathcal{P}' = (\mathcal{M} \cup \mathcal{M}_A, \mathcal{D}, rules \sqcup inst_{A,d,\mathcal{S}} \circ rules_A)$.

1: $rules^i_A \leftarrow inst_{A,d,\mathcal{S}} \circ rules_A$
2: **for all** class $C$ of application $A$ **do**
3: **for all** method $m$ defined in class $C$ **do**
4: **if** verify_method(A.C.m) = FAIL **then**
5: **return** FAIL
6: $ruleswait \leftarrow ruleswait \sqcup rules^i_A$
7: **for all** $B$ such that $wait(B) = \emptyset$ **and** $rules(B) = \emptyset$ **do**
8: $rules \leftarrow rules[B \mapsto ruleswait(B)]$ // *$A$ is the only possible $B$ at the 1st iteration*
9: **for all** $B'$ such that $B \in wait(B')$ **do**
10: $wait \leftarrow wait[B' \mapsto wait(B') \setminus \{B\}]$
11: clean *unsolved* and *ruleswait*

**Algorithm 3.2.** Loading of an application $A$ with policy $(\mathcal{M}_A, \mathcal{L}_A, rules_A)$ in a domain $d$ of $\mathcal{S}$.

The application will become *selectable* as soon as all its dependencies will be resolved, this means that all the called methods must be methods of selectable applications, and that all the inherited classes must also be classes of selectable applications. Thus,

$$\mathcal{U}' = \begin{cases} \mathcal{U} & \text{if } \forall A.C.m \in \mathcal{M}_A, \forall A'.C'.m' \in \mathcal{I}(A.C.m), A' \in \mathcal{A} \setminus \mathcal{U} \\ & \qquad \wedge (A.C \leq A'.C') \Rightarrow (A' \in \mathcal{A} \setminus \mathcal{U}) \\ \mathcal{U} \cup \{A\} & \text{otherwise} \end{cases}$$

**Lemma 1.** *Let $(A_i)_i$ be a list of applications and $\mathcal{S}$ be the system*

$$\mathcal{S} = install((A_n, P_{A_n}), d_n, \ldots install((A_0, P_{A_0}), d_0, \mathcal{S}_\emptyset))$$

*where $\mathcal{S}_\emptyset = (\mathcal{D}, \emptyset, \emptyset, \delta_\emptyset, (\emptyset, \mathcal{D}, rules_\emptyset))$ ($\delta_\emptyset$ and $rules_\emptyset$ are defined only on $\emptyset$) and each application passes the verification to be loaded in its domain. Then $\mathcal{S}$ is secure. Proof of this lemma is given in Appendix A.1.*

*Removal.* Note that removing an application does not break the consistency of the properties our algorithms ensure. So no additional checking is necessary.

### 3.2 Addition of New Domains

Designers of applications may want to change the application by adding new domains, for example to offer services to new partners.

**Definition 6 (On-device verification of access from new domains).** *Let $M$ be a set of methods of an application $A$ and $D$ a set of domains. Access to the methods $M$ can be granted from the domains $D$ if for each method $A.C.m \in M$:*

$$\forall A'.C'.m' \in \mathcal{I}(A.C.m), D \subseteq rules(A'.C'.m') \tag{4}$$

$$\forall A'.C', A'.C'.m \leq A.C.m \Rightarrow D \subseteq rules(A'.C'.m) \tag{5}$$

Note that (4) and (5) of Definition 6 correspond respectively to (1) and (3) of Definition 5. We do not need to recheck (2) since the set of domains from which calls to $A.C.m$ are granted can only grow.

Once the extended policy is verified, we update the system thus:

$$add((M, D), A, (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, (\mathcal{M}, \mathcal{D}, rules))) = \\ (\mathcal{D}, \mathcal{A}, \mathcal{U}, \delta, (\mathcal{M}, \mathcal{D}, rules \sqcup (m \in M \mapsto D))).$$

This extension of the set of domains from which access is granted can be checked for consistency following the Algorithm 3.3.

```
1: for all A.C.m of M do
2:    if A.C ≤ A'.C' and ∃A'.C'.m and D ⊈ rules(A'.C'.m') then
3:       return FAIL
4:    for all invoke A'.C'.m' bytecode do
5:       if D ⊈ rules(A'.C'.m') then
6:          return FAIL
7: rules ← rules ⊔ (A.C.m ∈ M ↦ D)
```

**Algorithm 3.3.** Consistency verification when granting access to $M$ from $D$.

### 3.3 Integration in GlobalPlatform/Java Card

The algorithms described in Section 3 have several practical requirements:

1. to have access to application's code before it is installed and selectable;
2. to know control flow policies of shared methods before their code is analyzed;
3. to keep persistent data related to control flow policies of already loaded applications across installation of new applications;
4. to prevent an installation or an update if it would break the security policy of already loaded applications.

For these requirements to be met, it is mandatory to integrate the algorithms in the application loading process. To ensure that verifications cannot be bypassed, our algorithms have to be integrated either directly in the implementation of GlobalPlatform, and more precisely the LOAD command implementation, either in a *Controlling Authority*, as described in GlobalPlatform specifications.

Satisfaction of the requirement 2 is intimately linked to the specific development and deployment processes of Java Card. The Connected Edition of Java Card 3.x (as well as standard Java) can directly load traditional class files, so control flow policies are simply inserted as special attributes in the bytecode *before* methods bytecode. In other Java Card releases (2.x and Java Card 3.x Classic Edition), all class files required to instantiate an applet have to be bundled in a CAP file. The conversion into CAP file involves many tasks such as bytecode verification (types, operand stack, valid instruction set, *etc.*), bytecode transformation (typed instructions), instantiation of static containers, and bytecode reordering for efficient loading in card's memory in a single pass. Our algorithms are designed to deal with such a constraint. To store control flow policies, we rely on special containers, called *custom components*. As an alternative, or if custom components are already used for other purposes, encoded control flow policies can be embedded into static components of CAP files. In both cases the conversion process is amended to include the control flow policies.

To deal with addition of new domain(s) to the policy of a method, we provide APDU commands implemented in a simple applet installed in the controlling authority domain or the card issuer domain to load the policy update. We rely on PKI mechanism of security domains to establish a secure channel with the domain that hosts the shared method whose policy has to be updated. If it succeeds, we process control flow policy verification to accept or reject this update thanks to the Algorithm 3.2.

### 3.4 Application to a Multi-application Use Case for Smart Cards

To illustrate the techniques presented in the two previous sections, we consider a multi-purpose smart card on which different stakeholders install their applications. This use case follows the style of existing use cases from the literature [1, 2] with enhancements of the deployment scenarios in order to include post-issuance (un)installation of applications.

The Figure 1 gives an overview of the use case. At the beginning, the card is issued with an electronic purse from Bank implemented in the `Purse` applet

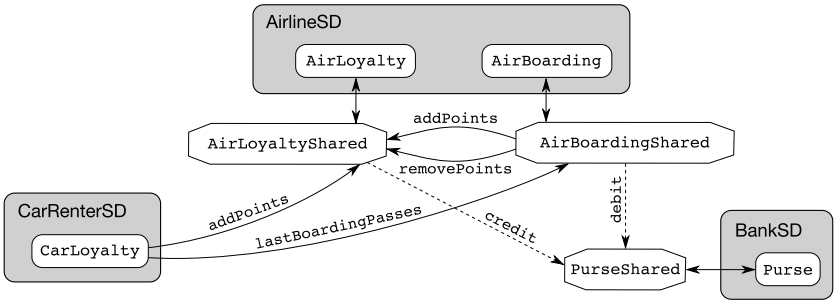



**Fig. 1.** Overview of applications interactions in the considered use case

installed in the BankSD security domain with privileges delegated to BANK. This applet shares some services to debit and to credit the purse through a registered instance of the `PurseShared` object. BANK wants to strictly ensure that applications that uses its services are allowed to, according to its own security policy, and of course that authorized applications do not relay service invocations from non-authorized ones. When the card is issued, control flow policies of the methods declared in `PurseShared` are empty, *i.e.* no application outside the domain BankSD can use `Purse`'s services.

Later, the card issuer creates two new security domains CarRenterSD for the company CARRENTER and AirlineSD for the company AIRLINE, and grants them the necessary privileges to manage the content of their respective domains. CARRENTER wants to install its loyalty application `CarLoyalty` in its domain, while AIRLINE wants to install its loyalty application `AirLoyalty` and `AirBoarding`, an application to manage electronic boarding passes. AIRLINE's applications interact in the following way: some points are credited when a boarding pass is "consumed", and some points are debited to pay additional charges at check-in, for example in case of excess luggage. AIRLINE's applications offer some services only to its partner CARRENTER through registered instances of `AirBoardingShared` and `AirLoyaltyShared` objects: `CarLoyalty` can credit some loyalty points to `AirLoyalty`, and can check last boarding passes from `AirBoarding` to offer discount prices to their customers. AIRLINE thus permits the methods defined in `AirLoyaltyShared` and `AirBoardingShared` to be called from CarRenterSD. `AirLoyalty`, `AirBoarding` and CARRENTER can be safely installed at this point since none of them use `Purse`'s services, and their interactions are allowed.

Finally, AIRLINE decides to subscribe to BANK's services and consequently updates its applications: `AirBoarding` wants to debit the purse when the amount of loyalty points is too low to pay additional charges at check-in and to buy duty-free goods according to the currently applicable boarding pass. Conversely, `AirLoyalty` wants to credit `Purse` when its amount of gathered loyalty points exceed some predefined amount. BANK consequently updates the control flow policy of the shared methods `debit` and `credit` of `PurseShared` registered by `Purse` through APDU commands. This update is straightforward since no application outside BankSD actually uses these methods. To update its applications, AIRLINE first

uninstall its existing applications on the card. Then, AIRLINE installs its new `AirBoarding` application that remains not selectable because of unresolved dependencies with `AirLoyalty`. It is important to notice that this behavior has nothing to do with functional dependencies but that it is induced by our incremental analysis that requires the control flow policies of `AirLoyalty` and `AirBoarding` to be known to verify `CarLoyalty`. However, installation of `AirLoyalty` is rejected by our verification process because its shared method `addPoints` does not respect the control flow policy attached to the method `credit` of the `PurseShared`. The control flow policy of the method `credit` states that it can be called only from the security domain AirlineSD, but the method `addPoints` invokes the method `credit` and the policy of `addPoints` permits CarRenterSD to call it. In fact, the new loyalty application of AIRLINE could be used by `CarLoyalty` to credit the purse of BANK via the loyalty application of AIRLINE while CARRENTER has not paid to use the services supplied by BANK.

## 4 Related Work

Several methods have been proposed to strengthen the security of open multi-application systems [3], but few are devoted to small devices such as mobile phones or smart cards. The work of Huisman *et al.* [4, 5] is the closest to our work. Their model is proved to be sound and complete for properties expressed in simulation logic, and is implemented in a tool that accepts Java bytecode as input. However, their proposal is not dedicated to be embedded on small devices, and it lacks concrete means for ensuring deployment of verified applets only, especially in an "open" context where it is crucial that composition is done on-device. Barthe *et al.* [6] have a similar approach to Huisman *et al.* but rely on temporal logic and lack the same features. Stack inspection [7] could also be applied to enforce control flow policies but this monitoring approach cannot be concretely applied on smartcards without significant run-time speed penalty. The static analysis of control flow paths proposed by [8] could also be modified to enforce the control flow policies described in this paper, but it is on off-device approach not desgined to fit the requirements of constrained systems.

Concretely, few proposals have be made for static on-device verification of some properties, especially security properties. Leroy [9] exhibits problems encountered to verify properties, like type-checking or stack under/over-flow, on-device for at loading-time only for Java bytecode. Ghindici *et al.* [10, 2] have followed a similar approach for secure information flow purposes. They rely on a sound and complete model to compositionally verify on-device proofs (inspired by proof-carrying code methods [11]) computed off-device and then embedded in Java bytecode. As Leroy's work, it does not require code signing. In fact, our work and the work presented of Ghindici *et al.* are complementary as they do not focus on the same things: they are following data while we are focusing services uses. For this purpose they use PCC-like mechanism with off-device computation of proofs while our model does not use any off-device calculus.

Certification based-mechanisms are in fact widely used in small devices. Code signing is the simplest kind of such mechanisms. This process is used in several existing devices such as smart cards (SIM cards, credit cards, *etc.*) but also iPhone, Android and JavaME. Although code signing provides a simple but strong way to ensure both origin and integrity of code, it is not designed to deal with application behaviors, especially their interactions with each other.

Relying on code signing, many systems restrict authorized behaviors according to the origin of the application, as recommended by the *Application Security Framework* of the *Open Mobile Terminal Platform*[6]. A first simple example comes from the Android platform where an application can specify that its shared services must only be invoked by code signed with the same signature. *Security-by-contract* approaches [12, 1, 13, 14, 15] also rely on code signing to bind application code with some *contracts* that specify the relevant features of an application (e.g. use only HTTPS connections, use a shared service of application $A$, *etc.*) that can include interactions with other applications and/or its host platform. Contracts are in practice not inferred automatically from application code, but written by developers. Contracts have to be enforced on-device by additional mechanisms, such as those described in this paper. Moreover some security-by-contract approaches such as [1] are devoted to closed systems where everything can be computed off-device and thus without any dynamic loading of code unknown before issuance of the target device.

McDaniel *et al.* [16] have introduced more refined features in Android contracts along with the Saint framework to enforce them at run-time. This approach, as Kirin rules [17] for malware detection, is far more sophisticated than the model presented in this paper, but suffers from a strong drawback: it is not sensitive to collusion of applications and can thus be easily cheated.

## 5 Conclusion

In this paper we have proposed a proved model for controlling methods invocations on open and constrained Java-based systems that support dynamic (un)loading of applications. This model has several strong features. It is suitable for autonomous systems because everything is computed on-device without any third-party. It also fits requirements of small constrained devices, such as Java-enabled smart cards, because verification is performed on-the-fly at installation-time in an incremental way with very low memory requirements.

Future works will offer support for deeper modifications on-device, after issuance, especially extended modifications of control flow policies for already loaded applications. In addition, it could also be interesting to propose an implementation for the Android platform, which is mostly Java-based, to enhance existing security mechanisms with an efficient way to detect applications collusion.

[6] Open Mobile Terminal Platform (`http://www.omtp.org`)

## References

[1] Bieber, P., Cazin, J., Wiels, V., Zanon, G., El-Marouani, A., Girard, P., Lanet, J.L.: The PACAP Prototype: A Tool for Detecting Java Card Illegal Flow (chapter 3). In: Attali, I., Jensen, T. (eds.) JavaCard 2000. LNCS, vol. 2041, pp. 25–37. Springer, Heidelberg (2001)

[2] Ghindici, D., Simplot-Ryl, I.: On Practical Information Flow Policies for Java-Enabled Multiapplication Smart Cards. In: Grimaud, G., Standaert, F.X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 32–47. Springer, Heidelberg (2008)

[3] Yoshioka, N., Washizaki, H., Maruyama, K.: A survey on security patterns. Progress in Informatics (5), 35–47 (2008)

[4] Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)

[5] Huisman, M., Gurov, D., Sprenger, C., Chugunov, G.: Checking absence of illicit applet interactions, a case study. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 84–98. Springer, Heidelberg (2004)

[6] Barthe, G., Gurov, D., Huisman, M.: Compositional verification of secure applet interactions. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 15–32. Springer, Heidelberg (2002)

[7] Besson, F., Blanc, T., Fournet, C., Gordon, A.D.: From stack inspection to access control: a security analysis for libraries. In: Proceedings of the 17th workshop on Computer Security Foundations (CSFW 2004), p. 61. IEEE Computer Society, Los Alamitos (2004)

[8] Sistla, A.P., Venkatakrishnan, V.N., Zhou, M., Branske, H.: CMV: automatic verification of complete mediation for java virtual machines. In: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2008), pp. 100–111. ACM, New York (2008)

[9] Leroy, X.: Bytecode verification on Java smart cards. Software – Practice & Experience 32(4), 319–340 (2002)

[10] Ghindici, D., Grimaud, G., Simplot-Ryl, I.: An Information Flow Verifier for Small Embedded Systems. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 189–201. Springer, Heidelberg (2007)

[11] Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 106–119. ACM, New York (1997)

[12] Bieber, P., Cazin, J., Girard, P., Lanet, J.L., Wiels, V., Zanon, G.: Checking Secure Interactions of Smart Card Applets: extended version. Journal of Computer Security 10, 369–398 (2002)

[13] Dragoni, N., Massacci, F., Naliuka, K., Siahaan, I.: Security-by-contract: Toward a semantics for digital signatures on mobile code. In: López, J., Samarati, P., Ferrer, J.L. (eds.) EuroPKI 2007. LNCS, vol. 4582, pp. 297–312. Springer, Heidelberg (2007)

[14] Dragoni, N., Massacci, F., Schaefer, C., Walter, T., Vetillard, E.: A Security-by-Contract Architecture for Pervasive Services. In: SECPerU, pp. 49–54 (2007)

[15] Ion, I., Dragovic, B., Crispo, B.: Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In: ACSAC, pp. 233–242 (2007)

[16] Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in Android. In: Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009), pp. 340–349. IEEE Computer Society, Los Alamitos (2009)
[17] Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: CCS 2009, Chicago, IL, USA, pp. 235–245. ACM, New York (November 2009)

# A Appendices

## A.1 Proof of Lemma 1

*Proof.* We prove by induction on the number of methods installed that, for any path from some $m_1$ (we abbreviate full names here) to some $m_2$ in the call graph for $\mathcal{S}'$, $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_2)$ and that for all methods $m_1$ and $m_2$ such that $m_1 \leq m_2$, $rules(m_2) \subseteq rules(m_1)$, which entails the result. This is obvious for $\mathcal{S}_\emptyset$. Let $m$ be a method added to the system in domain $d$ after passing the verification. And let $m_1$ and $m_2$ be two methods such that there is a path $m_1 \dots m_{i-1}.m.m_{i+1} \dots m_2$ in the call graph. By the induction hypothesis, we get $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_{i-1})$ and $\{\delta(m_{i+1})\} \cup rules(m_{i+1}) \subseteq rules(m_2)$.

By definition of the call graph, there is some method $m' \in \mathcal{I}(m_{i-1})$ with $m \leq m'$. If $m' = m$, by Rule 2, we get $rules(m_{i-1}) \subseteq rules(m)$. Otherwise, there is an edge from $m_{i-1}$ to $m'$ so we get $rules(m_{i-1}) \subseteq rules(m')$ by the induction hypothesis. And by Rule 3, we get $rules(m') \subseteq rules(m)$.

We also know from the call graph that there is some method $m'' \in \mathcal{I}(m)$ with $m_{i+1} \leq m''$. By Rule 1, we get $rules(m) \subseteq rules(m'')$ and by induction hypothesis we know $rules(m'') \subseteq rules(m_{i+1})$. Summing up those results, we get $\{\delta(m_1)\} \cup rules(m_1) \subseteq rules(m_2)$.

Let $m_1$ and $m_2$ be two methods such that $m_1 \leq m_2$. If both methods are different from $m$, $rules(m_2) \subseteq rules(m_1)$ entails from the induction hypothesis. If $m_1 = m$, we get the result by Rule 3. We cannot have $m_2 = m$ and $m_1 \neq m$ since the Java loading mechanism ensures that $m_1$ is loaded after $m$. □

# Efficient Symbolic Execution for Analysing Cryptographic Protocol Implementations

Ricardo Corin[1,2] and Felipe Andrés Manzano[1]

[1] FaMAF, Universidad Nacional de Córdoba, Argentina
[2] CONICET

**Abstract.** The analysis of code that uses cryptographic primitives is unfeasible with current state-of-the-art symbolic execution tools. We develop an extension that overcomes this limitation by treating certain concrete functions, like cryptographic primitives, as *symbolic functions* whose execution analysis is entirely avoided; their behaviour is in turn modelled formally via rewriting rules. We define concrete and symbolic semantics within a (subset) of the low-level virtual machine LLVM. We then show our approach sound by proving operational correspondence between the two semantics. We present a prototype to illustrate our approach and discuss next milestones towards the symbolic analysis of fully concurrent cryptographic protocol implementations.

## 1 Introduction

Despite a large amount of research devoted to formally analyze cryptographic protocols (see e.g. [8,6,3,4,18,10]), the thorough analysis of existing industrial protocol implementations (like e.g. OpenSSL) remains largely unexplored. The difficulty is that exploring real low-level protocol code is much more complex than the formal models that are fed to protocol analyzers. For instance, implementations need to deal with memory manipulation, bitarray arithmetics, and intricate message formatting, all of which are abstracted away in formal models.

Previous research in this area focuses on extracting and verifying formal models from implementation code. This includes the work of Goubault [10], where protocols written in C are abstracted into horn-clauses. Another direction is taken by Bhargavan et al. [2], which develops an implementation of TLS [7] coded in ML. In this case, the implementation is interoperable with other industrial implementations and can be analyzed by automatically extracting and verifying its corresponding formal model (using Proverif [3] and Cryptoverif [4]). Still, these tools are not applicable to legacy or binary code. Some quite recent works attempt to address this; see e.g. the roadmap in [1] for extracting formal models from C legacy code (or earlier work for Java [11]); other work [16] takes the inverse approach: generating monitors from formal models that prevent invalid or malicious messages forbidden by the protocol specification.

Recently, methods based on symbolic execution (developed initially by [12]) have been proved to scale very well to real life non-protocol code. Here, the code is dynamically explored through all its branches looking for implementation bugs

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 58–72, 2011.

like memory manipulation errors. In order to avoid trying the entire (arbitrarily large) input space, program inputs are assumed to be symbolic variables that remain uninstantiated (but become constrained) at execution time. Whenever the program reaches a conditional branch that operates on symbolic terms, the execution is forked; each branch is taken, and the condition (or its negation) is added to a store of path conditions. Tools like Bitblaze [17], Catchconv [15] and KLEE [5] have been succesfully used to find bugs in real code like **gnutils**. We focus on KLEE, which works on top of the LLVM [14], a virtual machine that is becoming popular for its efficiency and flexibility. There are several compilers that output LLVM code (e.g., gcc and clang) for many architectures, using powerful optimizations that benefit from its type-rich framework. LLVM code is closer to machine code and lower-level than a language like C; thus, performing analysis at this level is more realistic and can catch compiler-introduced bugs. KLEE uses STP [9], a bitarray SMT solver to detect unfeasible branches. Additionally, whenever a program reaches an unsafe state (e.g. memory corruption), STP can be used to generate a counterexample by providing triggering values for the (symbolic) inputs.

Our aim is to develop a symbolic execution analysis that can be applied to code that uses cryptographic primitives, like security protocols. Existing symbolic execution tools can not deal with such code, as the search space size blows up when it explores the insides of cryptographic functions.

In this work, we propose an extension that avoids such difficulty by replacing concrete functions with "symbolic" ones, thus preventing their exploration. In order to specify the behaviour of symbolic functions and to allow the execution to progress, symbolic functions are endowed with rewriting rules that detail abstractly their properties (e.g., that decryption inverts encryption).

We aim for fully formal results, and so start by rigourously formalising the LLVM language and its semantics. In summary, our net contributions are:

1. A formalization of the LLVM, with its syntax and concrete ideal semantics (Section 2).
2. A formalization of symbolic execution (Section 2.4). We develop an alternative symbolic semantics, and formally prove that symbolic execution is sound, in the sense that symbolic executions correspond to concrete ones (Lemma 1 in Section 2).
3. Symbolic functions (Section 3). We introduce the concept of symbolic functions that abstract away concrete functions we are not interested in analyzing. Not every concrete function can be turned into symbolic, so we identify the necessary conditions. Then, we give a new semantic rule that enables the symbolic execution of a symbolic function.
4. Efficient code analysis and Rewriting rules (Section 3.3). The behaviour of symbolic functions is given in terms of rewriting rules that specify how different symbolic terms may be rewritten to simpler terms, so that standard symbolic execution can progress. This results in efficient symbolic analysis that would be impossible to achieve otherwise.

5. Soundness (Section 3.4). We prove operational correspondence between the semantics with symbolic functions and the concrete ones (Theorem 1).
6. Prototype (Section 4). We develop a prototype coded as a non-intrusive patch of KLEE, and use it to illustrate our method with an example.

## 2 LLVM

To the best of our knowledge, the syntax and semantics of LLVM have not been formalized yet, besides the natural language reference given at the LLVM website [14]. Thus, we start by formalizing a subset of the LLVM language and provide both concrete and symbolic operational semantics.

### 2.1 Syntax

We illustrate LLVM code by means of an example.

```
i8 dec () {
   i8 c = inp();
   if (c <= 0)
     return 0;
   return c-1;
}
```

```
define i8 @dec() {
entry:
    %c = call i8 @inp()
    %cond = icmp slet i8 %c, 0
    br i1 %cond, label %true_b, label %false_b
true_b:
    ret i8 0
false_b:
    %res = sub i8 %c, 1
    ret i8 %res
}
```

On the left we show a small C program with its corresponding LLVM code on the right. The LLVM code decreases an input `%c` by `1` while it is still positive, otherwise returns `0`. The function `@dec` returns an integer of one byte (type `i8`), and has three *basic blocks*, each tagged with a label: `entry`, `true_b`, and `false_b`. Identifiers starting with `@` are globally accessible, while identifiers starting with `%` are local bindings. This code uses instruction `call` to input a byte using the `inp` function[1], then stores in `%cond` the result of comparing the input byte `%c` to `0`; there, execution branches (using the `br` instruction) to the `true_b` or `false_b` label. If we jump to `false_b`, `%c` is decreased by `1`. Finally, execution ends with by executing the return instruction `ret`.

The complete LLVM language contains many more instructions than the ones shown in this example; the avid reader can refer to the informal description [14].

### 2.2 Semantics

We provide two different semantics: one concrete and one symbolic. The former is *ideal* and intuitive, although difficult to reason with. The latter allows for symbolic execution and enables efficient path exploration. We show an operational correspondence between both semantics in Lemma 1, at the end of this section.

[1] *inp* is defined by us to simplify the management of interactive inputs.

*Concrete Contexts.* LLVM specifies two kinds of storages: bindings and memory.

The bindings can be either global or local: local bindings $\mathcal{L}$ maps identifiers starting with `%` to typed values, for instance `%cond` to `(i8, 0)`; global bindings $\mathcal{G}$ maps identifiers starting with `@` to (typed) memory addresses; for instance `@dec` maps to a memory location with the first instruction of function 'dec'.

We model memory as a partial map $\mathcal{M}$ from addresses to bytes.

The execution state is represented by a *context*, defined as a tuple $\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle$. Here, the program counter $pc$ is an address (s.t. $\mathcal{M}(pc)$ holds the first byte of the next instruction to be executed). The stack of function frames $fs$ is a sequence of *function frames*, where each frame $(id, \mathcal{L}, ret, \mathcal{A})$ contains all the data locally needed in context of a function call: $id$ is the local binding name from the calling function where to put the result of the current function in execution; $\mathcal{L}$ are the local bindings, and $ret$ is the return address where to continue executing after this function returns. Finally, $\mathcal{A}$ is a set of memory addresses reserved during the execution of this function.

*Initial context.* We assume an initial context $\langle pc_0, \mathcal{M}_0, \mathcal{G}_0, (id_0, \mathcal{L}_0, ret_0, \emptyset)\rangle$ in which the operating system has loaded the LLVM code into memory $\mathcal{M}_0$, and populated the bindings appropiately. $pc_0$ is the memory address of the first instruction of a *@main* function. $id_0$ and $ret_0$ are addresses (their values are irrelevant since our semantics does not model an operating system; executions simply get stuck at the last step and no value is returned). $\mathcal{L}_0$ contains the parameters to the *@main* function.

### 2.3 Concrete Semantics

Table 1 shows concrete semantic rules. All the operations on bitarrays refer to actual, concrete operations. Since we store code in memory, we need to parse its opcode to know which instruction we are executing. This is modelled by the auxiliary function $op_{\mathcal{M}}(pc)$.

Rules (SUB) and (ICMP) perform simple bitarray arithmetics. In the former, the operator $op_2$ is subtracted from $op_1$ and the result is placed in binding $id$. (Here, auxiliary function $v(x, \mathcal{L})$ evaluates to the pair $(type, value)$ defined in $\mathcal{G}$ or $\mathcal{L}$.) In the latter, the bit with the result of the comparison is stored in $id$. Rule (CALL) is used to jump to a subroutine. We set as next program counter the starting address $addr$ of $f$, and create a new function frame containing the argument parameters ($id_i$ mapping to $v_i$), and as return address $nxt_{\mathcal{M}}(pc)$. Rule (RET) returns to the calling function: it pops the function frame, jumps to address $ret$, and frees the used local memory $\mathcal{A}$.

Rule for input (INP) takes a byte $b$ (of type $i8$) and stores in the local binding $id$. This rule is non-deterministic and effectively forks computation in 256 ways, one for each possible input value. Rules (BRT) and (BRF) are conditional branches. Rules (ALLOCA), (LOAD), (STORE), are used for memory manipulation; here, functions *pck* and *unpack* store expressions into memory bytes (notation $\{p\}_n$ means $\{p, \ldots, a + n\}$). (ALLOCA) finds a pointer $p$ in memory

where to allocate the chunk of memory. (LOAD) looks up for the memory chunk starting in $x_p$, and unpacking it and returning it in $v_p$, (STORE) makes the inverse operation.

*Concrete execution traces.* The semantic rules define a step relation $\longrightarrow$. For convenience, we add a label $l$ to $\longrightarrow$ and write $\xrightarrow{l}$, where $l$ is the instruction execution (i.e., $l = op_{\mathcal{M}}(pc)$). A sequence of (formal) labels is a trace $tr$, and we define $c \xrightarrow{tr}{}^{*} c' \doteq \exists c_1, \ldots, c_n : c \xrightarrow{l_1} c_1 \rightarrow \ldots \xrightarrow{l_n} c_n \rightarrow c'$ with $tr = l_1 \ldots l_n$.

### 2.4 Symbolic Semantics and Symbolic Execution

Although the concrete semantics is intuitive and easy to understand, it is not easy to implement and to analyze on top of it (e.g. due to the non-deterministic behaviour of rule (INP) and the existential quantifiers in some of the rule preconditions like (ALLOCA)).

Here we consider an alternative *symbolic* semantics, that uses symbolic expressions. These expressions are either simple constants or symbolic variables $\iota$; more complex expressions are formed by applying operators $op_t$ of a type $t$.

**Definition 1.** *Symbolic expressions and conditions are given by:*

$$\begin{array}{ll} op_t & ::= +_t, -_t, /_t, *_t \\ exp & ::= (t, \iota) \mid (t, const) \mid exp\ op_t\ exp \mid exp.n \\ cop_t & ::= =_t, <_t, >_t, \&_t, |_t \\ cond & ::= exp\ cop_t\ exp \end{array}$$

Expressions are either basic typed constants $(t, const)$ or symbolic variables $(t, \iota)$, or formed by applying arithmetic operators $op_t$. Finally, for an expression $exp$ which is stored in memory as a sequence of bytes, we use projector $exp.n$ to project the $n$th byte. Condition expressions are formed by two expressions and one conditional operator $cop_t$. We assume a natural semantics to conditional expressions so we can decide whether a conditional expression $c\sigma$ holds or not, under a substitution $\sigma$ that maps symbolic variables to concrete bitarrays.

We use symbolic expressions and conditions to carry out the symbolic execution of an LLVM program. To this end we provide an alternative *symbolic* semantics. The first is to add a *constraint store* $\Sigma$ to contexts: $\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$ where $\Sigma$ is a sequence of symbolic conditions that we use to record the path conditions taken in each branch during execution. We also change the range of $\mathcal{M}$ and $\mathcal{L}$: they now map addresses and identifiers to symbolic expressions, and not concrete values.

**Definition 2 (Satisfiability of $\Sigma$).** *We write $\vdash \Sigma$ when there exists a substitution $\sigma$ s.t. every condition $c\sigma$ holds for every $c \in \Sigma$.*

In practice, this is decided by a bitarray SMT solver (STP [9] in our case).

**Table 1.** Concrete semantic rules (selected)

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{sub}\ \ t\ op_1, op_2 \qquad v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2})}{\begin{array}{c}\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \\ \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}\{id \rightarrow (t, x_{op_1} -_t x_{op_2})\}, ret, \mathcal{A}) :: fs\rangle\end{array}}\ \texttt{SUB}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{icmp}\ \ \texttt{cond}\ \ t\ op_1, op_2 \\ v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2}) \qquad \mathcal{L}' = \mathcal{L}\{id \rightarrow (\mathbf{i1}, x_{op_1}\ \texttt{cond}\ {}_t x_{op_2})\}\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs\rangle}\ \texttt{ICMP}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{call}\ f(a_0, a_1, \ldots a_n) \\ \mathcal{G}(f) = (t(t_0 id_0, t_1 id_1 \ldots t_n id_n), fp) \qquad \forall k : v(a_k, \mathcal{L}) = (t_k, v_k)\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle fp, \mathcal{M}, \mathcal{G}, ((t, id), \{id_0 \rightarrow v_0, \ldots id_n \rightarrow v_n\}, nxt_{\mathcal{M}}(pc), \emptyset) :: fs\rangle}\ \texttt{CALL}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = \texttt{ret}\ t\ rslt \\ fs = ((t, id), \mathcal{L}, ret, \mathcal{A}) :: ((t_1, id_1), \mathcal{L}_1, ret_1, \mathcal{A}_1) :: fs' \qquad v(rslt, \mathcal{L}) = (t, x_p)\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle ret, \mathcal{M} - \{\mathcal{A} \times *\}, \mathcal{G}, (id_1, \mathcal{L}_1\{id \rightarrow (t, x_p)\}, ret_1, \mathcal{A}_1) :: fs'\rangle}\ \texttt{RET}$$

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad b\ byte\ in\ \mathcal{L} \qquad \mathcal{L}' = \mathcal{L}\{id \rightarrow (\texttt{i8}, b)\}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs\rangle}\ \texttt{INP}$$

$$\frac{op_{\mathcal{M}}(pc) = \ \texttt{br}\ c\ \texttt{labelt}\ l_1\ \texttt{labelt}\ l_2 \qquad \mathcal{L}(l_1) = (\mathbf{label}, bb) \qquad v(c, \mathcal{L}) = (\mathbf{i1}, 1)}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle}\ \texttt{BRT}$$

$$\frac{op_{\mathcal{M}}(pc) = \ \texttt{br}\ c\ \texttt{labelt}\ l_1\ \texttt{labelt}\ l_2 \qquad \mathcal{L}(l_2) = (\mathbf{label}, bb) \qquad v(c, \mathcal{L}) = (\mathbf{i1}, 0)}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, fs\rangle}\ \texttt{BRF}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{alloca}\ t, i32\ n \qquad \exists p : \mathcal{A}' - \mathcal{A} = dom(\mathcal{M}' - \mathcal{M}) = \{p\}_{size(t)\times n} \\ p + size(t) \times n < K \qquad \mathcal{L}' = \mathcal{L}\{id \rightarrow (t*, p)\}\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (res, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}', \mathcal{G}, (res, \mathcal{L}', ret, \mathcal{A}') :: fs\rangle}\ \texttt{ALLOCA}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{load}\ t{*}\ p \\ v(p, \mathcal{L}) = (t, x_p) \qquad \{x_p\}_{size(t)} \subseteq dom(\mathcal{M}) \qquad v_p = unpck(t, \mathcal{M}(\{x_p\}_{size(t)}))\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (id, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (id, \mathcal{L}\{id \rightarrow (t, v_p)\}, ret, \mathcal{A}) :: fs\rangle}\ \texttt{LOAD}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = \ \texttt{store}\ t\ val, t{*}\ p \\ v(p, \mathcal{L}) = (t, x_p) \qquad v(val, \mathcal{L}) = (t, x_{val}) \qquad \{x_p\}_{size(t)} \subseteq dom(\mathcal{M})\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}\{x_p \rightarrow pck(t, x_{val})\}, \mathcal{G}, fs\rangle}\ \texttt{STORE}$$

**Symbolic semantics.** Besides changing the range of $\mathcal{M}$ and $\mathcal{L}$, all the previous arithmetic rules now deal with expressions, not concrete operations. Rule (ADD), for instance, has operation $+_t$ not being the concrete one but the symbolic expression operator.

Table 2 shows the new symbolic semantic rules wrt the previous concrete. Rule (SINP) now returns a fresh symbolic $\iota$ variable (instead of non-deterministacally choosing a byte). Rules for branching now (SBRT) and (SBRF) now are both applicable, as long as the (updated) constraint store $\Sigma'$ is solvable; this effectively models the forking of each execution branch.

**Table 2.** Symbolic semantic rules (selected)

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad \iota \textit{ fresh in } \mathcal{L} \qquad \mathcal{L}' = \mathcal{L}\{id \to (\texttt{i8}, \iota)\}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs, \Sigma\rangle}\texttt{SINP}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{ br } c \texttt{ labelt } l_1 \texttt{ labelt } l_2 \\ \mathcal{G}(l_1) = (\textbf{label}, bb) \qquad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 1) \qquad \vdash \Sigma' \end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma'\rangle}\texttt{SBRT}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{ br } c \texttt{ labelt } l_1 \texttt{ labelt } l_2 \\ \mathcal{G}(l_2) = (\textbf{label}, bb) \qquad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 0) \qquad \vdash \Sigma' \end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, fs, \Sigma'\rangle}\texttt{SBRF}$$

In the symbolic version of rule ALLOCA we enforce that both the return pointer ($p$ in rule ALLOCA) and the size $n$ of the requested memory are concrete. We are not interested on exploring different executions that arise from alloca returning different pointers in memory, or of symbolic sizes.

*Operational correspondence.* Given a symbolic execution trace $tr$, we let $\Sigma_{tr}$ denote the sequence of recorded path conditions at the end of executing $tr$. We say that $\Sigma_{tr}$ is $\sigma$-satisfiable when $c\sigma$ holds for each condition $c \in \Sigma_{tr}$. Given $\sigma$, we let $tr\sigma$ denote the result of applying $\sigma$ in each step of $tr$ and replacing applications of rule $SINP$ with rule $INP$ (and $SBRT$, $SBRF$), mapping each $\iota$ by $\sigma(\iota)$.

**Lemma 1 (Soundness of symbolic execution).** *Let $tr$ be a symbolic trace (i.e., one obtained with our symbolic semantics) with $\Sigma_{tr}$ $\sigma$-satisfiable. Then $tr\sigma$ is a valid concrete trace.*

This lemma formalizes the symbolic execution frameworks (e.g., KLEE). The converse to Lemma 1 does not quite hold: since we restrict to concrete memory sizes, programs that allocate memory based on inputs are not symbolically captured (see above).

## 3 Symbolic Functions

Our aim is to verify code that uses cryptographic primitives like encryption or hashing. This is difficult with current state-of-the-art symbolic execution tools. To see this, consider the piece of code:

```
int main () {
   int i;
   char buf[80];
   for (i=0; i<80; i++)
     buf[i]=inp();
   assert (0x12345678 == hash(buf,80));
   printf("OK\n");
}
```

where `hash` is a cryptographic hash function, like SHA-1 or MD5. This code is unfeasible to be symbolically analyzed, as one is basically asking for an input that inverts 0x12345678, which is exactly what `hash` is designed to make difficult. In this section we develop a method to abstract away from such calls, which we regard as "symbolic functions" (analogously to the abstraction of inputs into symbolic variables).

### 3.1 Symbolic Functions

The first thing to notice is that not every concrete function can be abstracted away into a symbolic one. A function that has arbitrary side-effects cannot be symbolically modelled as its behaviour is unknown and may change the execution state in an uncontrolled manner. Thus, we restrict ourselves to concrete *pure* functions that simply perform some (non-interactive) calculation; nevertheless this covers all the functions we are interested in (that is, cryptographic primitives). More precisely, symbolic functions satisfy:

1. The types and number of input and output parameters are statically known, and are simple pointers or basic (constant) values;
2. The function only returns values in the (previously allocated) memory locations given by its output parameters (that is, there are no side-effects);
3. The function is non-interactive and does not do any inputs (it's deterministic);
4. The function terminates in all of its possible inputs (that is, it never enters an infinite loop).

These conditions are needed to encapsulate all the behaviour of a function so that it can be easily abstracted away in a symbolic term.

### 3.2 Semantics

We are given a set of functions $f_1, \ldots, f_n$ which we are going to assume symbolic. For each symbolic function $f$ we assume given information about its parameters

and their lengths. For a parameter $p_i$ we let $len(f, p_i)$ to return its length (this can be either constant or another input parameter, which is then an identifier that needs to be looked up in the current state). We are also given a global precondition $pre_f$, modelled simply as a symbolic conditional expression, that specifies on which cases the function terminates.

We modify the symbolic semantics so that when a symbolic function is called a special SCALL rule is triggered Given a call to symbolic function $f$, the goal of rule SCALL is to avoid its execution, and simply create a symbolic term standing for each ouput parameter $o_i$ of $f$. We extend the grammar of symbolic expressions as follows: $exp ::= \ldots | \iota^c_{f,o_i}$. For each symbolic function $f$ with output parameter $pi$. A term $\iota^c_{f,pi}$ then represents the output $pi$ of a call to $f$ under context $c = \langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$.

We are now ready to define the SCALL rule:

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = id = \texttt{call}\ f(i_0, \ldots, i_n, o_0, \ldots o_m) \quad f \text{ symbolic} \quad \vdash \Sigma :: \{pre_f\} \\ \forall k : v(a_k, \mathcal{L}) = (t_k, v_k) \quad \mathcal{M}' = \mathcal{M}\{o_i \rightarrow \iota_{f,o_i,\mathcal{S}.j)}\}_{j=0\ldots len(f,o_i),i=0\ldots m} \\ \mathcal{S} = \langle pc, \mathcal{M}, \mathcal{G}, (res, \mathcal{L}, ret, \mathcal{A}) :: ffs, \Sigma \rangle \quad fs' = (res, \mathcal{L}\{id \rightarrow \iota_{f,o_i,\mathcal{S}}\}, ret, \mathcal{A}) :: ffs \end{array}}{\mathcal{S} \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}', \mathcal{G}, fs', \Sigma :: \{pre_f\} \rangle}\ \texttt{SCALL}$$

Briefly, this rule sets the symbolic outputs of $f$ as symbolic expressions, and continues to the next instruction.

### 3.3 Specifying the Behavior of a Symbolic Function

In formal methods for security (following Dolev-Yao [8]), behavior is usually modelled via simple rewriting rules stating that e.g. decryption inverts encryption. However, as we noticed above, in our setting we deal with functions that output several parameters, of variable lengths. Outputs of symbolic functions are disseminated in bytes in memory, which can then passed as parameters to other symbolic functions; at this point, a rewriting rule may specify that a given output can be rewritten.

Thus, our rewriting rules need to specify how whole sets of memory locations (bytes) can be substituted by another set of bytes. Moreover, we wish to allow several rewriting rules, each modelling a different behaviour. We assume that each rewriting rule has an associated precondition to it.

*Example 1.* Consider functions zip/unzip: `zip(inbuf, inlen, !outbuf, !outlen)` and `unzip(inbuf, inlen, !outbuf, !outlen)`. Both functions take two inputs and return two outputs. Assume that in memory location pointed by $o$ starts the (symbolic expression representing the) output *outbuf* of a call to *unzip*. Formally, right after calling *unzip*, we have in memory:

$$\{o + j \rightarrow \iota^c_{unzip,outbuf}.j\}_{j=0..outbuflen^c} \tag{1}$$

Here $c$ is the context at the time of calling *unzip*, say $c = \langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$, and $outbuflen^c$ denotes the actual value for parameter *outbuflen* in context $c$.

We have to look at the other input pointer parameter *inbuf* of unzip, say $i_1$; the rewriting rule has to enforce that the input comes from a call to *zip*:

$$\{i_1 + k \rightarrow \iota^{c'}_{zip,outbuf.k}\}_{k=0..outbuflen^{c'}} \tag{2}$$

Here $c'$ contains the provided input to zip, namely *inbuf* of length *inbuflen*, pointed by $i$:

$$\{i + k \rightarrow inbuf_k\}_{k=0..inbuflen^{c'}} \tag{3}$$

At this point we are ready to rewrite 1) with

$$\{o + k \rightarrow inbuf_k\}_{k=0..inbuflen^{c'}} \tag{4}$$

provided that the precondition (for this rule)

$$pre_0 = inbuflen^{c'} <= outbuflen^{c} holds. \tag{5}$$

In summary, each rewriting rule consists of a series of matches over a context: (1), (2), (3), and the checking of a precondition (4); if this holds we can rewrite (1) with (4).

We now can add a rule that rewrites terms:

$$\frac{\mathcal{M}' \text{ result of rewriting in } \mathcal{M} \text{ with precondition } p \qquad \Sigma' = \Sigma :: p \qquad \vdash \Sigma'}{\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma\rangle \longrightarrow \langle pc, \mathcal{M}', \mathcal{G}, fs, \Sigma'\rangle}\texttt{REWR}$$

We need rewriting systems that are *decidable*. For the case of cryptography this is a well-studied problem; we can in fact restrict to rewriting systems which are *s*ubterm convergent, which is sufficient for decidability [13].

### 3.4 Operational Correspondence under Symbolic Functions

Calling symbolic functions with rule (SCALL) introduces symbolic function expressions which are of course not possible to solve with a bitarray solver like STP. However, such expressions may disappear by rewriting (that is, applications of the (REWR) rule). In that case, we can then call the bitarray solver and continue the execution. We thus need to separate in $\Sigma$ the constraints that depend on symbolic functions from those that do not.

**Definition 3.** *We say that $\Sigma$ is* fully $\sigma$-satisfiable *when $\Sigma$ is $\sigma$-satisfiable and every constraint in $\Sigma$ does not have occurrences of symbolic function expressions. We say that $\Sigma$ is* partially $\sigma$-satisfiable *when $\Sigma'$ is $\sigma$-satisfiable and $\Sigma'$ is the result of removing each constraint with symbolic functions from $\Sigma$.*

In practice, after calling a (constructor) symbolic function, execution continues for a while (possibly hitting a branch conditional that does not include an occurrence of a symbolic function), and then finally reaches a (deconstructor) symbolic function that is then rewritten off. Thus, it may happen that a partially solvable $\Sigma$ becomes fully solvable. Thus, we let rules for symbolic branching (that use $\vdash \Sigma$) to apply when $\Sigma$ is either fully or partially $\sigma$ satisfiable.

**Definition 4.** *We write* $[tr]$ *to denote the resulting trace of replacing each call to symbolic function* $f_i$ *using rule SCALL with a corresponding computation subtrace, and removing any call to rule REWR. We say that* $\Sigma_{tr}$ *is* functionally satisfiable *if* $\Sigma_{[tr]}$ *is satisfiable.*

**Theorem 1 (Soundness of Symbolic Execution under Symbolic Functions).** *Let tr be a symbolic trace (i.e., one obtained with our symbolic semantics with a given set of symbolic functions).*

1. *If* $\Sigma_{tr}$ *is fully* $\sigma$*-satisfiable, then* $[tr\sigma]$ *is a valid concrete trace.*
2. *If* $\Sigma_{tr}$ *is partially* $\sigma$*-satisfiable and functionally satisfiable, then* $[tr\sigma]$ *is a valid concrete trace.*

The first item (Theorem 1(1)) says that if we reach an execution that has no occurrences of symbolic functions (e.g., we rewrote them off), then we can use the underlying bitarray solver to find an exact assignment for inputs ($\sigma$) that reaches the same context concretely. The second item (Theorem 1(2)) says that even if we could not remove all symbolic functions, it *could* be possible to find a corresponding concrete execution (although this now depends on the path conditions that are still undecided).

## 4 Prototype

We developed a proof-of-concept prototype that works for symbolic functions that have a single output parameter (although they can take many inputs) of fixed length (this works for encryption/decryption primitives like Rijndael, or the hashing function SHA-1). As future work we plan to develop a general implementation that can deal with general symbolic functions as defined above.

KLEE's internal representation of symbolic expresions cannot handle our symbolic functions natively, since it only deals with expressions built from simple symbolic variables. We initially envisaged to augment KLEE's representation, but decided against so rather than modifying KLEE's internals we chose a less intrusive approach that works as an add-on wrapper for symbolic functions. KLEE already deals with environment and system calls by providing a hand-written library versions (for instance, a simpler version of `libc`). We follow a similar approach and code a wrapper that contains for each symbolic functions a code stub that works as follows:

1. The stub maintains a table that record the different calls made to it, along with the passed parameters and a unique *call identifier* related to that call;
2. Before accepting the passed parameters, the function preconditions are asserted, and the parameters are checked to see whether valid values and memory locations were provided;
3. If the same parameters were passed in an earlier call, so they exist already in the table, return the call identifier as response;
4. If new inputs are passed, try to deconstruct the call using the rewriting rules; if possible, return the (deconstructed) parameters in the table;

5. If the call cannot be deconstructed, a new entry is generated in the table, with a fresh random value generated as call identifier and provided as output.

The advantage of this approach is that all this code is fed along with the original program that can be analysed directly by KLEE+STP, since at this point we have implemented the symbolic functions themselves as a library.

Note that on item (3) KLEE will generate symbolic comparisons when running the table lookups. So this enables KLEE to find conditions over the symbolic input that was put (or copied) in the stubs' internal tables.

On item (4) if it can decontruct the arguments, a possibly symbolic value is taken from the tables and returned. If a case in wich undeconstructable arguments are passed, it simply fills the output with random fresh data, also saving that in the internal tables. We have assumed that new symbolic function expressions will not be used in any comparison outside the intervening functions.

*Example 2 (Encrypting and Decrypting).* We consider the C interface of Rijndael/AES, from the cryptographic API taken verbatim from RFC:

```
struct rijndael_ctx;
void rijndael_set_key(rijndael_ctx * ctx,
                      u_char *key, int key_len, int do_encrypt);
void rijndael_decrypt(rijndael_ctx * ctx,
                      u_char *clear, u_char *crypt);
void rijndael_encrypt(rijndael_ctx * ctx,
                      u_char *crypt, u_char *clear);
```

Several implicit preconditions over the memory buffers and parameters passed to these functions apply:

- `ctx` points to a `sizeof(*ctx)` sized memory buffer
- `key` points to a string of `key_len/8` bytes being the key
- `key_len` is either `64` or `128` or `256`.
- `clear` and `crypt` shall point to memory buffers of 16 bytes.

Now consider the following simple code:

```
int main() {
  int i;
  rijndael_ctx ctx;
  char input[16], encrypted[16], decrypted[16];
  for (i=0; i<16; i++) input[i]=inp();
  rijndael_set_key (&ctx, KEY, KEY_LEN, 0);
  rijndael_encrypt (&ctx, input, encrypted);
  rijndael_decrypt (&ctx, encrypted, decrypted);
  if (decrypted[0] == 'A')
      return 1;
  return 0;
}
```

Here, 16 bytes are read from input, then a rijndael context is set with a hard-coded key and its length. Then we encrypt and immediately decrypt the input data. In summary, we aim at being able to translate the symbolic conditions applied over the `decrypted` text to the original clear text `input`. We need to explore the two obvious paths, avoiding the pitfall of exploring the complexity of the crypto calls. The two trivial paths are walked using an `input` starting or not with the character 'A'.

We verified that if we mark the input as symbolic and we use the original `rijndael` implementation, KLEE fails to terminate and find the two simple path conditions (again, we tested a reasonable amount of time using the same system as in the previous experiments). This is not surprising of course since the cryptographic functions are designed on purpose to behave exactly like that.

The implementation of the symbolic function stubs for `rijndael` follows the description in Table 3. Using this implementation KLEE founds the two paths instantly.

**Table 3.** Rijndael stubs

**rijndael_set_key:** pushes a fresh symbolic value to `ctx` pointed memory, also it saves the relation `key[], key_len, do_encrypt -> ctx[]` in its internal table.

**rijndael_encrypt:** pushes a fresh symbolic value to `crypt` pointed memory, also it saves the relation `ctx[], clear[] -> crypt[]` in its internal table.

**rijndael_decrypt:** is a potential deconstructor. It seraches the tables for a `clear` text to return. All original values `(key, key_len, do_crypt, crypt)` shall match the apropiate rewriting rule. If there is no applicable rule it pushes a fresh symbolic value on `clear` pointed memory and saves the relation `ctx[], crypt[] -> clear[]` in its own table.

## 5 Conclusions and Future Work

The results reported in this paper are, although promising, just the first step towards being able to fully analyze cryptogaphic protocol implementations. As future work, we need to address the following issues:

- *Symbolic concurrent execution.* To analyze crypto protocols, we need to extend from sequential code to support concurrency. In this new setting, consider an initiator and a responder of a given protocol: we have an adversary that chooses inputs to both parties, and can in addition touch en-route messages. In this setting it is interesting to model how different symbolic inputs flow from one thread to another (we do not know of any standard symbolic execution framework for dealing with plain, non-crypto concurrency).
- *Computational and Symbolic Cryptography.* So far we have not focused on which kind of attacker model we are using; in this paper we only show how to substitute concrete functions (like crypto primitives) with symbolic counterparts (which are in turn implemented via tables), but have not focused on establishing security properties. In order to be able to replace concrete

crypto primitives by their ideal counterparts, though, we need to prove that the program behaves soundly: for instance, we can substitute ciphertexts only when the used cryptographic key has not been manipulated or leaked in any way by the program, and only passed to the relevant encryption function. Otherwise, using tables and rewriting rules is unsound. As future work, we are working on performing the relevant checks to ensure that we can safely proceed in this manner. Note that indeed if we prove that cryptographic materials are always used correctly then we can establish strong computational results that hold even if we consider realistic PPT adversaries.

– *Generic security properties.* KLEE tests for memory safety properties by automatically embedding tests in each state. Once we can replace concrete functions with symbolic ones (as discussed above), we wish to encode higher-level properties, that depend on the specification of the code we are analysing. In the case of cryptographic protocols, we need to be able to encode correspondence assertions [18] that relate different parts of the (joint concurrent) memory state.

We conclude by mentioning that the machinery developed in this paper can be used to abstract away from any functions, not necesarily cryptographic, that we want to avoid to analyse but still need to specify abstractly. Examples include functions like CRC32, or ZIP/UNZIP.

**Acknowledgements.** We wish to thank Cédric Fournet and the anonymous reviewers for their useful comments.

## References

1. Aizatulin, M., Gordon, A., Jurgens, J., Nuseibeh, B.: Verifying implementations of security protocols in c, `http://users.mct.open.ac.uk/ma4962/files/abstract202010.pdf`
2. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for tls. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM Conference on Computer and Communications Security, pp. 459–468. ACM, New York (2008)
3. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: CSFW, pp. 82–96. IEEE Computer Society, Los Alamitos (2001)
4. Blanchet, B.: A computationally sound mechanized prover for security protocols. IEEE Trans. Dependable Sec. Comput. 5(4), 193–207 (2008)
5. Cadar, C., Dunbar, D., Engler: D.r.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224 (2008)
6. Corin, R.: Analysis Models for Security Protocols. PhD thesis, University of Twente (2006)
7. Dierks, T., Rescorla, E.: The Transport Layer Security (tls) protocol. RFC 4346, Internet Engineering Task Force (April 2006), `http://www.ietf.org/rfc/rfc4346.txt`
8. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)

9. Ganesh, V., Dill, D.: Stp: A decision procedure for bitvectors and arrays, http://theory.stanford.edu/~vganesh/stp
10. Goubault-Larrecq, J.: Csur: Static analysis of C code (2002), http://www.lsv.ens-cachan.fr/csur/ Written in OCaml (12648 lines)
11. Jürjens, J.: Security analysis of crypto-based java programs using automated theorem provers. In: ASE, pp. 167–176. IEEE Computer Society, Los Alamitos (2006)
12. King, J.C.: Symbolic execution and program testing. ACM Commun. 19(7), 385–394 (1976)
13. Kremer, S., Delaune, S., Kremer, S.: Computing knowledge in security protocols under convergent equational theories (March 2009)
14. Lattner, C., Adve, V.: The LLVM language reference manual, http://llvm.org/docs/LangRef.html
15. Molnar, D.A., Wagner, D.: Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical report (2007)
16. Pironti, A., Jürjens, J.: Formally-based black-box monitoring of security protocols. In: Massacci, F., Wallach, D.S., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 79–95. Springer, Heidelberg (2010)
17. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
18. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols (1993)

# Predictability of Enforcement*

Nataliia Bielova and Fabio Massacci

University of Trento, Italy
lastname@disi.unitn.it

**Abstract.** The current theory of runtime enforcement is based on two properties for evaluating an enforcement mechanism: *soundness* and *transparency*. Soundness defines that the output is always good ("no bad traces slip out") and transparency defines that good input is not changed ("no surprises on good traces"). However, in practical applications it is also important to specify how bad traces are fixed so that the system exhibits a reasonable behavior. We propose a new notion of *predictability* which can be defined in the same spirit of continuity in real-functions calculus. It defines that there are "no surprises on bad input". We discuss this idea based on the feedback of an industrial case study on e-Health.

## 1 Introduction

Run-time monitoring is a well known technique to control an untrusted application that runs in an otherwise secure environment. During its run the application receives some inputs from the environment and produces some (tentative) outputs. If the monitor considers these i/o sequences legal according to some policy then it will let them pass, otherwise it will block them or somehow change them. This simple and intuitive description applies for a variety of monitors: from usage control and DRM [16], to control of downloaded applications in .NET mobile code [7], or from Javascript confinements in browsers [17] to Enterprise Service Bus enforcement for web services [10]. It can be implemented by wrappers as in Google's Caja, by inlining hooks with a separate policy decision point as in [7] or by inlining the monitor code as in Polymer or PSLANG [8].

While many research proposals and tools exist on the market, we have not found major deployments in the field. A reason may be that the overhead penalty is still significant but, based on our experience on a concrete case study, we argue that there is a deeper, more foundational reason.

Italian hospitals must guarantee the compliance of many business processes involving large amounts of money (reimbursements from public health authorities for drugs dispensation), significant privacy concerns (drugs can be related to HIV or other serious illnesses), major safety considerations (drugs might have

* We would like to thank Marta Zambetti, Marco Nalin, Andrea Micheletti and Daniela Marino from the Hospital San Raffaele for many useful discussions that helped to shape our proposal. This work has been partly supported by the EU under the projects EU-IP-MASTER, EU-FET-IP-SecureChange and EU-NoE-NESSoS.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 73–86, 2011.

serious side-effects), and compliance with many regulations. These regulations can change frequently and a run-time monitor could guarantee the compliance of each process with minor efforts: no change to processes or ESB architectures, we deploy an updated policy and we are done.

Unfortunately, a monitor must offer some guarantees to the hospital on what happens when things are not according to the policy. This is the point where we found a foundational gap. At present the only offered formal guarantees are

**transparency** the monitor will never touch a good execution;
**soundness** the monitor will never output a trace violating the policy.

These properties are currently used in all state-of-the-art papers on runtime enforcement [4,9,14,20,21]. The recently introduced property of completeness [14] is implied by transparency and soundness. In all practical settings these two properties are necessary but not sufficient.

**Reality Check 1.** *What the risk manager of the hospital wants to know is "What the monitor* normally *does when a doctor's action does not respect the policy?" Does it abort the whole transactions if a research protocol number is not entered (Schneider Security Automata [19])? Does it alert the head of the department if we are prescribing a drug out of stock (Pretschner's usage control [18])? Does it wait till the doctor opens the therapeutical notes before committing the transaction to the audit logs (Ligatti's longest valid prefix automata [13])?*

While we can implement a concrete enforcement monitor to give the desired answer, there is no general, principled guarantee in the same way that we have for soundness and transparency.

**Reality Check 2.** *In medical terms, the "protocol" in charge of caring for bad traces is too underspecified (the nurse will somehow deliver* a *right drug). Tolerance of error is accepted in the medical domain. Often drugs cannot be dosed exactly at the milliliter, yet one must be able to give some indication of the errors (or safety margins) that the protocol might tolerate. Here the only thing we could say that it depends on the nurse dispensating the drug (i.e. our particular implementation of the run-time monitor).*

### 1.1 The Contribution of This Paper

To address this problem we propose a new theoretical notion beside transparency and soundness to describe the behavior of the run-time monitor when it takes care of bad traces. To this extent we need two contributions:

- A formal notion of distance to clarify the meaning of "being close" to the original input or to a legal trace
- The equivalent formal $\epsilon - \delta$ notion of boundedness and continuity that we have in the real Calculus.

Once we have a notion of distance we can generalize the notions of transparency and soundness to weaker notions that apply to bad and good traces

**Table 1.** Properties of enforcement mechanisms

| **Name** | **Brief description** |
|---|---|
| Soundness | every trace is mapped into *some* valid trace |
| Transparency | every valid trace is mapped into itself |
| Bounded Map | every trace is mapped into a trace close to *one* valid trace |
| Boundedness | every trace is mapped into a trace close to *some* valid trace |
| Conditional Boundedness | every trace that is close to some valid trace is mapped into a trace close to *some* (possible other) valid trace |
| Predictability | every trace that is close to some valid trace is mapped into a trace close to *the same* valid trace |

alike: bounded map, boundedness, conditional boundedness and (our final proposal) predictability. We give their brief descriptions in Table 1.

Boundedness corresponds intuitively to a weakening of soundness: every trace should be mapped not to some valid trace but to an "almost valid" trace, which is close to the valid one. Transparency is only defined for valid traces (they should not be changed) and by weakening it we define conditional boundedness, which specifies that "almost valid" traces should be mapped to (possibly other) "almost valid" traces. Both these notions are not sufficient to deal with real life situations. Predictability is the notion that deals with traces close to valid ones and maps them to the predictable (closest) output.

The next section presents our running example (§2). Then we introduce some standard notation (§3) and discuss the applicability of various metrics(§4). The next steps (§5-6) present and discuss the different notions generalizing soundness and transparency from Table 1. We conclude the paper by discussing some open problems (§7).

## 2 Running Example

The case study is based on a healthcare process of drug dispensation. Hospitals accredited with the Public National Health Service are in charge of administering drugs and providing diagnostic services to patients. These hospitals are obliged to claim the cost of drug dispensation or diagnostic services to the Regional Healthcare Authority.

In the region of Lombardia, the process called "File F" is used by hospitals to refund the drugs administered and/or supplied by the hospitals' outpatient departments to the patients that are not hospitalized (we sketch some steps of the process in Fig. 1). This process is highly regulated and requirements are often subject to changes. Regulations span from national or regional health service legislation related to drug reimbursements to data protection laws, from health specific standards such as HL7 to ISO-type standards or whose adoption follows from compliance to best practices.

In order to give an idea of the sheer volume of regulation, the simple process of authorization and accounting for the dispensation and reimbursement of drugs is subject to the following (not exhaustive) set of (local) regulations interpreting

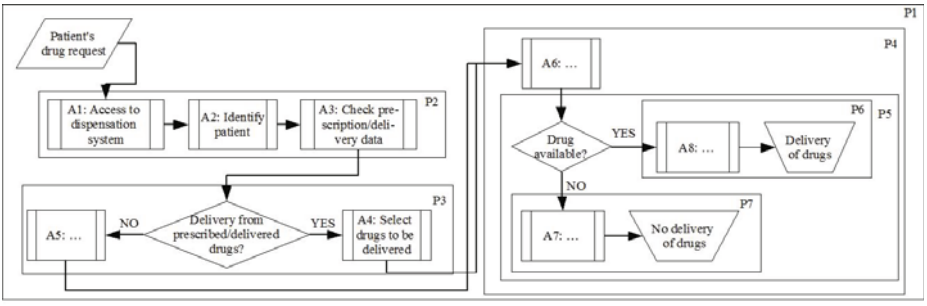



**Fig. 1.** The Top-Level process

national and EU laws: regional directive 17/SAN 3.4.1997, amended by directive No. 5/SAN 30_1_2004, Circular No. 45/SAN 23_12_2004, Note 30.11.2007 H1.2007.0050480, Note 27.3.2008 H1.2008.0012810, and Note 04.12.2008 H1.-2008.0044229 etc. Already in this partial sample we have had a quick turn-around of less than 6 months. In many cases compliance has to be almost immediate. For this kind of processes run-time monitors could be an effective solution.

Our running example is the drug selection subprocess of the drug dispensation process (denoted with A4 in Fig. 1). Its main steps are the following ones (in brackets we write the abbreviation of the step used in the paper):

1. The doctor fills list of drugs for his patient and selects one drug (Dis).
2. If the drug is highly sensitive, reviewing therapeutical notes is needed. They will be shown to the doctor and he has to review them (Tnn; Rtn). Otherwise therapeutical notes can be emitted (TnNn).
3. The system checks drug's submission to Research program and in case the drug is registered (Dr) shows the notification to the doctor. Then the doctor should insert the research protocol number (Irpn), a number of the protocol according to which the drug can be given to the patient. If the drug is not registered to Research program, then the doctor skips this step (DNr).
4. The doctor performs other actions needed for the drug prescription (Dpres).

## 3 Standard Notations of Enforcement

The set of observable process actions is denoted by $\Sigma$ and a set of possible actions to be executed is T. A *trace* is a finite or infinite sequence of actions; the set of all finite sequences over $\Sigma$ is denoted by $\Sigma^*$, the set of infinite sequences is $\Sigma^\omega$, and the set of all sequences is $\Sigma^\infty$. By $\sigma$ we refer to a trace and by $\cdot$ we refer to an empty trace. We write $\sigma;\tau$ to denote concatenation of two sequences, where $\sigma$ must be finite. A trace consisting of actions requested for execution is a *tentative execution*. A runtime monitor $E : \Sigma^\infty \to \mathrm{T}^\infty$ transforms tentative executions into a sequences of actions that will be actually executed on the system.

A *security policy* is a set of traces $P \subseteq \Sigma^\infty$. A policy $P$ is a *security property* if there exists a predicate $\widehat{P}$ over the traces, such that $\forall\sigma \in \Sigma^\infty : \widehat{P}(\sigma) \Leftrightarrow \sigma \in P$.

**Reality Check 3.** *Information-flow policies cannot be evaluated by looking at a single trace but must be evaluated by comparing a trace with other possible executions. This characteristic makes them totally "un-interesting" for stakeholders. They (or their hospital) can be held financially or penally liable only for what actually happened, so only actual traces matter.*

So in the rest of the paper we use interchangeably the policy $P$ with its corresponding predicate $\widehat{P}$. The trace $\sigma$ that satisfies the property $\widehat{P}$ is called *valid*, and the trace that does not satisfy the property is called *invalid*.

*Example 1.* The security policy $P$ consists of the following traces:

**SimpleRun** Dis; TnNn; DNr; Dpres,
**NoteRun** Dis; Tnn; Rtn; DNr; Dpres,
**ResearchRun** Dis; TnNn; Dr; Irpn; Dpres,
**NoteResearchRun** Dis; Tnn; Rtn; Dr; Irpn; Dpres,

and their closure under concatenation: for every $\sigma, \sigma' \in P : \sigma;\sigma' \in P$. Notice that an empty trace **NoRun** also satisfies the policy.

*Example 2.* Let us now make some examples of invalid traces with respect to the policy $P$. The doctors might forgot to click the "I have read the Therapeutical Note" button and rather close the window (Ctw). A similar event could happen for the step in which research protocol numbers are not inserted (Cpw), or he might skip all steps altogether. These alternatives give us the following traces

**CloseProt** Dis; TnNn; Dr; Cpw; Dpres
**SkipAll** Dis; Dr; Dpres
**CloseNoteProt** Dis; Tnn; Ctw; Dr; Cpw; Dpres

**Definition 1.** *An enforcement mechanism $E$ is* sound *iff* $\forall\sigma \in \Sigma^\infty : E(\sigma) \in P$. *It is* transparent *iff* $\forall\sigma \in \Sigma^\infty : (\sigma \in P \Rightarrow E(\sigma) = \sigma)$.

Fig. 2 graphically shows soundness and transparency, inputs on the left side of the figure marked with $\Sigma^\infty$ and outputs on the right side marked with $\mathrm{T}^\infty$. The gray area denotes invalid sequences and the white area denotes valid ones.

Transparency means that the traces in the white area of the input are mapped into *the same* traces (at the same position) in the white area of the output. Soundness means that all the traces shall be mapped into the white area. However, it is not specified where exactly the points from gray area are mapped.

The valid traces ResearchRun and NoteResearchRun are mapped into the same traces in the output, while invalid ones (CloseProt, SkipAll, CloseNoteProt) are mapped into good traces that are chosen arbitrarily.

In the original definition of Bauer et al. [2] the equal ("$\approx$") relation is used in the right part of the last statement. In another recent work Khoury and Tawbi [11] discussed possible semantics of this relation, however it is not possible to define a semantics that can be used in all domains. Hence, we use an equivalence relation in order to compare this definition with the new notions we propose in this paper. Ligatti and Reddy [14] have proposed the notion of completeness instead. It can be easily shown that transparency implies completeness and since transparency is necessary here, we don't discuss it further.

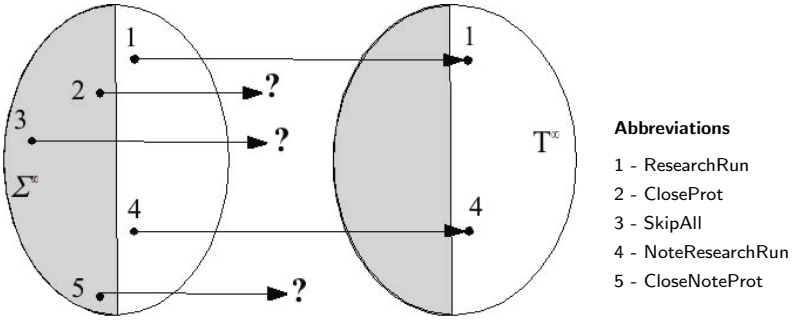



**Fig. 2.** Sound and Transparent enforcement mechanism

## 4 Metrics and Distances

**Definition 2.** *A* metric[1] *on a set $S$ is a function $d : S \times S \to \mathbb{R} \cup \{\infty\}$ such that (a) $d(\sigma, \tau) \geq 0$, (b) $d(\sigma, \tau) = 0$ if and only if $\sigma = \tau$, (c) symmetry: $d(\sigma, \tau) = d(\tau, \sigma)$, (d) triangular inequality: $d(\sigma, \tau) \leq d(\sigma, \sigma') + d(\sigma', \tau)$.*

The pair $(S, d)$ is called a *metric space* and the number $d(\sigma, \tau)$ is called the *distance between the elements* $\sigma$ and $\tau$. If all conditions but symmetry hold, then $d$ is called a *quasi-metric*. We propose several concrete metrics that will be used in the paper. Each of them has passed our reality check.

**Reality Check 4.** *When a distance is a finite number we can compare how far two situations (some potentially illegal traces) are from the legal situation. For medical staff these two situations can be perceived as qualitatively similar with a different degree of gravity. The notion of $\infty$ can be used to represent distance between traces perceived so qualitatively different to be incomparable. An example: a wrong action compromising the health of a patient takes place.*

We discuss here some concrete distances between the traces that will be useful in comparing tentative executions with the output of an enforcement mechanism.

**Definition 3.** *The* suppressing distance *between two finite traces is a total function $d_S : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$, such that*

$$d_S(a\sigma, b\sigma') = \begin{cases} \infty & \text{if } a\sigma = \cdot \text{ and } b\sigma' \neq \cdot \\ |a\sigma| & \text{if } b\sigma' = \cdot \\ d_S(\sigma, \sigma') & \text{if } a = b \\ 1 + d_S(\sigma, b\sigma') & \text{if } a \neq b \end{cases} \tag{1}$$

Distance $d_S$ counts number of actions that must be suppressed to obtain the second trace from the first one. The last rule says that if the first actions of the two sequences are different then the action in the first sequence is suppressed.

[1] The concepts of metrics and metric spaces are adapted from [1,6,15].

**Reality Check 5.** *The intuition behind suppression is that a bad trace is close to a good trace if the actions we have to undo are few. This can be explained to the operator and can be acceptable for an administrative procedure (albeit annoying for the operator involved). For example the monitor could block the process if the number of bad actions exceeds a given threshold or, preferably, it could undo all bad actions bringing us back to the point where we started the transaction that has gone awry.*

It is obvious that suppressing distance does not satisfy the symmetry property of metrics. Hence, the suppressing distance is not a metric but a *quasi-metric*.

*Example 3.* A doctor is selecting a drug (Dis) for which therapeutical notes are needed (Tnn). However at the time to review them, he is interrupted (e.g. in case of an emergency, interruption by another colleague etc.). When he comes back, he has to start running the process again because it timed out. The second time the doctor successfully finishes the process. So, the tentative execution is Dis; Tnn; NoteRun and the distance to the good trace NoteRun is 2. Notice that the distance from the good trace to the bad one is $\infty$ because no number of suppressions can transform the good trace into the bad trace.

This distance already discriminates between different run-time monitors:

*Example 4.* If the mechanism that is used to enforce this policy is a security automaton, it will stop executing the process as soon as something wrong happens (action Dis after Tnn in our case). Then, $E_{SA}(\mathsf{Dis};\mathsf{Tnn};\mathsf{NoteRun}) = \cdot$. If we use the suppressing distance to compare the outputs, $d_S(\cdot, E_{SA}(\mathsf{NoteRun})) = \infty$. The iterative suppression automaton [3] would have suppressed the initial prefix:

$$d_S(E_{IS}(\mathsf{Dis};\mathsf{Tnn};\mathsf{NoteRun}), E_{IS}(\mathsf{NoteRun})) = d_S(\mathsf{NoteRun},\mathsf{NoteRun}) = 0 \quad (2)$$

To compare outputs of more enforcement mechanisms we can use another metric that counts the number of replaced actions.

**Reality Check 6.** *While enforcing a process in the hospitals, an enforcement mechanism could be entitled only to correcting small errors without changing the protocol used by the operators (such as patient identification, patient consent and blood sampling before blood transfusion). If the doctor forgot to fill one field in the form, the mechanism can insert a default value. On the other hand, insertions of new steps by the monitor to compensate a bad event are not be allowed because a different protocol might have different medical or legal consequences and those can only be judged by an expert.*

**Definition 4.** *The* replacing distance *between two finite traces is a total function* $d_R : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$*, such that*

$$d_R(a\sigma, b\sigma') = \begin{cases} 0 & \text{if } a\sigma = \cdot \text{ and } b\sigma' = \cdot \\ \infty & \text{if } a\sigma = \cdot \text{ xor } b\sigma' = \cdot \\ d_R(\sigma,\sigma') & \text{if } a = b \\ 1 + d_R(\sigma,\sigma') & \text{if } a \neq b \end{cases} \quad (3)$$

Distance $d_R$ counts number of replacements to obtain one trace from another. If the traces have different length, the distance is $\infty$ (as expected as they clearly belong to different protocols). The replacing distance is a *metric*.

A more general definition of distance was originally proposed by Levenshtein [12]. This distance counts number of insertions, suppressions and replacements needed to obtain one trace from another.

**Definition 5.** *The* Levenshtein distance *between two finite traces is a total function* $d_L : \Sigma^* \times \Sigma^* \to \mathbb{N}$*, such that*

$$d_L(a\sigma, b\sigma') = \begin{cases} |b\sigma'| & \text{if } a\sigma = \cdot \\ |a\sigma| & \text{if } b\sigma' = \cdot \\ d_L(\sigma, \sigma') & \text{if } a = b \\ 1 + \min(d_L(\sigma, \sigma'), d_L(a\sigma, \sigma'), d_L(\sigma, b\sigma')) & \text{if } a \neq b \end{cases} \tag{4}$$

Let's make some examples of replacing and Levenshtein distances between the valid traces (Ex. 1) and the invalid ones (Ex. 2).

*Example 5.* The replacing distance can be equal to the Levenshtein distance: $d_R(\mathsf{NoteResearchRun}, \mathsf{CloseNoteProt}) = d_L(\mathsf{NoteResearchRun}, \mathsf{CloseNoteProt}) = 2$ since action Rtn is replaced with Ctw and action Irpn is replaced with Cpw. The replacing distance between CloseProt and NoteResearchRun is $\infty$ because NoteResearchRun is simply longer than CloseProt. But the Levenshtein distance counts the inserted and replaced actions and the distance is equal to 3.

## 5 From Sound to Bounded Monitors

Building on metrics over the sequences we propose to use the notions from measure theory. In the following definitions $(\Sigma^\infty \cup \mathrm{T}^\infty, d)$ is a metric space, a map $E : \Sigma^\infty \to \mathrm{T}^\infty$ is an enforcement mechanism and a security policy is a set $P \subseteq \Sigma^\infty \cap \mathrm{T}^\infty$. As a starting point we assume that all monitors in this paper are transparent.

**Reality Check 7.** *The actions in the systems corresponds to actions of doctors and nurses (or administrative staff) who are knowledgeable and accountable for their actions. Their point of view is that the choice of a* legitimate *course of action is due to a contextual knowledge not available to the system. A system that would change their decisions, when those actions conform to the policy of the hospital, would be unacceptable.*

The next step is generalizing the notion of soundness. We start from a classical definition of bounded map that Fig. 3 shows graphically. Even though the division of traces into good and bad is not relevant for the definition of boundedness, we keep it in the figure to ease the comparison with other notions.

**Definition 6.** *A function* $E : \Sigma^\infty \to \mathrm{T}^\infty$ *is* bounded *if the subset* $\{E(\sigma) : \sigma \in \Sigma^\infty\} \subseteq \mathrm{T}^\infty$ *is bounded. Formally,* $\exists \tau \in \mathrm{T}^\infty : \exists \varepsilon > 0 : \forall \sigma \in \Sigma^\infty : d(E(\sigma), \tau) \leq \varepsilon$.

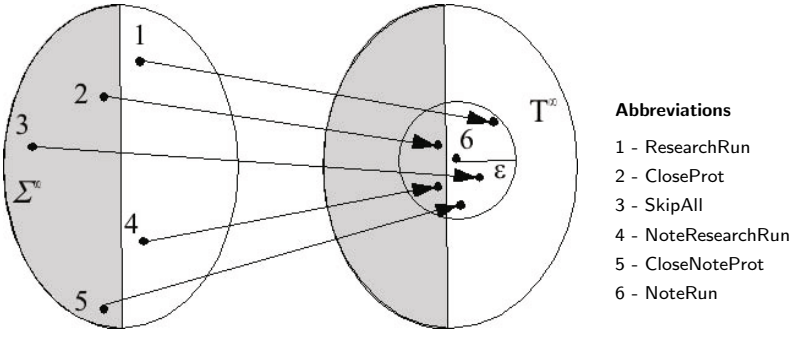



**Fig. 3.** Bounded map

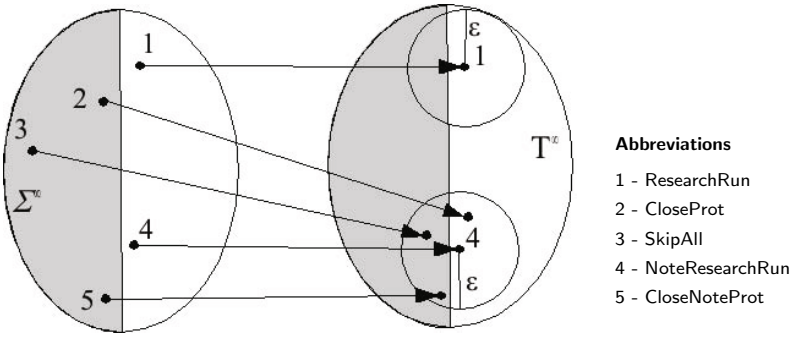



**Fig. 4.** Bounded within $\varepsilon$ enforcement mechanism

Let us project this notion to the theory of enforcement mechanisms. We get a mechanism that transforms all sequences to some sequence close to $\tau$. This is not what users are expecting. The user's policy usually contains several good sequences (see Ex. 1). Hence we should adapt the definition to map bad sequences to different good sequences in the policy.

**Definition 7.** *An enforcement mechanism* $E : \Sigma^\infty \to \mathrm{T}^\infty$ *is* bounded within $\varepsilon$ *iff* $\forall \sigma \in \Sigma^\infty : \exists \sigma_P \in P : d(E(\sigma), E(\sigma_P)) \leq \varepsilon$.

This notion says that an output of enforcement mechanism is always within the distance $\varepsilon$ from some good execution, for $\varepsilon > 0$ we are weakening the notion of soundness. Fig. 4 shows the bounded within $\varepsilon$ enforcement mechanism.

There is a fundamental difference between boundedness and boundedness within $\varepsilon$. Boundness means that there is one single trace $\tau$ in the possible outputs such that *all* the outputs of $E$ are in the radius $\varepsilon$ from $\tau$. Boundedness within $\varepsilon$ means that for every possible output there is a valid trace such that the distance between them is smaller or equal than $\varepsilon$.

*Example 6.* An enforcement mechanism $E$ enforces the policy from Ex. 1 in the following way (we also show the distance to NoteResearchRun):

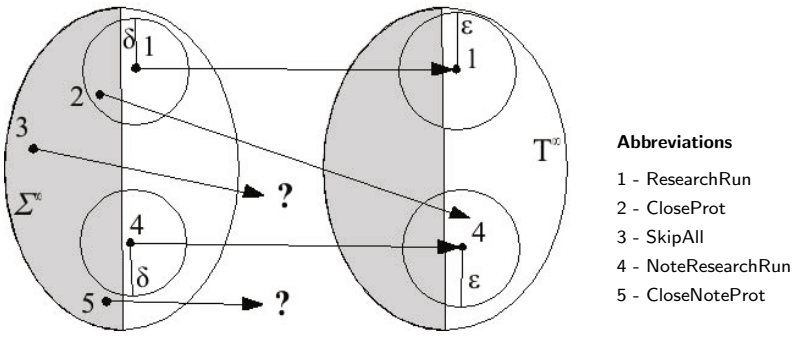



**Fig. 5.** Conditionally bounded within $\varepsilon$ enforcement mechanism

$E(\mathsf{CloseNoteProt}) = \mathsf{CloseNoteProt}$, $d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$
$E(\mathsf{SkipAll}) = \mathsf{CloseNoteProt}$, $d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$
$E(\mathsf{CloseProt}) = \mathsf{NoteRun}$, $d_L(\mathsf{NoteRun}, \mathsf{NoteResearchRun}) = 2$

Since the output of $E$ is always at distance 2 from some good execution NoteResearchRun then $E$ is bounded within $\varepsilon = 2$.

We can refine the notion by imposing also that "almost valid traces" should be mapped to some "almost valid traces". We call it *conditional boundedness within* $\varepsilon$ and show graphically in Fig. 5.

**Definition 8.** *An enforcement mechanism* $E : \Sigma^\infty \rightarrow \mathrm{T}^\infty$ *is* conditionally bounded within $\varepsilon$ *iff* $\exists \delta > 0 : \forall \sigma \in \Sigma^\infty : (\exists \sigma'_P \in P : d(\sigma, \sigma'_P) \leq \delta \Rightarrow \exists \sigma_P \in P : d(E(\sigma), E(\sigma_P)) \leq \varepsilon)$.

*Example 7.* An enforcement mechanism $E$ transforms the sequences of actions in the following way (we also show the distance to some good sequence):

$E(\mathsf{CloseNoteProt}) = \mathsf{CloseProt}$, $d_L(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1$
$E(\mathsf{SkipAll}) = \mathsf{NoRun}$, $d_L(\mathsf{NoRun}, \mathsf{NoRun}) = 0$
$E(\mathsf{CloseProt}) = \mathsf{NoteRun}$, $d_L(\mathsf{NoteRun}, \mathsf{NoteRun}) = 0$

Obviously, $E$ is bounded within $\varepsilon = 1$. It is conditionally bounded within $\varepsilon = 1$ because there is a $\delta = 3$ such that for every sequence there is a valid trace at most at distance 3: $d_L(\mathsf{CloseNoteProt}, \mathsf{NoteResearchRun}) = 2$, $d_L(\mathsf{SkipAll}, \mathsf{NoRun}) = 3$ and $d_L(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1$.

**Reality Check 8.** *Boundedness and conditional boundedness can refine soundness but are still unacceptable: if a doctor skips key steps altogether (SkipAll), a bounded EM can transform it into a sequence CloseNoteProt. A conditionally bounded, sound and transparent EM, when the doctor closes the window (CloseProt) instead of inserting the protocol (ResearchRun), can transforms this tentative execution into another, completely different one (NoteRun). The problem is that some actions in the outcome are not a direct transformation of the actions of the doctors. Since actions carry liabilities it is important that the actions of the EM are always linked to corresponding actions by the doctor.*

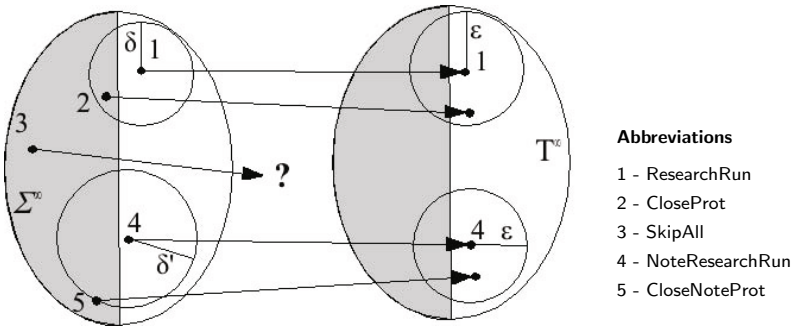



**Fig. 6.** Predictable within $\varepsilon$ enforcement mechanism

**Table 2.** Properties of enforcement mechanism

| Name | Pre-Condition | Post-Condition |
|---|---|---|
| Soundness | | for every trace the output is always some valid trace |
| Transparency | if input is a valid trace | output is *the same* valid trace |
| Bounded Map | | there is one valid trace such that output is always within $\varepsilon$ from that trace |
| Boundedness | | output is always within $\varepsilon$ from some valid trace |
| Conditional Boundedness | if input is within $\delta$ from some valid trace | output is within $\varepsilon$ from some valid trace |
| Predictability | if input is within $\delta$ from a valid trace | output is within $\varepsilon$ from *the same* valid trace |

## 6 Predictability

Our notion of predictability within $\varepsilon$ is inspired by the classical notion of continuous functions. Let $(\Sigma^\infty \cup \mathrm{T}^\infty, d)$ and $(\Sigma^\infty \cup \mathrm{T}^\infty, d')$ be metric spaces.

**Definition 9.** *A map* $E : \Sigma^\infty \to \mathrm{T}^\infty$ *is* continuous *if at every trace* $\sigma \in \Sigma^\infty$ *the following holds:* $\forall \varepsilon > 0 : \exists \delta > 0 : \forall \sigma' \in \Sigma^\infty (d(\sigma, \sigma') < \delta \Rightarrow d'(E(\sigma), E(\sigma')) < \epsilon)$.

In conditional boundedness we proposed to limit an output when an input is "almost good". But as a Reality Check 8 shows, an input and its output should be compared to *the same* valid trace.

**Definition 10.** *An enforcement mechanism* $E$ *is* predictable within $\varepsilon$ *if for every trace* $\sigma_P \in P$ *the following holds:* $\forall \nu \geq \varepsilon : \exists \delta > 0 : \forall \sigma \in \Sigma^* : (d(\sigma, \sigma_P) \leq \delta \Rightarrow d'(E(\sigma), E(\sigma_P)) \leq \nu)$.

Informally, it says that for every valid trace there always exists a radius $\delta$, such that all the traces within this radius are mapped into the circle with radius $\varepsilon$ from this trace. We show it in Fig. 6.

*Example 8.* The enforcement mechanism $E$ from Ex. 7 is conditionally bounded within $\varepsilon = 1$, but not predictable within $\varepsilon = 1$ since there exists $\sigma_P = \mathsf{ResearchRun}$ such that $\exists \nu = 2 : \forall \delta > 0 : \exists \sigma \in \Sigma^* : (d(\sigma, \sigma_P) \leq \delta \wedge d(E(\sigma), E(\sigma_P)) > \nu)$ where $\sigma = \mathsf{CloseProt}$, then $d(\mathsf{CloseProt}, \mathsf{ResearchRun}) = 1 \leq \delta$ and $d(E(\mathsf{CloseProt}), E(\mathsf{ResearchRun})) = d_L(\mathsf{NoteRun}, \mathsf{ResearchRun}) = 3 > 2$.

Table 2 shows all the notions so far and the new notion of predictability.

## 7 Conclusions

In this paper we have discussed how to go beyond the (only) two classical properties used to evaluate an enforcement mechanism: *soundness* and *transparency*. Soundness specifies that the output is always good and transparency guarantees that good input is not changed. However those two characteristics alone are not sufficient to discriminate between enforcement mechanisms. The key issue is to specify how bad input is fixed into good output.

We have introduced several notions that could describe predictable behavior and checked them against the industrial case study on e-Health. The idea behind *predictability* is that there are "no surprises on bad inputs".

An apparent limitation of our work is that we don't deal with infinite traces. We have actually considered some mathematical functions over infinite traces from [5] but we found a practical obstacle:

**Reality Check 9.** *The end of the fiscal year effectively terminates any trace in the eyes of our stakeholders.*

Further, all simple and natural metrics that we considered from [5] were not adequate from one perspective or another (we give some examples in the appendix).

Another open issue is the analysis of existing mechanisms to identify which one is predictable. The practically interesting question is whether a $\varepsilon$ for predictability can be extracted from a security policy expressed as an automata.

The second issue revolves around edit automata as an enforcement mechanism and the characterization of predictable policies. A key question is whether policies of a certain form do always (or never) have predictable enforcement mechanism. Under some definition of convexity we could prove that convex policies always have predictable enforcement mechanisms within a bound fixed on the border, but the natural definition, while mathematically sound, is not intuitive enough to pass our reality check. More research is needed.

## References

1. Pontryagin, L.S., Arkhangel'skii, A.V. (eds.): General topology I: basic concepts and constructions, dimension theory. Springer, Heidelberg (1990)
2. Bauer, L., Ligatti, J., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Int. J. of Inform. Sec. 4(1-2), 2–16 (2005)

3. Bielova, N., Massacci, F., Micheletti, A.: Towards practical enforcement theories. In: Jøsang, A., Maseng, T., Knapskog, S.J. (eds.) NordSec 2009. LNCS, vol. 5838, pp. 239–254. Springer, Heidelberg (2009)
4. Brown, A., Ryan, M.: Synthesising monitors from high-level policies for the safe execution of untrusted software. In: Chen, L., Mu, Y., Susilo, W. (eds.) ISPEC 2008. LNCS, vol. 4991, pp. 233–247. Springer, Heidelberg (2008)
5. Chatterjee, K., Doyen, L., Henzinger, T.A.: Expressiveness and closure properties for quantitative languages. Comp. Research Repository, abs/1007.4018 (2010)
6. Cohn, D.L.: Measure Theory. Birkhauser, Basel (1980)
7. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security-by-contract on the .net platform. Information Security Technical Report 13(1), 25–32 (2008)
8. Erlingsson, U.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Cornell University (2003)
9. Falcone, Y., Fernandez, J.-C., Mounier, L.: Enforcement monitoring wrt. the safety-progress classification of properties. In: Proc. of 24th ACM Symp. on Applied Computing – Software Verif. and Test. Track, pp. 593–600. ACM Press, New York (2009)
10. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An enterprise service bus for access and usage control policy enforcement. In: IFIPTM 2010. IFIP Advances in Information and Communication Technology, vol. 321, pp. 63–78. Springer, Heidelberg (2010)
11. Khoury, R., Tawbi, N.: Using Equivalence Relations for Corrective Enforcement of Security Policies. In: Kotenko, I., Skormin, V. (eds.) MMM-ACNS 2010. LNCS, vol. 6258, pp. 139–154. Springer, Heidelberg (2010)
12. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8), 707–710 (1966); An English translation of the "Physics Sections" of the Proceedings of the Academy of Sciences of the USSR
13. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. on Inform. and Sys. Security 12(3), 1–41 (2009)
14. Ligatti, J., Reddy, S.: A theory of runtime enforcement, with results. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 87–100. Springer, Heidelberg (2010)
15. Matthews, S.G.: Partial metric topology. In: Proceedings of the 8th Summer Conference, Queen's College, vol. 728, pp. 183–197. Annals of the New York Academy of Sciences (1994)
16. Park, J., Sandhu, R.: The UCON ABC usage control model. ACM Trans. on Inform. and Sys. Security 7(1), 128–174 (2004)
17. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In: Proc. of ACM Symp. on Inform. Comp. and Comm. Security, pp. 47–60. ACM Press, New York (2009)
18. Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: Proc. of ACM Symp. on Inform. Comp. and Comm. Security, pp. 240–244. ACM Press, New York (2008)
19. Schneider, F.B.: Enforceable security policies. ACM Trans. on Inform. and Sys. Security 3(1), 30–50 (2000)
20. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Inform. and Comp. 206(2-4), 158–184 (2007)
21. Yun, D., Chander, A., Islam, N., Serikov, I.: Javascript instrumentation for browser security. In: Proc. of the 34th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang., pp. 237–249. ACM Press, New York (2007)

## A Examples of Levenshtein Distances

**Table 3.** The Levenshtein distances between some sequences

| | NoRun | ResearchRun | CloseProt | SkipAll | NoteResearchRun | CloseNoteProt |
|---|---|---|---|---|---|---|
| NoRun | 0 | - | - | - | - | - |
| ResearchRun | 5 | 0 | - | - | - | - |
| CloseProt | 5 | 1 | 0 | - | - | - |
| SkipAll | 3 | 2 | 2 | 0 | - | - |
| NoteResearchRun | 6 | 2 | 3 | 3 | 0 | - |
| CloseNoteProt | 6 | 3 | 2 | 3 | 2 | 0 |
| NoteRun | 5 | 3 | 3 | 3 | 2 | 3 |

## B Examples of (Unsatisfactory) Metrics on Infinite Traces

Here we discuss two natural distances from [5] that have a clear mathematical intuition for our domain and allow to obtain finite numbers when comparing infinite traces. The first option is to use an economic approach and consider the discounted distance that discounts the remote differences in the sequence :

**Definition 11.** *The* discounted distance *between two infinite traces is a total function* $d_D : \Sigma^\omega \times \Sigma^\omega \to \mathbb{N}$*, such that*

$$d_D(a\sigma, b\sigma') = \begin{cases} |a\sigma| & \text{if } b\sigma' = \cdot \\ |b\sigma'| & \text{if } a\sigma = \cdot \\ d_D(\sigma, \sigma') & \text{if } a = b \\ 1 + \frac{1}{k}\, d_D(\sigma, \sigma') & \text{if } a \neq b \end{cases} \tag{5}$$

In this definition $k \in \mathbb{R}$, $k \neq 0$ is a discounting factor.

**Reality Check 10.** *The first function can be acceptable from the perspective of a risk manager as later events have less risk of being detected or of having consequences within the year. It is less acceptable for doctors: a wrong drug is a wrong drug, no matter if delivered at the beginning or the end of the fiscal year.*

Another approach that works for replacing-type distances is to attribute a weight to each replacement and consider the maximum of such weights.

**Reality Check 11.** *The maximum weight determines what can at worst happen and can be satisfactory from the point of view of an operator: at worst it deviated so and so from the ideal trace. It is not satisfactory from a risk management perspective, as it cannot distinguish between a trace where such deviations are rare from trace where they are frequent.*

At the end the only acceptable metrics boils down to considering what happens during a limited slot and project it to infinity. But then we could simply consider the finite slot.

Hence, the definition of mathematically simple and meaningful metrics for infinite traces is still open for us (assuming we should consider them at all).

# SessionShield: Lightweight Protection against Session Hijacking

Nick Nikiforakis[1], Wannes Meert[1], Yves Younan[1],
Martin Johns[2], and Wouter Joosen[1]

[1] IBBT-DistriNet
Katholieke Universiteit Leuven
Celestijnenlaan 200A B3001
Leuven, Belgium
{nick.nikiforakis,wannes.meert,yves.younan,wouter.joosen}@cs.kuleuven.be
[2] SAP Research - CEC Karlsruhe
martin.johns@sap.com

**Abstract.** The class of Cross-site Scripting (XSS) vulnerabilities is the most prevalent security problem in the field of Web applications. One of the main attack vectors used in connection with XSS is session hijacking via session identifier theft. While session hijacking is a client-side attack, the actual vulnerability resides on the server-side and, thus, has to be handled by the website's operator. In consequence, if the operator fails to address XSS, the application's users are defenseless against session hijacking attacks.

In this paper we present SessionShield, a lightweight client-side protection mechanism against session hijacking that allows users to protect themselves even if a vulnerable website's operator neglects to mitigate existing XSS problems. SessionShield is based on the observation that session identifier values are not used by legitimate client-side scripts and, thus, need not to be available to the scripting languages running in the browser. Our system requires no training period and imposes negligible overhead to the browser, therefore, making it ideal for desktop and mobile systems.

**Keywords:** session hijacking, client-side proxy, http-only.

## 1 Introduction

Over the past decade, users have witnessed a functional expansion of the Web, where many applications that used to run on the desktop are now accessible through the browser. With this expansion, websites evolved from simple static HTML pages to dynamic Web applications, i.e. content-rich resources accessible through the Web. In this modern Web, JavaScript has proven its usefulness by providing server offloading, asynchronous requests and responses and in general improving the overall user experience of websites. Unfortunately, the de facto

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 87–100, 2011.

support of browsers for JavaScript opened up the user to a new range of attacks, of which the most common is Cross-site scripting (XSS[1]).

In XSS attacks, an attacker convinces a user's browser to execute malicious JavaScript code on his behalf by injecting this code in the body of a vulnerable webpage. Due to the fact that the attacker can only execute JavaScript code, as opposed to machine code, the attack was initially considered of limited importance. Numerous incidents though, such as the Sammy worm that propagated through an XSS vulnerability on the social network MySpace [34] and the XSS vulnerabilities of many high-impact websites (e.g., Twitter, Facebook and Yahoo [40]) have raised the awareness of the security community. More recently, Apache released information about an incident on their servers where attackers took advantage of an XSS vulnerability and by constant privilege escalation managed to acquire administrator access to a number of servers [1].

Today, the Open Web Application Security Project (OWASP) ranks XSS attacks as the second most important Web application security risk [28]. The Web Hacking Incident Database from the Web Application Security Consortium states that 13.87% of all attacks against Web applications are XSS attacks [4]. These reports, coupled with more than 300,000 recorded vulnerable websites in the XSSed archive [40], show that this problem is far from solved.

In this paper, we present SessionShield, a lightweight countermeasure against session hijacking. Session hijacking occurs when an attacker steals the session information from a legitimate user for a specific website and uses it to circumvent authentication to that website. Session hijacking is by far the most popular type of XSS attack since every website that uses session identifiers is potentially vulnerable to it. Our system is based on the observation that session identifiers are strings of data that are intelligible to the Web application that issued them but not to the Web client who received them. SessionShield is a proxy outside of the browser that inspects all outgoing requests and incoming responses. Using a variety of methods, it detects session identifiers in the incoming HTTP headers, strips them out and stores their values in its own database. In every outgoing request, SessionShield checks the domain of the request and adds back the values that were previously stripped. In case of a session hijacking attack, the browser will still execute the session hijacking code, but the session information will not be available since the browser never received it. Our system is transparent to both the Web client and the Web server, it operates solely on the client-side and it doesn't rely on the Web server or trusted third parties. SessionShield imposes negligible overhead and doesn't require training or user interaction making it ideal for both desktop and mobile systems.

The rest of this paper is structured as follows: In Section 2, we describe what sessions are and how XSS attacks are conducted followed by a detailed survey concerning a well-known protection mechanism, namely HTTP-Only cookies. In Section 3, we present the architecture and details of SessionShield. We evaluate

[1] Cross-site scripting is commonly abbreviated as XSS to distinguish it from the acronym of Cascading Style Sheets (CSS).

our system in Section 4 and provide implementation details in Section 5. We discuss related work in Section 6 and finally conclude in Section 7.

## 2 Background

### 2.1 Session Identifiers

The workhorse protocol of the World Wide Web, the HyperText Transfer Protocol (HTTP) and its secure counterpart (HTTP**S**) are by design stateless. That means that a Web application cannot track a client between multiple requests unless it adds a separate tracking mechanism on top of the HTTP(S) protocol. The most commonly used tracking mechanism are sessions identifiers. A session identifier (SID) is a unique string of random data (typically consisting of numbers and characters) that is generated by a Web application and propagated to the client, usually through the means of a cookie. After the propagation of the session, every request initiated by the client will contain, among others, the session identifier that the application entrusted him with. Using session identifiers, the Web application is able to identify individual users, distinguish simultaneously submitted requests and track the users in time. Sessions are used in e-banking, web-mail and virtually every non-static website that needs to enforce access-control on its users. Sessions are an indispensable element of the modern World Wide Web and thus session management support exists in all modern Web languages (e.g., PHP, ASP and JSP).

Session identifiers are a prime attack target since a successful capture-and-replay of such an identifier by an attacker provides him with instant authentication to the vulnerable Web application. Depending on the access privileges of the user whose id was stolen, an attacker can login as a normal or as a privileged user on the website in question and access all sorts of private data ranging from emails and passwords to home addresses and even credit card numbers. The most common way of stealing session identifiers is through Cross-site Scripting attacks which are explained in the following section.

### 2.2 Cross-Site Scripting Attacks

Cross-site scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application. In XSS attacks, the attacker convinces the victim's browser to execute JavaScript code on his behalf thus giving him access to sensitive information stored in the browser. Malicious JavaScript running in the victim's browser can access, among others, the contents of the cookie for the running domain. Since session identifiers are most commonly propagated through cookies, the injected JavaScript can read them and transfer them to an attacker-controlled server which will record them. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim.

XSS vulnerabilities can be categorized as *reflected* or *stored*. A reflected XSS vulnerability results from directly including parts of the HTTP request into the corresponding HTTP response. Common examples for reflected XSS issues include search forms that blindly repeat the search term on the results-page or custom 404 error pages. On the other hand, a stored XSS vulnerability occurs whenever the application permanently stores untrusted data which was not sufficiently sanitized. If such data is utilized to generate an HTTP response, all potentially injected markup is included in the resulting HTML causing the XSS issue. Stored XSS was found in the past for instance in guestbooks, forums, or Web mail applications.

Code listing 1 shows part of the code of a search page, written in PHP, that is vulnerable to a reflected XSS attack. The purpose of this page is to read one or more keywords from the user, search the database for the keyword(s) and show the results to the user. Before the actual results, the page prints out the keyword(s) that the user searched for. The programmer however has not provisioned against XSS attacks, and thus whatever is presented as a query, will be "reflected" back in the main page, including HTML and JavaScript code. An attacker can hijack the victim's session simply by sending him the following link:

```
http://vulnerable.com/search.php?q=</u><script>
document.write('<img src="http://hacker.com/
session_hijack.php?ck=' + document.cookie +'">');
</script>
```

When the user clicks on the above link, his browser will initiate a `GET` request to `vulnerable.com`. The `GET` parameter `q` will be added to the resulting page that the server sends back to the user's browser. The victim's browser will start rendering the page and once it reaches the "Search results for:" part, it will create an image URI which contains the values stored in the user's cookie and ask for that image from the attacker-controlled server. The attacker's script will record the cookie values and enable the attacker to masquerade as the victim at the `vulnerable.com`.

**Code Listing 1.** Code snippet vulnerable to an XSS attack

```
<?php
  session_start();
  ...
  $search_query = $_GET['q'];
  print "Search results for: <u> $search_query </u>";
  ...
?>
```

### 2.3 HTTP-Only and Sessions

Developers realized from early on that it is trivial to hijack Web sessions in the presence of an XSS vulnerability. In 2002, Microsoft developers introduced the notion of `HTTP-Only` cookies and added support for them in the release of Internet Explorer 6, SP1 [22]. `HTTP-Only` is a flag that is sent by the Web application to the client, along with a cookie that contains sensitive information, e.g., a session identifier. It instructs the user's browser to keep the values of that cookie away from any scripting languages running in the browser. Thus, if a cookie is denoted as `HTTP-Only` and JavaScript tries to access it, the result will be an empty string. We tested the latest versions of the five most common Web browsers (Internet Explorer, Firefox, Chrome, Safari and Opera) and we observed that if the Web application emmits the `HTTP-Only` flag the cookie is, correctly, no longer accessible through JavaScript.

In an attempt to discover whether the `HTTP-Only` mechanism is actually used, we crawled the Alexa-ranked top one million websites [35] and recorded whether cookies that contained the keyword "sess" where marked as `HTTP-Only`. We chose "sess" because it is a common substring present in the session names of most major Web languages/frameworks (see Section 3.2, Table 2) and because of the high probability that customely named sessions will still contain that specific substring. We also provisioned for session names generated by the ASP/ASP.NET framework that don't contain the "sess" string. The results of our crawling are summarized in Table 1. Out of 1 million websites, 418,729 websites use cookies in their main page and out of these, 272,335 cookies contain session information[2]. We were surprised to find out that only a 22.3% of all websites containing sessions protected their cookies from session stealing using the `HTTP-Only` method. Further investigation shows that while 1 in 2 ASP websites that use sessions utilize `HTTP-Only`, only 1 in 100 PHP/JSP websites does the same.

These results clearly show that `HTTP-Only` hasn't received widespread adoption. Zhou et al. [42] recently made a similar but more limited study (top 500 websites, instead of top 1 million) with similar findings. In their paper they acknowledge the usefullness of the `HTTP-Only` mechanism and they discuss possible reasons for its limited deployment.

## 3 SessionShield Design

SessionShield is based on the idea that session identifiers are data that no legitimate client-side script will use and thus should not be available to the scripting languages running in the browser. Our system shares this idea with the `HTTP-Only` mechanism but, unlike `HTTP-Only`, it can be applied selectively to a subset of cookie values and, more important, it doesn't need support from Web applications. This means, that SessionShield will protect the user from session hijacking regardless of the security provisioning of Web operators.

[2] Cookies that contained the `HTTP-Only` flag but were not identified by our heurestic are added to the "Other/With HTTP-Only" column.

**Table 1.** Statistics on the usage of HTTP-Only on websites using session identifiers, sorted according to their generating Web framework

| Session Framework | Total | With HTTP-Only | Without HTTP-Only |
|---|---|---|---|
| PHP | 135,117 (53.2%) | 1,736 (1.3%) | 133,381 (98.7%) |
| ASP/ASP.NET | 60,218 (23.5%) | 25,739 (42.7%) | 34,479 (57.3%) |
| JSP | 12,911 (5.1%) | 113 (0.9%) | 12,798 (99.1%) |
| Other | 64,089 (18.2%) | 33,071 (51.6%) | 31,018 (48.4%) |
| Total | 272,335 (100%) | 60,659 (22.3%) | 211,676 (77.8%) |

The idea itself is founded on the observation that session identifiers are strings composed by random data and are unique for each visiting client. Furthermore, a user receives a different session identifier every time that he logs out from a website and logs back in. These properties attest that there can be no legitimate calculations done by the client-side scripts using as input the constantly-changing random session identifiers. The reason that these values are currently accessible to client-side scripts is because Web languages and frameworks mainly use the cookie mechanism as a means of transport for the session identifiers. The cookie is by default added to every client request by the browser which aleviates the Web programmers from having to create their own transfer mechanism for session identifiers. JavaScript can, by default, access cookies (using the `document.cookie` method) since they may contain values that the client-side scripts legitimately need, e.g., language selection, values for boolean variables and timestamps.

### 3.1 Core Functionality

Our system acts as a personal proxy, located on the same host as the browser(s) that it protects. In order for a website or a Web application to set a cookie to a client, it sends a `Set-Cookie` header in its HTTP response headers, followed by the values that it wishes to set. SessionShield inspects incoming data in search for this header. When the header is present, our system analyses the values of it and attempts to discover whether session identifiers are present. If a session identifier is found, it is stripped out from the headers and stored in SessionShield's internal database. On a later client request, SessionShield queries its internal database using the domain of the request as the key and adds to the outgoing request the values that it had previously stripped.

A malicious session hijacking script, whether reflected or stored, will try to access the cookie and transmit its value to a Web server under the attacker's control. When SessionShield is used, cookies inside the browser no longer contain session identifiers and since the attacker's request domain is different from the domain of the vulnerable Web application, the session identifier will not be added to the outgoing request, effectively stopping the session hijacking attack.

In order for SessionShield to protect users from session hijacking it must successfully identify session identifiers in the cookie headers. Our system uses

two identification mechanisms based on: a) common naming conventions of Web frameworks and of custom session identifiers and b) statistical characteristics of session identifiers.

### 3.2 Naming Conventions of Session Identifiers

**Common Web Frameworks.** Due to the popularity of Web sessions, all modern Web languages and frameworks have support for generating and handling session identifiers. Programmers are actually adviced not to use custom session identifiers since their implementation will most likely be less secure from the one provided by their Web framework of choice. When a programmer requests a session identifier, e.g., with `session_start()` in PHP, the underlying framework generates a random unique string and automatically emmits a `Set-Cookie` header containing the generated string in a `name=value` pair, where `name` is a standard name signifying the framework used and `value` is the random string itself. Table 2 shows the default names of session identifiers according to the framework used[3]. These naming conventions are used by SessionShield to identify session identifiers in incoming data and strip them out of the headers.

**Common Custom Naming.** From the results of our experiment in Section 2.3, we observed that "sess" is a common keyword among custom session naming and thus it is included as an extra detection method of our system. In order to avoid false-positives we added the extra measure of checking the length and the contents of the value of such a pair. More specifically, SessionShield identifies as session identifiers pairs that contain the word "sess" in their name and their value is more than 10 characters long containing both letters and numbers. These characteristics are common among the generated sessions of all popular frameworks so as to increase the value space of the identifiers and make it practically impossible for an attacker to bruteforce a valid session identifier.

### 3.3 Statistical Characteristics of Session Identifiers

Despite the coverage offered by the previous mechanism, it is beyond doubt that there can be sessions that do not follow standard naming conventions and thus would not be detected by it. In this part we focus on the fact that session identifiers are long strings of symbols generated in some random way. These two key characteristics, length and randomness, can be used to predict if a string, that is present in a cookie, is a session identifier or not. This criterion, in fact, is similar to predicting the strength of a password.

Three methods are used to predict the probability that a string is a session identifier (or equivalently the strength of a password):

1. **Information entropy:** The strength of a password can be measured by the information entropy it represents [6]. If each symbol is produced

[3] On some versions of the ASP/ASP.NET framework the actual name contains random characters, which are signified by the wildcard symbol in the table.

**Table 2.** Default session naming for the most common Web frameworks

| Session Framework | Name of Session variable |
|---|---|
| PHP | phpsessid |
| ASP/ASP.NET | asp.net_sessionid<br>aspsessionid*<br>.aspxauth*<br>.aspxanonymous* |
| JSP | jspsessionid<br>jsessionid |

independently, the entropy is $H = L \cdot \log_2 N$, with $N$ the number of possible symbols and $L$ the length of the string. The resulting value, $H$, gives the entropy and represents the number of bits necessary to represent the string. The higher the number of necessary bits, the better the strength of the password in the string. For example, a pin-code consisting out of four digits has an entropy of 3.32 bits per symbol and a total entropy of 13.28.

2. **Dictionary check:** The strength of a password reduces if it is a known word. Similarly, cookies that have known words as values are probably not session identifiers.
3. $\chi^2$: A characteristic of a random sequence is that all symbols are produced by a generator that picks the next symbol out of a uniform distribution ignoring previous symbols. A standard test to check if a sequence correlates with a given distribution is the $\chi^2$-test [16], and in this case this test is used to calculate the correlation with the uniform distribution. The less the string is correlated with the random distribution the less probable it is that it is a random sequence of symbols. The uniform distribution used is $1/N$, with $N$ the size of the set of all symbols appearing in the string.

Every one of the three methods returns a probability that the string is a session identifier. These probabilities are combined by means of a weighted average to obtain one final probability. SessionShield uses this value and a threshold to differentiate between session and non-session values (see Section 5 for details).

## 4 Evaluation

### 4.1 False Positives and False Negatives

SessionShield can protect users from session hijacking as long as it can successfully detect session identifiers in the incoming HTTP(S) data. In order to evaluate the security performance of SessionShield we conducted the following experiment: we seperated the first 1,000 cookies from our experiment in Section 2.3 and we used them as input to the detection mechanism of SessionShield. SessionShield processed each cookie and classified a subset of the values as sessions identifiers and the rest as benign data. We manually inspected both sets of values and we recorded the false positives (values that were wrongly

detected as session identifiers) and the false negatives (values that were not detected as session identifiers even though they were). SessionShield classified 2,167 values in total (average of 2.16 values/cookie) with 70 false negatives (3%) and 19 false positives (0,8%).

False negatives were mainly session identifiers that did not comply to our session identifier criteria, i.e a) they didn't contain both letters and numbers or b) they weren't longer than 10 characters. Session identifiers that do not comply to these requirements are easily brute-forced even if SessionShield protected them. With regard to false positives, it is important to point out that in order for a website to stop operating correctly under SessionShield, its legitimate client-side scripts must try to use values that SessionShield classified as session identifiers. Thus the actual percentage of websites that wouldn't operate correctly and would need to be white-listed is less than or equal to 0,8%.

### 4.2 Performance Overhead

In an effort to quantify how much would SessionShield change the Web experience of users, we decided to measure the difference in page-download time when a page is downloaded: a) directly from the Internet; b) through a simple forwarding proxy [9] and c) through SessionShield. Using `wget`, we downloaded the top 1,000 Internet websites [35] and measured the time for each.

In order to avoid network inconsistencies we downloaded the websites locally together with the HTTP headers sent by the actual Web servers. We used a fake DNS server that always resolved all domains to the "loopback" IP address and a fake Web server which read the previously-downloaded pages from disk and replayed each page along with its original headers. This allowed us to measure the time overhead of SessionShield without changing its detection technique, which relies on the cookie-related HTTP headers. It is important to point out that SessionShield doesn't add or remove objects in the HTML/JavaScript code

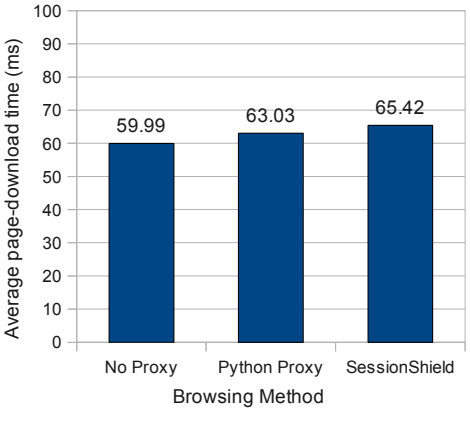



**Fig. 1.** Average download time of the top 1,000 websites when accessed locally without a proxy, with a simple forwarding Python-proxy and with SessionShield

of each page thus the page-rendering time isn't affected by its operation. Each experiment was repeated five times and the average page-download time for each method is presented in Fig. 1. SessionShield's average time overhead over a simple Python proxy is approximately 2.5 milliseconds and over a non-proxied environment is 5.4 milliseconds. Contrastingly, popular Web benchmarks show that even the fastest websites have an average page-download time of 0.5 seconds when downloaded directly from the Internet [38].

Since our overhead is two orders of magnitute less than the fastest page-download times we believe that SessionShield can be used by desktop and mobile systems without perceivable performance costs.

## 5 Implementation

We decided to prototype SessionShield using Python. We used an already implemented Python proxy, TinyHTTPProxy [9], and added the session detection mechanisms that were described in Section 3. The advantage of implementing SessionShield as a stand-alone personal proxy instead of a browser-plugin relies on the fact that cookies residing in the browser can still be attacked, e.g. by a malicious add-on [18]. When sensitive data are held outside of the browser, in the data structures of the proxy, a malicious add-on will not be able to access them. On the other hand, a browser plugin can transparently support HTTPS and provides a more user-friendly install and update procedure.

The threshold value of SessionShield, mentioned in Section 3.3, was obtained by using the known session identifiers from our `HTTP-Only` experiment in Section 2.3 as input to our statistical algorithm and observing the distribution of the reported probabilities.

## 6 Related Work

*Client-side approaches for mitigating XSS attacks:* Noxes [15] is a defensive approach closely related to ours – A client-side Web proxy specifically designed to prevent session identifier (SID) theft. Unlike our approach, Noxes does not prevent the injected JavaScript to access the SID information. Instead, Noxes aims to deprive the adversary from the capability to leak the SID value outside of the browser. The proposed technique relies on the general assumption that dynamically assembled requests to external domains are potentially untrustworthy as they could carry stolen SID values. In consequence, such requests are blocked by the proxy. Besides the fact that this implemented policy is incompatible with several techniques from the Web 2.0 world, e.g., Web widgets, the protection provided by Noxes is incomplete: For example, the authors consider static links to external domains to be safe, thus, allowing the attacker to create a subsequent XSS attack which, instead of `script`-tags, injects an HTML-tag which statically references a URL to the attackers domain including the SID value.

Vogt et al. [36] approach the problem by using a combination of static analysis and dynamic data tainting within the browser to track all sensitive information, e.g., SID values, during JavaScript execution. All outgoing requests that are recognised to contain such data are blocked. However, due to the highly dynamic and heterogenous rendering process of webpages, numerous potential hidden channels exist which could lead to undetected information leaks. In consequence, [32] exemplified how to circumvent the proposed technique. In comparison, our approach is immune to threats through hidden channels as the SID never enters the browser in the first place.

Furthermore, browser-based protection measures have been designed that disarm reflected XSS attacks through comparing HTTP requests and responses. If a potential attack is detected, the suspicious code is neutralized on rendering-time. Examples for this approach include NoScript for Firefox [20], Internet Explorer's XSS Filter [31], and XSSAuditor for Chrome [3]. Such techniques are necessarily limited: They are only effective in the presence of a direct, character-level match between the HTTP request and its corresponding HTTP response. All non-trivial XSS vulnerabilities are out of scope. In addition, it is not without risk to alter the HTTP response in such ways: For instance, Nava & Lindsay [25] have demonstrated, that the IE XSS Filter could cause XSS conditions in otherwise secure websites. Finally, to confine potentially malicious scripts, it has been proposed to whitelist trusted scripts and/or to declare untrusted regions of the DOM which disallow script execution [10,23,5,21]. All of these techniques require profound changes in the browser's internals as well as the existence of server-side policy information.

*Security enhancing client-side proxies:* Besides Noxes, further client-side proxies exist, that were specifically designed to address Web application vulnerabilities:

RequestRodeo [12] mitigates Cross-site Request Forgery attacks through selectively removing authentication credentials from outgoing HTTP requests. As discussed in this paper, the majority of all Web applications utilize the SID as the de facto authentication credential. In consequence, RequestRodeo (and its further refinements, such as [33]) could benefit from our SID detection algorithm (see Section 3) in respect to false positive reduction.

HProxy [27] is a client-side proxy which protects users from SSL stripping attacks and from malicious JavaScript code that a Man-In-The-Middle (MITM) attacker could have added in a page to steal Web credentials. In order to differentiate between original and "added" JavaScript, HProxy takes advantage of the user's browsing habits to generate a template of each script, recording the static and the dynamic parts of it. If, at a later time, a script is detected which doesn't adhere to its original template, HProxy marks it as an attack and doesn't forward it to the browser.

*Server-side approaches:* The majority of existing XSS prevention and mitigation techniques take effect on the Web application's server-side. We only give a brief overview on such related work, as this paper's contributions specifically address client-side protection:

Several approaches, e.g., [29,26,8,41], employ dynamic taint tracking of untrusted data on run-time to identify injection attacks, such as XSS. Furthermore, it has been shown that static analysis of the application's source code is a capable tool to identify XSS issues (see for instance [17,39,14,37,7]). Moreover, frameworks which discard the insecure practice of using the string type for syntax assembly are immune against injection attacks through providing suitable means for data/code separation [30,11]. Jovanovic et al. [13] use a server-side proxy which rewrites HTTP(S) requests and responses in order to detect and prevent Cross-site Request Forgery. Finally, cooperative approaches spanning server and browser have been described in [2,19,24].

## 7 Conclusion

Session hijacking is the most common Cross-site Scripting attack. In session hijacking, an attacker steals session-containing cookies from users and utilizes the session values to impersonate the users on vulnerable Web applications. In this paper we presented SessionShield, a lightweight client-side protection mechanism against session hijacking. Our system, is based on the idea that session identifiers are not used by legitimate client-side scripts and thus shoudn't be available to the scripting engines running in the browser. SessionShield detects session identifiers in incoming HTTP traffic and isolates them from the browser and thus from all the scripting engines running in it. Our evaluation of SessionShield showed that it imposes negligible overhead to a user's system while detecting and protecting almost all the session identifiers in real HTTP traffic, allowing its widespread adoption in both desktop and mobile systems.

## Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, the Research Fund K.U.Leuven and the EU-funded FP7-project WebSand.

## References

1. Apache.org, `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`
2. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E.P., Karagiannis, T.: xjs: Practical xss prevention for web application development. In: Proceedings of the 1st USENIX Conference on Web Application Development, WebApps 2010 (2010)
3. Bates, D., Barth, A., Jackson, C.: Regular expressions considered harmful in client-side XSS filters. In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010). ACM, New York (2010)
4. Web Application Security Consortium. Web Hacking Incident Database
5. Erlingsson, U., Livshits, B., Xie, Y.: End-to-end Web Application Security. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS 2007) (May 2007)

6. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th International Conference on World Wide Web (WWW 2007). ACM, New York (2007)
7. Geay, E., Pistoia, M., Tateishi, T., Ryder, B., Dolby, J.: Modular String-Sensitive Permission Analysis with Demand-Driven Precision. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009 (2009)
8. Halfond, W.G.J., Orso, A., Manolios, P.: Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In: Proceedings of the 14th ACM Symposium on the Foundations of Software Engineering, FSE (2006)
9. Hisao, S.: Tiny HTTP Proxy in Python
10. Jim, T., Swamy, N., Hicks, M.: Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In: Proceedings of the 16th International World Wide Web Conference (WWW 2007) (May 2007)
11. Johns, M., Beyerlein, C., Giesecke, R., Posegga, J.: Secure Code Generation for Web Applications. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 96–113. Springer, Heidelberg (2010)
12. Johns, M., Winter, J.: RequestRodeo: Client Side Protection against Session Riding. In: Proceedings of the OWASP Europe 2006 Conference (2006)
13. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Proceedings of IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm) (2006)
14. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In: IEEE Symposium on Security and Privacy (May 2006)
15. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In: Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006) (April 2006)
16. Knuth, D.E.: The Art of Computer Programming, vol. 2. Addison-Wesley Publishing Company, Reading (1971)
17. Livshits, B., Lam, M.S.: Finding Security Vulnerabilities in Java Applications Using Static Analysis. In: Proceedings of the 14th USENIX Security Symposium (August 2005)
18. Ter Louw, M., Lim, J.S., Venkatakrishnan, V.N.: Extensible web browser security. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 1–19. Springer, Heidelberg (2007)
19. Louw, M.T., Venkatakrishnan, V.N.: BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In: IEEE Symposium on Security and Privacy (Oakland 2009) (May 2009)
20. Maone, G.: NoScript Firefox Extension (2006)
21. Meyerovich, L.A., Livshits, B.: Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In: Proceedings of 31st IEEE Symposium on Security and Privacy (SP 2010) (2010)
22. Microsoft. Mitigating Cross-site Scripting With HTTP-only Cookies
23. Mozilla Foundation. Content Security Policy Specification (2009)
24. Nadji, Y., Saxena, P., Song, D.: Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: Network & Distributed System Security Symposium, NDSS 2009 (2009)
25. Nava, E.V., Lindsay, D.: Our favorite XSS filters/IDS and how to attack them. Presentation at the BlackHat US Conference (2009)

26. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: the 20th IFIP International Information Security Conference (May 2005)
27. Nikiforakis, N., Younan, Y., Joosen, W.: HProxy: Client-side detection of SSL stripping attacks. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 200–218. Springer, Heidelberg (2010)
28. OWASP Top 10 Web Application Security Risks
29. Pietraszek, T., Berghe, C.V.: Defending against Injection Attacks through Context-Sensitive String Evaluation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 124–145. Springer, Heidelberg (2006)
30. Robertson, W., Vigna, G.: Static Enforcement of Web Application Integrity Through Strong Typing. In: Proceedings of the USENIX Security Symposium, Montreal, Canada (August 2009)
31. Ross, D.: IE 8 XSS Filter Architecture/Implementation (August 2008)
32. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 86–103. Springer, Heidelberg (2009)
33. De Ryck, P., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 18–34. Springer, Heidelberg (2010)
34. WhiteHat Security. XSS Worms: The impending threat and the best defense
35. Alexa: The Web information company
36. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium, NDSS 2007 (2007)
37. Wassermann, G., Su, Z.: Static Detection of Cross-Site Scripting Vulnerabilities. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany. ACM Press, New York (May 2008)
38. Performance Benchmark - Monitor Page Load Time — Webmetrics
39. Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In: 15th USENIX Security Symposium (2006)
40. XSSed — Cross Site Scripting (XSS) attacks information and archive
41. Xu, W., Bhatkar, S., Sekar, R.: Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In: 15th USENIX Security Symposium (August 2006)
42. Zhou, Y., Evans, D.: Why Aren't HTTP-only Cookies More Widely Deployed? In: Proceedings of 4th Web 2.0 Security and Privacy Workshop, W2SP 2010 (2010)

# Security Sensitive Data Flow Coverage Criterion for Automatic Security Testing of Web Applications

Thanh Binh Dao[1] and Etsuya Shibayama[2]

[1] Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 O-okayama Meguro Tokyo Japan
dao3@is.titech.ac.jp
[2] Information Technology Center, The University of Tokyo, 2-11-16 Yayoi Bunkyo-ku Tokyo Japan
etsuya@ecc.u-tokyo.ac.jp

**Abstract.** Common coverage criteria for software testing, such as branch coverage and statement coverage, are often used to evaluate the adequacy of test cases created by automatic security testing methods. However, these criteria were not originally defined for security testing. In this paper, we discuss the limitation of traditional criteria and present a study on a new criterion called security sensitive data flow coverage. This criterion aims to show how well test cases cover security sensitive data flows. We conducted an experiment of automatic security testing of real-world web applications to evaluate the effectiveness of our proposed coverage criterion, which is intended to guide test case generation. The experiment results show that security sensitive data flow coverage helps reduce test cost while keeping the effectiveness of vulnerability detection high.

**Keywords:** automatic security testing, web application, coverage criteria, data flow.

## 1 Introduction

In security testing of web applications, the selection of coverage criteria for adequacy evaluation of test cases is based on a trade off between test cost and vulnerability detection effectiveness. Traditional coverage criteria such as branch coverage and statement coverage used in general software testing [17] have also been widely applied to security testing [16,12]. Branch coverage-based methods try to generate test cases to execute as many branches of the program under test as possible. They provide high effectiveness for vulnerability detection but may cost much time because many test cases unrelated to vulnerabilities may be executed. When security testing is done frequently in software development process, it is necessary to reduce test cost so branch coverage may not be a suitable criterion. On the other hand, statement coverage only requires any statement in the program source code to be executed at least once. When using statement coverage in security testing, security sensitive sinks such as database query function

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 101–113, 2011.

*mysql_query* become the targets of coverage measurement. Because statement coverage is completely insensitive to control structures, it may not be able to find vulnerabilities which depend on certain control flows of the program. Therefore, there is a need of new coverage criteria that can help automatic security testing reduce test cost while keeping vulnerability detection effective.

In this paper, we propose new coverage criterion called security sensitive data flow coverage as another choice for automatic security testing when low cost but effective security test is needed. Security sensitive data means the values of variables which are passed to the security sensitive functions during the execution of the program. Any change to these data could affect the revealing of vulnerabilities. To get high vulnerability detection effectiveness, this coverage requires test cases to cover as many security sensitive data flows as possible.

To evaluate the effectiveness of our proposed coverage criterion, we conducted an experiment of security testing of three real-world web applications. We used the automatic security testing tool Volcano which was developed by us in the previous work [7]. Volcano does white-box security testing to find SQL Injection vulnerabilities in web applications written in PHP language. We modified Volcano so that it is able to generate test cases based on the traditional coverage criteria and the proposed criterion.

This paper makes the following contributions:

- We discuss the problems of traditional coverage criteria when using in security testing of web applications.
- We propose security sensitive data flow coverage criterion to deal with those problems.
- We show the effectiveness of the proposed coverage criterion through experiment results.

In this paper, section 2 describes web application vulnerabilities with SQL Injection vulnerability in detail and existing security testing techniques. Section 3 presents the problems of traditional coverage criteria when using in security testing. Section 4 presents our proposed coverage criterion for dealing with those problems. Section 5 presents our experimental evaluation on real-world web applications. Section 6 gives a discussion. Section 7 presents an overview of related work and section 8 concludes this paper.

## 2 Background

### 2.1 Web Application Vulnerabilities

Many web application vulnerabilities have been well documented and the mitigation methods have also been introduced [1]. The most common cause of those vulnerabilities is the insufficient input validation. Any data originated from outside the program code such as input data provided by user through web forms should always be considered malicious because it may contain content unexpected by developers. SQL Injection, Remote code execution or Cross-site Scripting are

very common vulnerabilities of that type [3]. Below we give a brief introduction to SQL Injection vulnerability though the security testing method presented in this paper is not limited to this vulnerability.

SQL injection vulnerability allows an attacker to illegally manipulate database by injecting malicious SQL codes into the value fields of input parameters of an http request sent to victim website.

List 1 shows a program that uses the database query function *mysql_query* to get the name of the user specified by the GET input parameter *username* and print the result to the client browser. A normal http request with the input parameter *username* looks like "http://example.com/index.php?username=bob". The dynamically created database query becomes "SELECT * FROM users WHERE username='bob' AND usertype='user'".

```
1: <?php
2: $result = mysql_query("SELECT * FROM users WHERE
      username='" . $_GET['username'] . "' AND usertype='user'");
3: show_result($result);
4: ?>
```

*List 1: An example of program in PHP containing an SQL Injection vulnerability*

This program is vulnerable to SQL Injection attacks because *mysql_query* uses the input value of *username* without sanitizing malicious codes. A mallicious code can be a string that contains SQL symbols or keywords. If an attacker send this request "http://example.com/index.php?username=alice'--", the query becomes "SELECT * FROM users WHERE username='alice'--' AND usertype='user'". In this query, the quote character encloses previous quote, and the symbol "--" comments out the subsequent text. The program will be forced to illegally get and show secret information of user *alice* instead of *bob*.

To protect from SQL Injection in this case, the program must escape all SQL symbols and remove malicious keywords in user-provided input before using to create the database queries. As another countermeasure, developers can use prepared statement instead of using function *mysql_query* [2].

### 2.2 Automatic Security Testing

Security testing methods for web applications can be classified into black-box testing and white-box testing. Black-box testing is a method that executes the test without knowing the internal structure and operation of the web applications under test. Generally, it sends attack requests that contain malicious codes to the web site and then searches the response html pages for signatures such as error messages to detect vulnerabilities [10,4,5]. Black-box testing is fast and useful especially when the source code of the web application program is not available. However, one of the limitation is that the test result cannot show exactly where the vulnerabilities are in the program code. Furthermore, without

knowing anything about the source code, black-box testing may not be able to create effective test cases and may miss many vulnerabilities.

On the other hand, white-box testing, also known as structural testing, is based on analysis the internal structure and operation of the web application. Static analysis is one of the techniques widely used [6,11,18]. Based on well understanding of the program code, although much time is required for code analysis, they can create useful test cases to find vulnerabilities and can achieve better results than black-box testing.

In the previous work, we created a tool called Volcano, a white-box automatic security testing tool to find SQL Injection vulnerabilities in web applications written in PHP language [7]. Security testing by Volcano is separated into two steps as described below.

`Step 1:` Volcano acts as black-box testing that uses initially provided URLs to generate test cases for the target web application. New test cases is created from static links and web forms found in the response html pages. Attack requests are created by injecting prepared SQL attack codes into the value fields of input parameters. Step 1 finishes when no more new links or forms are found.

`Step 2:` Volcano uses input data generation techniques to create test cases in order to execute branches which are not executed by previous test cases. The purpose is to increase branch coverage with the hope that by exploring more branches the test will find more vulnerabilities. In addition to the algorithm of generating input data for solving character string predicate [8], a similar algorithm is also applied for numerical comparison predicate.

Volcano uses taint tracking technique to track taint information of input data and detect vulnerabilities by finding the tainted SQL symbols and keywords in dynamically generated SQL queries [9]. The effectiveness of Volcano has been shown by the results of the experiments on real-world web applications.

## 3 Traditional Coverage Criteria

In this section, we discuss the limitation of traditional coverage criteria when using in security testing.

List 2 shows an example of a web application written in PHP language. The program fist declare function *query* at line 2. This function is a wrapper function of the security sensitive sink *mysql_query* called at line 3. The program starts to execute from line 6. If the value of input parameter $_GET['mode'] is "news" and the input parameter $_GET['newsid'] is specified in the http request, the program will create and execute an SQL query at line 7 to get the *news* content corresponding to the *newsid* from the database server. Otherwise, an SQL query $sql is created at line 10. This query string is extended at line 12 to include the condition for getting data corresponding to the article id *articleid*. The program finally get the information of web access status from the database at line 18 and display it to the client web browser.

At line 7, because $_GET['newsid'] is used directly, the program is vulnerable to SQL Injection attacks at this point. The program is also vulnerable at

```
 1: <?php
 2: function query($query) {
 3:   return mysql_query($query);
 4: }
 5:
 6: if ($_GET['mode'] == "news" && isset($_GET['newsid'])) {  // (1)
 7:   $content = query("SELECT * FROM news WHERE newsid="
                 . $_GET['newsid']);
 8:   /* code to show content here */
 9: } else {
10:   $sql = "SELECT * FROM articles";
11:   if (! empty($_GET['articleid'])) {    // (2)
12:     $sql .= " WHERE articleid=" . $_GET['articleid'];
13:   }
14:   $content = query($sql);
15:   /* code to show content here */
16: }
17:
18: $counter = query("SELECT * FROM access_counter");
19: /* code to show counter here */
20: ?>
```

*List 2: An example of a web application written in PHP language*

the execution of line 14 because the value of variable $sql used to call security sensitive sink *mysql_query* includes the value of input variable $_GET['articleid'] from line 12 without sanitizing. The query in line 18 does not contain any input data so it is safe with SQL Injection attacks.

To find both two vulnerabilities in this program, a security testing tool must generate test cases to execute both branches of the *if* condition (1) and also the TRUE branch of the *if* condition (2).

Consider the situation when the security testing tool finished some test cases, and assume that only the FALSE branch of the *if* condition (1) and the FALSE branch of the *if* condition (2) were executed. So no vulnerability was found at this step. We will describe how the security testing tool generates further test cases when the test case generation is based on branch and sink coverage criteria.

### 3.1 Branch Coverage-Based Security Testing

Branch coverage criterion requires each branch in the program to be executed at least once. Thus, the security testing tool will try to generate new test cases so that the TRUE branch of the *if* condition (1) and the TRUE branch of the *if* condition (2) execute. With this web application program, test cases for executing those branches can be created by setting the values of input parameters properly. By generating test cases in which malicious SQL codes are injected into the value of $_GET['newsid'] and $_GET['articleid'], the security testing tool can detect all two vulnerabilities.

In general, branch coverage criterion is good for exhaustive testing of web applications. However, when there are many branches that do not contain operations related to vulnerabilities, generating test cases for executing those branches only cost time without finding any more vulnerabilities.

### 3.2 Sink Coverage-Based Security Testing

Sink coverage criterion requires all calls to security sensitive sinks to be executed at least once during the test.

In List 2, because line 18 is executed without any condition, the coverage of the security sensitive sink *mysql_query* reaches 100% right after the first test case. Thus the security testing whose test case generation is guided by sink coverage will stop to generate test cases for executing the TRUE branch of the *if* condition (1) and (2). As the result, no vulnerability will be found.

In general, sink coverage is one of the most simple coverage criteria and it helps the test to stop early, but the effectiveness of vulnerability detection is low.

### 3.3 Problem Summary and Our Approach

The space of test cases of branch coverage-methods is too large, but with statement coverage, it is too confined. There is a need of an intermediate criterion that is more suitable for security testing.

## 4 Security Sensitive Data Flow Coverage Criterion

To address the problems of traditional coverage criteria when using in security testing, we take into account the data flow of security sensitive data. Security sensitive data means the values of variables which are passed to the security sensitive functions during the execution of the web application program. Any change to these data could affect the revealing of vulnerabilities. We call those type of variables *security sensitive variables*. A branch is *security sensitive* if it contains:

- calls to security sensitive sinks (e.g. mysql_query)
- propagation of the values of security sensitive variables through assignments

For example, in List 2, variable $sql is security sensitive because its value is passed to the function *query* at line 14, and then to the security sensitive sink *mysql_query* at line 3. We can see that, if the *if* condition (2) is False, the True branch is not executed and the value of input variable $_GET['articleid'] is not assigned to variable $sql. In this case, the security testing could not find the SQL Injection vulnerability caused by using unsanitized input data here. Of course, if the assignment at line 12 does not contain any input variable, testing the True branch of the *if* condition (2) will not find any vulnerability.

We claim that, by creating test cases that can at least test all security sensitive branches, high effectiveness of vulnerability detection can be achieved.

In this section, we will define our security sensitive data flow coverage criterion and describe how to use this criterion to guide test case generation for security testing.

### 4.1 Definition

Security sensitive data flow coverage (SSDFC) is measured by the percentage of security sensitive branches executed by test cases over all security sensitive branches existing in the program.

$$SSDFC = \frac{No.\ of\ executed\ security\ sensitive\ branches}{No.\ of\ total\ security\ sensitive\ branches}$$

Each loop in the program is counted once so the total number of security sensitive branches is finite. Branches are counted by summed up the number of branches of all conditional statements.

### 4.2 Security Sensitive Data Flow Coverage-Based Security Testing

Test cases for executing a branch should be created if the target branch is security sensitive. In practice, even if a security sensitive branch of an *if* condition was executed, the opposite branch should also be executed. It is because security sensitive branch may contain code to sanitize input data and successfully bypassing it may help reveal vulnerabilities. The algorithm of test case generation of security sensitive data flow coverage-based security testing is shown in List 3.

```
For each conditional statement such that one and only one of its branch
has not yet been executed
    If any of its branches are security sensitive
        Generate test cases to execute the unexecuted branch
    End
End
```

*List 3: Algorithm of test case generation guided by security sensitive data flow coverage criterion*

In List 2, test cases should be created to execute the False branch of the *if* condition (1) (the *else* block) and True branch of the *if* condition (2) because the value of security sensitive variable $sql is changed in line 10 and 12. Although the value assigned to variable $sql at line 10 is a constant string, we conservatively consider this change is critical and the code need to be tested. If attack code was injected into the value of input variable $_GET['articleid'], the vulnerability of directly using input data here will be detected.

### 4.3 Determining Security Sensitive Branch

We discuss how to determine whether a branch is security sensitive in practice. The first case is that the target branch was not executed.In this case, we have

to do static analysis on this branch. The second case is that the target branch was executed by some previous test cases. As mentioned above, if this branch is security sensitive, then test cases for executing the opposite branch should also be executed. In this case, we can do dynamic analysis of the execution trace of this branch (e.g. the execution trace is collected by Volcano from inside modified PHP interpreter). However, if the branch contains other conditional statements that only one of its branches was executed, it may be unable to do dynamic analysis due to the lack of information of these unexecuted branches. Hence, static analysis is selected for both cases.

First, the analysis searches for calls to security sensitive sinks inside the target branch using the static call graph of the web application under test. If it found any call to security sensitive sinks, then the target branch is security sensitive and the analysis finishes. While precise static analysis can be done to generate the call graph, we chose context-insensitive analysis to make the implementation in our tool Volcano simple.

Secondly, the analysis finds if there is a variable assigned inside the target branch such that its value later reaches calls to security sensitive sinks. For doing that, all calls to security sensitive sinks in the subsequent code executed after the execution of the target branch are listed up first. The subsequent code is determined from the control flow graph in which each loop is counted once. To find the data flow of each variable from the assignment point in the target branch to each call to security sensitive sink, an intra-procedural and inter-procedural analysis with some conservative approximations was chosen to implement into Volcano. Alias analysis was not taken into account for simplicity. The analysis basically searches for variable assignments and function calls and maintains the information of the relationship between variables. If a variable is used to call a function such that that function calls security sensitive sinks in its execution (determined by the call graph mentioned above), then that variable is considered to be security sensitive.

### 4.4 Discussion

While there are some conservative approximations in the analysis above, they do not affect the preciseness of the vulnerability detection because taint tracking technique is used to precisely detect vulnerabilities.

For detecting vulnerabilities such as SQL Injection vulnerabilities which caused by insufficient input validation, people can think of a coverage criterion that considers data flow of tainted data originated from user input. However, it may be not practical because there are many features of scripting languages that make it difficult to define or measure the coverage. For example, function *extract* in PHP is used to extract all GET, POST input parameters into the symbol table so that they can be used as global variables without explicit declaration in the program. Another problem is that, not all input data are related to vulnerabilities. They may be used for control flow of executing certain branches which are not related to revealing of any vulnerability.

**Table 1.** Web applications used in the experiment. "No. of Conditions" indicates the number of conditional statements and "No. of Sinks" is the number of calls to security sensitive sink *mysql_query* in the program.

| Web Application | Description | Lines of Code | No. of Conditions | No. of Sinks |
|---|---|---|---|---|
| jobhut | Job board site | 2278 | 94 | 1 |
| easymoblog | Blog site | 9996 | 687 | 86 |
| phpaaCMS | CMS site | 13434 | 368 | 4 |

## 5 Experiment

In this section, we present an experiment to evaluate the cost-effectiveness of our proposed coverage criterion when using for automatic security testing of web applications written in PHP language to find SQL Injection vulnerabilities, in comparison with branch and statement coverage criteria.

### 5.1 Experiment Setup

To select real-world web applications for the experiment, we searched the Cyber Security Bulletins of US-Cert [19] for finding web applications which are written in PHP, use MySQL database and contain SQL Injection vulnerabilities. We randomly selected three web applications which are shown in Table 1.

Apache web server version 2.2, MySQL database server version 5.0 and the security testing tool Volcano which is written in Java are installed in a desktop PC with CPU Intel i7 2.85GHz, 3GB RAM, Ubuntu Linux OS.

We modified the algorithm of test cases generation of Volcano so that it can do security testing based on three type of coverage criteria: branch coverage, sink coverage and the proposed security sensitive data flow coverage.

Volcano uses one SQL attack code "1' or 1=1 --" to inject into the values of input parameters of http requests. Vulnerabilities are counted by the number of different vulnerable SQL query patterns at each function calls to security sensitive sink *mysql_query*. Two queries have the same pattern if they have the same query structure with the same set of parameters, regardless of the value of these parameters. For example, two queries "SELECT * FROM users WHERE user='bob'" and "SELECT * FROM users WHERE user='alice'" have same pattern, while "SELECT * FROM users WHERE user='bob'" and "SELECT * FROM users WHERE user='1' or 1=1 --'" are different because the latter contains SQL keyword 'or'. By using this counting method, we assume that the queries with different patterns are created from different execution paths of the program, so they must be shown as different vulnerabilities.

### 5.2 Experiment Results

Table 2 shows the results of our experiment. Row `Step 1` is the execution result of the first step (black-box testing) of security testing of Volcano. Three rows BC,

**Table 2.** Experiment results of security testing of web applications

| **jobhut** | | **Time** | **TCase** | **Vuln** | **ExC** | **GenTg** | **T&F** | **Sink** | **SSDFC** |
|---|---|---|---|---|---|---|---|---|---|
| Step 1 | | 20 | 105 | 12 | 63 (67%) | - | 24 | 1 (100%) | 35/59 (59%) |
| Final Result | BC | 152 | 386 | 19 | 79 (84%) | 28 | 51 | 1 (100%) | 46/59 (77%) |
| | SSDFC | 66 | 210 | 19 | 79 (84%) | 14 | 37 | 1 (100%) | 46/59 (77%) |
| | SC | 33 | 105 | 12 | 63 (67%) | 0 | 24 | 1 (100%) | 35/59 (59%) |

| **easymoblog** | | **Time** | **TCase** | **Vuln** | **ExC** | **GenTg** | **T&F** | **Sink** | **SSDFC** |
|---|---|---|---|---|---|---|---|---|---|
| Step 1 | | 222 | 188 | 7 | 267 (38%) | - | 46 | 31 (36%) | 30/103 (29%) |
| Final Result | BC | 4676 | 941 | 10 | 268 (39%) | 188 | 54 | 37 (43%) | 32/103 (31%) |
| | SSDFC | 1115 | 390 | 10 | 268 (39%) | 33 | 48 | 37 (43%) | 32/103 (31%) |
| | SC | 948 | 283 | 7 | 267 (38%) | 22 | 47 | 37 (43%) | 31/103 (30%) |

| **phpaaCMS** | | **Time** | **TCase** | **Vuln** | **ExC** | **GenTg** | **T&F** | **Sink** | **SSDFC** |
|---|---|---|---|---|---|---|---|---|---|
| Step 1 | | 50 | 255 | 17 | 103 (27%) | - | 25 | 4 (100%) | 21/55 (38%) |
| Final Result | BC | 187 | 639 | 31 | 117 (32%) | 80 | 54 | 4 (100%) | 45/55 (82%) |
| | SSDFC | 133 | 518 | 31 | 117 (32%) | 40 | 54 | 4 (100%) | 45/55 (82%) |
| Result | SC | 75 | 255 | 17 | 117 (32%) | 0 | 25 | 4 (100%) | 21/55 (38%) |

SSDFC, SC show the final execution results of Volcano which are based on branch coverage, security sensitive data flow coverage and sink coverage respectively. Note that the `Final Result` contains the result of `Step 1`. Column `Time` shows the execution time in second. `TCase` is the number of test cases. Column `Vuln` shows the total number of vulnerabilities detected by Volcano. Column `ExC` reports the total conditional statements executed during the test. Column `GenTg` shows the number of conditional statements whose True or False branches were the target of the test case generation to create test case for executing them. Column `T&F` shows the number of conditional statements whose both True and False branches were executed by test cases. Column `Sink` denotes the numbers of executed security sensitive sink. Finally, column `SSDFC` shows the security sensitive data flow coverage for each test, in which the denominator is the total number of security sensitive branches found in the source code and the numerator is the total number of security sensitive branches executed by test cases.

For all web applications, the number of test cases created by SSDFC-based method is fewer than BC-based method and its execution time reduced up to about 4 times in the case of "easymoblog". Execution time of SC-based method was smallest because the number of generated test cases was fewer than the other two methods. In each web application, vulnerabilities found by BC-based method and SSDFC-based method were the same. This implies that to find vulnerabilities in these web applications, SSDFC-based method is as effective as BC-based method. On the other hand, SC-based method found fewer vulnerabilities and all of them have been found in step 1. For the cases of "jobhut" and "phpaaCMS", the reason is that, in Step 1, all security sinks have been executed so sink coverage reached 100%, so no more test cases were generated (GenTG is 0). For the case of "easymoblog", only 36% of security sensitive sinks were executed in Step 1, so after that, 95 test cases were created to try to execute 22

branches that contains some other calls to security sensitive sinks. As the result, 43% of security sensitive sinks were executed. However, SC-based method did not find any more vulnerabilities.

The numbers in column `GenTg` shows the differences between each method. While BC-based method tried to generate test cases for executing many branches, only a small part of them are the targets of test case generation of SSDFC-based methods. Remember that those branches are targeted because they are security sensitive or the opposite branches of them are security sensitive. In the case of "phpaaCMS", although test case generation of BC-based method targeted many branches, the final result of `T&F` is the same with SSDFC-based method. For other two web applications, even though BC-based method provides better result of `T&F`, it could not find more vulnerabilities than SSDFC-based method. This means, in this experiment, many unnecessary test cases were created by BC-based method. Combined with the results shown in the column `SSDFC`, we can conclude that, security sensitive data flow coverage is a good coverage criterion to show the effectiveness of test cases created by security testing methods.

Overall, the experiment results show that security sensitive data flow coverage was successful to help security testing reduce test cost significantly while keeping high effectiveness of vulnerability detection in comparison with traditional coverage criteria.

## 6 Discussion

In software development, security testing is done frequently while coding. Methods which can effectively find vulnerability in reasonable time are preferred. Developers can choose branch coverage for security testing when they have time (e.g. nightly test) or want to do the test in which effectiveness of vulnerability detection is more concerned (e.g. acceptance test). The proposed security sensitive data flow coverage criterion can be one of the good choices for the purpose of frequent test. Sink coverage can be used for fast test to collect some basic information relating to security issues such as the number of security sensitive sinks and how easily test cases can be generated to cover all security sensitive sinks.

The experimental results of finding vulnerabilities shows that, our analysis method for determining security sensitive branches was sufficient enough to find vulnerabilities as many as BC-based method did in the selected web applications. For other real-world web applications, other analysis such as aliasing may be needed to achieve high effectiveness. However, doing complex analysis may raise the test cost. Again, choosing what analysis methods for the test depends on the purpose of the test with regard to the trade off between test cost and effectiveness.

## 7 Related Work

Smith et al. addressed the limitation of statement coverage and introduce target statement coverage (which is the same as sink coverage) and input variable

coverage [12]. Input variable coverage measures the percentage of input variables tested at least once by the test cases out of total number of input variables found in any target statement in the web application program. They considered only input variables which are dynamically user-assigned and appear in SQL statements in the production code. However, in practice, a web application may not directly use input variables in SQL statement, for example it may split or concatenate string value of input variables before use. Hence, simply counting the input variables in all SQL statements may not provide accurate coverage result. Another limitation is that this coverage criterion does not consider data flow of input variables so security testing based on this input variable coverage based method may miss some security sensitive branches and could not find some vulnerabilities. Our proposed security sensitive data flow coverage can solve these problems.

Some coverage criteria are proposed for adequacy evaluation of testing of database interaction in applications [13,14,15]. There are two types of criteria that focus on data-flow and on the structure of the SQL queries sent to the database. Data interaction points, in the case of security testing, are similar to security sensitive sinks in that they are defined as any statement in the application code where SQL queries are issued to the relational database management system [13]. Not only concentrate on testing database interaction, our proposed coverage criterion can be used for security testing to find many type of vulnerabilities caused by insufficient input validation.

## 8 Conclusion

In this paper, we discussed the problems of traditional coverage criteria, branch coverage and statement coverage, when using in automatic security testing to evaluate the adequacy of automatically generated test cases. To address these problems, we proposed security sensitive data flow coverage criterion which measures the percentage of security sensitive branches executed by generated test cases. The experiment results show that, the proposed security sensitive data flow coverage is a good intermediate coverage criterion that can help to reduce test cases while keeping the vulnerability detection effectiveness high.

## References

1. The Open Web Application Security Project: Vulnerability Category, `http://www.owasp.org/index.php/Category:Vulnerability`
2. The Open Web Application Security Project: SQL Injection Prevention Cheat Sheet, `http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`
3. Symantec Corporation: Five common Web application vulnerabilities, `http://www.symantec.com/connect/articles/five-common-web-application-vulnerabilities`
4. Chinotec Technologies Company: Paros, `http://www.parosproxy.org`

5. Acunetix Web Vulnerability Scanner (2008), http://www.acunetix.com/
6. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP 2006), pp. 258–263. IEEE Computer Society, Washington, DC (2006)
7. Dao, T.-B., Shibayama, E.: Idea: Automatic Security Testing for Web Applications. In: Massacci, F., Redwine Jr., S.T., Zannone, N. (eds.) ESSoS 2009. LNCS, vol. 5429, pp. 180–184. Springer, Heidelberg (2009)
8. Zhao, R., Lyu, M.R.: Character String Predicate Based Automatic Software Test Data Generation. In: Proceedings of the Third International Conference on Quality Software (QSIC 2003), p. 255. IEEE Computer Society, Washington (2003)
9. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: Twentieth IFIP International Information Security Conference, SEC 2005 (2005)
10. Huang, Y.-W., Huang, S.-K., Lin, T.-P., Tsai, C.-H.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International Conference on World Wide Web (WWW 2003), pp. 148–159. ACM, New York (2003)
11. Benjamin Livshits, V., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium (SSYM 2005), vol. 14, p. 18. USENIX Association, Berkeley (2005)
12. Smith, B., Shin, Y., Williams, L.: Proposing SQL statement coverage metrics. In: Proceedings of the Fourth International Workshop on Software Engineering For Secure Systems (SESS 2008), pp. 49–56. ACM, New York (2008)
13. Halfond, W.G.J., Orso, A.: Command-Form Coverage for Testing Database Applications. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), pp. 69–80. IEEE Computer Society, Washington, DC (2006)
14. Surez-Cabal, M.J., Tuya, J.: Using an SQL coverage measurement for testing database applications. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT 2004/FSE-12), pp. 253–262. ACM, New York (2004)
15. Kapfhammer, G.M., Soffa, M.L.: A family of test adequacy criteria for database-driven applications. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11), pp. 98–107. ACM, New York (2003)
16. Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL Injection and cross-site scripting attacks. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 199–209. IEEE Computer Society, Washington, DC (2009)
17. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)
18. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: IEEE Security and Privacy Symposium (2008)
19. Cyber Security Bulletins, US-Cert, http://www.us-cert.gov/cas/bulletins/

# Middleware Support for Complex and Distributed Security Services in Multi-tier Web Applications

Philippe De Ryck, Lieven Desmet, and Wouter Joosen

IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{philippe.deryck,lieven.desmet}@cs.kuleuven.be

**Abstract.** The security requirements of complex multi-tier web applications have shifted from simple localized needs, such as authentication or authorization, to physically distributed but actually aggregated services, such as end-to-end data protection, non-repudiation or patient consent management. Currently, there is no support for integrating complex security services in web architectures, nor are approaches from other architectural models easily portable. In this paper we present the architecture of a security middleware, aimed at providing a reusable solution bringing support for complex security requirements into the application architecture, while addressing typical web architecture challenges, such as the tiered model or the lack of sophisticated client-side logic. We both evaluate the security of the middleware and present a case study and prototype implementation, which show how the complexities of a web architecture can be dealt with while limiting the integration effort.

**Keywords:** middleware, multi-tier architecture, security, web application, non-repudiation.

## 1 Introduction

The online availability of more and more everyday services has heightened the complexity of web applications as well as the sensitivity of their assets, such as customer data, financial data or medical information. This evolution has also affected the security requirements, which have shifted from simple localized needs, such as authentication or authorization, to physically distributed but actually aggregated services, such as end-to-end data protection, non-repudiation or patient consent management.

The simple security requirements have been addressed by support for authentication schemes and access control models, both in distributed application architectures [9,10] as in web architectures [2,4,13]. Support for achieving complex security requirements is limitedly available for distributed application architectures, either as a middleware solution (e.g. [5,22]) or as a complementary software product (e.g. [18]). For web architectures however, there is no specific support

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 114–127, 2011.

for achieving complex security requirements, nor are the previously mentioned approaches easily portable to such architectures.

In this paper we present the architecture of a security middleware, aimed at providing a reusable solution bringing support for complex security requirements into the application architecture. The middleware offers a generic support layer, which supports both the implementation and integration of specific complex and distributed security services into a web architecture. We will show that the security middleware provides complete protection and is tamper-proof. Additionally, we conduct an extensive case study focused on a concrete security requirement, fair mutual non-repudiation[14,23], which can informally be defined as "the inability to subsequently deny an action or event"[5]. This case study illustrates the practical applicability of the middleware as well as the feasibility to achieve complex security requirements in an existing application.

The remainder of this paper is structured as follows. Section 2 introduces a motivating example and identifies the main challenges for supporting complex security requirements in web architectures. Section 3 presents the security middleware architecture and discusses implementation details using the non-repudiation case study as an example. Section 4 presents the implemented prototype and the evaluation of the middleware solution. We conclude the paper by discussing potential improvements and some suggestions for future developments (Section 5) and a summary of the contributions of this paper (Section 6).

## 2 Motivation and Background

We demonstrate the need for complex security requirements by means of an online banking system. The security requirements of such a system have evolved from ensuring basic security, such as secure communication or preventing unauthorized access, to complex security properties, for instance *non-repudiation for financial transactions*. Looking at the current level of non-repudiation for transactions in such applications, it seems that the non-repudiation is based on the successful execution of an authentication scheme, where even the very robust offline two-factor authentication schemes used in European banking systems are unable to provide non-repudiation guarantees for specific transactions. Basing non-repudiation on an authentication scheme is a flawed approach, since non-repudiation is based on evidence generated from business data, transactions in this case, where an authentication scheme is clearly not. To achieve non-repudiation for financial transactions in an online banking system, the business operations leading to the execution of a transaction need to be augmented with a non-repudiation security service. This service ensures the correct execution of a non-repudiation protocol that achieves fair mutual non-repudiation [14,23] between both participating parties, before the transaction is executed.

A different example are e-health systems, which are becoming increasingly popular and provide access to sensitive medical information, which needs to be shared among a patient, involved doctors, pharmacists, etc. As required by law [21], medical data can only be shared with a valid *patient consent*. Checking

patient consents when sharing data is an example of a complex security service, which requires extensive support in the application tier.

These examples already suggest some properties of different kinds of *complex security services*, such as achieving non-repudiation, patient consent management or end-to-end data protection. Compared to *simple security services*, such as an authentication or authorization service, a complex security service is tightly coupled to business functionality, such as the execution of financial transactions or accessing medical records. Additionally, most complex security services are also distributed, since there is an active involvement of the user and the server, which requires additional communication between the client and server. Examples of such interactive participation are the involvement in a non-repudiation protocol, the encryption of information or the creation of patient consents.

Support for simple security services has been integrated in distributed application architectures, such as CORBA [9,10], as well as in web architectures, such as JavaEE [4,13] or web services [17]. Complex security services on the other hand are not well supported, as will be discussed later in this section. Before discussing the existing support, we will first discuss several key challenges and requirements for integrating complex and distributed security services in web architectures.

### 2.1 Challenges for Complex Security Services in Web Architectures

When integrating complex security services in web architectures, the specific properties of this architectural model need to be taken into account. The web architecture is a tiered model, where the functionality is separated over four distinct tiers [20]. Figure 1 shows two different versions of the web architecture, with (a) being a full-fledged four tier web architecture such as JavaEE, mainly used in large enterprise applications, and (b) being a more lightweight version, where the web and application tier are located in one physical tier, but still remain logically separated from each other, as used by popular frameworks such as Spring and Struts.

Earlier work has already identified important issues with existing security technology, from which we can extract specific requirements for a security middleware. One of these requirements is compliance with the basic principles for

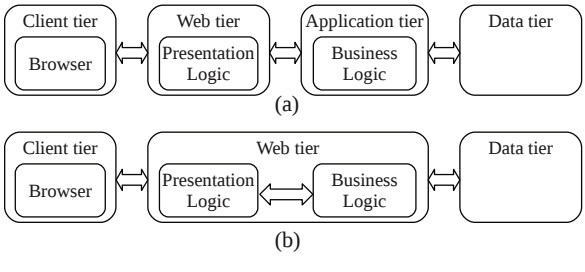



**Fig. 1.** (a) A typical enterprise web architecture with four distinct tiers. (b) A lightweight web architecture with business logic embedded in the web tier.

protection mechanisms [3,8], which state that (i) all paths to the functionality under protection need to be covered, (ii) the mechanism itself should be tamper-proof and (iii) the mechanism should be transparent to the application under protection, except for the specifically added security features. Apart from these security requirements, the middleware should be easy to integrate, without requiring intensive changes to the codebase of the application under protection, to ensure its practical applicability [2].

Next to these basic requirements, web architectures pose three interesting design challenges, that will need to be taken into account:

**C1. Coupling with business functionality:** to be able to augment the protection of business operations, the complex security service needs to be tightly coupled with the business functionality. For example to achieve non-repudiation on financial transactions, the security service will have to be coupled to the operation that executes a financial transaction.

**C2. Request/response model:** client/server communication in a web architecture is a uni-directional request/response model. This means that a message to the client can only be sent as a response to an earlier request. Additionally, a business transaction can consist of multiple request/response pairs.

**C3. User involvement:** a complex security service is typically distributed, where an active involvement of the user is required. This involvement requires both the ability to execute client-side code and to execute an out-of-band communication protocol, such as a non-repudiation protocol or the transfer of encrypted data.

Next to the middleware requirements and design challenges, the implementation of middleware for complex security services also poses specific challenges. One of them is the need for an interoperable representation of service-specific data, a lesson learned from earlier approaches [2]. Additionally, the correct implementation of a security protocol, if any is used, is truly important, since a vulnerability in such a protocol can render the entire security service useless.

### 2.2 Support for Complex Security Services

Generic support for complex and distributed security services is very limited to non-existent, in all types of architectures, which is why a complex security service is often developed from scratch, as part of an application. One observation is that the existing support is mainly suited for a machine-to-machine environment, without much user-involvement. The remainder of this section will discuss these application-extrinsic approaches to support complex security services. These approaches are mainly limited to one specific security service, non-repudiation, but will be discussed with the goal of leveraging them to support complex and distributed security services in a web architecture.

One of the earliest approaches is the CORBAsec specification [9], which addresses complex security services in the third level of the specification, by proposing an API for non-repudiation. Several challenges with the CORBAsec

specification [2], such as transparency and a fixed evidence format, were addressed by Wichert et al. [22], who propose a generic transparent CORBA non-repudiation service, based on code annotations. Continued work by Cook [5] has led to a middleware solution to achieve mutual non-repudiation in a business-to-business context, on top of the JavaEE platform. The middleware supports transparent integration by using a container service of the platform, which is available both at the client and at the server side. The last approach, which is based on the approach of Wichert, supports non-repudiation in an effective and secure manner, and can even be extended to other complex security services. It is however not directly applicable in a web architecture, due to the dependence on container services at the client-side, but it is nonetheless inspiring.

Support for complex security services is also available in other contexts, albeit for machine-to-machine communication. One example in the field of web services is a transparent non-repudiation service [1], which executes a non-repudiation protocol in the background. This service can also be used for similar security services, which depend on a protocol execution. Another approach for web services is attaching a non-repudiation log to messages [19], a technique less suitable to support other complex and distributed security services, since it is not aimed at strong interactivity between client and server. A totally different approach is the use of a standardized security interface, as provided by GSS-API [15], which is aimed at achieving robust interoperability for implementations of a security service. Although the specification of GSS-API is rather limited to origin authentication, integrity and confidentiality, it could be extended to support more complex security services as well.

The discussed approaches towards supporting complex security services are not directly applicable in a web architecture, but provide sufficient inspiration. Especially the work of Cook, which provides non-repudiation in a business-to-business context, presents an interesting way to integrate a complex security service on the server-side. When trying to adapt the discussed approaches to fit in a web architecture, the previously identified problems and challenges reoccur.

## 3 Middleware Support

The goal of the security middleware is to augment the protection of the business logic of a web application, by providing an easy way to integrate a complex and distributed security service in the application. This can be achieved by means of a security middleware, which can be integrated with the architecture of the application under protection and addresses the specific web architecture challenges. Figure 2 provides a high-level overview of the tiered web architecture, where the white blocks represent the application under protection. The flow in such an application always starts from the client, based on the request/response model, which sends a request to the web tier. The web tier processes the request and will eventually invoke a business operation, located in the business tier. The business logic processes the actual request, potentially using persistent data from the data tier, and returns a response to the web tier, which sends a response to the client.

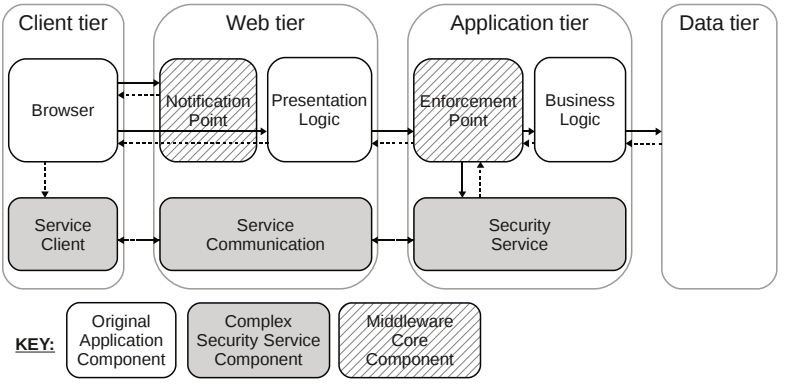



**Fig. 2.** The high-level architecture of the security middleware in a four tier web architecture

The first challenge for integrating a complex security service in a web architecture is achieving tight coupling with the business functionality of the application under protection. This can be achieved by integrating the security service into the application tier, similar to the approach of Cook [5]. Figure 2 shows how the *enforcement point* effectively couples the *security service* to the *business logic*.

Involving the user in the complex and distributed security service poses a second major challenge. Solving this challenge means both the ability to execute client-side code and the ability to communicate with the security service at the application tier. The former is represented by the *service client* in Figure 2 and can be achieved using locally installed software, such as a browser plugin, or remotely downloaded code, such as JavaScript code, a Java Applet, Flash code, etc. Addressing the latter requires security service-specific out-of-band communication. Direct communication between the client and application tier is not possible, due to the architectural model as well as the underlying infrastructure, which contains perimeter firewalls and web application firewalls. Therefore, we have chosen to integrate the out-of-band communication with the normal communication channel through the web tier, as shown by the *service communication* component in Figure 2.

In the remainder of this section, we will discuss the design details of the security middleware, applied to the non-repudiation case study which has been performed. In this case study, the support for complex and distributed security services is used to integrate non-repudiation for financial transactions in an online banking application [6]. For the details of the concept of non-repudiation, which are not really relevant for this paper, we refer to the literature [5,14,23]. Even though the discussion is focused around non-repudiation to avoid an abstract explanation of how the middleware works, other complex security services, such as end-to-end data protection or patient consent management, are also supported.

### 3.1 Detailed Design Based on a Non-repudiation Case Study

Figure 3 (a) shows the operations that lead to the execution of a financial transaction. Figure 3 (b) illustrates the changes introduced by the integration of a complex security service, a process which will be discussed next. To achieve non-repudiation for financial transactions, the non-repudiation service needs to be coupled with the "confirm wire transfer" operation. The coupling is achieved on the level of method invocations using interceptor techniques [7] in the enforcement point, similar to the approach taken by Cook [5]. The enforcement point knows which operations need to be protected by means of an application-specific configuration file. Apart from coupling the security service to a business operation, the enforcement point addresses the need of the security service to have access to extensive business information. For instance, the non-repudiation service needs details about the financial transaction and the user within the system, to construct a plaintext, which is the basis for the non-repudiation evidence.

A complex and distributed security service requires client-side support and user participation, which is not available in a web browser. Providing this client-side support can be done using a locally installed piece of software, a very impractical approach due to deployment and management issues. A better alternative is the use of remotely downloaded code, of which multiple forms are widely supported (JavaScript, Java, Flash, etc.). The prototype implementation uses a Java Applet, since Java offers strong smartcard support, which the prototype uses for identity management and cryptographic operations.

The initiation of a complex security service is not trivial. Following the request/response model of a web architecture, the client should initiate the communication process, but does not know when non-repudiation is required. The enforcement point does know this, but there is no way for the enforcement point or security service to send a message to the client tier, without potentially causing unwanted effects in the presentation logic (e.g. triggering generic exception handling mechanisms). This problem has been addressed by introducing the *notification point*, which is configured to know which incoming requests will lead to the invocation of a protected business operation. It intercepts such requests and responds to the client with a security notification. Receiving such a notification informs the client of the non-repudiation requirement, ensures that the client-side applet is loaded and automatically re-sends the original web request through the web tier. The notification point recognizes this repeated request and allows it to be handled by the presentation logic, from where it will result in the invocation of a protected business operation, which is handled by the enforcement point. In the mean time at the client-side, the security notification has been passed on to the applet, which starts executing the complex security service independently from the containing page.

The final part of the design is the realization of an out-of-band communication channel, which has to pass through the web tier, in order not to violate the tiered model of the web architecture. Therefore a communication component, which converts incoming web requests to method invocations for the security service and does the reverse with the responses, is provided by the web tier. The service


(a) Original application

(b) Application with an integrated non-repudiation service

**Fig. 3.** Sequence diagram showing the execution of a financial transaction

client can communicate directly with the web tier using Java Sockets if desired, but can also re-use the already established communication channel between the browser and the web tier, using a JavaScript communication mechanism provided by the page. The latter approach has the advantage that the security features of the existing communication channel, such as SSL protection, are also available for the security service-specific communication.

The discussion of the middleware in the non-repudiation case study shows how specific challenges within a web architecture can be addressed, but does not explain how other complex and distributed security services can be supported. Figure 4 shows the internal design of the security middleware, which clearly shows the generic middleware layer, aimed at supporting multiple complex security services, as well as the non-repudiation-specific layers.

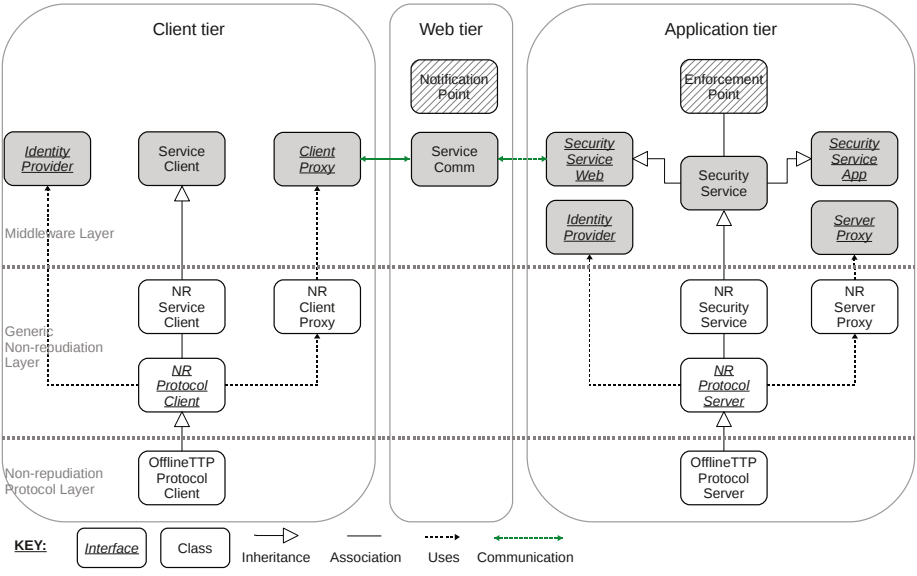



**Fig. 4.** The detailed design of the security middleware and a specific complex security service (non-repudiation)

The concrete implementation of complex and distributed security services on top of the middleware layer can be achieved by providing customized operations in the *service client* and *security service*. Using security service-specific configuration files, the service can be configured to fit the application requirements. The middleware layer provides abstractions for the concrete details of integration in the application, which allows the easy and rapid implementation of a complex security service, such as patient consent management or end-to-end data protection. An example of this are the generic and configurable enforcement point and notification point. Both the client and server-side offer an *identity provider*, which simplifies the management of identity information, such as certificates, smartcards, etc. The *client proxy* at the client side provides abstractions for communication with the server or with external parties. The server-side equivalent is the *server proxy*, which simplifies communication with external parties.

## 4 Prototype and Evaluation

This section presents the evaluation of the middleware. The actual integration of a prototype of the middleware with a prototype of a banking application shows the practical applicability as well as the configuration effort. Additionally, we discuss how we have addressed the basic requirements and design or implementation challenges. Finally, we also present the results of a security analysis, indicating the security of the security middleware.

### 4.1 Prototype Implementation and Configuration

We have implemented the case study of the online banking application, with the requirement of achieving non-repudiation for financial transactions. The banking system is designed and implemented independently from the security middleware, to assess the practicality of the integration process. Although the implemented prototype offers a subset of the designed functionality, it does include important security features such as strong authentication and authorization. The business logic of the partial implementation of the banking application counts 3514 lines of code. Next to the banking application, we have also developed a prototype of the security middleware and the non-repudiation security service. The generic middleware layer counts 3219 lines of code, the non-repudiation support layer counts 1407 lines and the specific protocol implementation, in the form of state machines, counts 4294 lines. Detailed design information can be found in [6] and the source code is available for download[1].

Using the developed prototypes, we have configured the non-repudiation service to be integrated with the banking application, as previously shown in Figure 3. The configuration and integration process is purely declarative, by means of XML configuration files, which allows the middleware to be integrated, without changing the original application code. The amount of configuration instructions to integrate the middleware is dependent on the number of operations that need to be protected. In the non-repudiation case study, we protected two sensitive business operations (executing financial transactions and creating new customers), which resulted in the following configuration cost: 57 XML elements[2] have been created and 131 source lines of code (Java) have been written. This last cost is to be attributed to the retrieval of business information, which forms the plaintext of the non-repudiation protocol.

### 4.2 Analysis of the Security Middleware

Next to the practical applicability, addressing the design challenges is an important aspect of the security middleware. The principal design challenges, more specifically (a) coupling with business functionality, (b) the request/response model and (c) user involvement have been addressed in the design of the middleware, as discussed in the previous section. Service-specific data is represented using an XML format, which is open and easily readable. A detailed protocol analysis is out of scope for this paper, but the implementation of the non-repudiation protocol has been achieved using a careful conversion to state machines, which have literally been implemented.

The security properties of the middleware, as well as the effectiveness of the provided protection are verified using a detailed security analysis using STRIDE [11]. The security analysis examines the claim that *protected business operations will only be executed if the security service has been contacted and has approved*

[1] `http://people.cs.kuleuven.be/philippe.deryck/papers/essos2011/src.zip`

[2] The XML element count does not include structural elements, only elements with application-specific values.

*the pending execution.* This means that if the security service is not satisfied and signals a problem, the business operation will not be executed. The details of a specific security service are regarded as a black box in the security analysis, because the middleware supports multiple different security services.

The claim will be investigated for a specific attacker profile: the attacker can carry out active attacks on information that passes the trust boundaries of the system. This includes messages sent between client and server, but also messages sent between two client components, or messages between the server and another party. Internal attacks on the server infrastructure or components will not be considered (e.g. spoofing attacks between the web and application tier). The analysis also makes certain environment assumptions, such as absence of business logic flaws in the application under protection, secure communication channels, a trusted codebase for client and server platform and reliable certificate validation using the Online Certificate Status Protocol (OCSP) [16].

After analyzing the middleware solution and carefully applying STRIDE to all elements, we have concluded that the only way to have a protected business operation executed is by getting a positive answer from the security service. This positive answer can only be obtained if the security service has fulfilled the required security properties. Concretely for the non-repudiation security service, this can only happen if the non-repudiation protocol ends in the *SUCCESS* state, which ensures the possession of the required evidence. Any attempt to tamper with the security service-specific data, such as the communication messages or identity information, will be detected before the security service gives its answer. This includes the invocation of the servlet using potentially compromised[3] JavaScript operations. Therefore, we can conclude that a protected business operation can only be executed if the security service has been executed successfully, for instance if both parties possess the appropriate non-repudiation evidence.

## 5 Discussion

In this section we discuss potential improvements and some suggestions for future developments. These include a discussion of lack of support for transactions, a description of how the security middleware can be implemented for lightweight web architectures, potential automation using tool support and a future approach by integrating the support in the application platforms.

**Transactional Support.** One problem with invoking a security service before the method invocation takes place, is that it might provide certain security properties about the invocation, but says nothing about the result. A simple approach is to make the security service provide protection for the response too [5]. The problem with this approach lies in dealing with malicious clients or

[3] Using XSS attacks in the application under protection, the JavaScript code of the security middleware could be compromised, but can not be used to subvert the server-side security service using falsified evidence.

communication failures. What if the client refuses to execute a non-repudiation protocol for the response? Can the operation be undone?

The appropriate solution for this problem involves the support for transactions, using a transaction service, which supports a rollback operation in case the security service fails on the response of the invocation. Sensitive operations desiring this behavior should be implemented with transaction support, or should be modified to do so.

**Lightweight Web Architectures.** As discussed in the introduction, not all web architectures follow the four tier architecture. For web architectures with only three tiers, the use of the Spring framework is very popular, even for more complex business systems. The conversion of the EJB implementation to a Spring implementation is straightforward. The only difference is the implementation of the enforcement point, which is no longer located in the application tier. The enforcement point can be implemented as a bean in the web tier, and will operate on business operations, which still are methods of Java objects. The interception mechanism is changed from EJB Interceptors to Spring Method Interceptors [12].

**Tool Support.** The integration process can be greatly simplified using automated tool support. The automation of the application-independent integration is straightforward, and the application-dependent integration can be partially supported as well. By means of a tool, the integrator can select the business operations that need to be protected and automatically configure the servlets requiring a notification message. The integration process is further supported by aiding with the configuration of the business data required by the security service.

**Native Support.** The current implementation has been developed to be compatible with current browser and server technology, as is the de facto standard for web application solutions. In that light, some of the proposed solutions can be simplified by evolving towards built-in support for complex security services. An important improvement to modern browser technology, would be the possibility to execute complex security protocols, such as non-repudiation, directly from the browser, much like authentication now. A second improvement is possible at the server-side, where application servers can provide context-spanning services, that allow the application tier to simply notify a client of a specific security requirement, without the presentation logic of the web tier having to deal with it. These two functionality improvements can simplify the design of the middleware to the *enforcement point* and the *security service*.

## 6 Conclusion

Based on the observation that there is currently no support for integrating complex and distributed security services into a multi-tier web application and that approaches in other architectures are not readily portable to a web architecture, we have analyzed what challenges need to be overcome to provide such support.

We discussed the architectural and detailed design of a security middleware which provides this support, while addressing the aforementioned challenges. We have implemented the security middleware and used a practical evaluation approach, based on a case study of an online banking system. In this evaluation, we have shown that the middleware can be used to integrate complex security services into a multi-tier web application, while providing strong security guarantees.

**Acknowledgements.** This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT and the Research Fund K.U. Leuven.

## References

1. Agreiter, B., Hafner, M., Breu, R.: A fair non-repudiation service in a web services peer-to-peer environment. Computer Standards & Interfaces 30(6), 372–378 (2008)
2. Alireza, A., Lang, U., Padelis, M., Schreiner, R., Schumacher, M.: The challenges of corba security, pp. 61–72 (2000)
3. Anderson, J.P.: Computer security technology planning study volume ii. Technical report, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bredford, MA (October 1972)
4. Ball, J., Carson, D.B., Evans, I., Haase, K., Jendrock, E.: The java ee 5 tutorial. Sun Microsystems, Santa Clara (2006)
5. Cook, N.: Middleware Support for Non-repudiable Business-to-Business Interactions. PhD thesis, School of Computing Science, Newcastle University (2006)
6. Debie, E., De Ryck, P.: Non-repudiation middleware for web-based architectures. MSc thesis, Katholieke Universiteit Leuven (2009)
7. DeMichiel, L., Keith, M.: Enterprise javabeans, version 3.0. Sun Microsystems (2006)
8. Erlingsson, U., Schneider, F.: Irm enforcement of java stack inspection. In: IEEE Symposium on Security and Privacy, pp. 246–255 (2000)
9. Object Management Group. Security service specification v1.8 (March 2002)
10. Object Management Group. Corba specification (January 2008), `http://www.omg.org/spec/CORBA/3.1/`
11. Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft Press, Redmond (2006)
12. Johnson, R., et al.: Spring java application framework - reference documentation (2009)
13. Koved, L., Nadalin, A., Nagaratnam, N., Pistoia, M., Shrader, T.: Security challenges for enterprise java in an e-business environment. IBM Systems Journal 40(1), 130–152 (2001)
14. Kremer, S., Markowitch, O., Zhou, J.: An intensive survey of fair non-repudiation protocols. Computer Communications 25(17), 1606–1621 (2002)
15. Linn, J.: Rfc2743: Generic security service application program interface version 2, update 1. RFC Editor United States (2000)
16. Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: Rfc2560: X. 509 internet public key infrastructure online certificate status protocol-ocsp (1999)
17. Nadalin, A., Kaler, C., Hallam-Baker, P., Monzillo, R., et al.: Web services security: Soap message security 1.0 (ws-security 2004). OASIS Standard, 200401 (2004)

18. Nenadic, A., Zhang, N., Barton, S.: Fides–a middleware e-commerce security solution. In: Proceedings of the 3rd European Conference on Information Warfare and Security, pp. 295–304 (2004)
19. Parkin, S., Ingham, D., Morgan, G.: A message oriented middleware solution enabling non-repudiation evidence generation for reliable web services. In: Malek, M., Reitenspieß, M., van Moorsel, A. (eds.) ISAS 2007. LNCS, vol. 4526, pp. 9–19. Springer, Heidelberg (2007)
20. Singh, I., Johnson, M., Stearns, B.: Designing enterprise applications with the J2EE platform. Addison-Wesley Professional, Reading (2002)
21. Tribble, D.A.: The health insurance portability and accountability act: security and privacy requirements. American Journal of Health-System Pharmacy 58(9), 763 (2001)
22. Wichert, M., Ingham, D., Caughey, S.: Non-repudiation evidence generation for corba using xml (1999)
23. Zhou, J., Gollmann, D.: Evidence and non-repudiation. Journal of Network and Computer Applications (1997)

# Lightweight Modeling and Analysis of Security Concepts*

Jörn Eichler

Fraunhofer Institute for Secure Information Technology SIT,
Rheinstr. 75, 64295 Darmstadt, Germany
Joern.Eichler@sit.fraunhofer.de

**Abstract.** Modeling results from risk assessment and the selection of safeguards is an important activity in information security management. Many approaches for this activity focus on an organizational perspective, are embedded in heavyweight processes and tooling and require extensive preliminaries. We propose a lightweight approach introducing SeCoML – a readable language on top of an established methodology within an open framework. Utilizing standard tooling for creation, management and analysis of SeCoML models our approach supports security engineering and integrates well in different environments. Also, we report on early experiences of the language's use.

**Keywords:** Risk Assessment, Information Security Management, Security Engineering, DSML.

## 1 Introduction

Flexibility and adaptability are key drivers for success of today's enterprises. Therefore smart and tailored processes and applications are crucial, especially for small and medium sized enterprises (SME) [22]. The execution of information technology (IT) related projects to develop or adapt applications to the needs of the organization, the integration of applications to better support business processes, and the adaptation of IT-supported business processes to changing needs in the market is a recurring task in order to provide the necessary infrastructure. Those projects are confronted with scarce resources, especially on expert knowledge outside the organization's core competences. Hence, security engineering activities have to be addressed on the basis of restricted knowledge, delivering quick results and a flexible integration in existing tool chains and processes [4].

A vital part of the security engineering activities in these projects is the assessment and treatment of IT security risks [2]. To conduct risk assessment and risk treatment IT security related information about the environment is necessary. Security models created to support information security management systems (ISMS) can provide valuable information [15]. To distinguish security models for that purpose from others the term security concept is common and will be used in the following [8]. Security

* The work presented in this paper was partly developed in the context of the project Alliance Digital Product Flow (ADiWa) that is funded by the German Federal Ministry of Education and Research. Support code: 01IA08006F.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 128–141, 2011.

concepts capture corporate assets, their dependencies and protection requirements, threats to the assets, as well as necessary and implemented safeguards to protect them. Using security concepts, environmental protection requirements and design constraints for new or adapted applications based on consolidated information can be introduced early in the security engineering process. Furthermore, impacts on the environment by the applications to be developed or adapted can be analyzed, and different design alternatives can be evaluated.

The development and integration of a new service into the organization's environment to exchange data between business partners might serve as example. The new service has to meet different security requirements depending e.g. on the protection requirements of the data to be communicated as well as those of the applications and business processes utilizing this service. Furthermore the integration of the new service affects the existing infrastructure as new requirements for systems or communication nodes supporting the new service might be imposed.

A variety of methods and tools for risk assessment in the context of ISMS have been presented focusing on different aspects of risk assessment or targeting different environments [14]. Unfortunately only very few methodologies are suitable for the use in SME environments. Moreover, candidate methodologies identified in the following sections focus on an organizational perspective and bring only general or heavyweight tooling to support users documenting their security concept and analyzing it.

We propose a lightweight approach for modeling security concepts introducing the Security Concept Modeling Language (SeCoML). SeCoML is a domain specific modeling language (DSML) with a readable textual syntax. It is based on the IT Baseline Protection Methodology (IT-BPM, also known as IT Grundschutz [8]), a well known methodology in SME environments conformant to accepted international standards. SeCoML allows for an easy creation, validation, and analysis of security concepts. Its modularity provides support for reusability of models, incremental creation of security concepts, and adaptability. Applying current frameworks for model driven software development, state of the art tooling is provided to use SeCoML effectively, and to integrate it in existing tool chains.

The rest of this paper is structured as follows. Section 2 identifies requirements, selects an appropriate information security management (ISM) approach, provides some background on IT-BPM, and discusses related work. SeCoML and corresponding tooling is presented in section 3. In section 4, we report early experiences using SeCoML in SME environments. We summarize our results in section 5 and discuss further research topics.

## 2 Requirements, Background, and Related Work

In this section, at first we identify basic requirements considering a lightweight approach to model security concepts. Then, IT-BPM as underlying ISM approach for SeCoML is selected and some background on IT-BPM is provided. The section closes with a discussion of related work.

### 2.1 Basic Requirements

To allow for a beneficial use of security concepts in integration and adaption projects of SMEs we identified the following basic requirements for a lightweight modeling and analysis approach:

(R1) The security concept needs an established methodology as foundation that is appropriate for the use in the targeted environment.
(R2) At least a semi-formal modeling of the security concept must be possible.
(R3) Incremental creation, refinement, and analysis of security concepts as well as modular partitioning of security concepts should be supported.
(R4) (Technical) assistance in the creation of valid security concepts should be available.
(R5) Security concepts and tooling must integrate in existing tool chains and processes without extensive preliminaries.

Generally security concepts are created and maintained in the course of the initiation and operation of an ISMS. A modeling approach should therefore build upon an established methodology for the targeted environment to become feasible and accepted (R1). Furthermore only an at least semi-formal modeling of security concepts allows for a precise common understanding and (semi-) automated validation and analysis [5] (R2). Usually security concepts are created in multiple (incremental) steps involving different stakeholders. In the lifetime of an ISMS security concepts are refined for different purposes (reporting, auditing, analysis etc.) [8]. Initially only core assets are included into the security concept and other assets, additional threats etc. are refined on demand. Parts of the security concepts might be reused to reflect organizational subdivisions and subsidiaries. Incremental creation, refinement, and modular partitioning are also important to support differing scenarios (e.g. to analyze design alternatives in integration and adaption projects) (R3).

To leverage security concepts the creation, manipulation, and validation should be assisted by respective tooling targeting not only security experts [29] (R4). To allow for a lightweight approach modeling artifacts as well as corresponding tooling must be easy to integrate into existing tool chains (e.g. development tool chains including application lifecycle management, document management, management information systems for security certification and ISMS operation), and independent of individual tooling or heavyweight processes (e.g. requiring multiple stakeholders to participate in every project and several activities or presupposing large documentation) (R5).

### 2.2 Information Security Management

As security breaches based on IT issues gain media interest, more and more organizations are introducing internal initiatives to improve their protection of their IT infrastructure, processed data, and other IT-related assets. One keyword accompanying those initiatives is ISM. Understanding information security (IS) as preservation of confidentiality, integrity, and availability of information as well as other properties as authenticity and accountability, information security management refers to the process of the implementation and ongoing management of IS in an organization [19].

Targeting not only the (cost efficient) improvement of their IS but also indications of trust that consumers or partners can have in the organization's IS, standards and corresponding certifications are becoming an important foundation of the initiatives. Important international standards in the field of ISM are:

- ISO/IEC 13335-1 [18] is a general guide for initiating and implementing the IT security management process focusing on concepts and models of IT security. Other parts of the current ISO/IEC 13335 series describes techniques for IT security risk management and guidance for network security.
- ISO/IEC 27001 [19] provides requirements for an ISMS. An ISMS is part of the overall management system to establish, implement, operate, monitor, review, maintain and improve IS. ISO/IEC 27001 is one of the very few international ISM standards that allow for certification. ISO/IEC 27001 is part of a series that includes also complementing standards on risk management, metrics, measurement, and implementation guidance.

Analyzing risk assessment and management methodologies best suited for SMEs in Europe, the European Network and Information Security Agency (ENISA) identified six methodologies [14]: the Austrian IT Security Handbook [9], the Dutch A&K Analysis [24], EBIOS [12], IT-BPM, MEHARI [11], and OCTAVE-S [1]. We used ENISA's report to select from the methodologies those that fulfill the requirements R1 and R5 and therefore

- provide special information for SMEs,
- are compliant to international standards listed above,
- provide interfaces to other organizational processes, and
- are applicable without consultancy support.

Only the Austrian IT Security Handbook and IT-BPM comply with all criteria. MEHARI and the A&K Analysis for example lack interfaces to other organizational processes, OCTAVE-S is not compliant to international standards, and Ebios requires consultancy support. IT-BPM is widely used not only in Germany but also in Austria, Switzerland, and other countries, documentation is also given in English, and technical guidelines covering various application domains (e.g. [3]) as well as examples including security concepts for current developments and real world applications are publicly available (e.g. [16]). We therefore decided to use the IT-BPM as basis for our approach.

### 2.3 IT Baseline Protection Methodology

The ISO standards explicate ISM and ISMS generically. IT-BPM tries to bridge the gap between those generic descriptions and a practical implementation. It provides detailed guidance for an organization for establishing an ISMS compliant with ISO/IEC 27001.

The IT-BPM security process comprises four phases: process initiation, creation of the security concept, implementation of the security concept and maintenance and improvement of the security concept (cf. Figure 1). The creation of the security concept encompasses six steps. The structure analysis as first step details interdependencies between business processes, applications, and existing IT infrastructure. For

those items the protection requirements are determined using an ordinal rating scale and – based on the IT-BPM catalogues – appropriate safeguards are selected and adapted. As IT-BPM uses an (extended) baseline approach the following basic check generates a target/actual comparison. If the protection requirements are exceeding the normal rating or elements of the organization's structure are not covered by the catalogues a supplementary risk analysis is executed to assess the risks and integrate corresponding safeguards in the security concept.

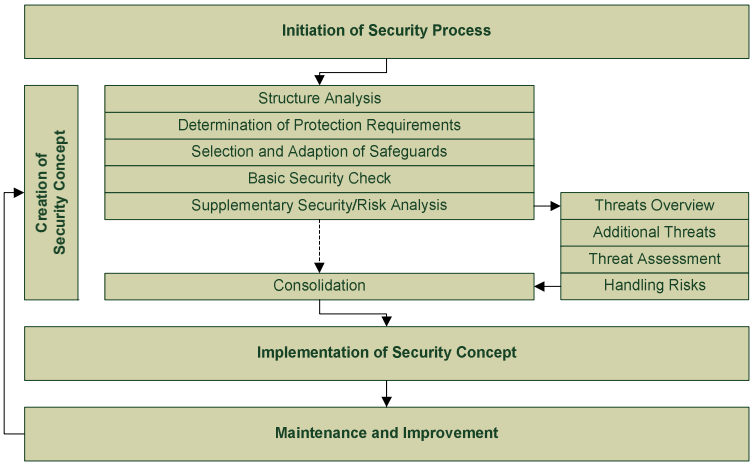



**Fig. 1.** Security process of the IT-BPM [8]

To implement the IT-BPM most organizations use the checklists and forms of the IT-BPM directly (using word processors and spreadsheets) or deploy a dedicated ISMS tool that supports the IT-BPM. An approach that offers (semi-) formal modeling and fulfills the requirements given in section 2.1 is not available.

### 2.4 Related Work

Beside the risk assessment and ISM methodologies discussed previously, several model- and/or modeling language-based approaches have been presented recently. Zambon et al. present in [31] a model-based risk assessment approach for IT infrastructures. Introducing QualTD they suggest a time dependency model and techniques to analyze risks on availability and the propagation of incidents in an IT architecture. Confidentiality or integrity are not considered in QualTD models. In [7] den Braber et al. present CORAS, a method to conduct security risk analysis that uses a customized graphical DSML based on UML. CORAS follows the AS/NZS 4360:2004 standard [28] that does not offer special support for the targeted environment. Other authors propose further modeling approaches and tooling in the area of risk assessment but do not use a methodology appropriate for the targeted environment or do not provide support for incremental creation or modular partitioning of their models [13, 23, 25].

Further model-driven approaches highlight the gap between security models, system design models and implementation. The focus is on the generation of implementation artifacts from security (enriched) models providing modeling languages and

transformations. Prominent in this domain is SecureUML presented by Basin et al. [6] for access control modeling and infrastructure generation. Wolter et al. [30] derive security policies from business process models with security annotations; Rodriguez et al. use analogue models to derive design models [27]. Creation and analysis of security models using DSMLs and OCL is presented also in [5], targeting access control. UMLsec presented by Juerjens [20] uses extensions to UML to include security relevant information in design models and to analyze security properties of those models. In [17] UMLsec is integrated with the heuristic requirements editor HeRA and the security standard ISO 14508 Common Criteria to comprise with SecReq a security engineering methodology to elicit security requirements and trace them to design models. Together the approach provides valuable insights and guidance for security engineering but is rather heavyweight and directed to security professionals.

The aim to support flexible and adaptable applications and processes touches the agenda of Agile development and Agile security engineering (e.g. [10]). The special requirements of the targeted environment with regard to security engineering are also analyzed by Bartsch et al. in [4] addressing authorization rules for SME applications providing a dedicated DSML and a corresponding enforcement implementation.

## 3 Modeling Security Concepts with SeCoML

To support a lightweight approach to model and analyze security concepts based on IT-BPM in SME environments we developed SeCoML. In the following we will present the modeling language SeCoML, describe how to analyze security concepts modeled using SeCoML in the course of development, adaption, and integration projects, and sketch the tooling that we implemented for an effective use of SeCoML.

### 3.1 The Modeling Language

The design of SeCoML was guided methodically by the software language engineering approach from Kleppe in [21]. We present SeCoML describing the abstract syntax model (provided as Ecore[1] metamodel) first. A short summary of properties of the concrete syntax model and the syntax mapping of SeCoML follows. Examples of security concepts modeled using SeCoML are given in section 3.3. Informal descriptions detailing the semantics are given in [8]. As formal semantics for IT-BPM security concepts are not defined we documented our understanding formulating constraints for the metamodel using OCL [26].

The concepts necessary to model an IT-BPM security concept are reflected in the metamodel of SeCoML. Figure 2 shows an overview of the core concepts and their relations. The following paragraphs will give a short introduction to the metamodel following the security process of IT-BPM (cf. the phase “Creation of Security Concept” in Figure 1).

The structure analysis is the first step of the creation of a security concept. The `Asset` and its relation `supports` in the metamodel capture the results of the structure analysis (i.e. the organization’s assets and their interdependencies considering protection requirements).

[1] `http://www.eclipse.org/modeling/emf/`

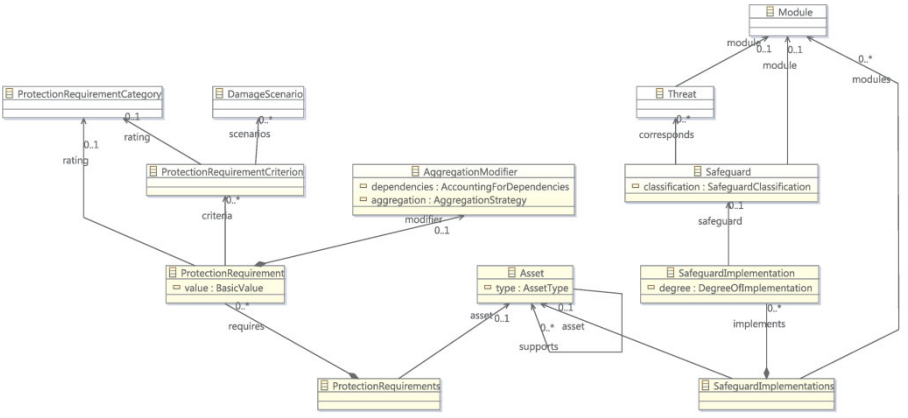



**Fig. 2.** Core metamodel of SeCoML

In the next step of the security process, protection requirements (`ProtectionRequirements`) considering the basic values confidentiality, integrity, and availability are analyzed and documented for each asset. Protection requirements are rated using an ordinal rating scale from normal to very high (`ProtectionRequirementCategory`). The ratings are based on applicable criteria (`ProtectionRequirementCriterion`) depicted in damage scenarios (`DamageScenario`). The derivation of protection requirement ratings follows aggregation strategies given by IT-BPM (`AggregationStrategy`). In the normal case the rating is derived from the highest rating of applicable criteria and dependent assets' ratings (called "maximum principle" and "accounting for dependencies"). Alternative aggregation strategies can be chosen for each protection requirement to cover risk accumulation or distribution increasing or decreasing derived ratings. Also, protection requirements can be rated independently from dependent assets (`AggregationModifier`).

Appropriate safeguards to meet the protection requirements are selected in the following step of the security process. Therefore, modules from the IT-BPM catalogues are assigned to assets (`Module`). Modules cover consolidated threat scenarios and recommended safeguards for various components, procedures, and IT systems (`Threat`, `Safeguard`). Safeguards are tagged to indicate their importance with regard to later certifications of the organization's ISMS (`SafeguardClassification`).

To perform a basic security check the implementation status for each safeguard is evaluated (`SafeguardImplementation`). The degree of implementation is recorded using one of the following statuses: `Unnecessary`, `Yes`, `Partially`, and `No`. After this evaluation a target/actual comparison for the aspired classification level can be derived and a corresponding implementation plan can be developed.

One distinguishing property of SeCoML is the support for the derivation of protection requirement ratings following the different aggregation strategies of the IT-BPM. Ratings are derived using the "maximum" and "accounting for dependencies" principle per default but consider also deviating aggregation strategies. "Cumulation" increases, "distribution" decreases the derived rating. The following example demonstrates how the semantics are defined in the metamodel:

```
context ProtectionRequirement::maxRatingCriteria() : ProtectionRequirementCategory
  body: if criteria->notEmpty() then criteria.rating->sortedBy(ordinal)->first()
  else rating endif
context ProtectionRequirement::maxRatingSA() : ProtectionRequirementCategory
  body: requirements.asset.supports.requires.requires->select(value=self.value)
  .criteria.rating->sortedBy(ordinal)->first()
context ProtectionRequirement::maxDerivedRating() : ProtectionRequirementCategory
  body: if modifier.dependencies=AccountingForDependencies::NoDependencies
  then maxRatingCriteria()else maxRatingCriteria()->union(maxRatingSA()->
  asSet())->sortedBy(ordinal)->first() endif
context ProtectionRequirement::derivedRating : ProtectionRequirementCategory
  derive: if modifier.aggregation=AggregationStrategy::Cumulation
  then maxDerivedRating().higher()
  else if modifier.aggregation=AggregationStrategy::Distribution
  then maxDerivedRating().lower()else maxDerivedRating() endif endif
context ProtectionRequirements
```

The textual syntax of SeCoML has been defined using a domain specific language provided by Xtext[2] resembling the Extended Backus-Naur Form. A concept for namespaces has been included to support recurring names in different packages. To allow for better reusability, modularization, and separation of concerns models can be split using multiple resources (e.g. files or databases). The metamodel respects the incremental nature of results of different phases of the security process. One security concept can be split into several resources, each resource can potentially be used in several security concepts (e.g. reusing the threat and safeguard catalogues). Examples of SeCoML resources are presented in Figure 3.

In comparison with graphical approaches the textual syntax of SeCoML has the main advantages that models can be manipulated, searched, compared, and put under version control using efficient and commonly used tools and techniques. Furthermore it is impossible to break the model in such a way that it cannot be reopened with the delivered editors (and other editors as well), and that they can be fixed using the same tools if the metamodel or the syntax is adapted in future versions. It thus fits very well to our lightweight approach and frequently changing environments.

### 3.2 Analysis of Security Concepts

An obvious aim of the analysis of security concepts is to support mandatory steps in the security process (e.g. checking whether all assets are analyzed concerning their protection requirements and safeguard implementations, or the generation of target/actual comparisons to develop implementation plans etc.).

Not that obvious are questions concerning dependencies within the security concept (e.g. dependencies of assets' protection requirement ratings and the influence of given rating criteria or the choice of aggregation strategies). The evaluation of design or implementation alternatives requires answers to these questions. We provide with SeCoML a lightweight basis to analyze those questions.

The following paragraphs introduce a short example security concept and exemplify several analysis operations. Generally, security concepts entail much more elements and therefore underline the necessity for corresponding analysis support (e.g. more than 3000 threats and safeguards are given in the IT-BPM catalogues, more than 25 criteria are recommended as starting point).

[2] http://www.eclipse.org/Xtext/

**Table 1.** Example security concept (assets, requirements and safeguard implementation)

| Asset | Supports | Requires | Implements |
|---|---|---|---|
| A1: Production | | R1: C:N (C2, C3), R2: I:N (C2), R3: A:H (C1) | S1:Y, S3:Y |
| A2: Server | A1 | R4: C:N, R5: I:N, R6: A:N:Dis | S1:Y, S2:U, S3:Y |
| A3: Communication | A2 | R7: C:N, R8: I:N, R9: A:N | S1:Y, S2:Y, S3:Y |

Table 1 shows a very small example security concept. All elements have an identifier (given before the colon). The basic values are abbreviated using their initial character (Confidentiality, Integrity, Availability) as well as the rating (Normal, High, Very high). In brackets aggregation strategies (Acc: Accumulation, Dis: Distribution) as well as criteria (Cx) are given to substantiate the rating. Safeguards (Sx) are given with the degree of implementation (Y: Yes, N: No, P: Partly, U: Unnecessary). All safeguards are from module M1 and cover the following threats: S1 and S2 threat T1, S2 and S3 threat T2, both from module M1 as well. The criteria C2 and C3 apply for assets with normal protection requirements, C1 applies for those with high protection requirements.

In the example asset A1 (a production application) requires normal protection with regard to confidentiality following from criteria C2 and C3 (requirement R1) but high protection with regard to availability following from criteria C1. It implements safeguards S1 and S3. The corresponding server A2 runs the production application A1, its protection requirements derive from the supported assets (A1). The protection requirement rating considering availability is reduced because of the risk distribution aggregation strategy (R6).

In addition to the validations based on the constraints of the metamodel analysis, operations can be used to answer questions within the evaluation of security concepts. The following examples show some of the common queries in the analysis of security concepts that are provided with SeCoML. Ad-hoc queries can utilize these operations as well to further analyze the security concept.

- Which asset lack consideration for a given safeguard depending on the chosen modules for that asset? The operation `Safeguard::getMissingSafeguardImplementations` returns the implementation of {A1} since the security concept does not consider safeguard {S2}.
- If a given safeguard fails, are there assets that will be unprotected with regard to a threat (i.e. implement no other safeguard that covers that threat)? In our example the failure of S1 results in the assets {A1, A2} that are not protected with regard to threat {T1} (`Safeguard::unprotectedAssetsOnFail()`).
- Which protection requirements are affected (directly and indirectly) by altering a given protection requirement criterion definition, leading possibly to changed protection requirement ratings? Changing criterion C2 affects {R1, R2, R4, R5, R7, R8} in our example (`ProtectionRequirementCriterion::affectedRequirements()`).

### 3.3 Implementation and Integration in the Tool Chain

To use SeCoML efficiently, we implemented an editor for security concepts formulated in SeCoML and integrated the editor with application lifecycle management tooling (versioning, branching, tagging and merging based on Apache Subversion[3], lightweight project management based on Edgewall Software Trac[4] and Eclipse Mylyn[5]) as well as an analysis console (using Interactive OCL from Eclipse MDT[6]) together in a security workbench as Eclipse application.

The core component – the SeCoML editor – was implemented using Eclipse Modeling Framework[7] (EMF) and Xtext[8]. We customized several parts of the editor including labeling and filtering of entities in outlines and content assist, scoping for content assist, formatting of the concrete syntax, validation, quick fixes, and templates to insert new entities quickly. Figure 3 shows a screenshot of the workbench.

The upper part (1) shows the textual editor for SeCoML demonstrating content assist (pop-up showing possible basic values) and constraint-based validation (underlined text). For easy navigation in textual representations of the security concept the outline view can be used (4). Another view (2) presents an alternative, synchronized tree editor for the same resource that is analyzed using an interactive console (3).

To integrate the workbench further more in the existing tool chain we generated and adopted transformations using EMF and XSL[9] to acquire information given e.g. in Microsoft Excel files (excerpts from a configuration management database) and transform it into SeCoML and back (e.g. for target/actual comparisons).

Considering the requirements (cf. section 2.1), SeCoML is able to meet all of them: With IT-BPM an established methodology that is appropriate for the use in the targeted environment is used as foundation (R1). The definition of SeCoML comprising an Ecore metamodel enriched with OCL constraints, a corresponding concrete syntax model and mapping model provides solid ground for semi-formal modeling of security concepts (R2). Incremental creation and refinement as well as modular partitioning of security concepts is supported by SeCoML given the modular metamodel, the support for namespaces, and the possibility to split security concepts into multiple resources (R3). Delivering state of the art editors for SeCoML offers assistance in the creation and validation of security concepts (R4). The combination of the modular metamodel, the corresponding textual syntax and the implementation of accompanying tooling on the basis of an open and well established (model driven software development) framework allows for an easy integration in existing tool chains without extensive preliminaries (R5).

---

[3] http://subversion.apache.org/
[4] http://trac.edgewall.org/
[5] http://www.eclipse.org/mylyn/
[6] http://www.eclipse.org/modeling/mdt/?project=ocl
[7] http://www.eclipse.org/modeling/emf/
[8] http://www.eclipse.org/Xtext/
[9] http://www.w3.org/Style/XSL/

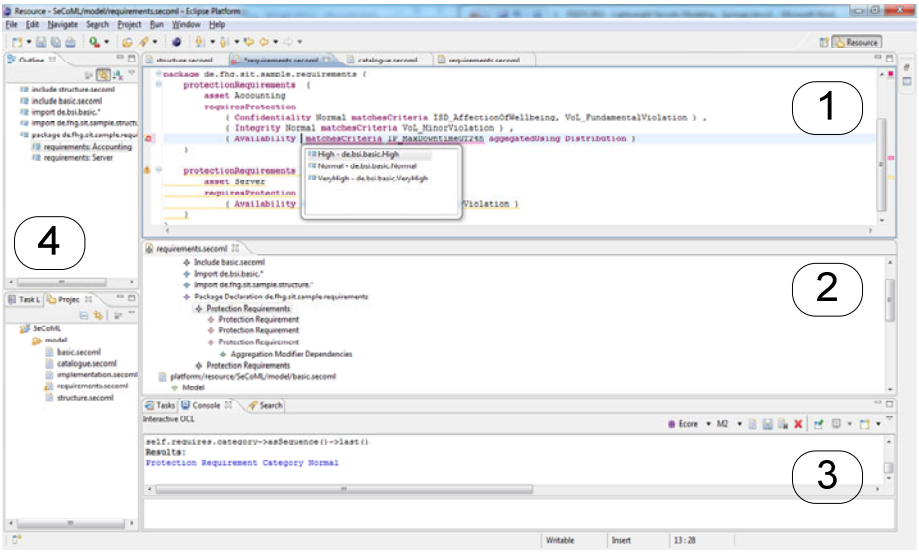



**Fig. 3.** Editor and analysis workbench for SeCoML

## 4 Early Experience with SeCoML

In order to evaluate SeCoML and its corresponding tooling with respect to its use in the targeted environments, we employed SeCoML in projects with SMEs. In the latest case the SME executed a project to integrate a new communication service to exchange production data with business partners. Therefore the existing security concept of the information security management system should be validated and different solutions for the service integration should be analyzed and presented.

In that project we used SeCoML mainly together with the security officer of the SME, a project manager and several domain experts (for IT infrastructure, operations and the production application). The security officer as well as domain experts for the IT infrastructure used SeCoML regularly as end users. Based on the security concept modeled in SeCoML we discussed alternatives to deploy the new communication service and integrated the chosen alternative in the security concept. Additionally we conducted interviews with the participants to capture their subjective views of the feasibility of the application of SeCoML.

In the course of the project SeCoML proved to be very helpful. Several errors and inconsistencies in the existing security concept were identified. Most errors had been induced by the wrong application of aggregation strategies for protection requirement ratings but also protection requirement criteria had been applied inconsistently. The use of a known methodology and terminology as basis for SeCoML helped to work with SeCoML very quickly. Additionally, the ad-hoc analysis gave solid ground to the discussions about implementation alternatives and the propagation of protection requirement ratings. Providing the security workbench allowed for an easy integration in the existing tool chain (e.g. versioning security concepts for different scenarios along with the sources in the repository).

On the downside the application of SeCoML in larger enterprises had been questioned as the textual approach does not allow for a fine grained access control to elements of the security concept. Also, the appearance of the security workbench attracts people with background as software developer more. For long-term maintenance of the security concept the stakeholders voted for an additional form-based user interface.

Within the project executed, the given environment, and the intended use the participants rated SeCoML as a very helpful approach to improve the projects execution and result.

## 5 Summary and Outlook

With SeCoML we contribute a lightweight approach to model security concepts on top of an established methodology appropriate for the environment of SMEs. Providing with SeCoML a textual DSML for security concepts establishes a solid foundation for security concept modeling and analysis. The modular design of the metamodel in combination with corresponding properties of the textual syntax supports the lightweight approach. SeCoML integrates well in existing tool chains and processes and delivers state of the art tooling for the creation, validation and analysis of security concepts. Therefore, SeCoML leverages the use of security concepts in the course of development, adaption, and integration of applications and supports corresponding security engineering activities. Early experiences underline the suitability of SeCoML for the use in SME environments.

Further research will analyze the use of additional models to support security engineering and document security engineering best practices using our lightweight approach. Also the integration and transformation of existing models to facilitate the security concept creation, modification, and validation will be examined.

## References

[1] Alberts, C., Dorofee, A., Stevens, J., Woody, C.: OCTAVE®-S implementation guide, version 1.0 (2005), http://www.sei.cmu.edu/reports/04hb003.pdf

[2] Anderson, R.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley & Sons, Chichester (2001)

[3] Bartels, C., Kelter, H., Oberweis, R., Rosenberg, B.: Technical guidelines for the secure use of RFID – application area trade logistics. Tech. Rep. TR 03126-4, Bundesamt für Sicherheit in der Informationstechnik (2009)

[4] Bartsch, S., Sohr, K., Bormann, C.: Supporting agile development of authorization rules for SME applications. In: Bertino, E., Joshi, J.B.D. (eds.) CollaborateCom 2008. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 10, pp. 461–471. Springer, Heidelberg (2009)

[5] Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information and Software Technology 51(5), 815–831 (2009)

[6] Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Transactions on Software Engineering and Methodology 15(1), 39–91 (2006)

[7] den Braber, F., Hogganvik, I., Lund, M., Stølen, K., Vraalsen, F.: Model-based security analysis in seven steps – a guided tour to the CORAS method. BT Technology Journal 25(1), 101–117 (2007)
[8] Bundesamt für Sicherheit in der Informationstechnik: BSI-Standard 100-2: IT-Grundschutz methodology (2008), https://www.bsi.bund.de/cae/servlet/contentblob/471430/publicationFile/27993/standard_100-2_e_pdf.pdf
[9] Österreich, B.: Österreichisches Informationssicherheitshandbuch (2007), http://www.a-sit.at/pdfs/OE-SIHA_I_II_V2-3_2007-05-23.pdf
[10] Chivers, H., Paige, R., Ge, X.: Agile security using an incremental security architecture. In: Baumeister, H., Marchesi, M., Holcombe, M. (eds.) XP 2005. LNCS, vol. 3556, pp. 57–65. Springer, Heidelberg (2005)
[11] Club de la Sécurité Informatique Français (CLUSIF): Méthodologie d'Analyse des Risques Informatiques et d'Optimisation par Niveau, MEHARI (2010)
[12] Direction Centrale de la Sécurité des Systèmes d'Information, Premier Ministre: Expression des Besoins et Identification des Objectifs de Sécurité (EBIOS) - Méthode de Gestion des Risques (2010), http://www.ssi.gouv.fr/IMG/pdf/EBIOS-1-GuideMethodologique-2010-01-25.pdf
[13] Ekelhart, A., Fenz, S., Neubauer, T.: AURUM: A framework for supporting information security risk management. In: Proceedings of the 42nd Hawaii International Conference on System Sciences (2009)
[14] European Network and Information Security Agency: Risk assessment and risk management methods: Information packages for small and medium sized enterprises, SMEs (2006), http://www.enisa.europa.eu/act/rm/files/deliverables/information-packages-for-small-and-medium-sized-enterprises-smes/at_download/fullReport
[15] Evans, R., Tsohou, A., Tryfonas, T., Morgan, T.: Engineering secure systems with ISO 26702 and 27001. In: 5th International Conference on System of Systems Engineering (2010)
[16] Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH: Übergreifendes Sicherheitskonzept der Telematikinfrastruktur (2008), http://www.gematik.de/upload/gematik_DS_Sicherheitskonzept_V2.4.0_4493.zip
[17] Houmb, S., Islam, S., Knauss, E., Jürjens, J., Schneider, K.: Eliciting security requirements and tracing them to design: an integration of Common Criteria, heuristics, and UMLsec. Requirements Engineering 15(1), 63–93 (2009)
[18] ISO/IEC: ISO/IEC 13335-1: Information technology – security techniques – management of information and communications technology security – part 1: Concepts and models for information and communications technology security management (2004)
[19] ISO/IEC: ISO/IEC 27001: Information technology – security techniques – information security management systems – requirements (2005)
[20] Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
[21] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, Reading (2008)
[22] Laforet, S., Tann, J.: Innovative characteristics of small manufacturing firms. Journal of Small Business and Enterprise Development 13(3), 363–380 (2006)

[23] Mayer, N., Heymans, P., Matulevicius, R.: Design of a modelling language for information system security risk management. In: Proceedings of the 1st International Conference on Research Challenges in Information Science, pp. 121–131 (2007)
[24] Ministerie van Binnenlandse Zaken en Koninkrijksrelaties: Afhankelijkheids- en kwetsbaarheidsanalyse (1996)
[25] Normand, V., Félix, E.: Toward model-based security engineering: developing a security analysis DSML. In: Proceedings of the First International Workshop on Security in Model Driven Architecture, SEC-MDA (2009)
[26] Object Management Group: Object constraint language (OCL) specification (2006), http://www.omg.org/spec/OCL/2.0/
[27] Rodríguez, A., Fernández-Medina, E., Piattini, M.: Towards CIM to PIM transformation: From secure business processes defined in BPMN to use-cases. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 408–415. Springer, Heidelberg (2007)
[28] Standards Australia/Standards New Zealand: AS/NZS 4360:2004: Risk management (2004)
[29] Talhi, C., Mouheb, D., Lima, V., Debbabi, M., Wang, L., Pourzandi, M.: Usability of security specification approaches for UML design: A survey. Journal of Object Technology 8(6), 103–122 (2009)
[30] Wolter, C., Menzel, M., Schaad, A., Miseldine, P., Meinel, C.: Model-driven business process security requirement specification. Journal of Systems Architecture 55(4), 211–223 (2009)
[31] Zambon, E., Etalle, S., Wieringa, R., Hartel, P.: Model-based qualitative risk assessment for availability of IT infrastructures. In: Software and Systems Modeling, pp. 1–28 (2010)

# A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications

Linda Ariani Gunawan, Frank Alexander Kraemer, and Peter Herrmann

Department of Telematics
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
{gunawan,kraemer,herrmann}@item.ntnu.no

**Abstract.** We describe a highly automated and tool-supported method for the correct integration of security mechanisms into distributed applications. Security functions to establish and release secure connections are provided as self-contained, collaborative building blocks specifying the behavior of several parties. For the security mechanisms to be effective, the application-specific model needs to fulfill certain behavioral properties, for instance, a consistent start and termination. We identify these properties and show how they lead to correct secured applications.

## 1 Introduction

Security is a significant aspect in the design and implementation of networked systems. Despite this fact, security is still an aspect that many developers consider as one of the last steps during system development [1]. One of the reasons is that developing secure applications needs substantial expertise [2,3]. This level of expertise may exceed what one can expect from average developers who are rather experts in their specific application domains. For security experts, on the other side, it may likewise be difficult to cope with the domain-specific applications. In addition, even when security mechanisms themselves are sufficiently understood, their integration into an application needs careful consideration in order to be effective (see, for instance, [4] for integrating TLS [5] into application layer protocols). This means that both knowledge of a specific application domain and of the appropriate security mechanisms are needed.

Since security is an aspect that spreads and entangles with many components in an application [6,7], it can be difficult to separate this aspect from the application's functional part. However, from our experience with model-driven design, analysis and refinement of distributed applications, we see that there are cases in which, given that certain preconditions hold, some security mechanisms can be effectively integrated into the system by a highly automated process which we can support by tools. The employed strategy here is three-fold:

1. We check that an initially unsecured system fulfills certain necessary structural and behavioral properties.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 142–155, 2011.

2. We automatically encapsulate parts of the specification with security mechanisms.
3. The protected specifications are integrated into the complete system model.

The preconditions address mainly functional properties and can be easily understood by domain experts and checked by tools. Due to the high degree of automation, the process of introducing the security mechanisms only requires basic knowledge. As a result, domain experts can spend most of their attention on their respective fields, while security experts may focus on the provision of general security mechanisms that can be applied to an entire class of systems.

In this paper, we present a highly automated and tool-supported approach that extends our previous method on model-driven engineering SPACE [8] with its tool-suite Arctis [9] and implements the strategy described above. As will be detailed later, the method benefits from the collaborative specification style of SPACE to model both functional and security aspects. We show that the secured applications produced by the method fulfill important security goals and hence the method correctly integrates security mechanisms into functional models.

### 1.1 Collaborative Specification Style

To specify a distributed application, we use a specification style that is based on collaborative building blocks [8]. These are the major design units which can cover both local behavior within components and interactions between them. As we will later see, this has the benefit that security mechanisms for communication, which are inherently collaborative, can be expressed by self-contained building blocks. Moreover, the specification style enables a rapid application development since, as shown in [10], on average more than 70 % of a system specification can be taken from reusable building blocks provided in various libraries [11]. The semantics of the specification is formally defined in [12] which makes it possible to guarantee important system properties, e.g., the correct usage of building blocks and the boundedness of communication, by the model checker included in Arctis [9].

As an example, we built an application for a telemedical consultation that enables patients to consult a physician by chat. Since the physician cannot determine the exact time when a patient can be served, the application contains an active waiting queue, in which the patient registers and gets informed about when the consultation begins. Figure 1 shows the specification of the application. It is a UML 2.0 activity consisting of two partitions, *patient* and *physician*, which denote the two participating entities for this system. The activity is composed from six building blocks that refer to subordinate activity diagrams (ignore for now the one labelled *Secure Chat*). The blocks have pins on their frames that are used to compose their behavior. A starting pin (see the legend in Fig. 1) is used to start a block which is only allowed when it is idle. Once a block is started, data can be passed in either direction via streaming pins. Termination of a block is modeled by terminating pins. The blocks *u1: Queue UI* and *u2: Chat UI* on the patient side encapsulate the user interfaces for the queueing function and the

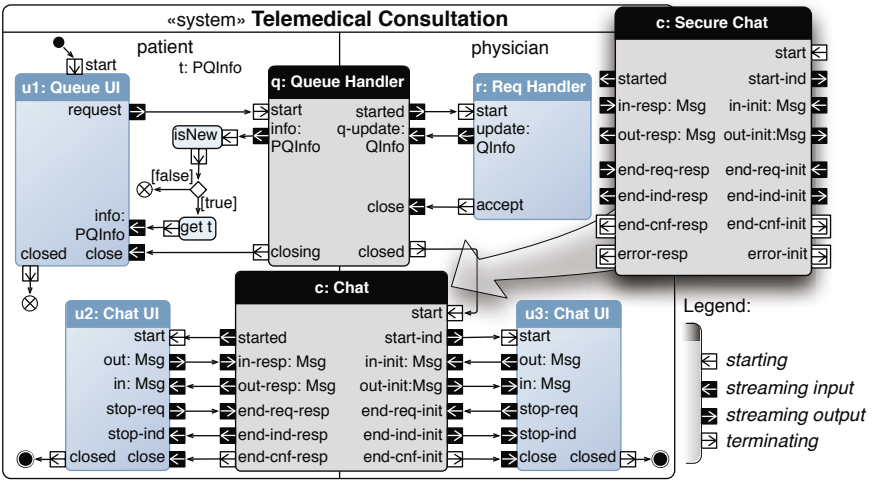



**Fig. 1.** Medical Consultation System

actual chat. The physician part has similar blocks, i.e., *r* and *u3*. The building blocks *q: Queue Handler* and *c: Chat* are assigned to both participants. They are collaborations, and their task is to manage interactions between the patient and the physician. The protocol to handle the queueing is encapsulated in *q: Queue Handler*, while *c: Chat* contains the necessary interactions for the chat.

The application is initiated on the patient side with the initial node (•) that starts *u1: Queue UI*. Then, via this UI, the patient can issue a request to get medical consultation, which is expressed by a flow from the streaming output *request* to the starting pin of *q: Queue Handler*. A signal carrying this request is sent to the physician part which further starts block *r: Req Handler*. Thereafter, the queue information in this block is appended, encapsulated in an *QInfo* object, and sent via pin *update*. Block *q* receives this, extracts necessary data, and forwards the data via streaming output *info*. This information is futher displayed to the patient. Updates on the queue information are also sent periodically in the same manner. However, an update is forwarded to the patient UI only if it is new, as denoted by operation *isNew* which contains a corresponding Java method and the two choices following a decision node (⋄). When it is the patient's turn to be served, the block *r* emits an event *accept* and terminates. This event triggers the closing of blocks *u1* and *q*. Then, the *c: Chat* block is started.

As depicted in Fig. 1, the start of the chat collaboration also initiates the related UIs. Thereafter, both participants can send and receive messages at will via pins *in* and *out* on their respective UIs. This message exchange is governed by block *c: Chat* as shown by the flows through the corresponding pins. One side may decide at any time to terminate the chat, which is also managed by the chat block. To handle the termination consistently, it declares its intention via either pin *end-req-resp* or *end-req-init*, whereupon the other side receives an indication. Any remaining chat messages that are obtained by the block before

the indication are sent. Thereafter, the patient and subsequently the physician sides of the chat collaboration are closed. This also accounts for a situation in which both sides decide to terminate at the same time.

### 1.2 Security Goals

In the following, we focus on the protection of signals carrying sensitive or private information exchanged between two entities. Our method to integrate this protection into a distributed application such as the telemedical consultation in Fig. 1 ensures the fulfillment of three security goals as motivated below.

Due to the computation and resource penalty from executing cryptographic operations (like encryption or digital signature generation), the secure connection is generally only used as long as required. Therefore, in the life span of a distributed application, there may exist unsecure and secure phases. It is important to ensure all signals that require protection are only transferred in a secure mode. We formulate this goal as follows:

**G1** While in secure mode, all signals are transferred with protection. In other words, during the secure mode either a correct, secured transmission of flows occurs or an attack is detected.

To avoid vulnerabilities due to the interference of application-specific communication with the security functions, no application signals can be transferred during secure mode establishment and termination. Therefore, our method realizes the following goal:

**G2** The four phases, namely *unsecure mode*, *secure mode establishment*, *secure mode* and *secure mode termination*, are all distinct, i.e., signal transmissions belonging to different phases do not interleave.

As shown in [13,14,15], combining several secured security mechanisms can potentially lead to vulnerabilities. To avoid this problem, the applications obtained by using our method must not contain duplication of security mechanisms. We formulate this last goal as follows:

**G3** The protected system specifications do not contain duplication of security mechanisms.

### 1.3 Overview of the Method

Figure 2 illustrates how our method transforms a functionally correct but unprotected system specification through three systematic steps into a protected one that fulfills the goals listed above. As a first step, the specification is assessed in order to determine which collaborations need to be protected. Risk-based assessment like in [16] can be applied here. This step is performed manually since the information to be protected is different in each application. Thereafter, using our analysis tools, both the system specification and the to-be-secured collaborations are checked for the fulfillment of properties which we will later discuss in detail.

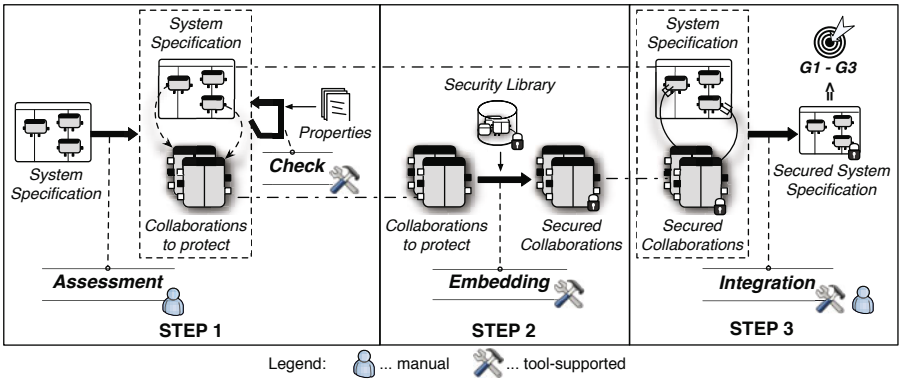



**Fig. 2.** Security Integration Method with Tool-Support

When all the properties are fulfilled, the next step is to embed security mechanisms into the collaborations to protect. These mechanisms that are also modeled as reusable collaborative building blocks are taken from a security library. This step is highly automated utilizing graph transformation techniques and produces, for each block to protect, a secured one that integrates the security blocks with the corresponding original block.

The last step to obtain a protected system specification is to integrate secured collaborations into the unprotected blocks. This step includes replacing the collaborations to be secured with their corresponding protected collaborations as illustrated by the *Secure Chat* block in Fig. 1. The substitution process is also highly automated. However, a manual inspection may still be needed since the new block contains additional pins as will be described further in Sect. 3.3.

In the following, we will describe the dedicated building blocks used to protect collaborations in Sect. 2. Thereafter, in Sect. 3 we detail the integration method illustrated in Fig. 2 and apply it to our telemedical consultation example. The discussion in Sect. 4 provides a proof-sketch that shows how specifications treated by our method fulfill the security goals stated above. We end with a discussion of related approaches in Sect. 5 and concluding remarks in Sect. 6.

## 2 Building Blocks for Secure Connections

As shown in Fig. 2, security functions are integrated by dedicated blocks. These blocks require certain mechanisms integrated into the underlying runtime-support system responsible for executing components, which we describe first.

### 2.1 Preparing the Runtime Support-System

To execute an application, our tool Arctis [9] automatically generates a software component for each partition of a system specification [17,18]. As an example, the system in Fig. 1 is realized by one component for the patient and one for the

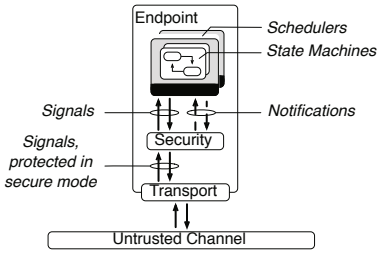



**Fig. 3.** An Endpoint

physician. Each component consists of four parts, namely state machines, schedulers, a security module, and a transport module, as depicted in Fig. 3. *State Machines* are automatically generated from the collaboration-oriented models and contain the application-specific execution logic in the form of states and transitions. Every transition is triggered by an event dispatched by the *Schedulers*. The *Transport Module* simply sends and receives signals using the underlying channels and performs the necessary serialization of data.

To protect signals conveyed by the untrusted channel, we use the *Security Module* between the schedulers and the transporter. This module handles the establishment and termination of secure modes with other endpoints. When operating in a secure mode, all signals between any state machine handled by a pair of endpoints are protected by encryption and integrity measures. We employ symmetric encryption and keyed message authentication code which use generated keys derived from a shared secret that is negotiated during a secure mode creation. SSLEngine [19], a transport-independent Java implementation of TLS [5], is used to provide this security feature.

In order to correctly apply the protection, the security module needs to communicate with the application logic through the following two mechanisms:

1. The application may obtain a handle to the security module and invoke particular methods.
2. The security module may send a notification about an occurrence of a certain event to the application. This notification is sent by using an internal signal to a specified state machine via its corresponding scheduler.

### 2.2 Building Block for the Secure Mode Establishment

The establishment of a secure mode (SM) between two components needs the cooperation of the security modules in both endpoints, which is why it is encapsulated as the collaboration shown in Fig. 4. After the block is activated, the initiator gets the required parameters which are encapsulated in an *SMParam* object and obtained via the block *m1: Key Manager*.[1] Next, the operation *prepare* is invoked in which the parameters are passed to the corresponding security module as

[1] This block manages the public/private keys of an entity and thus is part of a public key infrastructure. For brevity, this block is not discussed further in this paper.

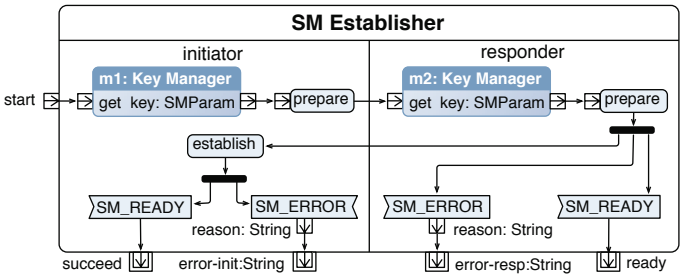



**Fig. 4.** Secure Mode Establisher Block

described by the first mechanism of communication explained in Sect. 2.1. Subsequently, the responder also obtains and sets its parameters, and is then ready to receive a notification from its own security module using the second communication mechanism. This notification is either *SM_READY* or *SM_ERROR*, indicating a successful respective failed attempt. Since these events are mutually exclusive, the terminating pins following these nodes are shown with additional boxes. After the preparation, the initiator starts the setup via operation *establish* that obtains a handler of its security module and invokes a provided method in the module. Thereafter, it is ready to receive a signal communicating the result.

The outcome of the establishment is communicated consistently by the security modules of both participants to their respective application partitions. If the secure connection is successfully established, the responder and subsequently the initiator get signal *SM_READY*. When the security module on the initiator side detects an error, the initiator and later the responder get *SM_ERROR*. Conversely, the responder and thereafter the initiator receive notifications if an error is detected by the responder's security module.

The establishment process implements the TLS Handshake protocol [5]. During the handshake, both communicating entities are mutually authenticated by means of a public key certification mechanism. Moreover, a shared secret is negotiated, from which the symmetric keys are generated. The TLS protocol guarantees that the negotiation is secure and reliable, i.e., the shared secret is only accessible to the participants and a modification in the messages is detected.

### 2.3 Building Block for the Secure Mode Termination

Termination of an SM must be handled carefully in order to thwart a truncation attack [5]. In our method, this process is executed by a pair of security modules implementing the TLS Close Alert protocol [5], and encapsulated in block *SM_Terminator*. Similarly to the establisher block, the terminator block is also a collaboration between an initiator and a responder. However, it does not necessarily mean that an entity that takes the initiator role for the setup must also be the initiator of termination, since these roles can be assigned independently. The details of block *SM Terminator* is not shown here as it resembles the successful case of the establishment block. It is worth pointing out that this termination

does not necessarily stop the transport module. Depending on its specification, the application may continue to communicate in the unprotected mode.

### 2.4 Building Block for the Secure Mode Error Listener

A security exception, such as a failed integrity check, may occur when transferring protected signals. If a security module of an endpoint detects this, it discards subsequent messages, sends an error alert to its peer, and informs the related application. Thus, the application must be prepared to receive this notification. We provide the *SM Err Listener* block (see Fig. 5) to implement this functionality. Upon activation, the block is ready to listen for a notification until it is stopped or a security exception does take place.

## 3 Integration of the Security Mechanisms

As outlined in Sect. 1.3, the integration of the security mechanisms is a process of three steps, which we will describe below.

### 3.1 Step 1: Risk Assessment and Check of Preconditions

First, a risk-based assessment is performed in order to determine which collaborations need protection. For the system in Fig. 1, block *t: Queue Handler* does not need to be secured due to the low value of information contained in the block. In contrast, collaboration *c: Chat* may transfer private information, e.g., medical records, and thus must be protected.

Next, the system level specification and the collaborations to be secured are checked for some properties. In order to apply our method of integrating security blocks in a highly automated way, we require that the system level specification fulfills the following properties:

- **S1** The system level does not directly contain any flows that cross partitions, i.e., all communication is encapsulated by collaborations.
- **S2** All collaborations have exactly two participants.
- **S3** While a collaboration to be secured is active, no other collaboration between the same pair of participants may be active as well.

Properties **S1** and **S2** are structural and can be checked by the syntactic inspection tool in Arctis, while the behavioral property **S3** is ensured by model checking. Our experience shows that many applications, including the one depicted in Fig. 1, can be designed to satisfy these properties. A specification that does not conform to the rules can be changed by, for instance, introducing collaborations that encapsulate other building blocks and direct communications.

In order to ensure that all application-specific signals of a secured collaboration are protected effectively, we identify the following properties that a collaboration to be protected must fulfill:

- **C1** The collaboration must have exactly one starting pin, which means that the collaboration is initiated by one party only.

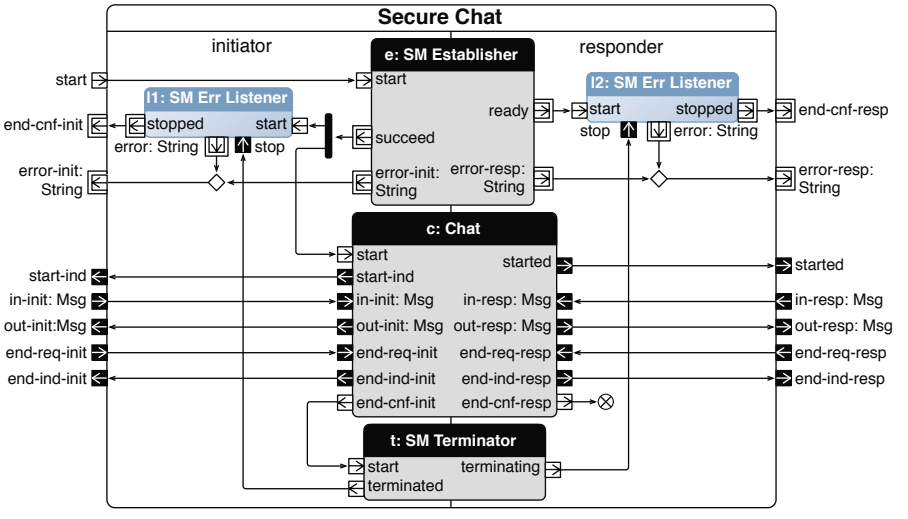



**Fig. 5.** Building Block for the Secure Chat

**C2** When a terminating pin of an activity partition is reached, there must not be any signals in the input queue of the partition or any other activity node that could trigger further behavior.

**C3** The collaboration to be secured does not directly or indirectly contain any security blocks.

The structural properties **C1** and **C3** can be checked by a syntactic inspection, while property **C2** is behavioral and is ensured by model checking. An analysis of the telemedical example shows that the system specification fulfills the properties **S1** .. **S3**, and while **C1** .. **C3** are satisfied by the *Chat* block.

### 3.2 Step 2: Embedding Security Functions

Once all properties are satisfied, a collaboration to be protected and the blocks from Sect. 2 are automatically composed into a larger collaboration. Applying this step to block *Chat* results in the *Secure Chat* collaboration depicted in Fig. 5. *e: SM Establisher* and *t: SM Terminator* are built-in before and after *c: Chat*. The flow from the pin *succeed* of the establisher block to the starting pin of the application shows that the chat can only begin after a successful secure mode setup. Likewise, the termination is started when the *c: Chat* block is fully terminated, i.e., on the initiator side. In each partition, a local block *SM Err Listener* is added to make the collaboration ready to receive an error notification.

Note that the pins of block *Secure Chat* are the same as *c: Chat*'s plus pins *error-init* and *error-resp*. These additional terminating pins are used to report a security exception that may occur. Through them, a notification may indicate either a failure during SM establishment or when the blocks *SM Err Listener* report exceptions between a successful setup and a normal termination. Thus,

after being started, collaboration *Secure Chat* may proceed normally as the chat application or is stopped at any time due to an error.

### 3.3 Step 3: Integrating the Secured Collaborations

The collaborations marked for protection in the system specification are replaced by their secured counterparts, as illustrated with collaboration *Chat* in Fig. 1, which is replaced by the *Secure Chat*. As noted above, the secured blocks are from the outside structurally and behaviorally similar, but with exception pins added. In general, it is up to the application to decide, what should happen upon such an exception. For the given example, one solution would be to inform users locally via the UIs and terminate both user clients. Since the collaborations to protect are on system level, the changes due to the additional pins are manageable; in the end, this is the necessary join point between a security function and an application logic that cannot be hidden.

## 4 Discussion and Proof

We ensure the correct integration of the secure mode solution by showing that the integration result, e.g., a secured, distributed application, fulfills the three security goals stated in Sect. 1.2. Before sketching the proof, we summarize the structural details of integrating the solution into an application-specific collaborative block $C$ in so-called *implementation directives* as follows:

- **I1 Establishment**: Block *SM Establisher* is built-in directly before the start of collaboration $C$, i.e., the termination of the block directly triggers the start of $C$. In particular,
  - **I1.1** Pins *error-init* and *error-resp* of block *SM Establisher* must never lead to the start of collaboration $C$.
  - **I1.2** Pin *succeed* of block *SM Establisher* must directly lead to the start of a local block *SM Err Listener*.
  - **I1.3** Pin *ready* of block *SM Establisher* must directly lead to the start of another local block *SM Err Listener*.
  - **I1.4** Pin *error* of both local blocks from **I1.2** and **I1.3** must not lead to the start of any other collaboration between the two participants of $C$ including block *SM Terminator*.
- **I2 Termination**: Block *SM Terminator* is built-in after the termination of collaboration $C$ and before any other collaboration attached to the same pairs of participants is activated. In particular,
  - **I2.1** Pin *terminated* of block *SM Terminator* leads to the termination of local block *SM Err Listener* from **I1.2**.
  - **I2.2** Pin *terminating* of block *SM Terminator* leads to the termination of local block *SM Err Listener* from **I1.3**.

The property **S2** which states that all collaborations have exactly two participants guarantees that our integration method correctly applies the security

functions designed for two entities. The rest of the properties for the system level specification and for the collaborations to be protected (Sect. 3.1) together with the implementation directives and the behavior of the security module in the execution environment satisfy the security goals **G1** to **G3** (Sect. 1.2). In the following, we will give the sketches of the corresponding refinement proofs:

The fulfillment of **G1** that ensures all signals are transferred protected during a secure mode phase is as follows: Due to **S3**, during the activity of a secured collaboration, no other collaboration between the same participants is active. Further, **S1** guarantees that there is no direct flow between those participants. Therefore, all communication between the two participants takes place in the secured collaboration. Due to **I1** and **I2**, the secured collaboration is only active after a successful SM establishment and before the start of SM termination. The security module of the execution environment assures that all signals exchanged between the participants of the secured collaboration are protected and a reception without resulting in the sending of error notification to the application means that no attack was detected.

A security attack can be detected by the application since **I1.2** and **I1.3** guarantee that during the lifetime of a secured collaboration the two local blocks of type *SM Err Listener* on both participants of the secured collaboration are active. Thus, an error will be detected due to the behavior of the security module in the execution environment. Furthermore, **I1.4** guarantees that the error leads to an error state and not into the normal proceeding of the function. ■

To prove **G2** claiming that all four phases are distinct, we show that in a normal condition they occur in sequence as follows:

- *unsecure mode* → *secure mode establishment*
  Due to **I1**, the *secure mode establishment* phase occurs before the secured collaboration is active. Further, **S3** guarantees that during this establishment process, no other collaboration between the participants of the secured collaboration is active. Due to **S1**, there is no other communication between the participants. Consequently, phases *unsecure mode* and *secure mode establishment* are mutually exclusive and the former leads directly to the later.
- *secure mode establishment* → *secure mode*
  Because of **I1** and **I1.1**, the secured collaboration is only started upon successful finishing of the SM establishment process.
- *secure mode* → *secure mode termination*
  This is guaranteed by **I2** and **C2**. Due to **I2**, the SM termination only begins when the secured collaboration is inactive. Moreover, **C2** guarantees that the secured collaboration contains no other signal in its queues.
- *secure mode termination* → *unsecure mode*
  Due to **I2** and **S3**, any collaboration between the participants of the secured collaboration can only be activated when SM termination process ends properly. Further, **I2.1** and **I2.2** guarantee that the remaining security function, i.e., listening for exception, is terminated as well. This shows the sequence of phase *secure mode termination* to phase *unsecure mode*. ■

The no-duplication mechanism goal of **G3** is fulfilled by **C3** that guarantees neither SM setup nor termination is executed more than once in order. ■

Employing the highly automated method described above contributes to make the task of developing secure applications less daunting. The integration of the secure mode solution can be applied to various application domains since protection of sensitive data is generally required in distributed systems. Moreover, the specification style makes changes in both functionalities and security aspects manageable. However, further investigation is needed to determine the applicability of the method for integrating other protections effectively.

As a proof of the practicability of our method, we also implemented the presented example using Arctis. The *c: Chat* block in Fig. 1 is replaced with block *c: Secure Chat*. To inform the users about an event of security exception (Sect. 3.3), the additional pins need to be connected to the UI blocks. Therefore, a streaming input that is connected to the pin *closed* is added to block *Chat UI*. Then, two components are generated automatically using Arctis. The component for the physician is running on the Java SE platform and the one for the patient on an Android phone.

## 5 Related Work

Some approaches and tool supports have been proposed to integrate security aspects in distributed applications. Middleware technologies such as Java RMI [20], Web Services and CORBA [21] are extended with protection support in order to create secure applications. Li et al. develop the RMI toolkit [22] that enables developers of RMI-based application to adopt security feature. Security standards are defined for CORBA in [23] and Web Services in [24]. The implementation of these approaches is different from our model-based method, since manual changes in the code is required.

Model-based secure system development methods have also been suggested. UMLsec [25] is a profile on security requirements that can be attached to UML diagrams to evaluate a specification for security. SecureUML [26] is an extension of UML to specify role-based access control policies. These approaches may still require much expertise on security since security solutions are developed manually. However, tools are also developed to help analysing the secured system.

Other work that attempts to integrate security concerns is aspect-oriented modeling. Here, security mechanisms are specified as aspects, and weaved into base specifications at join points. Although there is no standard, some approaches and tool-support have been proposed, see for example [6,7,27]. Both our method and aspect orientation try to integrate protection mechanism in a highly automated way. However, many of aspect-oriented approaches do not consider the functional changes after the integration. Our method limits the changes only on the system level specifications so that they are manageable.

## 6 Concluding Remarks

We presented a comprehensive method to integrate a secure communication mechanism, in which building blocks realizing dedicated security functions could be automatically and consistently integrated into an application if certain preconditions are met. We have shown that, given that these preconditions hold, the security goals are in fact fulfilled.

In the future, we will extend our method in several ways. Similar to the building blocks facilitating the secure mode, we will add further blocks to our security library to support also other security mechanisms like access control. Due to the formal basis of our method, we can analyze the specifications expressed by UML activities thoroughly. This enables tool support that can ensure the correct integration of security patterns [28], so that they are effective.

We will also exploit the formal nature of our collaboration-oriented models for the security analysis. The strategy here is two-fold: We analyze an existing functional specification for behavioral and structural properties that are relevant for the security aspect and provide a recommendation of adequate protection mechanisms based on these properties. To reveal other weaknesses, the models are translated to input for other security analysis tools such as Scyther [29].

## References

1. Mouratidis, H., Giorgini, P.: Integrating Security and Software Engineering: Advances and Future Vision. IGI Global (2006)
2. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons, Inc., Chichester (2008)
3. Lampson, B.W.: Computer Security in the Real World. Computer 37, 37–46 (2004)
4. Rescorla, E.: SSL and TLS: Designing and Building Secure Systems. Addison-Wesley, Reading (2001)
5. Dierks, T., Rescorla, E.: The Transport Layer Security Protocol (TLS) version 1.2. The Internet Engineering Task Force (IETF), RFC 5246 (August 2008)
6. Georg, G., Ray, I., Anastasakis, K., Bordbar, B., Toahchoodee, M., Houmb, S.H.: An Aspect-Oriented Methodology for Designing Secure Applications. Information and Software Technology, Special Issue: Model-Driven Development for Secure Information Systems 51(5), 846–864 (2009)
7. Mouheb, D., Talhi, C., Lima, V., Debbabi, M., Wang, L., Pourzandi, M.: Weaving security aspects into uml 2.0 design models. In: Proceedings of the 13th Workshop on Aspect-Oriented Modeling, AOM 2009, pp. 7–12. ACM, New York (2009)
8. Kraemer, F.A.: Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks. PhD thesis, Norwegian University of Science and Technology (August 2008)
9. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software 82(12), 2068–2080 (2009)
10. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 571–585. Springer, Heidelberg (2009)

11. Arctis Website, http://www.arctis.item.ntnu.no/
12. Kraemer, F.A., Herrmann, P.: Reactive Semantics for Distributed UML Activities. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 17–31. Springer, Heidelberg (2010)
13. Datta, A., Derek, A., Mitchell, J.C., Pavlovic, D.: Secure Protocol Composition. In: Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering, FMSE 2003, pp. 11–23. ACM, New York (2003)
14. Krawczyk, H.: The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 310–331. Springer, Heidelberg (2001)
15. Cremers, C.: Compositionality of Security Protocols: A Research Agenda. Electronic Notes Theoretical Computer Science 142, 99–110 (2006)
16. Baskerville, R.: Information Systems Security Design Methods: Implications for Information Systems Development. ACM Computing Surveys 25(4), 375–414 (1993)
17. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Electronic Communications of the EASST, vol. 7. EASST (2007)
18. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4276, pp. 1613–1632. Springer, Heidelberg (2006)
19. SSLEngine from JSSE, http://java.sun.com/javase/6/docs/api/javax/net/ssl/SSLEngine.html
20. Java Remote Method Invocation, http://java.sun.com/javase/technologies/core/basic/rmi/
21. Object Management Group: Common Object Request Broker Architecture (CORBA/IIOP), version 3.1, formal/2008-01-08 (January 2008)
22. Li, N., Mitchell, J.C., Tong, D.: Securing Java RMI-Based Distributed Applications. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC 2004, pp. 262–271. IEEE Computer Society, Los Alamitos (2004)
23. Object Management Group: CORBA Security Service, version 1.8, formal/2002-03-11 (March 2002)
24. OASIS: Web Services Security, version 1.1 (February 2006)
25. Jürjens, J.: Secure System Development with UML. Springer, Heidelberg (2004)
26. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology 15(1), 39–91 (2006)
27. Pavlich-Mariscal, J., Michel, L., Demurjian, S.: Enhancing UML to Model Custom Security Aspects. In: Proceedings of the 11th Workshop on Aspect-Oriented Modeling, AOM 2007 (2007)
28. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad: Security Patterns: Integrating Security and Systems Engineering. Wiley Software Patterns Series. John Wiley & Sons, Chichester (2006)
29. Cremers, C.J.: The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008)

# An Architecture-Centric Approach to Detecting Security Patterns in Software

Michaela Bunke and Karsten Sohr

Technologie-Zentrum Informatik, Bremen, Germany,
{mbunke,sohr}@tzi.de

**Abstract.** Today, software security is an issue with increasing importance. Developers, software designers, end users, and enterprises have their own needs w.r.t. software security. Therefore, when designing software, security should be built in from the beginning, for example, by using security patterns. Utilizing security patterns already improves the security of software in early software development stages. In this paper, we show how to detect security patterns in code with the help of a reverse engineering tool-suite Bauhaus. Specifically, we describe an approach to detect the *Single Access Point* security pattern in two case studies using the hierarchical reflexion method implemented in Bauhaus.

## 1 Introduction

The increasing and diverse number of technologies that are connected to the Internet such as distributed enterprise systems or smartphones and other small electronic devices like the iPad brings the topic IT security to the foreground. We interact daily with these technologies and spend much trust on a well-established software development process. However, security vulnerabilities appear in the software on all kind of PC(-like) platforms. Hence, developers must more and more face security issues during software design and especially reengineering [26].

Today's security tools that aim to support developers during the implementation phase are based on static analysis. Chess et al. describe in depth how static analysis tools can detect bugs in code such as buffer overflows or simple race conditions [3]. These bugs can be used to exploit software. Using such analysis tools can increase the software's security by ensuring its stability through filtering out bugs at code level.

Beyond known static security analysis tools like Fortify [5], which work on source code, security modeling elements exist at the architectural level such as security patterns, which should enhance security at the software design stage. Security patterns can model security issues driven by the software's requirements. Similar to the design patterns introduced by Gamma et al. [6], security patterns describe a general reusable solution to a well-known problem. Yoder and Barcalow were the first that listed existing security patterns [29]. At that time, their popularity in the pattern community grew and many patterns have been published thereafter [12]. Yoshioka et al. [30] and Heymann et al. [12] give

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 156–166, 2011.

a good survey of the published security patterns till 2008. Design patterns and security patterns have several aspects in common, but are they similar?

VanHilst and Fernandez pointed out that "GoF Patterns are not security patterns" [27]. Like design patterns, which are also called *GoF patterns* [6], security patterns can be applied to implementing and structuring software systems, but they can also model security issues and security processes in enterprises such as "Enterprise Architecture Management Patterns" [4]. Based on this heterogeneity, the classification of security patterns is an important and often discussed issue [24,28,8,9]. Different approaches exist, some describe basic topologies like separating the patterns in application domains (e.g., software, enterprise management), others describe complex layered structures. In this paper, we will have a closer look on patterns that describe how to implement a specific security feature, i.e., so-called structural patterns [8].

Within the pattern community, various descriptive models for security patterns exist [24]. The POSA model described by Buschmann et al. [2] is frequently used to describe the context and usage of security patterns. Some descriptions make use of UML diagrams or rarely source code snippets to clarify the security pattern modeling during the design phase. Nevertheless, security patterns are mostly described highly abstract; so it is difficult to understand the benefit or use if one is not familiar with software design linked with security issues [12]. Those circumstances have an impact on the software design when you have to ensure security interests and select an appropriate security pattern for the software needs. Halkidis et al. show that the usage of security patterns can offer a reasonable protection against most common attacks [10].

Frequently, software has to be modified due to changing requirements, bugs and security flaws. Moreover, the reconstruction of patterns in grown systems is quite difficult. Maintenance programmers, however, must deal with such use cases. Given that patterns in general are the best-known solution of a recurring problem [6], security patterns should also be recognized during the maintenance process to guarantee security objectives and requirements. For example, a detected *Check Point* pattern [24] allows one to conclude that on this point in code relevant information will be validated before further steps in the application flow are carried out. If the developer knows about a security pattern in code, he is aware of what is going on in this code unit or component and can program according to this context. An improvement of software quality caused by detecting incorrect or not accomplished security patterns is conceivable. In this case, the developer can avoid bypassing this *Check Point* in further implementations.

For this maintenance process, there exist several reengineering tools to get a clear view about the software structure and behavior. However, only a few of those tools take the well-known design patterns into account to support program comprehension at that point. Some approaches are presented in [27]. Presently none of them support the detection of security patterns.

All shown factors do not support the application, comprehension and recognition of security patterns in software. Therefore, it is desirable to integrate security patterns with a program comprehension tool. This ensures that security

patterns are preserved during software maintenance process and their highlighting allow once well-directed implementations of new (security) features in software.

In this paper, we focus on the reengineering scope by discussing security patterns. Specifically, we use the Resource Flow Graph (RFG) representation provided by the reverse engineering tool-suite Bauhaus [21]. With this program representation, we are able to use the integrated program comprehension method called hierarchical reflexion method [16]. This method aims to reconstruct a software architecture by mapping a hypothetical architecture to the actual software architecture extracted from the source code. Our goal is to depict the existence of security patterns on an abstract architectural level. We also describe which methods can help programmers in preserving security patterns during the software maintenance process. To demonstrate the feasibility of our approach, we present two case studies. Here, security patterns are identified in a software architecture by using our program comprehension technique.

The remainder of this paper is structured as follows. In Section 2, we briefly describe the software analysis tool Bauhaus. There we will concentrate on the RFG and the hierarchical reflexion method. In Section 3, we present further steps in combining an architecture-based methodology with security pattern detection. This is followed by the description of the case studies in Section 4. After discussing related work, we give an outlook in Section 6.

## 2 The Bauhaus Tool

The Bauhaus tool-suite is a reverse engineering tool-suite that has been employed in several industry projects [21]. Bauhaus allows one to retain two abstractions from the source code. The low-level representation called Intermediate Language (IML) is an attributed syntax tree (an enhanced AST) that contains the detailed program structure information such as loop statements, variable definitions and name bindings. The RFG, a more abstract representation, works at a higher abstraction level and represents architecturally relevant information of the software. At present such a graph can be created for programs written in C, C++, Java, ADA and C#. The RFG is a hierarchical graph that consists of typed nodes and edges representing elements like types, components, and routines and their relations. The RFG's information is stored in structured in *views*, where each view represents a different aspect of the architecture, e.g. a call graph.

Several analyses are built upon this infrastructure to derive design and architectural information like the so-called *hierarchical reflexion analysis* [16]. This analysis extends the original analysis developed by Murphy et al. [18] to hierarchical systems. It starts with a hypothesis of the architecture and a mapping of existing implementation components onto architectural components provided by an human analyst. An automated analysis then determines convergences and differences among the architecture and the implementation model, the so-called *reflexion model*. Based on these findings, the architecture and mapping may be refined and the process will be repeated until the architecture model sufficiently describes the implementation.

Usually, this procedure will be used when a software system has to be modified but the documentation or the knowledge of it got lost. Besides this reconstruction the reflexion analysis can be used to check the present implementation against their architectural specification.

## 3 Security Aspects and the RFG

Sohr and Berger [25] depict some possibilities to accomplish a security analysis with the RFG. We resume on their point and discuss other security aspects that can be based upon the RFG. In contrast to their ideas, we will not focus on policies and RBAC extensions. Our focus is software quality assurance and program comprehension in conjunction with security patterns.

*Hierarchical reflexion method for security issues:* As already mentioned in Section 2, the hierarchical reflexion analysis is a well-known method to reconstruct software architectures. This method can be used to identify security patterns. These will be marked as potential patterns and can be used to show deficiencies in the software architecture. With an automated check against the real source code, there can be detected architecture violations such as calling a component not through a *Check Point* that can induce to security concept violations. This reflexion method is used in case studies, presented in Section 4 to detect a selected security pattern in a software's architecture.

*Security patterns at the architecture level:* As shown above, the RFG provides the ability to create new views. These views can be created containing only elements focusing on special purposes. Conceivable is a view containing elements that are supposed to belong to a single security pattern, possibly identified by the method described above. If one has identified more than one security pattern and created them on different views, one can create a new view by intersecting or uniting the view to visualize composed or merged security patterns as described in [23]. Possibly this process can be automated when the system knows several available or often occurring compositions of security patterns. Presenting such combinations to maintenance programmers may facilitate the realization of adequately and inadequately programmed pattern collaborations. For instance, consider the combination of *Single Access Point* and *Check Point* pattern [24], where maybe a badly implemented cooperation raise security leaks.

*Automatic detection, suggestions and learning:* Semi-automatic detection of security patterns is good, but is time-consuming and requires deep knowledge of the system. A better approach would be the automatic detection. However, this is not easy to realize as there are many challenges as described in Section 1. Maybe, there exists a pattern language that fulfills our needs for the description of security patterns at the architecture level. We then can transform these descriptions to the RFG model for an automatic pattern matching and are able to present the maintenance programmer pattern suggestions. These suggestions can be assessed or modified by the programmer to improve the security pattern

model. Thereby, we can collect pattern derivatives for improving the automatic or even the semi-automatic detection. Moreover, this collection gives the abstract appearance of security patterns a more clear shape that can be reused. This technique can be refined by using security anti- or misuse patterns to model an architecture's irregularities to be able to detect the incorrect usage of security patterns. The benefit of the sketched technique is that software systems can be post-checked and hardened before they will be released.

*Source and sink markers for pattern endings:* Information flow is an important issue concerning software security. A security view of the RFG can also model fractals of the information flow. If we combine the RFG with the other more-detailed code representations such as the IML, we can model the information flow between components. If we have detected a security pattern or compound, we can extract the pattern in a new view and highlight sources and sinks of the patterns. This would enhance the role-based view described by Sohr et al. [25] and give the opportunity to plug in a further information flow analysis to validate the pattern's behavior. To give an example consider the communication with a database or a password manager. In these cases, the visualization of security patterns' sinks and sources like architectural glue dots addresses the maintenance programmer. It will support and ensure the information flow comprehension while reconstructing the software system.

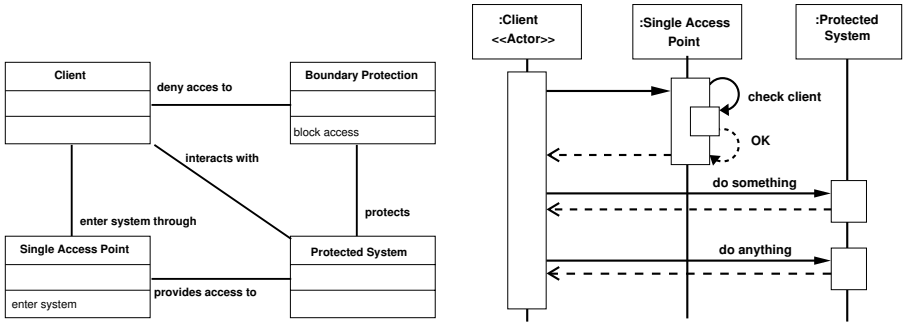
## 4 Early Case Studies

We now discuss our architecture-centric security pattern analysis in the context of two case studies. We selected the security pattern *Single Access Point* to demonstrate that we can identify this pattern within an abstract software representation.

We chose primarily the open source instant messenger client Spark [13] and an open source Android application named Simple Android Instant Messaging Application [17]. Both are Java-based programs, so we took the Java byte code and generated the software architecture in the RFG format. This is the starting point for using the hierarchical reflexion method to detect the *Single Access Point* pattern. First of all, we present the pattern in Section 4.1 and then we will have a closer look on the case studies in 4.2 and 4.3.

### 4.1 Single Access Point Pattern

A *Single Access Point* pattern [24] provides access to a system for external clients. Moreover, it ensures that the system cannot be damaged or misused by such clients. The idea behind this pattern is that an exclusive door to the system can be better protected and controlled than many. Fig. 1 depicts the UML diagrams for the *Single Access Point* security pattern. For this reason, many application clients such as twitter or instant messenger clients that provide any kind of access to other systems use a derivative of this pattern in order to provide clients (mostly, users) access to underlying services.

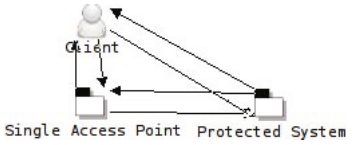



(a) *Single Access Point* UML diagram (b) *Single Access Point* Sequence Diagram

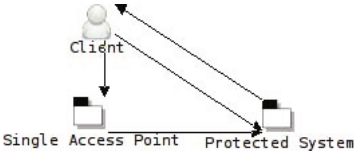
**Fig. 1.** *Single Access Point* [24]

According to the UML diagram in Fig. 1(a), we modeled a first hypothetical architecture (see Fig. 2(a)) containing the components `Client`, `Single Access Point`, and `Protected System`. The client that uses this software is represented by the `Client` component in the architecture. Given that the reflexion method is based on static code dependencies and our client in the case studies is a human, we obviously will never see any match in the dependencies with the system. Schumacher et al. mentioned that the `Boundary Protection` component of a protected system was often hard to show. This also applies to our case studies where the client is a human user that needs to know a user name and password to gain access to the protected system. In this case, the user's knowledge models the `Boundary Protection` component. Hence we skip this component as we cannot model it according to static analysis. We have not specified in depth the dependencies between components such as "calls", "references", and "inherits" to simplify the dependencies and modeled them as undirected dependencies according to the undirected edges in Fig. 1(a).

After our first attempt to model the architecture, we realized that the UML model was not adequate enough to represent the idea of the *Single Access Point* pattern. Thus, not every direction of the dependencies between the components may be allowed to ensure a secure behavior. For this reason, we give a more specific access point model according to the information given in Fig. 1(b). First, the `Client` interacts with the `Single Access Point` component, and thereafter the client can interact with the `Protected System`. Based upon this, we assume that the `Single Access Point` component allows or denies the user's request and informs the `Protected System` about the response. Possibly, the `Single Access Point` component instantiates a further window to allow the user to interact with the `Protected System` after the logon. Therefore, we specify that a proper behavior of the system is that the `Single Access Point` has dependencies to the `Protected System` to call, communicate or instantiate something after passing the `Single Access Point`. The corresponding hypothetical architecture is shown in Fig. 2(b).




(a) Hypothetical architecture




(b) Hypothetical architecture - specific dependencies

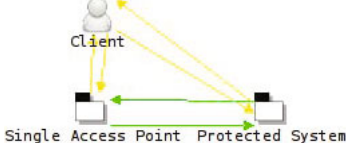
**Fig. 2.** Hypothetical architectures

Both hypothetical architectures will be used with the hierarchical reflexion method on the chosen applications to show and discuss distinctive features.

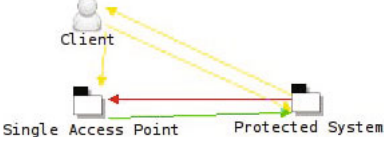
### 4.2 Case Study: Spark

Spark is an open source instant messenger client that provides a login screen and is expected to use this pattern [13]. It is a client that allows users to log on to an instant messenger network and then receive and write instant messages to other users.

Intuitively, we map the software components according to their names such as "LoginDialog", "LoginSettingsDialog" to the component `Single Access Point`. Then, we assume that the rest of the code is in package "org.jivesoftware" is the `Protected System`.




(a) Model check result against Fig. 2(a)




(b) Model check result against Fig. 2(b)

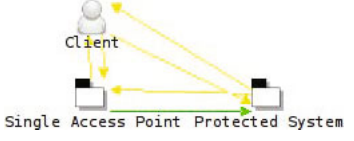
**Fig. 3.** Detection results - Spark

Fig. 3(a) shows the match between the hypothetical architecture and the real code architecture. As expected, outgoing and incoming edges of the `Client` component are marked as absence (yellow edges). The edges between `Single Access Point` and `Protected System` are marked as convergences. This shows that this pattern can be found in the software's architecture by using the reflexion method.

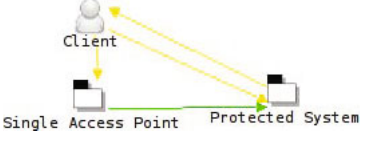
In Fig. 3(b), we try to detect a login behavior in Spark. The architecture match result is depicted in Fig. 3(b). The expected dependency between `Single Access Point` and `Protected System` is marked as convergence (green edge). However, there exist more dependencies than we have modeled, represented by the red edges. They arise from static field usages and class instantiations. This shows that the two identified components are bundled together and are not strictly separated in Spark as one might expect according to their task.

### 4.3 Case Study: Simple Android Instant Messaging Application

According to our experiences with Spark we will have a closer look on another open source application. The Simple Android Instant Messaging Application [17] is an example application for the mobile phone platform Android [7]. The author's intention to make this application freely available was to provide interested people with an example of an Android application and show how instant messaging can be provided easily. It communicates via the http protocol with a web server. This server is also used for user authentication. We assume that every component starting or ending with "Login" will indicate the `Single Access Point` in the source code. The rest belongs to the `Protected System` because it provides the instant messaging communication with the server. For the reflexion method, we used the same hypothetical architectures as depicted in Fig. 2(a) and 2(b).




(a) Model check result checking against the hypothetical architecture from Fig. 2(a)




(b) Model check result checking against the hypothetical architecture from Fig. 2(b)

**Fig. 4.** Detection results - Simple Android Instant Messaging Application

Fig. 4(b) depicts the architecture match result for the expected behavior. Here, the dependency between the components `Single Access Point` and `Protected System` is marked as convergence (green edge) and the dependencies to the client are marked as absence (yellow edges). This indicates that in the Simple Android Instant Messaging Application the components for `Single Access Point` and `Protected System` are separated in the code. This hypothesis is confirmed by Fig. 4(a) that shows an absence between `Protected System` and `Single Access Point`.

### 4.4 Conclusion

We demonstrated with these case studies that we were able to detect a security pattern within software using the components described in the pattern description. Besides the two case studies discussed, we have used this method to detect the *Single Access Point* pattern in two other software systems and to detect the *Runtime mix'n and match* design pattern [1] that is coupled with a *Check Point* security pattern. In particular, we detected it in the middleware of the open source platform Android [7].

Towards this case study we expect to be able to analyse more security patterns which apply the classification of structural patterns. They description should

also contain UML diagrams that clarify their structure and behavior. However, shown in this study even with the help of such diagrams it cannot be clearly decided whether a security pattern is modeled accurate or inaccurate. In our case Spark models the *Single Access Point* pattern according to the UML class diagram and the Simple Android Instant Messaging Application in compliance with both UML diagrams. Hence further researches on security patterns and their appearance in software architecture are reasonable.

With the introduced static examination, however, we are not able to consider if the system behaves in the expected way. Therefore, we will need more source code information as provided in the IML representation, a more specific static description of this pattern or even dynamic analysis information. Moreover, on detecting such patterns automatically or semi-automatically we have to deal with abstract descriptions that must be modeled differently for several application contexts. For example, in the shown case the client was a user that must enter his credentials. In another case, the client is possibly a web service that tries to use another web service.

## 5 Related Work

There exist a plethora of works for the static security analysis of software [3]. The works on static analysis for security often use the source code in order to detect common vulnerabilities such as buffer overflows or cross site scripting [5,20]. Another approach combines type-based security and annotations with dependence graph-based information flow control [11]. All aforementioned approaches do not deal with security patterns.

In addition, VanHilst and Fernandez [27] discuss the possibilities to detect security patterns using reverse engineering like [19]. They identify some problems that may occur during detection, but they do not describe a practical approach using the reflexion method. Concerning security and software architectures, Ryoo et al. [22] presented a basic approach to detecting architectural constructs and properties that make software less secure. Another idea on employing the software architecture for static security analysis was described by Sohr and Berger [25]. They use the RFG to check policies and permissions on Java EE and the Android platform. An approach to the automated verification of UMLsec models has been presented by Jürjens and Shabalin [14]. They present automated verification of UML models w.r.t. modeled security requirements. The precondition to this is that there exists a UML-modeled architecture for the software system. Our approach has the reverse engineering point of view, and we work with an abstract representation based on the software implementation for searching security patterns. Thus, our approach requires no UML documentation of the software architecture and represents always the actual software's implementation state.

## 6 Outlook

In this paper, we demonstrated that we are able to detect security patterns using the reflexion method. As indicated in Section 4.4, we plan to improve on

the automation degree of our detection process for instance using the incremental reflexion method [15]. In addition, we will search for the best arrangement of static and dynamic analysis techniques to support this goal. A further step in our work will be the examination of larger software systems to point out which security patterns are used and clarify their impact on software's architecture.

Section 1 indicates that many topics are open to discuss security patterns and reverse engineering. Thus we consider this work as a starting point for further approaches in security and program comprehension, bringing together the different research communities of reverse engineering and software security.

## References


1. Austrem, P.G.: Runtime mix'n and match design pattern. In: Proc. of the 15th Pattern Languages of Programs, pp. 1–8. ACM, New York (2008)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, Chichester (1996)
3. Chess, B., McGraw, G.: Static analysis for security. IEEE Security and Privacy 2, 76–79 (2004)
4. Ernst, A.M.: Enterprise architecture management patterns. In: Proc. of the 15th Pattern Languages of Programs, pp. 1–20. ACM, New York (2008)
5. Fortify Software. Fortify source code analyser (2009), `http://www.fortify.com/products`
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software. Addison Wesley, Reading (1995)
7. Google Inc. Android development (2010), `http://developer.android.com/index.html`
8. Hafiz, M., Adamczyk, P., Johnson, R.E.: Organizing security patterns. IEEE Software 24, 52–60 (2007)
9. Hafiz, M., Johnson, R.: Security patterns and their classification schemes. Technical report, Technical Report for Microsoft's Patterns and Practices Group (September 2006)
10. Halkidis, S.T., Chatzigeorgiou, A., Stephanides, G.: A qualitative analysis of software security patterns. Computers & Security 25(5), 379–392 (2006)
11. Hammer, C.: Experiences with pdg-based ifc. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 44–60. Springer, Heidelberg (2010)
12. Heyman, T., Yskout, K., Scandariato, R., Joosen, W.: An analysis of the security patterns landscape. In: Proc. of 3rd International Workshop on Software Engineering for Secure Systems. IEEE Computer Society, Los Alamitos (2007)
13. Jive Software. Spark - project page (2010), `http://www.igniterealtime.org/projects/spark/index.jsp`
14. Jürjens, J., Shabalin, P.: Automated verification of uMLsec models for security requirements. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 365–379. Springer, Heidelberg (2004)
15. Koschke, R.: Incremental reflexion analysis. In: European Conference on Software Maintenance and Reengineering. IEEE Computer Society Press, Los Alamitos (2010)
16. Koschke, R., Simon, D.: Hierarchical reflexion models. In: Proc. of 10th Working Conference on Reverse Engineering, pp. 36–45 (November 2003)

17. Mermerkaya, A.O.: Simple android instant messaging application - project page (2010), http://code.google.com/p/simple-android-instant-messaging-application/
18. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: Bridging the gap between design and implementation. IEEE Transactions on Software Engineering 27(4), 364–380 (2001)
19. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: Proc. of the 24th International Conference on Software Engineering, pp. 338–348. ACM, New York (2002)
20. Ounce Labs Inc. (2010), http://www.ouncelabs.com/
21. Raza, A., Vogel, G., Plödereder, E.: Bauhaus – A tool suite for program analysis and reverse engineering. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 71–82. Springer, Heidelberg (2006)
22. Ryoo, J., Laplante, P., Kazman, R.: In search of architectural patterns for software security. Computer 42, 98–100 (2009)
23. Schumacher, M.: Merging security patterns. In: Proc. of 6th European Conference on Pattern Languages of Programs (2001), http://www.voelter.de/data/workshops/europlop2001/merging_security_patterns.pdf
24. Schumacher, M., Fernandez, E., Hybertson, D., Buschmann, F.: Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, Chichester (2005)
25. Sohr, K., Berger, B.: Towards architecture-centric security analysis of software. In: Proc. of International Symposium on Engineering Secure Software and Systems. Springer, Heidelberg (2010)
26. The H Security. Number of critical, but unpatched, vulnerabilities is rising (2010), http://www.h-online.com/security/news/item/Number-of-critical-but-unpatched-vulnerabilities-is-rising-1067495.html
27. Van Hilst, M., Fernandez, E.B.: Reverse engineering to detect security patterns in code. In: Proc. of 1st International Workshop on Software Patterns and Quality. Information Processing Society of Japan (December 2007)
28. Washizaki, H., Fernandez, E.B., Maruyama, K., Kubo, A., Yoshioka, N.: Improving the classification of security patterns. In: Workshop on International Conference on Database and Expert Systems Applications, pp. 165–170 (2009)
29. Yoder, J., Barcalow, J.: Architectural patterns for enabling application security. In: Proc. of 4th Pattern Languages of Programs, Monticello/IL (1997)
30. Yoshioka, N., Washizaki, H., Maruyma, K.: A survey on security patterns. Progress in Informatics 5, 35–47 (2008)

# The Security Twin Peaks

Thomas Heyman[1], Koen Yskout[1], Riccardo Scandariato[1],
Holger Schmidt[2], and Yijun Yu[3]

[1] IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
first.last@cs.kuleuven.be
[2] Technische Universität Dortmund, Germany
holger.schmidt@cs.tu-dortmund.de
[3] Open University, United Kingdom
y.yu@open.ac.uk

**Abstract.** The feedback from architectural decisions to the elaboration of requirements is an established concept in the software engineering community. However, pinpointing the nature of this feedback in a precise way is a largely open problem. Often, the feedback is generically characterized as additional qualities that might be affected by an architect's choice. This paper provides a practical perspective on this problem by leveraging architectural security patterns. The contribution of this paper is the Security Twin Peaks model, which serves as an operational framework to co-develop security in the requirements and the architectural artifacts.

**Keywords:** security, software architecture, requirements, patterns.

## 1 Introduction

Often, the requirements specification is regarded as an independent activity with respect to the rest of the software engineering process. In fact, both literature and practice have pointed out that requirements cannot be specified in isolation and "thrown over the wall" to the designers and implementers of the system. In contrast, the requirements specification (describing the problem) and the architectural design (shaping a solution) are carried on concurrently and iteratively, while still maintaining the separation between the problem and solution space. This process of co-developing the requirements and the software architecture is referred to as the Twin Peaks model [22]. As depicted in Figure 1, the specification process (i.e., refinement) in the Twin Peaks model continuously jumps back and forth between the requirements and architectural peaks, in order to embrace the decisions made in the other peak.

Some work already exists that focuses on the forward transition from security requirements to software architectures [18,11,31,26]. This work leverages standardized solutions, such as security patterns. These solutions are related to the security requirements via traceability links, facilitating both the selection of the right architectural solutions and documentation of the rationale for the architectural choice [29]. This, in turn, facilitates impact analysis in face of change.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 167–180, 2011.

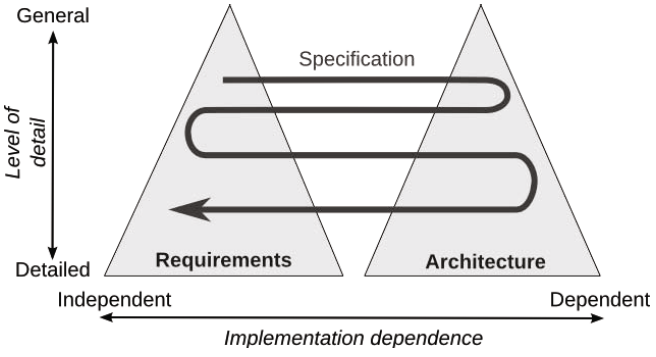



**Fig. 1.** The original Twin Peaks model [22]

Concerning the backward transition, even if the importance of the feedback from the architecture to the requirements is an established concept in the software engineering community, the literature fails in pinpointing the nature of this feedback in a precise and operational way. This is also true for software qualities such as security. Often, the feedback is generically characterized as additional qualities, such as performance, that might be affected by a security architectural choice [28].

This paper presents an elaboration of the original Twin Peaks model in the context of security, called the Security Twin Peaks. By leveraging architectural security patterns, the model provides constructive insights in the process of specifying and designing a security-aware system, by pinpointing interaction points between the software architect's and the requirements engineer's perspective. In particular, we illustrate that an architectural security pattern actually consists of three elements that are key with regard to the Twin Peaks: (1) components supporting the security requirement by fulfilling a security functionality, (2) roles (connecting the generic solution to the specific architecture) and the expectations on such roles, and (3) residual goals. As our main contribution, we show how these elements are related to the requirements specification and can be leveraged to drive the refinement process, thereby substantiating the Security Twin Peaks model. KAOS is used to represent (security) requirements [27]. However, the presented model is not specific to the chosen requirements engineering methodology.

The rest of this paper is organized as follows. The related work is presented in Section 2. Architectural security patterns are analyzed in Section 3, in order to identify the root causes of feedback from the architectural design to the requirements specification. The Security Twin Peaks model is introduced and discussed in Section 4. Finally, Section 5 presents the concluding remarks.

## 2 Related Work

*The problem peak in secure software engineering.* In the realm of security software engineering, Haley et al. [10,9] present a framework for representing security

requirements in an application context and for both formal and informal argumentation about whether a system satisfies them. The proposed argumentation process specifies several iterative steps for the problem part of the Twin Peaks. Mouratidis et al. [19,18,14] present a procedure to translate the Secure Tropos models [8] into UMLsec diagrams [16]. However, they do not provide explicit feedback from the chosen architectural solution back to the requirements phase.

*The solution peak in secure software engineering.* Côté et al. [5] propose a software development method using problem frames for requirements analysis and using architectural patterns for the design. For the benefit of evolving systems, evolution operators are proposed to guide a pattern-based transformation procedure, including re-engineering tasks for adjusting a given software architecture to meet new system demands. Through application of these operators, relations between analysis and design documents are explored systematically for accomplishing desired software modifications, which allows for reusing development documents to a large extent, even when the application environment and the requirements change. In parallel, Hall et al. [12] propose the A-struct pattern in problem frames to explore the relationship between requirements and architectures in a problem-oriented software engineering methodology. Both work deals with feedback for general software engineering problems, but they had not focused on specific difficulties in secure software development.

*Security patterns.* Many authors have advanced the field of security design patterns during the last years [30,17,3,24,25,6]. A comprehensive overview and a comparison of the different existing security design patterns can be found in [13], which establishes, among others, that the quality of the documentation of some existing security design patterns is questionable. A recent survey by Bandara et al. [1] compares various software engineering methods in application to address a concrete RBAC security pattern to reveal that there is still a need to systematically support the Security Twin Peaks by linking security requirements with security architectures. To shed more lights on the mechanisms for Secure Twin Peaks, another extensive survey on security requirements for evolving systems [21] categories the literature in terms of how an evolving system is related to its evolving requirements and changing contexts. Fernandez et al. [7] propose a methodology to systematically use security design patterns in UML activity diagrams to identify threats to the system and to nullify these threats during fine-grained design. Mouratidis et al. [20] present an approach to make use of security design patterns that connects these patterns to the results generated by the Secure Tropos methodology [8].

## 3 Architectural Security Patterns Revisited

Patterns are a well-known and recognized technique to build software architectures. This section revisits architectural security patterns and highlights the key elements that are used in Section 4 as stepping stones to link the solution domain (architectural peak) to the problem domain (requirements peak).

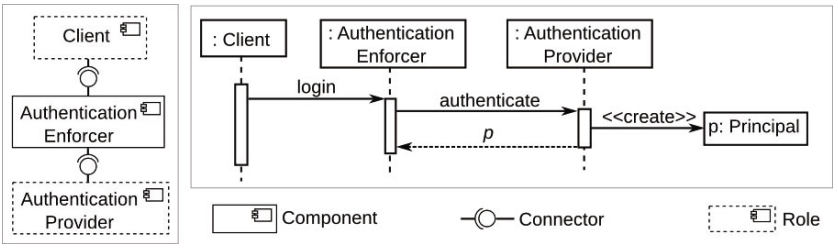



**Fig. 2.** Structure and behavior of the Authentication Enforcer pattern [25]

To illustrate the concepts, the Authentication Enforcer [25] pattern is used as an example. This pattern describes how to solve the problem of authenticating users in a systematic way by creating an authentication layer, and provides a number of different implementation alternatives to realize this layer in the architecture. One of the alternatives is the provider-based authentication method depicted in Figure 2. The solution consists of an Authentication Enforcer component that mediates all access requests originated by Clients and delegates the implementation of an authentication method to a third-party Authentication Provider component.

### 3.1 Key Notions for Co-development

Besides the many pieces of information that are traditionally documented in a pattern (e.g., the problem description and the known uses), we observe that an architectural security pattern can be seen, at its core, as a combination of three parts. They are:

*1) Components and behavioral requirements.* The participants of a pattern can be grouped into components that are newly introduced, and roles (further discussed in point 2) referring to components that are external to the pattern. A new component has a security-specific purpose, i.e., it adds new functionality to the system that is specific to a security requirement the system should uphold. This corresponds to the operationalization of a secondary functional requirement, as in Haley et al. [9]. Hence, new components introduce (finer-grained) behavioral requirements, which they fulfil and to which they can be linked, as clarified in Section 4.

Example. In the example, the Authentication Enforcer pattern introduces an Authentication Enforcer component, which encapsulates the authentication logic. This component is only needed to address the security problem statement.

*2) Roles and expectations.* Patterns are generic solutions that need to be instantiated in the context of concrete (possibly partial) architectures. Roles are used for that purpose. A role is a reference that needs to be mapped to a component (or sub-system) that is already present in the existing architecture. Hence, the roles provide the connective between the new components and the existing components, and define how both should interact.

Often, a pattern introduces expectations specific to its roles that need to be fulfilled by the concrete architecture. The pattern can impose constraints on both the way an external component is supposed to play a given role, as well as the way the external component interacts via the connectors with the rest of the patten internals. Hence, roles introduce finer-grained requirements in the problem domain.

Example. The Authentication Enforcer pattern introduces two roles: the Client, which is mapped to the actual component that invokes the Authentication Enforcer component, and the Authentication Provider, which needs to be mapped to a third-party system providing an authentication mechanism. One expectation that should be realized by the Authentication Provider role is that the result of the authentication process is passed back as a Principal object. The pattern specifies the responsibilities but does not dictate how the Authentication Provider should be implemented (e.g., it does not specify the authentication mechanism). The pattern also imposes certain expectations on the interaction between the Authentication Enforcer and the Client. It suggests to protect the confidentiality of credentials, especially during transit. For instance, in a web context, the pattern suggests to avoid clear-text communication.

*3) Residual goals.* These are security considerations to take into account when instantiating the pattern. For instance, the pattern might make (trust) assumptions on the environment in which the system is deployed, that fall outside the scope of the solution presented by the pattern. These residual goals are not under the responsibility of either the newly introduced components or the roles.

Example. One residual goal of the Authentication Enforcer pattern is to localize all authentication logic in the Authentication Enforcer component. Realizing this goal is out of scope of the pattern itself—it is impossible for the Authentication Enforcer pattern to enforce, in some way, that no other component contains custom authentication code. A residual goal externalizes this concern, placing the responsibility back in the hands of the software architect. Another residual goal is that it should be impossible for an attacker to obtain the user's credentials. This manifests itself in residual requirements such as "make credentials hard to forge" (e.g., implement a strong password policy) and "ensure that credentials do not leak" (e.g., store salted hashes locally, do not store the passwords in plain text).

As a final note, although this section focuses on architectural security patterns, the three parts presented above can be identified in any generic security architectural solution, irrespective from whether it is described as a pattern or not.

### 3.2 Revisiting the Pattern Documentation

The above three parts have been implicitly mentioned (often in a scattered way) in the literature, e.g., in the documentation of existing architectural patterns [4]. These patterns are usually documented by means of the following information [3]. The *problem and forces* describe the context from which the pattern emerged. The *solution* describes how the pattern resolves the competing forces and solves the problem. This solution consists of two parts. The *structure* of the

solution depicts the different participants that play a role in the pattern, and the relationships among them. The *behavior* of the solution describes the collaborations among the different participants, by which they realize their common goals and solve the problem. Apart from the solution, a pattern should document its *consequences*, that highlight both the strengths and weaknesses of the proposed solution. Finally, an *example* of the pattern in an easily understood software setting shows how this is applied in practise.

The three elements from the previous section are often present in a general pattern description. The participants from the solution description correspond to both the newly introduced components and roles. The behavior of the solution introduces the behavioral requirements on the new components of the pattern, and possibly also expectations on the roles. The consequences of the pattern (and in particular the weaknesses) identify potential residual goals. This clearly shows that the three key notions for co-development are not so abstract and are, in most cases, already implicitly documented in existing patterns. This work contributes to the subject by bringing these three parts to the front and illustrating the primary role they play in the context of the Security Twin Peaks.

## 4 The Security Twin Peaks

In the previous section, we discussed the fundamental parts comprising an architectural security pattern. In this section, these concepts are leveraged to outline a constructive process for co-developing secure software architectures and security requirements. Particular focus is placed on the feedback loop between architecture and requirements, and the more subtle intricacies that need to be taken into account.

This process is not a new development process or paradigm by itself. Rather, it gives constructive guidance on what is mostly left implicit, i.e., how to interleave the requirements and architectural peaks when designing a secure software system using security patterns or other generic security solutions. Hopefully, the awareness of this process contributes to the effectiveness of requirements engineering and software architecture design.

For the solution peak, we apply an attribute-driven architectural design approach, such as Bass et al. (ADD, [2]). In attribute-driven design, non-functional concerns such as performance, maintainability, security, and so on, are referred to as quality attributes, which are orthogonal to the functionality expected of the system and drive the design of the software architecture. Qualities are realized through fundamental design decisions, referred to as tactics (a.k.a. solution strategies). An architectural pattern (or style) is a domain-independent solution to a recurring problem, which packages and operationalizes a number of tactics.

For the problem peak, we apply a goal-based requirements engineering approach, such as the one by van Lamsweerde (KAOS, [27]). In goal-based methods, goals (prescriptive statements of intent that the system should satisfy) are used for requirements elicitation, analysis and elaboration. Agents (active system components playing a specific role in goal satisfaction) achieve these goals through cooperation.

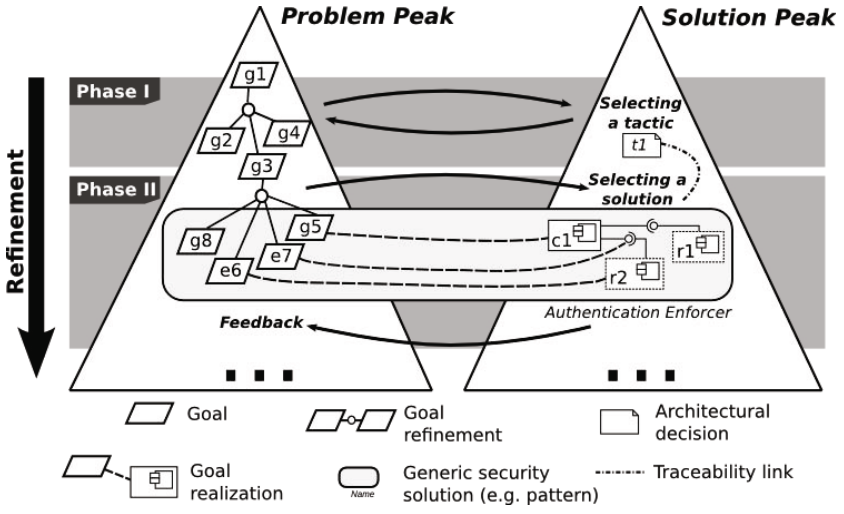



**Fig. 3.** The Security Twin Peaks

We illustrate the process on a simple application. Consider an online shop which allows customers to buy products over the Internet. Customers have an account to which costs are billed. For billing purposes, an important goal of the system is that all purchases are securely traced back to the customers.

### 4.1 Overview

The process is sketched in Figure 3. Table 1 complements the figure by explaining the meaning of the labels. The process progresses through 8 activities grouped in 2 phases. These phases are repeated over several iterations until the requirements specification and the architectural design are deemed as complete (i.e., detailed enough). In KAOS terminology, the goal decomposition stops once the leaf goals are realizable by software agents, i.e., a solution is selected and instantiated. In Phase I (activities 1-4), a tactic is selected to realize a goal. In Phase II (activities 5-9) a pattern is selected and instantiated. For each activity, a graphical representation is given of the current peak (with a filled triangle), and the transition between the peaks (with an arrow) where applicable. After a bird's-eye view on the whole process, each activity is described in more detail.

Act. 1. ▲ △ Select an **initial security goal** that will be refined in this iteration of the process. In the example, goal $g_1$ ('all purchases are securely traced') is selected. Obviously, $g_1$ would be part of a larger goal tree that is not shown here.

Act. 2. △ ▲ Choose and assess a **solution tactic** for the goal. In the example, a prevention tactic is chosen: users shall be authenticated before purchase orders can be placed ($t_1$). The architect (together with other

**Table 1.** The Security Twin Peaks: example

| Label | Type | Description |
|---|---|---|
| $g_1$ | Goal | All purchases are securely traced |
| $t_1$ | Tactic | User shall be authenticated before purchasing |
| $g_3$ | Goal | An identified user shall be authenticated |
| $g_5$ | Goal | Mediate requests and delegate authentication |
| $e_6$ | Expectation | Return a Principal object |
| $e_7$ | Expectation | Avoid clear-text communication |
| $g_8$ | Residual goal | Ensure that credentials do not leak |
| $c_1$ | Component | Authentication Enforcer |
| $r_1$ | Role | Client |
| $r_2$ | Role | Authentication Provider |

stakeholders), must decide whether, for example, the assurance gained from authentication outweighs the decrease in usability that goes together with enforcing authentication.

Act. 3. ▲ △ **Refine** the goal based on the chosen tactic. In the example, this leads to the introduction of sub-goals $g_2$ ('users shall be identified'), $g_3$ ('an identified user shall be authenticated') and $g_4$ ('only authenticated users can purchase products'). While manual refinement by an expert is possible, a problem decomposition pattern could be associated to the tactic. The decomposition pattern can provide extra guarantees of soundness, e.g., by ensuring that the set of sub-goals is indeed complete.

Act. 4. ▲ △ Check for **conflicts** between the newly introduced and the previous goals. Resolve conflicts where possible. If a conflict cannot be satisfactorily resolved, backtrack to Activity 2 to select a different solution. For instance, the identification goal $g_2$ may conflict with an anonymity goal elsewhere in the goal tree, aimed at protecting the customer's privacy. The stakeholders can, for example, weaken the anonymity goal so that customers can still anonymously browse the shop, but anonymity is no longer required when an actual purchase is made.

Act. 5. ▲ △ **Select** a sub-goal introduced in the previous step (e.g., $g_3$) that has to be resolved using an architectural security pattern.

Act. 6. △ ▲ Choose an **architectural security pattern** whose problem statement matches the selected sub-goal. In the example, the Authentication Enforcer pattern is chosen to resolve goal $g_3$. For instance, this can be a solution pattern that is linked to the used problem decomposition pattern. If no suitable solution is found, backtrack to activity 2 to select a different tactic.

Act. 7. △ ▲ **Instantiate** the architectural security pattern by performing the following activities:

(a) *Instantiate the newly introduced components* from the pattern in the architecture. In the example, an Authentication Enforcer component ($c_1$) is added to the architecture.

(b) Connect the new components to the rest of the architecture by *binding the roles* to concrete architectural elements. If no suitable

architectural elements are present already in the architecture, create them. In the example, the Client role ($r_1$) is mapped to the existing component that handles purchases, and the Authentication Provider role ($r_2$) is mapped to a to-be-introduced component, responsible for checking the customer's credentials.

If the instantiation of the pattern becomes infeasible, backtrack to activity 6 to select a different pattern.

Act. 8. ▲ △ **Update** the requirements model so that it corresponds to the new architecture, by performing the following activities.

(a) Introduce new requirements that describe the functionality of the newly introduced components. For instance, $g_5$ ('mediate requests and delegate authentication').

(b) Introduce the expectations that need to be achieved by the elements that play one of the pattern's roles. For instance, $e_6$ ('return a Principal object') is an expectation for role $r_2$ and $e_7$ ('avoid clear-text communication') is an expectation on the connector between $c_1$ and $r_2$. Note that this list of expectations is not complete for the given example.

(c) Add the residual goals described by the pattern to the goal tree. For instance, only $g_8$ ('ensure that credentials do not leak') is shown in the example.

Act. 9. ▲ △ Check for **conflicts** between the newly introduced and the previous goals. Resolve conflicts where possible. If a conflict cannot be satisfactorily resolved, backtrack to activity 6 to select a different solution.

### 4.2 Discussion

The previous process description should be complemented with the following considerations.

Activity 1. The security goal that is selected in this activity can originate from known requirements engineering techniques, which we do not consider further in this work. Also, the order in which goals are selected for refinement is not fixed, and should be decided by the requirements engineer and the other stakeholders. It should be noted, however, that additional security goals can be introduced by Activity 8 later in the process. These goals should also be considered for selection in a next iteration.

Activity 2. In choosing the solution tactic, the focus is on determining a suitable tactic to guide the goal decomposition, possibly led by a catalog of tactics. For instance, security can be handled by preventing attacks (e.g., authenticate users), detecting attacks (e.g., intrusion detection) or recovering from an attack (e.g., using audit trails) [2]. While not having a direct manifestation in the primary architectural artifacts (at this stage), choosing a tactic does involve the architectural peak, as potential architectural constraints need to be taken into account. This is why the architect should assess whether the current architecture is able to support the considered tactic. Also, care must be taken

to choose a tactic that does not conflict with the important qualities of the existing architecture. Note that it is still uncertain whether it is feasible to fulfill the goal using the chosen tactic. This only becomes apparent after a solution has been chosen and instantiated in Activity 7.

To determine the suitability of a tactic, various factors need to be taken into account and a risk assessment should be performed to decide whether the potential losses outweigh the implementation costs. For instance, the tactic can be too costly or even impossible to implement, e.g., a tactic may require full mediation, which is not supported by the current architectural environment.

Finally, notice that the selection of a tactic is an important architectural decision that belongs to the body of architectural knowledge. As shown in Figure 3, the tactic can be linked to the pattern that will be selected later on. In this respect, the tactic documents the rationale that will lead to the selection of a certain pattern and complements the rationale represented by linking the pattern and the goal it realizes.

In the online shop example, the prevention tactic is chosen: a user will be asked to authenticate before the shopping process continues, ensuring that the identity of the user is known before the billing procedure starts. An alternative tactic (while not as straightforward in the e-commerce context) would be detection and recovery: send the invoice to the address the user entered, without performing rigorous authentication first. If the bill is paid, the item gets shipped. Otherwise, the order is canceled.

Activity 3. Performing a goal decomposition based on a tactic represents the completion of a first round-trip between the problem peak and the solution peak. Note that the influence of architectural decisions on goal decomposition is mentioned in KAOS as well. In particular, in KAOS, a goal can be decomposed into several alternative branches and it is acknowledged that the selection of an alternative leads to a different architecture. In KAOS, the selection of an alternative is driven by soft goals (i.e., system qualities, development goals or architectural constraints) [26].

Activity 4 (and 9). In some cases, the new goals (resulting from a decomposition or introduced by a pattern instantiation) will fit naturally in the existing goal tree. However, conflicts may emerge and, hence, there is a need to explicitly incorporate conflict resolution in the secure development process.

Requirements engineering methodologies such as KAOS already define techniques to resolve conflicts (e.g., avoidance, restoration, anticipation or weakening) [27]. If the conflict cannot be sufficiently resolved, however, backtracking and selecting a different tactic or pattern can be considered. Of course, it can also be decided that the currently selected pattern remains in place, and the other (conflicting) part of the system is revisited.

Activity 5. The selection of a sub-goal initiates the second part of the process, where a concrete solution is chosen and instantiated. Like in Activity 1, the order of selection is left to the insight of the requirements engineer and the other stakeholders, which may mandate certain priorities.

Activity 6. The selection of the architectural pattern defines a traceability link, connecting the selected sub-goal and the pattern, i.e., the goal provides the rationale for the pattern. As mentioned before, this explicit relationship enriches the architectural knowledge.

Note also that, sometimes, a pattern may be able to solve a collection of goals simultaneously. This leads to more complex traceability links, but the outlined process can still be used.

Conversely, an architectural pattern may not be a complete match for the selected sub-goal, and can only fulfill a part of it. In this case, the goal can be additionally refined, such that one of its children match the pattern, or, alternatively, the initial refinement can be adapted.

Activity 7. By instantiating new components and connecting these to existing components, the software architecture gets refined. This refinement comes in two flavors. A pattern can extend an architecture with new components, while largely leaving the existing system untouched, as is the case with the Authentication Enforcer. As a second category of refinements, a pattern can substitute one or more components with a more refined subsystem. An example of such a pattern is the Secure Message Router [2], which can replace an existing message broker.

Of course, when instantiating the pattern, established software engineering practices need to be applied. For instance, related functionality should be grouped together. This also implies that the solution should be merged with existing components where possible. For instance, corresponding roles from different patterns can be mapped to the same component.

It can be expected that new components pose no difficulties to their introduction in the system, because they are independent from the context. Concerning the role bindings, however, more problems can arise. In some cases, it can be straightforward to select an existing component to fulfill a role, or to extend an existing component with new responsibilities. In general, however, the expectations imposed on a role might fundamentally conflict with other parts of the system. For example, consider a pattern dictating that communication between a new component and a role should be encrypted. If the element to which the role gets bound to requires that all its connections are plain-text to support auditing, then this clearly triggers a non-trivial conflict that prevents the pattern from being instantiated correctly. In general, it can be observed that conflicting expectations are the root cause of conflict among patterns, which are informally documented in pattern catalogs. Similarly, pattern languages (such as [30]) contain a cohesive set of patterns resolving each others residual goals as much as possible, while not introducing conflicts.

Activity 8. There are three distinct types of feedback, arising from the three parts of an architectural security pattern described in the Section 3. Each type of feedback introduces new elements in the requirements model.

The first type of feedback is the addition of new requirements assigned to the newly introduced components from the security pattern. These requirements describe the functionality of the new components and are necessary to ensure that all behavior implemented in the system is traceable to some requirement.

The second type of feedback is the set of expectations (i.e., constraints) imposed on the elements that play one of the roles from the pattern. It is then up to the architect to iterate over the architecture and assess whether the expectations (1) are already met by the component and the connectors involved in the corresponding role (e.g., it might be that the web server hosting the shop already supports SSL in order to securely transmit user credentials) or (2) require a refinement of these architectural elements so that they meet the expectations.

The third type of feedback is the set of residual goals. These goals are not assigned to a concrete element, because it is unspecified (from the pattern perspective) which element is responsible for them, or even how to achieve them. Therefore, they serve as candidate initial goals for refinement in a next iteration of the process. Obviously, it could be the case that the residual goals are already met by some sub-system of the architecture as is. In summary, the unbounded responsibility associated to residual goals distinguishes them from the previous type of feedback.

The goals generated by the feedback need to be reconciled with the goal tree. All goals introduced in this activity are prescribed by the pattern, and are necessary to ensure its correct functioning. As the pattern itself was selected to achieve the sub-goal selected in Activity 5 (e.g., $g_3$ in the example), it is natural to expect that the feedback goals need to be inserted as children of that sub-goal.

## 5 Conclusion

This paper has presented an elaboration of the Twin Peaks model, specific to co-developing secure software architectures and security requirements using patterns. The precise interaction points between architectural design and requirements engineering (in the context of software security) have been identified by decomposing the instantiation of an architectural security pattern into the interwoven process of (a) introducing new components and binding existing components to roles, and (b) introducing security behavioral requirements, expectations and residual goals. By pinpointing these interaction points, it is easier for the security-minded requirements engineer and software architect to predict where feedback might arise during the development process, and identify its root cause. Furthermore, this model can be leveraged to build more robust secure software engineering methods.

We plan to evaluate the secure twin peaks by applying it to other security requirements engineering methods such as SEPP [23], which is based on Jackson's problem frames [15]. Also, we would like to validate our approach in the context of industrial case studies. We believe that the explicit documentation of traceability links between architectural design and requirements analysis artifacts helps to (1) systematically evolve software systems, and (2) increase the applicability of security patterns in practice.

**Acknowledgements.** This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

## References


1. Bandara, A., Shinpei, H., Jürjens, J., Kaiya, H., Kubo, A., Laney, R., Mouratidis, H., Nhlabatsi, A., Nuseibeh, B., Tahara, Y., Tun, T., Washizaki, H., Yoshioka, N., Yu, Y.: Security patterns: Comparing modeling approaches. Technical Report 2009/06 (2009)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 1st edn. Addison-Wesley, Reading (1998)
3. Blakley, B., Heath, C., Members of The Open Group Security Forum: Security design patterns. The Open Group (2004)
4. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A system of Patterns. Wiley, Chichester (1996)
5. Côté, I., Heisel, M., Wentzlaff, I.: Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. International Journal of Cooperative Information Systems, World Scientific Publishing Company Special Issue of the Best Papers of the ECSA 2007 (December 2007)
6. Dougherty, C., Sayre, K., Seacord, R.C., Svoboda, D., Togashi, K.: Secure design patterns. Tech. Rep. CMU/SEI-2009-TR-010, Carnegie Mellon Software Engineering Institute (2009)
7. Fernandez, E.B., Larrondo-Petrie, M.M., Sorgente, T., Vanhilst, M.: A Methodology to Develop Secure Systems Using Patterns. In: Integrating Security and Software Engineering: Advances and Future Visions, pp. 107–126 (2007)
8. Giorgini, P., Mouratidis, H.: Secure tropos: A security-oriented extension of the tropos methodology. International Journal of Software Engineering and Knowledge Engineering 17(2), 285–309 (2007)
9. Haley, C.B., Laney, C.R., Moffett, D.J., Nuseibeh, B.: Security requirements engineering: A framework for representation and analysis. IEEE Transactions on Software Engineering 34(1), 133–153 (2008)
10. Haley, C.B., Moffett, J.D., Laney, R., Nuseibeh, B.: A framework for security requirements engineering. In: Proceedings of the International Workshop on Software Engineering for Secure Systems (SESS), pp. 35–42. ACM Press, New York (2006)
11. Haley, C.B., Nuseibeh, B.: Bridging requirements and architecture for systems of systems. In: Proceedings of the International Symposium on Information Technology (ITSim), vol. 4, pp. 1–8 (2008)
12. Hall, J.G., Rapanotti, L., Jackson, M.: Problem oriented software engineering: Solving the package router control problem. IEEE Transactions on Software Engineering 34(2), 226–241 (2008)
13. Heyman, T., Yskout, K., Scandariato, R., Joosen, W.: An analysis of the security patterns landscape. In: Proceedings of the International Workshop on Software Engineering for Secure Systems (SESS), pp. 3–10. IEEE Computer Society, Los Alamitos (2007)
14. Islam, S., Mouratidis, H., Jürjens, J.: A framework to support alignment of secure software engineering with legal regulations. Journal of Software and Systems Modeling (March 2010) (published online first)
15. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
16. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
17. Kienzle, D.M., Elder, M.C., Tyree, D., Edwards-Hewitt, J.: Security patterns repository (2002)

18. Mouratidis, H., Jürjens, J.: From goal-driven security requirements engineering to secure design. International Journal of Intelligent Systems – Special Issue on Goal-Driven Requirements Engineering 25(8), 813–840 (2010)
19. Mouratidis, H., Jürjens, J., Fox, J.: Towards a comprehensive framework for secure systems development. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 48–62. Springer, Heidelberg (2006)
20. Mouratidis, H., Weiss, M., Giorgini, P.: Modelling secure systems using an agent oriented approach and security patterns. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 16(3), 471–498 (2006)
21. Nhlabatsi, A., Nuseibeh, B., Yu, Y.: Security requirements engineering for evolving software systems: A survey. Journal of Secure Software Engineering 1(1), 54–73 (2009)
22. Nuseibeh, B.: Weaving together requirements and architectures. Computer 34(3), 115–117 (2001)
23. Schmidt, H.: A Pattern- and Component-Based Method to Develop Secure Software. Deutscher Wissenschafts-Verlag (DWV), Baden-Baden (April 2010)
24. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. Wiley & Sons, Chichester (2005)
25. Steel, C., Nagappan, R., Lai, R.: Core security patterns: Best practices and strategies for J2EE, web services, and identity management (2005)
26. van Lamsweerde, A.: From system goals to software architecture. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 25–43. Springer, Heidelberg (2003)
27. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Chichester (March 2009)
28. Weiss, M.: Modeling security patterns using NFR analysis. In: Integrating Security and Software Engineering, pp. 127–141. Idea Group, USA (2007)
29. Weiss, M., Mouratidis, H.: Selecting security patterns that fulfill security requirements. In: IEEE International Requirements Engineering Conference (2008)
30. Yoder, J., Barcalow, J.: Architectural patterns for enabling application security. In: Proceedings of the International Patterns Language of Programming (PLoP) Conference (1997)
31. Yskout, K., Scandariato, R., De Win, B., Joosen, W.: Transforming security requirements into architecture. In: Proceedings of the International Conference on Availability, Reliability and Security (AReS), pp. 1421–1428. IEEE Computer Society, Washington, DC (2008)

# Evolution of Security Requirements Tests for Service–Centric Systems

Michael Felderer, Berthold Agreiter, and Ruth Breu

Institute of Computer Science, University of Innsbruck, Austria
{michael.felderer,berthold.agreiter,ruth.breu}@uibk.ac.at

**Abstract.** Security is an important quality aspect of open service–centric systems. However, it is challenging to keep such systems secure because of steady evolution. Thus, security *requirements testing*, considering system changes is crucial to provide a certain level of reliability in a service–centric system. In this paper, we present a model–driven method to *system level security testing* of service–centric systems focusing on the aspect of requirements, system and test evolution. As requirements and the system may change over time, regular adaptations to the tests of security requirements are essential to retain, or even improve, system quality. We attach state machines to all model elements of our system- and test model to obtain consistent and traceable evolution of the system and its tests. We highlight the specifics for the evolution of security requirements, and show by a case study how changes of the attached tests are managed.

## 1 Introduction

Security requirements testing [1], i.e., the dynamic validation of security requirements such as authentication or integrity, is of high interest, especially for service–centric systems which are open. Basically, a *service–centric system* consists of a set of independent peers which provide and call services [2]. Service–centric systems are widely used, and they can be subject to modifications which may harm their security.

The work at hand shows how the evolution of a system, its infrastructure or its requirements influence security requirements and their tests.

According to [3], security requirements can be classified as follows:

- *Confidentiality* is the assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- *Integrity* is provided when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- *Authentication* is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization* provides access privileges granted to a user, program, or process.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 181–194, 2011.

- *Availability* guarantees timely, reliable access to data and information services for authorized users.
- *Non–repudiation* is the assurance that none of the partners taking part in a transaction can later deny of having participated.

This paper investigates the test evolution of security requirements, and regression testing based on the above classification. Note that our approach is independent of the concrete security requirements classification and can be based on other classifications, e.g., [4] as well. Defects or deviating behavior is normally detected by checking specific properties of a system during execution. These checks are called *assertions*. Security requirements are an adequate source for the definition of such assertions and provide additional constraints on functional requirements. Current approaches mostly tackle penetration or vulnerability tests and aim at the automatic *generation* of tests. We instead want to cover security requirement tests, called *security tests* in the sequel, from an acceptance point of view, i.e., testing whether security requirements are satisfied with regard to a functional requirement.

Our security test methodology takes into account that it is practically impossible to test security requirements completely, i.e., to provide evidence that security flaws are absent [5]. Instead of automatically generating test cases to be executed on the system, our approach allows for defining dedicated scenarios, i.e., functional tests, in which the compliance with security requirements is observed. We consider evolving scenarios and the consequences for security testing.

The remainder of this document is organized as follows. Section 2 details our testing approach by explaining our metamodel and the security test evolution process. The description of a case study in Section 3 applies the process to a real-world example. Finally, Sections 4 and 5 present related work and conclusions.

## 2 Test Evolution Methodology

On a very abstract level, we can distinguish among three different types of evolution for service–centric systems: evolution of *requirements*, evolution the *system*, and evolution of the *infrastructure*. For instance, the integration of a new security requirement is a requirements evolution, the adaptation of a service interface is a system evolution, and the modification of the implementation of a service or its deployment environment is an infrastructure evolution. Obviously, a change in the requirements may trigger changes in the system design or infrastructure. But a change in the design or in the infrastructure may also occur independently of a change in the requirements. Additionally, also changes on tests may occur.

In our approach, we attach a state machine to each changeable artifact, that defines its actual state, and triggers resp. receives events to compute new states. Changes in requirements are indicated by triggers on requirement elements and changes of the infrastructure or the system are indicated by triggers on service elements. Also tests itself have a state. Following the widely used classification in [6], the *type of a test* can be *evolution*, for testing novelties of the system,

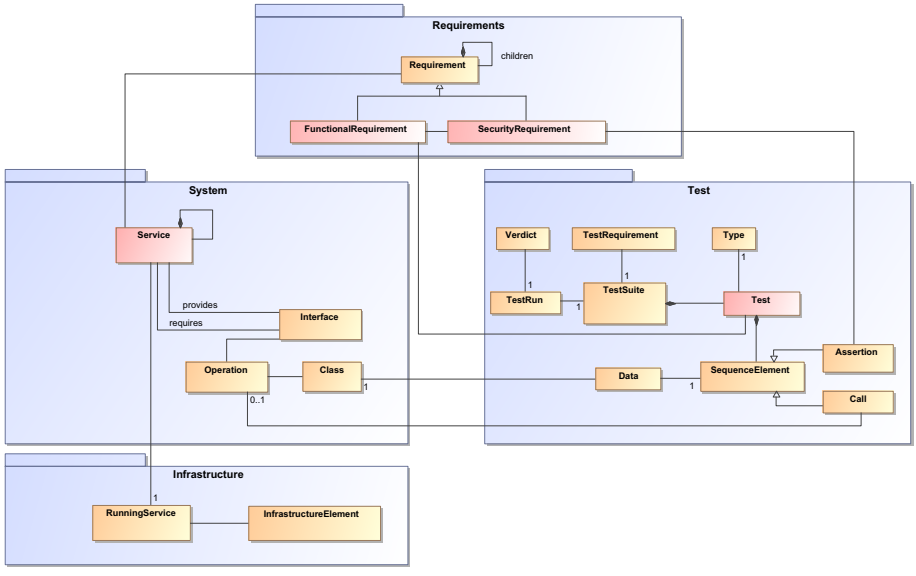



**Fig. 1.** Metamodel for Requirements, System, Infrastructure and Test

*regression*, for testing non–modified parts and ensuring that evolution did not unintentionally take place on other parts, *stagnation*, for ensuring that evolution did actually take place and changed the behavior of the system, and *obsolete*, for tests which are not relevant any more. Based on this test type, which is computed by states of model elements and a test requirement, a test suite for regression testing is determined.

In the following, we first explain the underlying metamodel (Section 2.1) and then the overall evolution process and its core process action of change propagation (Section 2.2).

### 2.1 Metamodel

Our model–based evolution process is based on the metamodel depicted in Fig. 1. It consists of the packages `Requirements`, `System`, `Infrastructure` and `Test`. In the remainder of this contribution we also refer to them as requirements, system and test model, respectively. The requirements model defines a hierarchy of `FunctionalRequirement` and `SecurityRequirement`. All requirements are connected to `Service` elements. A `FunctionalRequirement` is linked to `Test` and possibly `SecurityRequirement` elements; a `SecurityRequirement` is connected to `Assertion` and `FunctionalRequirement` elements. Therefore, a functional requirement has at least one test attached, and for all assigned security requirements an assertion in one of these tests exists. This guarantees that all specified security requirements of a functional requirement are also tested. Thus, in our approach *security requirements define security constraints on system functionality.*

The package `System` defines `Service` elements which contain provided and required `Interface` elements. Each interface consists of one or more `Operation` elements.

The package `Infrastructure` defines deployed services and their deployment environment by the elements `RunningService` and `InfrastructureElement`. To keep the description of our approach clear, we do not consider choreographies and orchestrations in the metamodel because they trigger change operations by analogy to plain services. Consequently, our view represents such workflows as `Service` elements.

The package `Test` defines all elements needed for testing service–centric systems. A `TestSuite` is a collection of `Test` elements. It is attached to `TestRequirement` elements, e.g., to select tests or define test exit criteria, and to `TestRun` elements assigning `Verdict` values to assertions. A `Test` has a `Type`, which can be either *evolution*, *stagnation*, *regression*, or *obsolete*, and consists of `SequenceElement` artifacts. A `SequenceElement` is either an `Assertion` defining how a verdict is computed or a `Call` element invoking a service operation and has some data assigned to the free variables of its assertions or calls.

The model elements `FunctionalRequirement`, `SecurityRequirement`, `Service`, and `Test` trigger further changes on other model elements and are highlighted in Figure 1. Each of these four types of model elements has a state machine attached, i.e., the state machines are defined at the meta layer, and instanciated for every concrete model element. The current state is stored in an attribute `state` which is not depicted in Fig. 1. The mechanism of mutual triggers by using these state machines is discussed in detail in the following section. At this point we only want to mention that this attribute and the type of tests can be used to define test requirements in the declarative language OCL [7].

### 2.2 Evolution Process

In this section we show how modifications of different model elements affect security tests and vice versa. Additionally we show, how tests are managed based on these modifications. The evolution process presented here, defines the steps that an arbitrary modification to the model induces to obtain an executable regression test suite for the updated model. Because our metamodel defines connections between the requirements model, the system model and the test model, we are able to propagate modifications to other relevant parts of the model via events and effects of state machines assigned to model elements. For example, if a service is undeployed, this mechanism enables to locate all service calls referring to this service and mark the according tests as "*not executable*". We first explain the overall evolution process, afterwards we present the change of model elements and the change propagation focusing on security requirements in more detail.

**Process Overview.** The evolution process is always initiated by a change of a model element. All model elements of the packages `Requirements`, `System` and `Test` can be modified. Changes can be either the creation of a model element (*add–operations*), the deletion of a model element (*remove–operations*), or the modification of a model element (*modify–operations*).

A change triggers an event involving state transitions in state machines which may generate new events. This may cause state changes in affected model elements, e.g., based on a service modification an assigned test state could transition from **executable** to **notExecutable**. This induces further changes to obtain consistent and executable models. We do not consider whether these changes are implemented automatically or manually. In practice, a semi–automatic update of the model elements, which highlights affected elements, may be the most practicable solution. After the change, the consistency and executability of the new model is checked which may imply new modifications if the check does not evaluate to true. The process of changing tests, requirements and services is repeated until the model is *consistent*, i.e., it contains no internal contradictions, and can therefore be transformed to executable test code.

Afterwards, tests are selected, i.e., a concrete test suite is computed from the set of all tests, based on test requirements. Test requirements define test selection criteria in OCL and typically consider the type of a test and the state of other model artifacts. A very general regression test selects all security tests that are supposed to pass, i.e., tests of type *evolution* or *regression* and is as follows:

```
context Model:
 Test::allInstances->select{ t |
  t.type='evolution' or t.type='regression'}
```

In the test execution phase, the tests of test suite are run. The result of the test execution is a test run which assigns verdicts to all assertions in the test suite. We abstract from the technical steps to make test models executable, which includes the transformation of test to executable test code and its execution against the running services, see [8]. If the test result is not as expected, e.g., if a test in the stagnation suite passes, the model has to be adapted again and the evolution process is executed iteratively.

The actions of the process are implemented by different tool sets. Because the process is *change-driven* [2], a *model repository* and a *modelling environment* are used for constructing an executable and consistent model. The model repository is metamodel–aware and manages model element changes using state machines as configuration input, i.e., according state changes are determined when an updated version of the model is committed. On the other hand, the modelling environment consists of a set of modelling tools to manipulate and validate models. The *test system* is then responsible for selecting and executing tests against the system under test.

**Change of model elements and change propagation.** All model elements can be changed, i.e., added, removed or deleted. The typical test design process, which is also reflected by the assigned state machines, starts with the definition of a new requirement. Afterwards, requirements are assigned to tests. The definition of tests may also induce changes to the system model because test model elements are connected to system model elements, e.g., for invoking service operations. Hence, an additional requirement may have impact on the rest of the model and raise the need for further changes. As we follow a test–driven development approach, the alignment of the requirement, test and system model is

an important step to keep the whole model consistent. Note that any part of the model can be subject to change and we also want to cover such cases with our approach, e.g., updates of infrastructural components or bug fixes to services.

Although changes may occur on all types of model elements of the metamodel depicted in Fig. 1, in the sequel we only consider changes to elements of type `FunctionalRequirement`, `SecurityRequirement`, `Service`, and `Test`. We do so because changes on arbitrary model elements of the packages `System` or `Infrastructure` can be regarded as changes to the assigned services. In the `Test` package, changes on sequence elements or data elements can be considered as changes on tests. Test suites resp. test runs are computed in the test selection resp. test execution phase but not changed by state transitions considered in this step.

For functional requirements, security requirements, services and tests, we define state machines describing how the states of these elements change. Each state transition defines a *trigger*, an optional *guard* condition and an optional *effect*, i.e., behavior to be performed when the transition fires. This means that the structure of transitions is of the following form: *trigger() [guard] / effect*. The effect part is used here to propagate changes by triggering state transition in other state machines. Simply put, the state machines show under which conditions a state transition occurs and which other elements are potentially affected.

Whenever a model element is removed, the assigned tests are assigned the type *obsolete*. In order to keep the state machines in this paper more readable, we do not consider the transitions to the obsolete state when removing model elements.

*Functional Requirements.* Fig. 2 shows the life cycle of `FunctionalRequirement` elements defined as a state machine.

When a new functional requirement $r$ is created, it is in state **defined**. As soon as the requirement is assigned to a test $t$, its state changes to **assigned**.

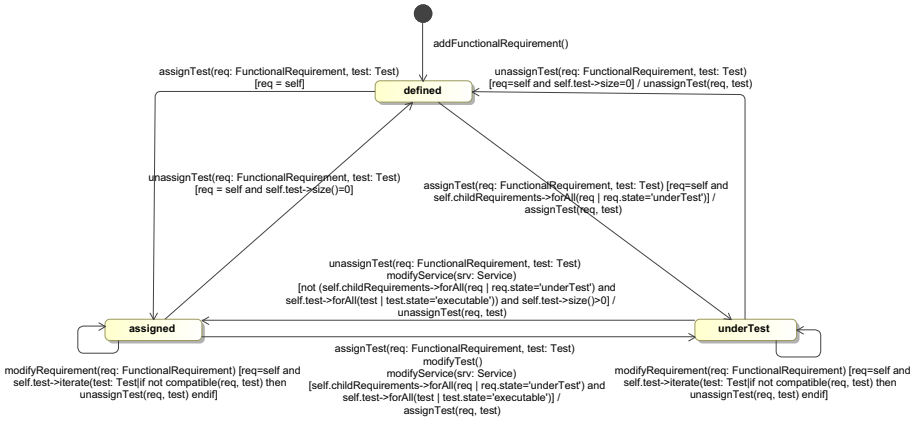



**Fig. 2.** State machine describing the life cycle of a `FunctionalRequirement`

The requirement can only reach the state **underTest** if all of its associated subrequirements are also in the same state. The requirement is *assigned* to at least one test and all assigned tests are *executable*. This transition can be fired either by assigning a test to $r$ or by modifying assigned tests or their services so that all tests become executable. This condition also takes into account that requirements are organized hierarchically. When $r$ is modified in state **assigned** or **underTest** the compatibility to all assigned tests has to be checked. If the requirement and a test are not compatible anymore, the connection among the two is removed, i.e., the *unassignTest()*-operation is triggered. The trigger *modifyRequirement* also triggers transitions in the state machine of tests as depicted in Fig. 5. The implications of this trigger are described in the section on tests. Furthermore, a requirement under test is always *assigned* to at least one test. If this is not fulfilled anymore, its state changes back to **defined**. In the subsequent semi–formal definitions, we determine the predicates *assigned* and *compatible* used as guards in the state machines.

**Definition 1 (Test–Requirement Assignment).** *The tuple set* $\mathcal{TR} \subseteq \mathcal{R} \times \mathcal{T}$ *denotes all assignments of tests to requirements. For every tuple* $(r, t) \in \mathcal{TR}$ *we say that* $t$ *is* assigned *to* $r$*. When a requirement* $r$ *is assigned to a test* $t$*, the tuple* $(r, t)$ *is added to the set* $\mathcal{TR}$*. We further define the function* $assigned_{TR}$ *as follows:*
$assigned_{TR} : \mathcal{R} \times \mathcal{T} \rightarrow Bool$*, where* $assigned_{TR}(r, t) :=$ **true** *if* $(r, t) \in \mathcal{TR}$ *and* **false** *otherwise.*

**Definition 2 (Test Compatibility).** *A test* $t \in \mathcal{T}$ *and a requirement* $r \in \mathcal{R}$ *are* compatible *if the test* $t$ *validates the requirement* $r$*.*

The assessment whether a test validates a requirement is a manual task because the requirements are always specified in an informal way. However, the compatibility only needs to be checked when a new test–requirement assigment is created, or when one of these elements is directly changed.

*Security Requirements.* Security requirements are different in the sense that we do not directly assign tests to them. Instead, a security requirement is assigned to a functional requirement which has to comply with it [9]. This implies that the adherence to security requirements needs to be part of the tests assigned to the functional requirement. Fig. 3 shows the life cycle of `SecurityRequirement` elements defined as a state machine.

When a new security requirement *sr* is created, it is in state **defined**. As soon as *sr* is connected to a functional requirement, *sr* moves into state **assigned**. Upon the assignment of the functional requirement with a test, *sr* changes its state to **underTest** because it is assumed that the test also checks for the adherence to this security requirement. This assumption is feasible because our security tests are functional. Whenever the connection between *sr* and its functional requirement is removed, *sr* goes back to state **defined**. If a requirement is changed and *sr* is currently assigned to a functional requirement, the compatibility to this requirement is re-evaluated. If the compatibility is not given anymore

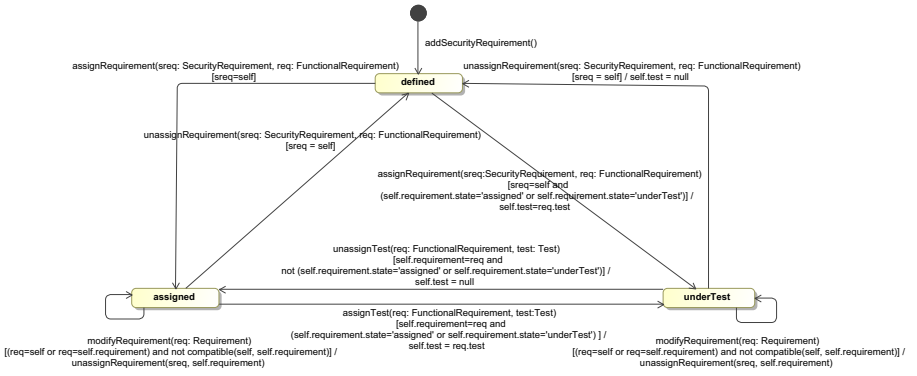



**Fig. 3.** State machine describing the life cycle of a `SecurityRequirement`

*sr* is disconnected from its functional requirement. Note that this modification takes into account both, the modification of a functional requirement and the modification of a security requirement.

**Definition 3 (Requirement Compatibility).** *A functional requirement r and a security requirement sr are* compatible *if in all cases where r is satisfied, also sr is satisfied.*

To clarify the meaning of requirement compatibility consider the following example. A functional requirement states that registered users are allowed to post messages on a bulletin board, and their usernames shall appear next to their posts. Consequently, one of the assigned security requirements states that users need to be authenticated. So far, these two requirements are *compatible* with each other. However, if we change the functional requirement so that for guest users the string "Guest" shall appear instead of the username, this requirement is not *compatible* to its assigned security requirement anymore. The reason for this is that the security requirement requires authenticated users, but the functional requirement also allows unauthenticated guest users. In other words, the functional requirement would be satisfied despite a violated security requirement.

Note that also in this case, the assertion of compatibility has to be checked manually due to the informal specification of requirements.

*Services.* Similar to requirements, every service $s \in \mathcal{S}$ has a state. Fig. 4 depicts the life cycle of services and its state transitions.

When a service is newly defined, it is in state **new**. The state changes to **specified** as soon as an interface is added to the service, because from that point on it can be referenced by tests. After a service is implemented, it can be deployed, which changes the state of the service to **executable**. Deployment creates a `RunningService` element and connects the service to it. However, deployment is only allowed if the service provides at least one interface. The deployment transition further triggers the operation *modifyService* which allows other model elements to react on the deployment of the service. Note that this operation also

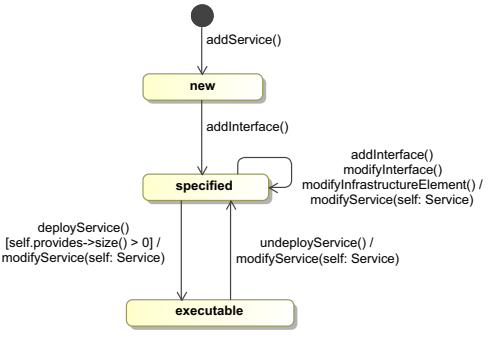



**Fig. 4.** State machine describing the life cycle of `Service` elements

receives the current service object (`self`) as input parameter. Before the interface or the infrastructural elements of a service can be modified again, the service has to be undeployed. Executable services always provide at least one interface. Every modification to a service, its provided or required `Interface` elements, or to subordinate services being part of this service is propagated to *assigned* tests via the trigger *modifyService*. The implications of this trigger are described next.

*Tests.* Every test $t \in \mathcal{T}$ has a state as well. As explained above, state transitions of tests are not only caused by direct modifications on elements of type `Test` but also triggered by requirements and services. The possible states and transitions of tests are depicted in Fig. 5.

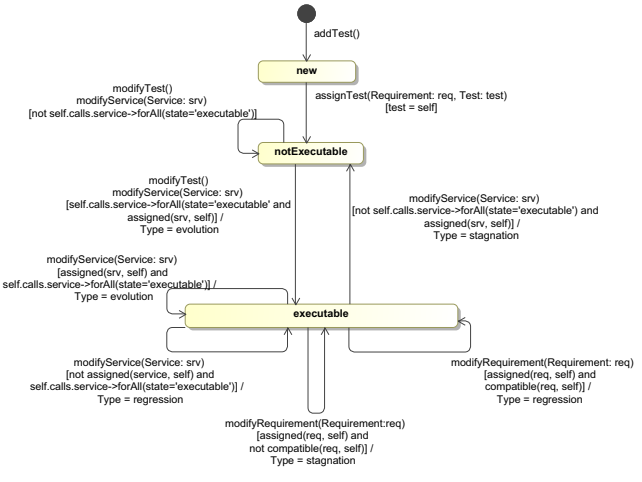



**Fig. 5.** State machine describing the lifecylce of `Test` elements

**Definition 4 (Test–Service Assignment).** *The tuple set $\mathcal{TS} \subseteq \mathcal{S} \times \mathcal{T}$ denotes all assignments of tests to services. When a `Call` element, contained in a test $t \in \mathcal{T}$, is associated to an `Operation` element of a service $s \in \mathcal{S}$, the tuple $(s, t)$ is added to $\mathcal{TS}$. We say that $s$ is* assigned *to $t$ if $(s, t) \in \mathcal{TS}$.*

*We further define the function* $assigned_{ST}$ *as follows:* $assigned_{ST} : \mathcal{S} \times \mathcal{T} \rightarrow Bool$*, where* $assigned_{ST}(s,t) :=$ **true** *if* $(s,t) \in \mathcal{TS}$ *and* **false** *otherwise.*

Note that `Call` elements can only be associated to `Operation` elements if their signature is compatible in terms of parameters and data types.

A test is in state **new** as long as it has no requirements assigned by the operation *assignTest*. Note that the same operation is also a trigger in the `FunctionalRequirement` state machine, i.e., this assignment causes a state transition for both, tests and requirements. After the assignment of a requirement, a test is in state **notExecutable** until all *assigned* `Service` elements are in state **executable**. When a test or an *assigned* service is modified such that all *assigned* services of a test are in state **executable**, the test also gets the state **executable**. A guard condition checks for adherence to this rule.

`Test` elements have an attribute `Type` assigned which can be used for test selection. Depending on the modifications to the model, the `Type` of a test is updated. If the state of a test goes from **notExecutable** to **executable** its `Type` is set to *evolution* because the test is the result of an evolution step. The same is true if a test is currently in state **executable** and one of its assigned services is subject to a modification. On the other hand, if a service is modified but the current test under consideration is not assigned with that service, the `Type` of the test is changed to *regression* because the test should not be affected by this modification.

Also the modification of requirements influences the `Type` attribute of executable tests. When a requirement assigned to the current test under consideration is modified such that the test and the requirement are incompatible, the type is set to *stagnation* because the test should fail now. If, on the other hand, the requirement and the test are still compatible, the type is set to *regression*.

## 3 Case Study

In this section we demonstrate our evolution methodology by a home gateway application. We provide only a short overview of the application as far as needed to discuss the evolution scenario. For more details, we refer to [10].

A home gateway is an appliance which is connected to an operator's network and provides certain services to customers, e.g., internet connection or video surveillance of rooms. As a service–centric system, it consists of the peers *Access Requestor* (AR), *Home Gateway* (HG), *Policy Enforcement Point* (PEP) and the *Policy Decision Point* (PDP). Following service–centric principles [2], each peer can be considered as a service providing and requiring interfaces defining the contents of information exchange. The AR is the client application to establish the connection to a HG and to use its functionalities. The HG is a device installed at the home of a customer controlling access to different resources. The enforcement of who is allowed to access which resources on the network is made by an internal component of the HG called PEP. The PEP receives the policy it has to enforce for a specific AR by the PDP.

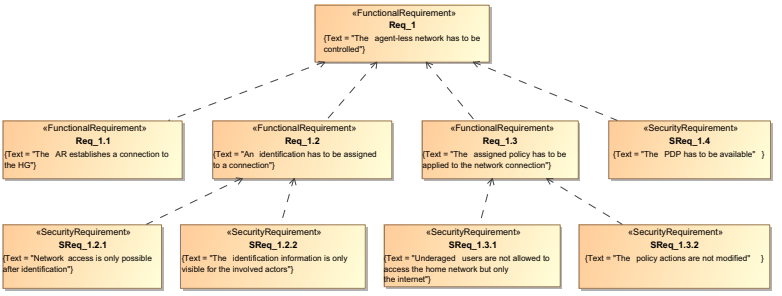



**Fig. 6.** Requirements model of the home networking case study

In Fig. 6 the requirements of the home gateway application are depicted.

The requirements taxonomy contains functional and security requirements for controlling the network access including security requirements for authentication (*SReq_1.2.1*), confidentiality (*SReq_1.2.2*), availability (*SReq_1.4*), authorization (*SReq_1.3.1*), and integrity (*SReq_1.3.2*).

We now demonstrate the effects by the addition of a security requirement and the subsequent modification of the corresponding functional requirement by explaining the tranistions in the state machines for functional requirements, security requirements and tests.

Initially, we assume that the agent–based network access is controlled and that all requirements but *SReq_1.4* are defined and in state **underTest**. Each requirement has at least one executable test assigned, and all services are in state **executable**. The assigned test for *Req_1* is *Test_1* testing an agent–based access scenario similar to the test depicted in Fig. 7 (details of the test are not relevant here).

In the first step, we add the security requirement *SReq_1.4* by *addSecurityRequirement()* which is then in state **defined**. Afterwards, the security requirement *SReq_1.4* is assigned to the requirement *Req_1* by the operation *assignRequirement(SReq_1.4,Req_1)* and moves into state **underTest** because the assigned requirement, *Req_1*, is in state **underTest**.

In the second step, *Req_1* is modified by changing it from an agent–based to an agent–less network as depicted in Fig. 6. This is caused by the event *modifyRequirement(Req_1)*. Assuming that the assigned test and the requirement are not compatible any more, the *unassignTest(Req_1,Test_1)* event is triggered and *Req_1* goes to state **defined** and the assigned security requirement *SReq_1.4* to state **assigned**. The unassigned test *Test_1* resumes in state *executable* and is assigned the type *stagnation* because it is not compatible with the modified requirement *Req_1*.

In the third step, we want to define a test which is compatible with *Req_1* again. The test *Test_2* depicted in Fig. 7 is defined via *addTest*. *Test_2* checks for agent–less network control, integrating the new service *NetworkFilter*. Then, *Req_1* and *Test_2* are assigned by *assignTest(Req_1, Test_2)*. Afterwards, *Req_1* is in state **assigned**, the security requirement *SReq_1.4* in state **underTest** and *Test_2* in state **notExecutable**. To make the test executable, the test is

modified by adding service calls and assertions. Also the new service *NetworkFilter* is added to the system model. After all modifications, the test is defined as in Fig. 7 (details of the test sequence itself are not relevant). After a *modifyTest()* triggered by the modifications on the test and as soon as the new service *NetworkFilter* has been specified and is executable, the event *modifyService(NetworkFilter)* triggers the transition of *Test_2* to the state **executable** and its type becomes *evolution*. This also causes the state of *Req_1* to transition from state **assigned** to state **underTest**.

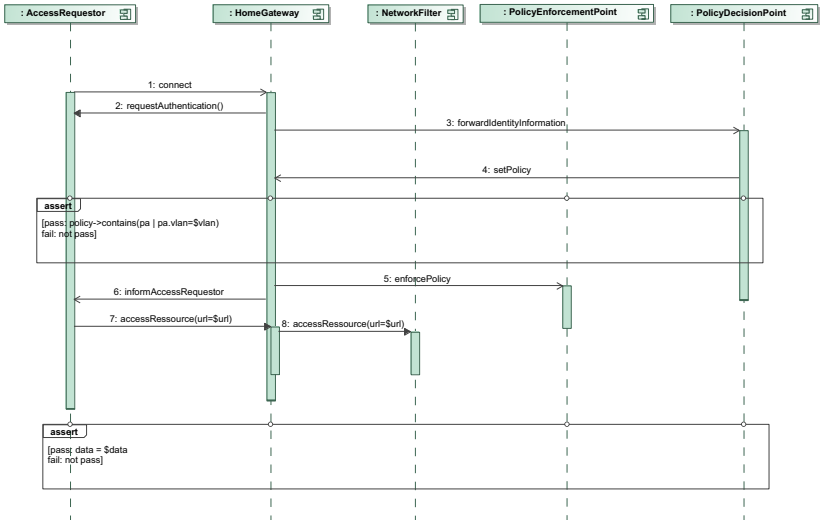



**Fig. 7.** Test for the agent–less network control

Finally, the test model is consistent and tests can be selected and then executed.

## 4 Related Work

The evolution of software and its implications for security has attained much attention in the last years. Many approaches consider the evolution of the system, either focusing on the source code or on the system model [11]. But only few approaches highlight the role of tests in the evolution process.

In [12] the interplay between software testing and evolution is investigated considering the influence of evolving tests on the system. The paper introduces *test–driven refactoring*, i.e., refactoring of production code induced by the restructuring of the tests. As in [12], we consider the relationship of tests and system evolution, but on the model–level and not on the code–level. To the best of our knowledge, there is actually no approach that considers the evolution of security tests.

Regression testing is one major activity to maintain evolving systems [13]. Most regression testing approaches are code–based [14]. Only a few model–based

regression testing approaches have been considered [15,16]. We integrate tests of security requirements into this process which has not been considered in any other model–based regression testing approach yet.

Several approaches to model–based testing of security requirements have been developed [17,18,19] but none of them considers evolution aspects.

For service–centric systems there are only a few approaches considering security testing [20,21] and regression testing [22]. None of these approaches combines them to regression testing of security requirements for service–centric systems. Our approach handles security testing for evolving requirements by regression testing.

## 5 Conclusions

We presented a methodology to maintain evolving security test models by attaching state machines to all model elements. The result is an evolution process overarching different system aspects which is initiated by adding, modifying or deleting model elements. These actions trigger change events and fire transitions in state machines. The model is then changed manually or automatically until it is consistent and executable. Based on a test requirement considering the actual state of model elements, a regression test suite is selected and executed. We have applied our methodology in a home networking case study. In the evolution process, we highlighted the special role of security requirements as requirements attached to functional requirements. By attaching security requirements to functional requirements, we are able to review the evolution of security requirements in connection with the evolution of functional requirements. The systems' compliance with security requirements is shown by the assigned functional requirements and only few additional effort is needed for the evolution management of security tests.

As a model–based approach to regression test selection our approach has advantages to test selection on the code level [16]. We are able to estimate the effort for testing earlier, tools for regression testing can be largely technology independent, the management of traceability at the model level is more practical because it enables the specification of dependencies at a higher level of abstraction, and no complex code analysis is required. Compared to other model–based approaches to regression testing, we model tests separately from the system. Our methodology keeps the test model consistent and optimal under consideration of security issues.

In the future, we will integrate risk–based security testing into our approach to also test negative security requirements and generalize it to non–functional properites other than security.

## Acknowledgements

This work is sponsored by the SecureChange project (EU grant number ICT-FET-231101), the MATE project (FWF project number P17380), and QE LaB – Living Models for Open Systems (FFG 822740).

## References


1. Bishop, M.: Computer Security: Art and Science. Addison Wesley, Reading (2003)
2. Breu, R.: Ten Principles for Living Models: A Manifesto of Change-Driven Software Engineering. In: CISIS 2010 (2010)
3. CNSS Instruction Formerly NSTISSI: National Information Assurance Glossary, Committee on National Security Systems, vol. 4009 (June 2006)
4. Common Criteria for Information Technology Security Evaluation, `http://www.commoncriteriaportal.org/thecc.html` [accessed: August 16, 2010]
5. Pfleeger, S., Cunningham, R.: Why measuring security is hard. IEEE Security Privacy PP(99) (2010)
6. Leung, H., White, L.: An approach for selective state machine based regression testing. In: Proceedings of Conference on Software Maintenance (1989)
7. OMG: Object Constraint Language Version 2.0 (2006)
8. Felderer, M., Fiedler, F., Zech, P., Breu, R.: Flexible Test Code Generation for Service Oriented Systems. In: QSIC 2009 (2009)
9. Hafner, M., Breu, R.: Security Engineering for Service–Oriented Architectures. Springer, Heidelberg (2008)
10. Felderer, M., Agreiter, B., Breu, R., Armenteros, A.: Security testing by telling teststories. In: Modellierung 2010 (2010)
11. Mens, T., Demeyer, S. (eds.): Software Evolution. Springer, Heidelberg (2008)
12. Moonen, L., van Deursen, A., Zaidman, A., Bruntink, M.: On the interplay between software testing and evolution and its effect on program comprehension. In: Software Evolution (2008)
13. Gorthi, R.P., Pasala, A., Chanduka, K.K., Leong, B.: Specification-based approach to select regression test suite to validate changed software (2008)
14. von Mayrhauser, A., Zhang, N.: Automated regression testing using dbt and sleuth. Journal of Software Maintenance 11(2) (1999)
15. Farooq, Q.u.a., Iqbal, M.Z.Z., Malik, Z.I., Nadeem, A.: An approach for selective state machine based regression testing. In: A-MOST 2007 (2007)
16. Briand, L.C., Labiche, Y., He, S.: Automating regression test selection based on uml designs. Inf. Softw. Technol. 51(1) (2009)
17. Julliand, J., Masson, P.A., Tissot, R.: Generating security tests in addition to functional tests. In: AST 2008 (2008)
18. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, p. 412. Springer, Heidelberg (2002)
19. Wimmel, G., Jürjens, J.: Specification-based test generation for security-critical systems using mutations. LNCS. Springer, Heidelberg (2002)
20. Barbir, A., Hobbs, C., Bertino, E., Hirsch, F., Martino, L.: Challenges of testing web services and security in soa implementations. In: Test and Analysis of Web Services. Springer, Heidelberg (2007)
21. Cova, M., Felmetsger, V., Vigna, G.: Vulnerability Analysis of Web–Based Applications. In: Testing and Analysis of Web Services (2007)
22. Penta, M.D., Bruno, M., Esposito, G., Mazza, V., Canfora, G.: Web services regression testing. In: Test and Analysis of Web Services (2007)

# After-Life Vulnerabilities: A Study on Firefox Evolution, Its Vulnerabilities, and Fixes*

Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen

Università degli Studi di Trento, I-38100 Trento, Italy
{massacci,neuhaus,vhnguyen}@disi.unitn.it

**Abstract.** We study the interplay in the evolution of Firefox source code and known vulnerabilities in Firefox over six major versions (v1.0, v1.5, v2.0, v3.0, v3.5, and v3.6) spanning almost ten years of development, and integrating a numbers of sources (NVD, CVE, MFSA, Firefox CVS). We conclude that a large fraction of vulnerabilities apply to code that is no longer maintained in older versions. We call these *after-life vulnerabilities*. This complements the Milk-or-Wine study of Ozment and Schechter—which we also partly confirm—as we look at vulnerabilities in the reference frame of the source code, revealing a vulnerabilitiy's future, while they looked at its past history. Through an analysis of that code's market share, we also conclude that vulnerable code is still very much in use both in terms of instances and as global codebase: CVS evidence suggests that Firefox evolves relatively slowly.

This is empirical evidence that the software-evolution-as-security solution—patching software and automatic updates—might not work, and that vulnerabilities will have to be mitigated by other means.

## 1 Introduction

The last decade has seen a significant push towards security-aware software development processes in industry, such as Microsoft's SDL [1], Cigital's BSIMM [2], and many other processes that are specific to other software vendors.

In spite of these efforts, software is still plagued by many vulnerabilities and the current trend among software vendors is to *counter the risk of exploits by software evolution*: security patches are automatically pushed to end customers, support for old versions is terminated, and customers are pressured to move to new versions. The idea is that, as new software instances replace the old vulnerable instance, the eco-system as a whole progresses to a more secure state.

Beside the social good (improvement of the security of the ecosystem) this model also has some significant economic advantages for software vendors: the simultaneous maintenance of many old versions is simply too costly to continue.

* This work is supported by the European Commission under projects EU-FET-IP-SECURECHANGE and EU-FP7-IST-IP-MASTER.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 195–208, 2011.

We call the time after which a software product is no longer supported that product's *end-of-life*[1]. Of course, having reached the end-of-life doesn't mean that the product is no longer in use: entire conferences are devoted to the issue of maintaining and wrapping legacy code. This product existence in *after-life* can have interesting security implications for the security of the ecosystem because a product should only reach end-of-life when

- the number of unpatched vulnerabilities in that product is small; or
- the number of active users of after-life code is small.

The key question that we try to address in this paper is whether there is some empirical evidence that software-evolution-as-a-security-solution is actually a solution, i.e., leads to less vulnerable software over time.

In this paper, we report our empirical findings on a major empirical study on the evolution of Mozilla Firefox. After studying 899 vulnerabilities in Mozilla Firefox from versions v1.0 to v3.6, we find:

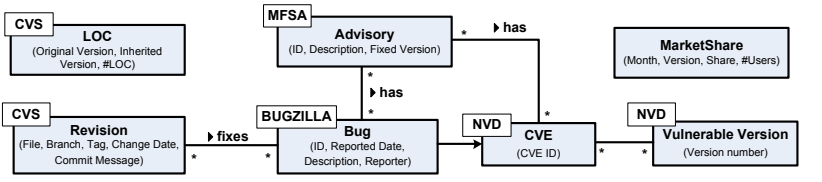
1. Many vulnerabilities for Firefox versions are discovered when these versions are well in their after-life. These *after-life vulnerabilities* account for at least 30% for Firefox v1.0.
2. Most disturbingly, there are still *many after-life instances* of Firefox.
3. There exists a statistically significant difference between *local* vulnerabilities (*i.e.*, those discovered and fixed in a same version) and *inherited* vulnerabilities (*i.e.*, those discovered in version $x$, but applicable also to some version $y < x$) or *foundational* (*i.e.*, inherited from the first version). By pure statistics change, foundational vulnerabilities are significant more than they should be, meanwhile, inherited on are significant less than they should be.
4. A possible explanation of this phenomenon is *slow code evolution*: a lot of code is retained between released versions. Code originating in Firefox v1.0 is over 40% of the code in v3.6.

These findings show that Ozment and Schechter's notion of foundational vulnerabilities [3] can be generalized to the more general case of *inherited* vulnerabilities. Ozment and Schechter's limitation of to foundational vulnerabilities (instead of the more general class of inherited ones) might be due to their particular case study, or might be due to a methodological specialty, which we analyze in §5. However we are not able to make sharp conclusions yet: foundational vulnerabilities are significantly more than they should be but inherited ones are significantly less than they should be. None of them is the majority.

These results could be explained by the quality of the slow evolution where code of v1.0 (*i.e.*, foundational code) largely dominates the inherited part: the good fraction of the code is the one that is retained.

The existence of many after-life vulnerabilities (contribution 1) and many after-life survivors (contribution 2), in spite of producers' efforts to eradicate

[1] Some companies provide security patches but not regular bugfixes for out-of-date products. In this case end-of-life is the point when all maintenance stops.




Rectangles denote tables; icons at the top left corner of tables denote the source database. Two additional tables (not shown) are used to trace back the evolution of individual lines of code from CVS and Mercurial, and Firefox's market share.

**Fig. 1.** Simplified database schema

them, is an interesting phenomenon by itself. It shows that security pushes during deployment and the active life of the project cannot eliminate vulnerabilities.

Further, it shows that software-evolution-as-security-solution is not really a solution: the fraction of vulnerable software instances that is globally present may be too high to ever attain herd immunity [4].

The remainder of this paper is structured as follows. First, we describe our methods of data acquisition and experiment setup (§2), after which we introduce important concepts such as foundational or inherited vulnerability (§3). Next, we report our detailed findings on after-life vulnerabilities (§4), revisit and challenge the "Milk or Wine" findings (§5), and present the data on software evolution (§6). Finally, we analyze the threats to validity (§7) and conclude by discussing our findings (§9).

## 2 Data Acquisition and Experiment Setup

Here we briefly describe how to acquire aggregated vulnerability data by the integration of a number of data sources; a more complete description is contained in a previous work [5].

Fig. 1 presents the abstract schema of our database and shows our data sources. The *Advisory* table holds a list of Mozilla Firefox-related Security Advisories (MFSA).[2] Rows in this table contain data extracted from MFSA (*e.g.*, vulnerability title, announced date, fixed versions); it also maintains links to Bugzilla and the CVE, captured in tables *Bug* and *CVE*, respectively. The *Bug* table holds Bugzilla-related information, including bugID, reported date, and description. Some Bugzilla entries are not publicly accessible, such as reported but unfixed vulnerabilities. In these cases, the reported date is set to the corresponding advisory's announced date. The *VulnerableVersion* table keeps the National Vulnerability Database (NVD) reported vulnerable version list for each CVE. The *Revision* table holds information of those code check-ins that fix bugs.

There are two additional tables that help to study the evolution of Firefox in term of code base, table *LOC*, and in term of global usage, table *MarketShare*. In

[2] Not all MFSAs relate to Firefox; some relate to other Mozilla products.

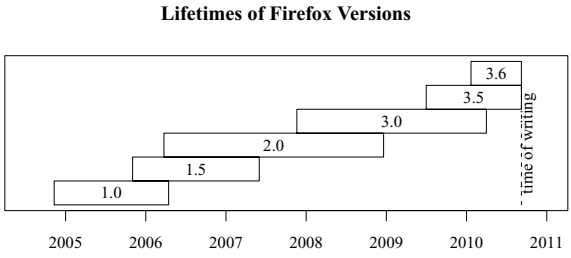



**Fig. 2.** Release and retirement dates for Firefox versions

order to estimate the market share and hence the number of users of a particular Firefox version, we collect data on the global evolution of Firefox usage since November 2007 from Net MarketShare[3]. This website collects browser data from visitors of about 40,000 websites. It provides monthly market share data free of charge, but does not show the absolute number of users. For this, we use Internet World Stats[4], which gives numbers of internet users since December 1995. This website does not provide data for every month from 2007, hence we use linear interpolation to calculate the missing values.

## 3 Versions and Vulnerabilities

We looked at six major Firefox versions, namely v1.0, v1.5, v2.0, v3.0, v3.5 and v3.6. Their lifetimes[5] can be seen from Fig. 2. At the time of writing, the versions of Firefox that were actively maintained were v3.5 and v3.6. Therefore, the rectangles representing version v3.5 and v3.6 actually extend to the right. There is very little overlap between two versions that are one release apart, *i.e.*, between v1.0 and v2.0, v1.5 and v3.0, v2.0 and v3.5, or v3.0 and v3.6. This is evidence of a conscious effort by the Mozilla developers to maintain only up to two versions simultaneously.

This picture seems to show a quick pace of software evolution as in the time span of slightly more than an year, we have a new version replacing an old one. As we shall see, this is not the case.

In order to find out to which version a vulnerability applies, we look at the NVD entry for a particular vulnerability and take the earliest version for which the vulnerability is relevant. For example, MFSA 2008-60 describes a crash with memory corruption. This MFSA links to CVE 2008-5502, for which the NVD asserts that versions v3.0 and v2.0 are vulnerable. The intuitive expectation (confirmed by the tests) is that the vulnerability was present already in v2.0 and that v3.0 inherited it from there.

Table 1 shows the breakdown of cumulative vulnerabilities affecting to each version of Firefox. An entry at column $x$ and row $y$ indicates the number of

[3] `http://www.netmarketshare.com/`
[4] `http://www.internetworldstats.com/emarketing.htm`
[5] `https://wiki.mozilla.org/Releases`

**Table 1.** Breakdown of Cumulative Vulnerabilities in Firefox

The Total row shows the cumulative vulnerabilities for each version. Other rows display the vulnerabilities that a version inherits from retrospective ones.

| Inherit from | Affected Version v1.0 | v1.5 | v2.0 | v3.0 | v3.5 | v3.6 |
|---|---|---|---|---|---|---|
| v1.0 | 334 | 255 | 184 | 142 | 45 | 13 |
| v1.5 | — | 227 | 119 | 15 | 0 | 0 |
| v2.0 | — | — | 149 | 23 | 1 | 1 |
| v3.0 | — | — | — | 102 | 35 | 5 |
| v3.5 | — | — | — | — | 73 | 41 |
| v3.6 | — | — | — | — | — | 14 |
| Total | **334** | **482** | **452** | **282** | **154** | **74** |

vulnerabilities applied to version $x$ inherits from version $y$. Obviously there are vulnerabilities applying for more than one version. Thus, they are counted several times in this table. It is a mistake if we sum all numbers and conclude about the total vulnerabilities of Firefox.

Based on versions that a vulnerability affects to, it can be classified into following sets:

- *Inherited vulnerabilities* are ones affect to a series of consecutive versions.
- *Foundation vulnerabilities* are inherited ones, but apply also to v1.0.
- *Local vulnerabilities* are known to apply to only one version.

Our definition of foundational vulnerability is weaker (and thus more general) than the one used by Ozment and Schechter [3]. We do not claim that there exists some code in version v1.0 that is also present in, say, v1.5 and v2.0 when a vulnerability is foundational. For us it is enough that the vulnerability applies to v1.0, v1.5 and v2.0. This is necessary because many vulnerabilities (on the order of 20–30%) are not fixed. For those vulnerabilities it is impossible, by looking at the CVS and Mercurial sources without extensive and bias-prone manual analysis, to identify the code fragment from which they originated.

When we tabulate which vulnerabilities affect which versions, we can in theory get $N = 2^6 - 1$ different results, depending on which of the six versions is affected.[6] If all $N$ combinations were equally likely, vulnerable versions separated by not vulnerable versions would be commonplace: there are $2^n - 1 - \sum_{k=0}^{n}(n - k) = 2^n - n(n+1)/2 - 1$ such arrangements, which we call *regressions*. For $n = 6$, that is 42 out of 63.

Once we exclude regressions, there are 6 combinations in which a vulnerability only applies to a single version, 5 combinations with foundational vulnerabilities and 10 inherited but not foundational vulnerabilities.

Another definition that we will use in this paper is the notion of *After-life Vulnerability*: a vulnerability which applies to a version which is no longer

[6] The only combination that is not allowed is when no version is affected. If no version is affected, why is it a vulnerability?

**Table 2.** Examples of inherited and foundational vulnerabilities

A vulnerability is *inherited* when it appears first in some version and affects a nonempty string of consecutive versions (and no others); it is also *foundational* if it applies to the first version, v1.0. Non-consecutive affected versions point to a regression, which we did not find in the data.

| Affected Version | | | | | | |
|---|---|---|---|---|---|---|
| v1.0 | v1.5 | v2.0 | v3.0 | v3.5 | v3.6 | Remark |
| | | × | × | | | Inherited (from v2.0) |
| | | × | × | × | | Inherited (from v2.0) |
| | × | × | × | × | × | Inherited (from v1.5) |
| × | × | × | × | | | Foundational |
| × | | | | | | Local |
| | | | × | | | Local |
| | × | | × | | | Regression (rare) |

maintained. Since a vulnerability can apply to many versions, the same vulnerability can be an after-life vulnerability for v1.0 and a current vulnerability for v3.6.

## 4 After-Life Vulnerabilities and the Security Ecosystem

Our first research question was whether many vulnerabilities were discovered that affected after-life versions.

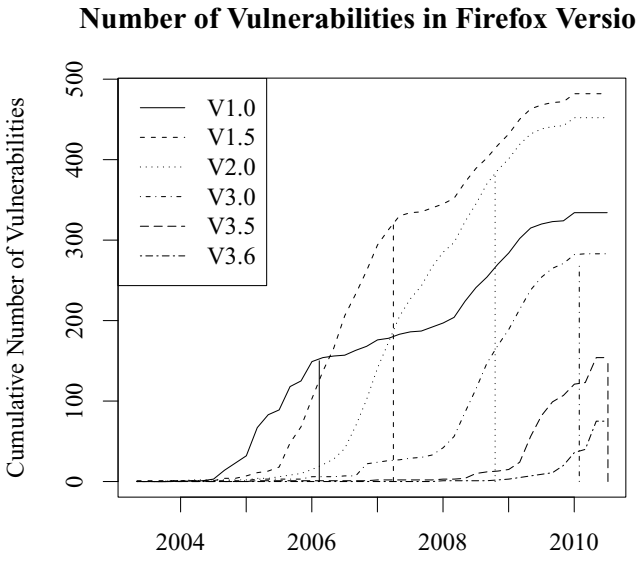
As Fig. 3 shows, this is indeed the case. This figure shows the cumulative number of vulnerabilities for each of the six Firefox versions versus time. We also marked the respective version's end of life. If there were no or just a few vulnerabilities discovered after end-of-life, the slopes of the curves should be zero, or close to zero, after that point. Since this is clearly not the case, after-life vulnerabilities do in fact exist.

In order to evaluate the impact of after-life vulnerabilities we consider the market share of the various versions and the attack surface of code that is around. The intuition is to calculate the LOC on each version that are currently available to attackers, either by directly attacking that version or by attacking the fraction of that version that was inherited in later versions.

Let $users(v, t)$ be the number of users of Firefox version $v$ at time $t$, and let $loc(p, c)$ be the number of lines of code that the $c$urrent version $c$ has inherited from the $p$revious version $p$. Then the total number of lines of code in version $c$ is $\sum_{1 \le p \le c} loc(p, c)$. In order to get an idea how much inherited code is used, we define a measure *LOC-users* as

$$LOC\text{-}users(v, t) = \sum_{1 \le p \le v} users(p, t) \cdot loc(p, v). \tag{1}$$

This is an approximation because the amount of code inherited into version $v$ varies with time, therefore, $loc(p, v)$ is time-dependent. In this way, we eliminate transient phenomena for this high-level analysis.




Cumulative number of vulnerabilities for the various Firefox versions. End-of-life for a version is marked with vertical lines. As is apparent, the number of vulnerabilities continues to rise even past a version's end-of-life.

**Fig. 3.** Vulnerabilities discovered in Firefox versions

Fig. 4 shows the development of the number of users and of LOC-users over time. It is striking to see the number of Firefox v1.0 go down to a small fraction, while the LOC-users for version v1.0 stays almost constant.[7]

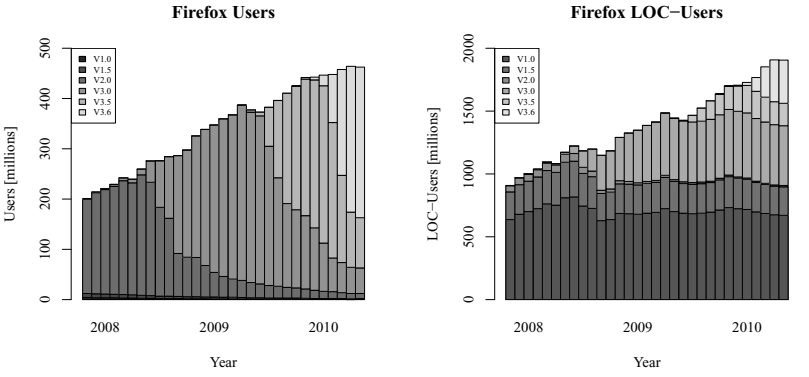
An important observation is that even the "small" fraction of users of older versions (full of after-life vulnerabilities that will never be fixed, as we have just shown) accounts for hundreds of thousands of users. You can imagine wandering in Florence and each and every person that you meet in the city still uses old Firefox v1.0.

This might have major implications in terms of achieving herd immunity as the number of vulnerable hosts would be always sufficient to allow propagation of infections [4].

## 5 "Milk or Wine" Revisited

Ozment and Schechter in their "Milk or Wine" study [3] look at vulnerabilities in OpenBSD and find evidence that most vulnerabilities are foundational, which they define as having been in the software in release 2.3, the earliest release for their study.

[7] A generalised linear model in which the residuals are modeled as an auto-regressive time series gives a statistically significant slope of $-0.27$ million LOC-users per month for version v1.5 ($p < 0.001$), and comparable numbers for the other versions.




Number of Firefox users (left) and LOC-users (right). While the number of users of Firefox v1.0 is very small, the amount of Firefox v1.0 *code* used by individual users is still substantial.

**Fig. 4.** Firefox Users vs LOC Users

To verify this finding in Firefox, we need the number of foundational and inherited vulnerabilities, which are not be able to calculate from Table 1. We obtain these values by counting a vulnerability for once based on the version that a vulnerability is reported, and the earliest version that it applies to.

Table 3 shows the number of vulnerability entries that were created during the lifetime of a version $x$ ("entered for"), where the earliest version to which it applies is $y$ ("applies to"). In the terms of Table 2, an entry in the table with a particular value for $x$ and $y$ means that the leftmost '$\times$' symbol is in the slot corresponding to $y$, and the rightmost '$\times$' symbol is in the slot for $x$. For example, there were 15 vulnerabilities that were discovered during the lifetime of v3.0, which also applied to v1.5, but not to any earlier version. Since we have no regressions (see above), these 15 vulnerabilities also apply to the intermediate v2.0. Therefore, the total vulnerabilities applies to version $x$ is sum of all entries which column is greater than $x$ and row is lesser than $x$. For instance, total vulnerabilities for v2.0 $= 42 + 97 + 32 + 13 + 104 + 15 + 126 + 22 + 1 = 452$.

We can now easily categorise the vulnerabilities in the table according to the categories that interest us: *inherited* vulnerabilities are the numbers above the diagonal (they are carries over from some previous version); *foundational* vulnerabilities are those in the first row, excluding the first element (they are carries over from version v1.0); and *local* vulnerabilities are those on the diagonal (they are fixed before the next major release and are not carried over).

The most important claim by Ozment and Schechter was that most vulnerabilities are foundational. If there were no difference between foundational vulnerabilities and others, all numbers in the table would be equal to $899/21 = 42.8$. Out of the 21 matrix entries, 5 would be foundational and 16 nonfoundational, so there would be $5 \cdot 899/21 = 214$ foundational and $16 \cdot 899/21 = 685$ nonfoundational vulnerabilities. We have 255 actual foundational and 565 actual nonfoundational vulnerabilities. A $\chi^2$ test on this data rejects this null hypothesis ($p = 1.3 \cdot 10^{-3}$).

**Table 3.** Vulnerabilities in Firefox

Number of vulnerabilities entered for a specific Firefox version (columns) and versions in which that vulnerability originated (rows). Since we look only at vulnerabilities that have been fixed fixed, the entries below the diagonal are all zero.

| first known to apply to | entered for v1.0 | v1.5 | v2.0 | v3.0 | v3.5 | v3.6 |
|---|---|---|---|---|---|---|
| v1.0 | 79 | 71 | 42 | 97 | 32 | 13 |
| v1.5 | — | 108 | 104 | 15 | 0 | 0 |
| v2.0 | — | — | 126 | 22 | 0 | 1 |
| v3.0 | — | — | — | 67 | 30 | 5 |
| v3.5 | — | — | — | — | 32 | 41 |
| v3.6 | — | — | — | — | — | 14 |

We can say that the vulnerabilities are not equally distributed, yet we cannot conclude that foundational vulnerabilities are the majority of vulnerabilities as argued by Ozment and Schechter, because they are not. They are actually less than a third of the total number of vulnerabilities. This is even more striking when compared with the actual fraction of the codebase that is still foundational.

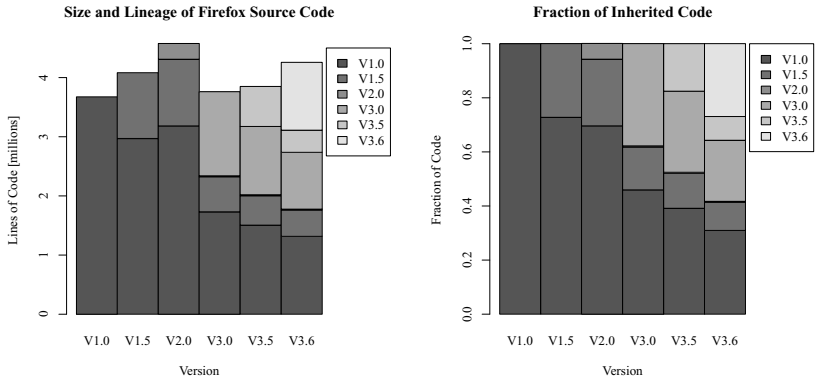
What about a weaker claim? maybe most vulnerabilities are inherited (but perhaps not necessarily foundational)? Using the same logic as above, we should now see $6 \cdot 899/21 = 257$ local and $15 \cdot 899/21 = 642$ inherited vulnerabilities; the actual counts are 426 and 473, respectively. This is strongly rejected ($p < 10^{-6}$).

Inherited vulnerabilities are far less than they should be, under the assumption of uniform distribution. These seem to show that Mozilla Firefox developers are doing a good job of vetting out vulnerabilities. In contrast, while we find that foundational vulnerabilities are more than they should be, we find no evidence that they are the majority.

We believe that the difference between our results and [3] can be explained by two factors. As we mentioned, Ozment and Schechter use a slightly different definition of "foundational": for them, a vulnerability is foundational if it was in the code when they started their study. For us, a vulnerability is foundational when it is inherited from version v1.0. However, they start their study at version v2.3, after a significant amount of development has already happened. Applied to our data, this would mean truncating Table 3, where we start at a later version, truncating all columns before, and adding all rows above that version. More formally, if we had started our study at version $v$, where $1 < v < 6$ we would get a new matrix $m'(v)$ with $6 - v + 1$ rows and columns where

$$m'_{jk}(v) = \begin{cases} \sum_{1 \le k \le v} m_{vk} & \text{if } j = 1, \\ m_{j+v-1,k+v-1} & \text{otherwise.} \end{cases} \tag{2}$$

In this aggregated matrix, much more weight is now given to the new first row, where foundational vulnerabilities originate, and the $p$-values for the corresponding $\chi^2$ test get less and less. This is therefore grounds to suspect that finer resolution (more intermediate versions) and truncation (not going back to




Evolution of Firefox codebase in absolute terms (left) and fraction of codebase (right). The left diagram shows the size of the Firefox releases, and the right diagram normalises all columns to a common height.

**Fig. 5.** Size and lineage of Firefox source code

v1.0, but to some later version) would in our study produce exactly the sort of foundational vulnerabilities that Ozment and Schechter found. In other words, the foundational vulnerabilities in their study might well be an artifact of the study design, which might disappear had more accurate data been available for releases earlier then 2.3.[8]

Other than that, when we repeat our tests with the data presented in their study (their Table 1 in [3]), we get high confidence that vulnerabilities are inherited there also, and also that they are not foundational under the uniform distribution assumption ($p < 10^{-6}$).

## 6 The Slow Pace of Software Evolution

While Fig. 2 seems to give a quick turn-around of versions, the actual picture in term of the code-base is substantially different. As Fig. 5 shows, the pace of evolution is relatively slow. Every version is largely composed by code inherited from old versions and the fraction of new code is a relatively small one.

Looking at this picture it is somewhat a surprise that our statistical tests reject the idea that foundational and inherited vulnerabilities are not the majority. At the same time the presence of after-life vulnerabilities, and even the frequency of zero-day attacks is no longer a big surprise.

The large fraction of code re-use reconciliates the seemingly contradictory information that vulnerability discovery is a hard process with the existence of zero-day attacks: when a new release gets out, the 40% of old code has been already targeted for over 6 months. Therefore the zero-day attack of version $v$ could well be in reality a six month mounting attack on version $v - 1$.

[8] They did of course not choose release 2.3 arbitrarily, but rather because it was the first release where vulnerability data was consistently and reliably documented. Therefore, they had no real choice for their initial version.

Observing this phenomenon from a different angle, if we assume that vulnerabilities are approximately uniformly distributed, then it is clear that most surviving vulnerabilities of version $v-1$ will also be present in version $v$.

There is a curious dip in the lines of code development for code inherited from v1.0. For v1.5, the number goes down, as expected, but for v2.0, it goes up again. So far, we do not have a good explanation of this phenomenon. A preliminary explanation is that it might be due to the way in which our algorithm calculates versions for merged branches.

## 7 Threats to Validity

**Errors in NVD.** we determine the Firefox version from which a vulnerability originated by looking at the earliest Firefox version to which the vulnerability applies, and we take that information from the "vulnerable versions" list in the NVD entry. If these entries are not reliable, we may have bias in our analysis. We have manually confirmed accuracy for few NVD entries, and an automatic large-scale calibration is part of our future work.

**Bias in CVS.** We only look at those vulnerabilities for which we can find fixes in the CVS. Fixes are identified by their commit messages, which must contain the Bugzilla identifier; therefore, fixes that do not have that form will escape us. This might introduce bias in our fixes (See [6]).

**Bias in data collection.** In addition to threats on parsing data collected from MFSA and CVS as described in previous work [5], we also have to extract lifetime of each source line. The extraction might bias our analysis if the extraction tool contains bug. We also apply the same strategy discussed in [5] to mitigate this issue.

**Ignoring the severity.** We deliberately ignore the severity of vulnerabilities in our study. Because current severity scoring system, Common Vulnerability Scoring System (CVSS), adopted by NVD and other ones (*e.g.*, qualitative assessment such such as critical, moderate used in Microsoft Security Bulletin) have shown their limitation. In accordance to [7], these systems are "inherently ad-hoc in nature and lacking of documentation for some magic numbers in use". Moreover, in that work, Bozorgi *et al.* showed that there is no correlation between severity and exploitability of vulnerabilities.

**Generality.** The combination of multi-vendor databases (*e.g.*, NVD, `Bugtraq`) and software vendor's databases (*e.g.*, MFSA, Bugzilla) only works for products for which the vendor maintains a vulnerability database and is willing to publish it. Also, the source control log mining approach only works if the vendor grant community access to the source control, and developers commit changes that fix vulnerabilities in a consistent, meaningful fashion *i.e.*, independent vulnerabilities are fixed in different commits, each associated with a message that refers to a vulnerability identifier. These constraints eventually limit the application of the proposed approach.

Another facet of generality, which is also our limitation, is that whether our findings are valid to other browsers, or other categories of software such as operating system? We plan to overcome this limitation by extending our study to different software in future.

## 8 Related Work

Fact Finding papers describe the state of practice in the field [8–10]. They provide data and aggregate statistics but not models for prediction. Some research questions picked from prior studies are *"What is the median lifetime of a vulnerability?"*[9], *"Are reporting rate declining?"* [9, 10].

Modeling papers propose mathematical models for vulnerabilities properties [10–14]. Here researchers provide mathematical descriptions of the vulnerability evolution such a thermodynamic model [14], or a logistics model [12]. Good papers in the group will provide experimental evidences that support the model, *e.g.*, [11–13]. Studies on this topic aim to increase the goodness-of-fit of their models *i.e.*, answer the question *"How well does our model fit the fact?"*.

Prediction papers try to predict defected/vulnerable component [8, 15–23]. The main concern of these papers is to find a metric or a set of metrics that correlate with vulnerabilities in order to predict vulnerable components.

Our paper can be classified in the fact finding group and the novelty of our approach and our findings is mostly due to our ability to dig into a database integrating multiple sources (albeit specialized to Mozilla Firefox).

## 9 Discussion and Conclusions

First, for the *individual*, we have the obvious consequence that running after-life software exposes the user to significant risk, which should therefore be avoided. Also, we seem to discover vulnerabilities late, and this, together with low software evolution speeds, means that we will have to live with vulnerable software and exploits and will have to solve the problem on another level. Vendors shipping patches faster will not solve the problem.

Second, for the *software ecosystem*, the finding that there are still significant numbers of people using after-life versions of Firefox means that old attacks will continue to work. This means that the penetrate-and-patch model of software security does not work, and that systemic measures, such as multi-layer defenses, need to be considered to mitigate the problem.

These phenomena revel that the problem of inherent vulnerabilities is merely a small part of the problem, and that the lack of maintenance of older versions leave software (Firefox) widely open to attacks. Security patch is not made available because it is not being deployed and because users are slow at moving to newer version of software.

In terms of understanding the interplay of the evolution of vulnerabilities with the evolution of software we think that the jury is still out. As we mentioned, foundational vulnerabilities are significantly more than they should be but are

less than a third of the total number of vulnerabilities. On the other side inherited vulnerabilities are almost half of the existing vulnerabilities but are significantly less than they should be according a uniform distribution model.

So we cannot in any way affirm that most vulnerabilities are due to foundational or anyhow past errors. We need to refine these findings by a careful analysis of the fraction of the codebase for each version.

These results have been possible by looking at vulnerabilities in a different way. Other studies have studied a vulnerability's *past*, i.e., once a vulnerability is known, we look at where it was introduced, who introduced it etc. In our study, we look at a vulnerability's *future*, i.e., we look at what happens to a vulnerability after it is introduced, and find that it survives in after-live versions even when it is fixed in the current release.

We plan to look also at fixes. In particular, CVS commit messages of the form "Fixes bug $n$", "#$n$", "Bug $n$", "bug=$n$", or related forms, where $n$ is a Bugzilla identifier make it possible to find the file(s) in which the vulnerability existed, the code that fixed it, and when it was fixed. This information is already present in our integrated database [5] and will be the subject of future studies.

We also plan to look at other software (either open-source or commercial products) to see whether our finding is applicable to other software. And therefore, should it have any significant influence on practise and give advice on how to move towards more secure software.

## References

1. Howard, M., Lipner, S.: The Security Development Lifecycle. In: Secure Software Development. Microsoft Press, Redmond (May 2006)
2. McGraw, G., Chess, B., Migues, S.: Building Security In Maturity Model v 1.5 (Europe Edition). Fortify, Inc., and Cigital, Inc. (2009)
3. Ozment, A., Schechter, S.E.: Milk or wine: Does software security improve with age? In: Proceedings of the 15th Usenix Security Symposium. USENIX Association, Berkeley (August 2006)
4. Hethcote, H.W.: The mathematics of infectious diseases. SIAM Review 42(4), 599–653 (2000)
5. Massacci, F., Nguyen, V.H.: Which is the right source for vulnerabilities studies? an empirical analysis on mozilla firefox. In: Proc. of MetriSec 2010 (2010)
6. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 121–130. Association for Computing Machinery. ACM Press, New York (2009)
7. Bozorgi, M., Saul, L.K., Savage, S., Voelker, G.M.: Beyond heuristics: Learning to classify vulnerabilities and predict exploits (July 2010)
8. Ozment, A.: The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In: Proceedings of 2nd Annual Workshop on Economics and Information Security, WEIS 2005 (2005)
9. Ozment, A., Schechter, S.E.: Milk or wine: Does software security improve with age? In: Proceedings of the 15th Usenix Security Symposium, USENIX 2006 (2006)

10. Rescorla, E.: Is finding security holes a good idea? IEEE Security and Privacy 3(1), 14–19 (2005)
11. Alhazmi, O., Malaiya, Y., Ray, I.: Measuring, analyzing and predicting security vulnerabilities in software systems. Computers & Security 26(3), 219–228 (2007)
12. Alhazmi, O., Malaiya, Y.: Modeling the vulnerability discovery process. In: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, pp. 129–138 (2005)
13. Alhazmi, O., Malaiya, Y.: Application of vulnerability discovery models to major operating systems. IEEE Trans. on Reliab. 57(1), 14–22 (2008)
14. Anderson, R.: Security in open versus closed systems - the dance of Boltzmann, Coase and Moore. In: Proceedings of Open Source Software: Economics, Law and Policy (2002)
15. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early predictors of vul. Journal of Software Architecture (2010)
16. Gegick, M., Rotella, P., Williams, L.A.: Predicting attack-prone components. In: Proc. of the 2nd Internat. Conf. on Software Testing Verification and Validation (ICST 2009), pp. 181–190 (2009)
17. Jiang, Y., Cuki, B., Menzies, T., Bartlow, N.: Comparing design and code metrics for software quality prediction. In: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering (PROMISE 2008), pp. 11–18. ACM, New York (2008)
18. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 33(9), 2–13 (2007)
19. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Communications and Computer Security (CCS 2007), pp. 529–540 (October 2007)
20. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, ESEM 2008 (2008)
21. Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings of the 4th Workshop on Quality of Protection (QoP 2008), pp. 47–50 (2008)
22. Zimmermann, T., Nagappan, N.: Predicting defects with program dependencies. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009 (2009)
23. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Proceedings of the 3th International Workshop on Predictor Models in Software Engineering (PROMISE 2007). IEEE Computer Society, Los Alamitos (2007)

# Authorization Enforcement Usability Case Study

Steffen Bartsch

Technologie-Zentrum Informatik TZI,
Universität Bremen, Bibliothekstr. 1, 28359 Bremen, Germany
sbartsch@tzi.org

**Abstract.** Authorization is a key aspect in secure software development of multi-user applications. Authorization is often enforced in the program code with enforcement statements. Since authorization is present in numerous places, defects in the enforcement are difficult to discover. One approach to this challenge is to improve the developer usability with regard to authorization. We analyze how software development is affected by authorization in a real-world case study and particularly focus on the loose-coupling properties of authorization frameworks that separate authorization policy from enforcement. We show that authorization is a significant aspect in software development and that the effort can be reduced through appropriate authorization frameworks. Lastly, we formulate advice on the design of enforcement APIs.

## 1 Introduction

Access control and, more specifically, authorization is a prevalent security requirement for multi-user applications, particularly for Web applications with the well-known advantage of efficiently serving large numbers of users. Research on authorization is focused on authorization models and algorithms for the enforcement of policies [1,5,12,33]. Only a fraction of the publications on authorization measures takes usability aspects into account, although it is widely accepted that a lack of security usability leads to vulnerabilities [32]. In case of authorization, the vulnerabilities may for example stem from end users circumventing imposed authorization measures to efficiently complete the work at hand [19].

Authorization usability can be categorized according to three perspectives: (1) the functional perspective of end users who may be hindered by inappropriate restrictions, (2) the security perspective of policy authors who elicit and codify authorization requirements into a policy, and (3) the development perspective of software developers who implement authorization enforcement, for example by placing adequate enforcement statements in the source code. This work primarily focuses on the development perspective of enforcement usability. Our main target is the effectiveness, that is the precise implementation of enforcement, and the efficiency, affected by the effort necessary to achieve the effectiveness. Developer usability is also concerned with the security perspective since the policy is often specified concurrently to development in case of custom software development. This development model is particularly wide-spread for agile development of Web applications.

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 209–220, 2011.

Enforcement usability is an important aspect of secure software development, since not adequately enforcing authorization leads to security risks [18]. A challenge is that, as a cross-cutting concern, authorization is often enforced on throughout the program. Consequently, authorization is difficult to test automatically in its entirety so that many authorization-related defects are only found in manual acceptance tests. A negative example of the usability are repetitive and complex enforcement statements because redundancy may lead to defects when developers implement evolutionary changes that are imminent in software development [22]. Interestingly, there has been little systematic work on authorization enforcement usability, despite the prevalent authorization requirements. To the best of our knowledge, there has been no prior work with a focus on how authorization is developed and the interplay of policy and enforcement changes in the development of custom applications.

This paper pursues to improve the understanding of how authorization affects software development. We assess the impact of authorization on software development in a case study of a business Web application. In particular, we study how the loose coupling of policy and enforcement, a well-known separation of concerns design principle [11], affects real world software development in this case study. The approach is an in-depth analysis of authorization in a real-world agile development project that employs a widely-used authorization framework. We evaluate the project's source code repository commits. Before describing the case study, we give a brief background on authorization usability and the framework that is used in the project. Following the study results, we offer advice on enforcement API design based on experiences from the case study.

## 2 Authorization Usability in Software Development

There are numerous authorization models, ranging from simple discretionary access control, such as access control lists, to logic-based models [27]. For the relevant authorization aspects in this paper, policy specification and authorization enforcement, each model has a specific form, differing in usability for policy creators and software developers.

### 2.1 Policy Specification

With many different authorization models and associated policy specification languages, the usability challenges naturally differ. For privacy policies, Reeder et al. identified five usability challenges: policy creators need to be able to efficiently group objects, use consistent terminology, understand the default rules, uphold the policy structure and solve rule conflicts [25]. In case of role-based policies, Brostoff et al. found that the primary challenges for policy creators lie in understanding the policy structure and the overall authorization paradigm [7].

Languages for authorization policy specification are usually defined as domain-specific languages (DSL) [10]. For the usability of an authorization language, the distance between humans and the syntax is important [24]. Inglesant et al.

studied how security experts formulated policies using a controlled natural language with an explicitly reduced distance. They also proposed an iterative process for policy specification, which is necessary for users to understand the authorization model [17]. Similarly, Stepien et al. suggest an intermediate natural language notation to simplify the creation of XACML policies [29]. Apart from policy languages, tool support for the creation of policies has been studied. Zurko et al. developed a policy editor for policies similar to RBAC (Role-based Access Control) based on usability testing and user-centered design that allowed novice users to produce meaningful results in under one hour [34]. Herzog and Shahmehri studied the usability of the Java policy tool and observed that help on semantics was missing. They also concluded that the tool is inadequate for expert users and promotes lax policies [16]. Lastly, security policy management does not only encompass the definition, but also decision-making regarding the policy. For this purpose, Rees et al. propose a framework for the development of high-level security policies [26].

### 2.2 Authorization Enforcement

Systematic authorization enforcement is not a new concept and has already been described in form of reference monitors in the 1970s [2]. Current authorization enforcement mechanisms come in a variety of approaches, often applied in combinations [23]. Operating system-based enforcement relies on operating system mechanisms that check permissions on the level of files or processes [15]. In runtime system-based mechanisms, the running program is encapsulated and every call to a protected component is first checked against a reference monitor; an example is the Java security model [14]. For language-based approaches, a compiler generates additional enforcement code for the specified authorization policy [13]. Similarly, aspect-oriented programming (AOP) can be employed to enforce authorization at the join points [20]. A related approach is the transformation of program bytecode to enforce program behavior, for example in Java programs to secure hosts against potentially malicious mobile code [23]. In these approaches, the program code is regarded as potentially hostile, requiring external supervision. In contrast, many authorization decisions are made in program code for external users, permitting a cooperative approach of program code and authorization. In these API-based approaches, enforcement hooks are placed into the source code, for example as `checkPermission` calls in Java [14] or as security hooks in the Linux kernel [18].

The challenges in authorization enforcement are the correct placement of enforcement statements. Errors in placement were for example found in Linux kernel modules [18]. Measures to enforce authorization as part of the architecture may be circumvented by accident through unanticipated control flow [28]. It is particularly difficult to achieve adequate authorization enforcement in software evolution, which often requires program comprehension for further development [31]. Frequently, information on why a certain state is reached is missing in development [21]. The API design also affects the enforcement comprehensibility [9,30]. Therefore, enforcement was ideally implemented once and could be left unmodified even

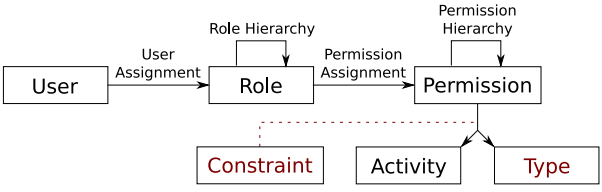



**Fig. 1.** `declarative_authorization` authorization model

when authorization requirements change [6]. Vice versa, the authorization policy would not need to be modified even with functional changes as long as the authorization requirements remain unchanged. Loose coupling of authorization policy and enforcement is needed as the separation of concerns [11].

## 3 Authorization Framework: declarative_authorization

Authorization in the case study is based on `declarative_authorization`[1], a widely-used open source authorization framework for the Web development platform Ruby on Rails [4]. It aims particularly at offering loose coupling of authorization policy and enforcement. Moreover, the framework supports secure software development with readable and maintainable authorization policies and enforcement statements.

The authorization model, depicted in Figure 1, is similar to classical RBAC [12] in that roles are assigned to users and permissions to roles with role and permission hierarchies. Instead of composing permissions of object and activity, the framework combines object types (contexts) and activities with constraints to increase the maintainability. Constraints are either defined on attributes of an object of the type, possibly spanning multiple nested object relations, or according to specific permissions on related objects. The policy language, although based on Ruby syntax, is aimed to be a human-readable and intuitively comprehensible textual DSL. A change management tool supports the modification by non-experts [3].

Authorization enforcement follows the typical model–view–controller (MVC)-approach that Ruby on Rails encourages, providing enforcement on multiple layers. Primarily, enforcement occurs on the model (object-relational mapping) and controller (business logic) layers. Declarative enforcement statements limit permissions on create, read, update, delete (CRUD) activities on the model and respective access through HTTP requests on controller actions. Additionally, the authorization policy is enforced through database query rewriting.

## 4 Authorization Development Case Study

To gain further insights on authorization in software development, we studied the development of a custom business Web application for a medium enterprise from

[1] `http://github.com/stffn/declarative_authorization`

the automotive supplier industries. At the time of examination, development had been ongoing for 2.5 years with small teams that employed agile development practices. The application had been in full productive use for 1.5 years.

As indicated in the introduction, we pursue the following research goals in this case study: First, what is the actual impact of authorization on agile software development? The hypothesis is that agile development causes frequent changes of functional requirements, including authorization requirements. Thus, a substantial part of the software development should affect authorization (H1). Second, how does a loose-coupling authorization framework help in authorization development? We expect that changes to the requirements often only affect either the policy or the enforcement (H2). Third, in which cases does the separation of concerns not work? Despite loose coupling, we still expect that there are authorization enforcement modifications for authorization requirement changes and vice versa (H3).

### 4.1 Methodology

For the study, we analyzed the commits to the development branch of the subversion repository over almost 1.5 years beginning with the introduction of the `declarative_authorization` plugin in the project, which superseded a very simple authorization mechanism. We target the efficiency aspect of developer usability and, thus, would need to study the effort that developers spend on different tasks. Since development effort is difficult to assess directly, our analysis is based on commit counts and the relation between the commits that touch authorization development aspects. The rational is that a developer generally risks to introduce defects and spends effort if a commit affects authorization, impacting the effectiveness and efficiency, respectively. We focused on two distinct aspects, policy changes and changes to authorization enforcement, and manually coded commits according to the reasons of modifications. Since a commit may contain several changes, all coding was non-exclusive.

Firstly, we determined the total number of commits on the development branch. For policy changes, we analyzed commits to the authorization rules configuration file and coded the commits into the categories that are listed in Table 1 using the commit log entries, inspection of the version differences and developer knowledge of underlying change reasons.

**Table 1.** Categories of reasons for authorization policy changes

| | |
|---|---|
| Refactoring | Changes to authorization policy for cosmetic and readability reasons without affecting the authorization test results |
| Bugfix | Authorization policy modification to fix bugs caused by the policy |
| Authorization requirement | Modifications related to changed authorization requirements without further application functionality changes. |
| Functionality changes | Authorization policy changes that are caused by functional changes to the application |

**Table 2.** Categories of reasons for authorization enforcement changes

| | |
|---|---|
| Bugfix | Bugfixes in application code that relate to enforcement |
| Requirement changes | Changes in enforcement that result from changed authorization requirements |
| Refactoring | General application refactoring that affects enforcement |
| Authorization-related Refactoring | Refactoring in application code to change authorization refactoring |
| Functional changes | Functional changes in application source code that affect enforcement |

For enforcement, we automatically analyzed the differences between versions and considered a commit enforcement-related if the changeset contained a change line with an enforcement statement. All enforcement commits were manually verified and categorized according to Table 2. For a more detailed analysis of enforcement commits, we also coded the commits according to the modification type, either addition, modification or removal.

### 4.2 Results

*Reasons for changes.* The results from the commit analysis are shown in Table 3. For each commit type, a line lists the quantity and, if applicable, the ratio against the total commit quantity and against the enforcement or policy-related commits. In addition, the raw categories are grouped in multiple cases as described below. Commits may fall in multiple categories, so that ratios do not sum up to 100%.

From the quantitative results in Table 3, we deduce several findings. Interestingly, almost a quarter (23%) of all commits are related to authorization, almost 10% to the policy. Looking at the impact of authorization requirement changes, the results show that 7% of all commits (76% of all policy-related commits) result from changed authorization requirements, including requirements from functionality changes. This result shows that there is a high number of authorization requirement changes as expected from agile development. In 67% of the commits relating to the policy, authorization behavior is changed without functionality changes, that is only authorization bugfix or requirements changes. Moreover, there are 33 commits caused by authorization requirement changes on the policy compared to only 9 for enforcement, preventing unnecessary code changes in one quarter of the changes. Only 8% of the enforcement change commits are related to authorization requirement changes without further functionality changes, so it again mostly seems to suffice to change the policy. These results suggest a loose coupling of policy and enforcement.

On further analyzing the enforcement-related commits, the results show that 72% of all enforcement-related changes are for general refactoring or functional changes. This result is important as it indicates that most enforcement changes are really necessary because the changes are part of the functional application development. While 9% of all commits involve enforcement changes that are

**Table 3.** Authorization commit reasons quantities

| | # commits | of total commits | of enforcement/ policy commits |
|---|---|---|---|
| **Total commits** | 610 | | |
| Total authorization-related | 137 | 22.5 % | |
| **Policy-related** | 58 | 9.5 % | |
| Functionality | 17 | 2.8 % | 29.3 % |
| Bugfixing | 8 | 1.3 % | 13.8 % |
| Policy refactoring | 15 | 2.5 % | 25.9 % |
| Authorization requirements | 33 | 5.4 % | 56.9 % |
| Authorization req.-related | 44 | 7.2 % | 75.9 % |
| Non-functional | 39 | 6.4 % | 67.2 % |
| **Enforcement-related** | 116 | 19.0 % | |
| Functionality | 52 | 8.5 % | 44.8 % |
| General refactoring | 35 | 5.7 % | 30.2 % |
| Authorization bugfixes | 11 | 1.8 % | 9.5 % |
| Authorization refactoring | 20 | 3.3 % | 17.2 % |
| Authorization requirements | 9 | 1.5 % | 7.8 % |
| Authorization-only | 37 | 6.1 % | 31.9 % |
| Non-authorization | 83 | 13.6 % | 71.6 % |

caused by functionality changes, functionality changes cause only policy changes in 3% of all commits. This suggests that the loose coupling of policy and enforcement also works in the reverse direction: functional changes are more likely to affect the enforcement than the policy.

*Enforcement modifications.* Although, ideally, no authorization-related modifications of enforcement should be necessary, one third (32%) of the enforcement changes relate to authorization bugfix, refactoring and requirement changes. We analyzed the reasons of authorization enforcement changes further in two ways: by examining the distribution of commits over time and by coding the commits by enforcement modification types. The distribution of enforcement commits over time relative to the total number of enforcement commits are shown in Figure 2. In the diagram, the most striking development is that authorization-related enforcement commits are decreasing after a transitional period of five months. Most of the authorization refactoring occurred in the transitional period. Later, the authorization enforcement stabilizes with only little refactoring necessary.

Further insights may be gained from the types of enforcement modifications, shown as a matrix for modification operations and reasons in Table 4. In this project, the addition and modification changes for authorization bugfixing can be primarily attributed to originally missing or wrong permission checks, for example found in acceptance tests. Similarly, addition enforcement commits for authorization requirements were caused by missing checks that were needed to enforce a requirement. This is also the reason for the modification commits, since

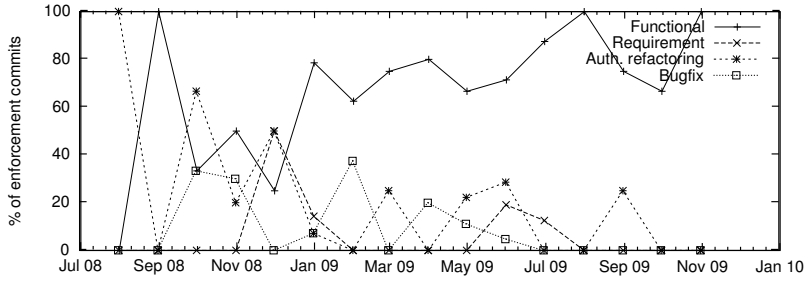



**Fig. 2.** Distribution of reasons for enforcement commits over time

the framework's enforcement statements either take a general context (object type) or specific object as basis for the decision. This may need to be modified for some authorization requirement changes.

### 4.3 Discussion

The findings are in line with the hypotheses. We observed that a significant part of the development was related to authorization. As we expected, agile development causes frequent changes to functional and authorization requirements over time (H1). Functionality-related authorization changes affected enforcement to a higher degree. Similarly, policy modifications are more often caused by authorization requirement changes. Thus, we found a loose coupling between policy and enforcement that improves the developer usability by reducing effort and preventing errors (H2). We still observed authorization requirements that affect enforcement and vice versa. The primary reasons were a large amount of transitional work in the first months after the framework introduction. To a lesser extent, enforcement additions and modifications were also necessary to fulfill changing authorization requirements (H3).

Since we conducted a single-project case study, we acknowledge several threats to the study's validity. For the scope validity, the results are mainly applicable to custom-developed Web applications that involve a non-trivial amount of authorization, preventing the definition of the entire authorization requirements in advance. The specific development process is also important, since the agile

**Table 4.** Enforcement-related commit quantities by modification reason and type

| | Addition | Modification | Removal | Total |
|---|---|---|---|---|
| Functionality | 29 | 26 | 5 | 52 |
| General refactoring | 4 | 31 | 3 | 35 |
| Authorization bugfixes | 7 | 5 | 0 | 11 |
| Authorization refactoring | 10 | 13 | 2 | 20 |
| Authorization requirements | 7 | 5 | 0 | 9 |
| Total | 49 | 67 | 10 | |

methods in this project caused continuously changing functional and authorization requirements. As most plan-based development processes also have changing requirements [22], we suppose that the effects will be similar in other process models. It would be interesting to investigate the influence of the process model in future research. The development context could also pose a threat to the validity since the development was conducted in the university realm. However, development was very focused on the product owner's requirements and was conducted by skilled developers of at least two years of experience with the development framework. We found no significant effects from trainings and no developer took security courses while on the project. Lastly, the specific programming language, Web development platform and authorization framework may have affected the results. We analyzed whether changes to the framework resulted in additional refactoring but the API was rather stable and we could only identify one commit in which non-backward compatible changes to the framework required refactoring in the application code. There was no authorization refactoring from newly introduced framework functionality. Still, we will study other authorization frameworks and platforms in future research.

Other threats to validity are posed by the study author's involvement in the development. While it is common in the humanities to conduct empirical work with active involvement of the study authors, for instance in action research [8], it is less common in computer science. Still, we believe that the involvement has not significantly influenced the results as the analysis was fully conducted *post-mortem* and, more importantly, the analysis was only decided after the studied period was over. Thus, the author should not have been biased in development. Moreover, developer involvement in the study was inevitable as the coding of commits was only possible with knowledge of the system and the development process. The primary source for a potential author bias in the study is the coding of commits into the categories. Some of the categories have fluent boundaries so that the threat of a biased coding is present, particularly with only one person having coded the commits. On the other hand, we were aware of this threat and separated the coding from the analysis phase to reduce the potential effect. While we cannot rule out the effect completely, we are confident that it did not significantly affect the results.

Lastly, threats to validity originate from the methodology of analyzing commit quantities instead of the actual development effort as indicated in the methodology section. Generally, the metric of commit quantity is no quantitative measure of the work effort, but gives an abstract approximation through the frequencies of occurrences. If authorization was touched in a development step, a developer needed to invest cognitive effort. To rule out issues from large and infrequent in contrast to small and frequent commits, we also analyzed the number of enforcement changes per commit. While means on the change quantities are problematic because of our non-exclusive coding, the categories of enforcement changes showed similar patterns of commits with high (up to 40 changes, six commits with more than 20 changes) and small numbers of changes. The large ones were mostly merge commits, encompassing several changes in one commit.

On the other hand, the combination of changes into single commits occurred at random and should thus not have impacted the validity of the ratios that were the foremost source for the findings.

## 5 Advice on Authorization Enforcement Design

From the findings in the case study and the analysis of the factors that lead to the results, we derive advice on the design of authorization enforcement. First, an API should facilitate the use of universally applicable enforcement statements that provide enough context for unanticipated authorization requirements, such as the accessed "branch" object to decide whether the activity "read this branch" is allowed. As demonstrated in the study, if the context is not meaningful enough, enforcement changes may be necessary when authorization requirements change.

Second, it helps when the same authorization rules can be used in multiple places. One example relates to associated objects and checking whether accessing a list of objects with a specific limitation is permitted as in "list branches of this company", re-using the specific permissions for "read branch". Another example is the defense-in-depth design principle of placing enforcement hooks on several levels. Defense-in-depth only requires limited additional implementation effort when the authorization decision is derived from one policy specification for multiple layers. The study shows that policy changes can thus be reduced.

Third, it is important for the enforcement statements to only pose a low implementation threshold and, thus, encourage implementing enforcement early-on. One approach is the "convention over configuration" paradigm, for example matching object types to controller names or permission names to the CRUD activities. The result is less code, improved readability and reduced redundancy, but a slightly steeper learning curve.

## 6 Conclusion

In this paper, we show that authorization can have a significant impact on software development and that loose coupling of authorization enforcement and policy specification improves authorization development. Based on the case study, we formulated advice on the enforcement API design. We indicated several limitations of the case study, mostly concerned with the project selection and the methodology. Still, the results should be relevant to a broader audience, given that previous publications are thin on authorization aspects in software development. Moreover, we are confident that the key findings, while in need of further validation in additional studies, already provide added value for authorization development. As part of this work, we also describe a methodology for authorization development analysis, facilitating the validation of the results in further development context and with other authorization frameworks in future work. Part of these efforts will be to study the differences to commercial off-the-shelf software development where policy specification is only added after release at deployment time.

## References


1. Ahn, G.J., Zhang, L., Shin, D., Chu, B.: Authorization management for role-based collaboration. In: IEEE International Conference on Systems, Man and Cybernetics, vol. 5, pp. 4128–4134 (October 2003)
2. Anderson, J.P.: Computer security technology planning study. Tech. Rep. ESD-TR-73-51, Deputy for Command and Management Systems, L.G. Hanscom Field, Bedford, MA (October 1972)
3. Bartsch, S.: Supporting authorization policy modification in agile development of Web applications. In: Fourth International Workshop on Secure Software Engineering (SecSE 2010). IEEE Computer Society, Los Alamitos (2010)
4. Bartsch, S., Sohr, K., Bormann, C.: Supporting Agile Development of Authorization Rules for SME Applications. In: 3rd International Workshop on Trusted Collaboration (TrustCol-2008). Springer, Heidelberg (2009)
5. Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur. 2(1), 65–104 (1999)
6. Beznosov, K., Deng, Y., Blakley, B., Barkley, J.: A resource access decision service for corba-based distributed systems. In: Computer Security Applications Conference, Annual, p. 310 (1999)
7. Brostoff, S., Sasse, M.A., Chadwick, D.W., Cunningham, J., Mbanaso, U.M., Otenko, S.: 'R-What?' development of a role-based access control policy-writing tool for e-scientists. Softw., Pract. Exper. 35(9), 835–856 (2005)
8. Cairns, P., Cox, A.L.: Research methods for human-computer interaction. Cambridge Univ. Press, Cambridge (2008)
9. Clarke, S.: Measuring API usability. Dr. Dobb's Journal (May 2004)
10. Consel, C., Marlet, R.: Architecture software using: A methodology for language development. In: Palamidessi, C., Glaser, H., Meinke, K. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 170–194. Springer, Heidelberg (1998)
11. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSA Workshop on the Application of Engineering Principles to System Security Design (2003)
12. Ferraiolo, D., Kuhn, R.: Role-based access controls. In: 15th NIST-NCSC National Computer Security Conference, pp. 554–563 (1992)
13. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, p. 11 (1982)
14. Gong, L., Ellison, G.: Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation. Pearson Education, London (2003)
15. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. ACM Commun. 19(8), 461–471 (1976)
16. Herzog, A., Shahmehri, N.: A usability study of security policy management. In: Security and Privacy in Dynamic Environments (SEC), vol. 201, pp. 296–306. Springer, Heidelberg (2006)
17. Inglesant, P., Sasse, M.A., Chadwick, D., Shi, L.L.: Expressions of expertness: the virtuous circle of natural language for access control policy specification. In: Proceedings of the 4th Symposium on Usable Privacy and Security, SOUPS 2008, pp. 77–88. ACM, New York (2008)
18. Jaeger, T., Edwards, A., Zhang, X.: Consistency analysis of authorization hook placement in the linux security modules framework. ACM Trans. Inf. Syst. Secur. 7(2), 175–205 (2004)

19. Johnson, M., Bellovin, S., Reeder, R., Schechter, S.: Laissez-faire file sharing. In: New Security Paradigms Workshop 2009 (2009)
20. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
21. Ko, A.J., DeLine, R., Venolia, G.: Information needs in collocated software development teams. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 344–353. IEEE Computer Society, Washington, DC (2007)
22. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9), 1060–1076 (1980)
23. Pandey, R., Hashii, B.: Providing fine-grained access control for java programs. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 449–473. Springer, Heidelberg (1999)
24. Pane, J.F., Ratanamahatana, C.A., Myers, B.A.: Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies 54(2), 237–264 (2001)
25. Reeder, R.W., Karat, C.M., Karat, J., Brodie, C.: Usability challenges in security and privacy policy-authoring interfaces. In: Baranauskas, M.C.C., Palanque, P.A., Abascal, J., Barbosa, S.D.J. (eds.) INTERACT 2007. LNCS, vol. 4663, pp. 141–155. Springer, Heidelberg (2007)
26. Rees, J., Bandyopadhyay, S., Spafford, E.H.: Pfires: a policy framework for information security. ACM Commun. 46(7), 101–106 (2003)
27. Samarati, P., de Capitani di Vimercati, S.: Access control: Policies, models, and mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 137–196. Springer, Heidelberg (2001)
28. Sohr, K., Berger, B.: Idea: Towards architecture-centric security analysis of software. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 70–78. Springer, Heidelberg (2010)
29. Stepien, B., Matwin, S., Felty, A.: Strategies for reducing risks of inconsistencies in access control policies. In: Proceedings of the International Conference on Availability, Reliability and Security (ARES 2010). IEEE Computer Society, Los Alamitos (2010)
30. Stylos, J., Clarke, S., Myers, B.: Comparing API design choices with usability studies: A case study and future directions. In: Proceedings of the 18th Workshop of the Psychology of Programming Interest Group (2006)
31. von Mayrhauser, A., Vans, A.M.: Program comprehension during software maintenance and evolution. Computer 28(8), 44–55 (1995)
32. Whitten, A.: Making Security Usable. Ph.D. thesis, CMU, cMU-CS-04-135 (2004)
33. Zhang, X., Oh, S., Sandhu, R.: PBDM: a flexible delegation model in RBAC. In: Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies, SACMAT 2003, pp. 149–157. ACM, New York (2003)
34. Zurko, M.E., Simon, R., Sanfilippo, T.: A user-centered, modular authorization service built on an RBAC foundation. In: IEEE Symposium on Security and Privacy. IEEE Computer Society, Los Alamitos (1999)

# Scalable Authorization Middleware for Service Oriented Architectures

Tom Goovaerts, Lieven Desmet, and Wouter Joosen

IBBT-Distrinet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{tomg,lieven,wouter}@cs.kuleuven.be

**Abstract.** The correct deployment and enforcement of expressive attribute-based access control (ABAC) policies in large distributed systems is a significant challenge. The enforcement of such policies requires policy-dependent collaborations between many distributed entities. In existing authorization systems, such collaborations are static and must be configured and verified manually by administrators. This approach does not scale to large and more dynamic application infrastructures in which frequent changes to policies and applications occur. As such, configuration mistakes or application changes might suddenly make policies unenforceable, which typically leads to severe service disruptions.

We present a middleware for distributed authorization. The middleware provides a single administration point that enables the configuration and reconfiguration of application- and policy-dependent interactions between policy enforcement points (PEPs), policy decision points (PDPs) and policy information points (PIPs). Using lifecycle and dependency management, the architecture guarantees that configurations are consistent with respect to deployed policies and applications, and that they remain consistent as reconfigurations occur. Extensive performance evaluation shows that the runtime and configuration overhead of the middleware scale with the size and complexity of the infrastructure and that reconfigurations cause minimal disruption to the involved applications.

**Keywords:** attribute-based access control, policy enforcement, policy deployment, middleware, service-oriented architectures.

## 1 Introduction

In open and dynamic distributed applications, it is common practice to enforce fine-grained authorizations using the Attribute-Based Access Control (ABAC) [13] model. The eXtensible Access Control Markup Language (XACML) [8] is the most widely deployed and used ABAC language in practice. In ABAC, policies are specified declaratively in terms of rules based on the attributes of the subject, resource and environment. Logically, the enforcement of ABAC policies is performed by three types of collaborating components, which we call *authorization*

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 221–233, 2011.

*components*: Policy Enforcement Points (PEPs) intercept access attempts, request authorization decisions and enforce those decisions, Policy Decision Points (PDPs) make authorization decisions for PEPs, and Policy Information Points (PIPs) provide values for required attributes. This work focuses on attributes that are pulled in by the PDP. Therefore, PEPs depend on PDPs for authorization decisions, and PDPs depend on PIPs for attributes.

In practice, managing and maintaining the effective deployment and enforcement of ABAC policies is a significant challenge. Besides the need for implementing custom PEPs and PIPs to bind policies to the business logic, policies must also be distributed to the relevant PDPs, and the interactions between PEPs, PDPs, and PIPs must be configured and managed such that the policies are correctly enforced. This problem is further complicated by the fact that (1) PEPs, PDPs and PIPs are located on distinct distributed nodes in the system and therefore must interact remotely and (2) frequent policy and application changes occur that affect the dependencies between authorization components and therefore require updates to the required interactions.

Existing work on ABAC enforcement mostly focuses on isolated PDP implementation issues, and does not address the fact that ABAC policy enforcement is effectively a complex and dynamically evolving workflow between distributed authorization components. The practical consequence is that PEP, PDP and PIP interactions are static and must be configured and verified manually by the security administrator. This approach does not scale to large and complex infrastructures, and can lead to severe application disruptions because policies might suddenly become unenforceable.

The main contribution of this paper is an extended and scalable distributed authorization middleware that supports the configuration and reconfiguration of interactions between remote authorization components in response to policy or application changes. To guarantee and maintain correctness of the configuration, the system performs dependency and lifecycle management based on policy evaluation contracts that state the capabilities and requirements of the authorization components. The middleware has been prototyped on the Apache ActiveMQ message broker. Extensive performance measurements show that the policy enforcement as well as the reconfiguration overhead scale to large and complex infrastructures of hundreds of authorization components and thousands of dependencies.

The rest of the paper is structured as follows. Section 2 discusses an example derived from an industrial collaboration that illustrates the configuration and policy deployment challenges that must be met. Section 3 presents the architecture of our middleware and Section 4 discusses its prototype implementation. We evaluate the presented solution in Section 5 and discuss related work in Section 6. Section 7 concludes the paper.

## 2 Motivation

This section illustrates the maintenance challenges of ABAC policies in a case study on a personal content management system. People acquire gigabytes of

personal digital content (documents, pictures, videos, etc.) and store it on many different locations. The PeCMan project [5] has developed a Personal Content Management (PCM) platform that offers a uniform user interface for managing and sharing personal content that is scattered over various devices and services. A core feature of the PCM system is that it is open and extensible with value-added services such as cloud storage, web publishing, face recognition and social network integration.

In order to protect access to services and content, the PCM system supports two types of authorization policies. First, system-level policies are defined by the owner of the PCM system and protect access to internal and third party services. An example of a system-level rule is: *documents uploaded to storage provider X may not be larger than 100MB*. Secondly, via a high level graphical editor, end-users themselves can define fine-grained policies that control access to shared content. An example policy rule is: *grant read access to my pictures only to friends that are older than 12*.

Figure 1 shows a simplified deployment view of a PCM system. The core system consists of an *index service* that keeps track of content location, a *metadata service* that stores, caches and searches content metadata, a *workflow engine* that orchestrates and composes value-added services into reusable processes and a *controller* that processes and mediates incoming client requests. HTTP-based clients are handled by a servlet in the web tier. Value-added services or adapters to external services are deployed on service containers on separate nodes.

Figure 1 also overlays the authorization infrastructure with the required authorization components, consisting of three PDPs and multiple application-specific PEPs and PIPs. The solid arrows show the dependencies between the

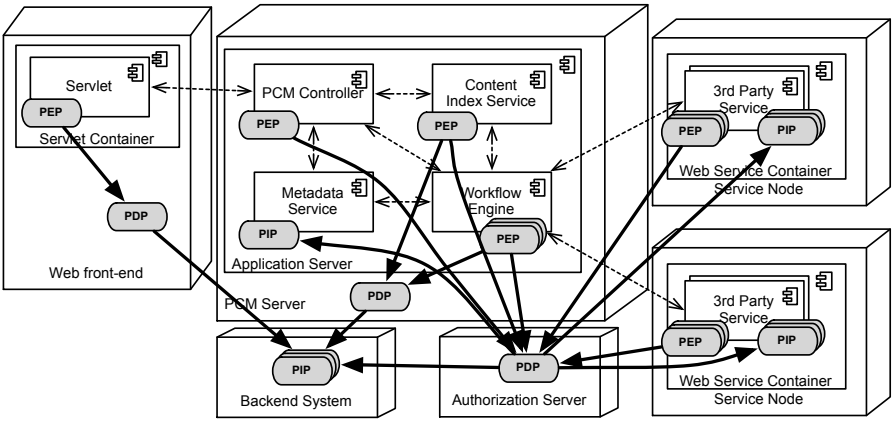



**Fig. 1.** Deployment view of the PeCMan architecture overlaid with the necessary authorization components and their dependencies. The dashed arrows represent architecture-level dependencies and the bold arrows represent dependencies between authorization components.

authorization components. It can be observed that there are already a large number of dependencies to manage in this (simplified) infrastructure. Furthermore, system must deal with two types of frequently occurring changes:

**Policy changes.** Because policies are system as well as user driven, policy changes occur very frequently. A new or updated policy may require new types of attributes. This requires a reconfiguration of the involved PDP with a PIP that offers those attributes. Second, a policy can apply to other types of resources. In this case, the administrator has to ensure that all PEPs that protect those resource must use the PDP.

**Application changes.** Application changes (for instance the deployment of a new third-party service) can introduce new PEPs that must be bound to one or more PDPs, they can change the interceptions and the required PDPs of an existing PEP, or they can break application-level PIPs that are required by existing PDPs.

To deal with such evolving, distributed software environments, the authorization infrastructure must be capable of (1) easily adding, reconfiguring or removing authorization components, as well as (2) guaranteeing the consistency of the deployed policies and their composed components. To the best of our knowledge, current authorization infrastructures, albeit modularized, do not support the necessary level of flexibility when changes must be managed and deployed because authorization components are tightly coupled to each other. Automatic (re)configuration has not been addressed in this context.

In addition, the authorization infrastructure must scale to a large number of authorization components and dependencies without sharp performance degradations, and changes to the authorization components or their interactions (triggered by policy changes or application changes) should impose a minimal disruption of the authorization system and of the applications. In the following section, we present the architecture of our authorization middleware, taking these requirements into account.

## 3 Architecture

An overview of the architecture of the proposed authorization middleware is shown in Figure 2. The middleware consists of three core ingredients. First, the PEPs, PDPs and PIPs are represented as first class components that are governed by a well-defined lifecycle model. Second, a message-based distribution layer mediates and supports the remote interactions between authorization components and supports flexible and efficient runtime adaptations to those interactions. Third, the authorization middleware provides a management component that functions as a centralized administrative interface. The management component is responsible for configuring and reconfiguring PEPs, PDPs and PIPs and their interactions while guaranteeing that configurations are consistent with respect to the deployed policies and applications. Section 3.1 discusses the authorization components and the lifecycle model in detail. Section 3.2 discusses the distribution layer and Section 3.3 discusses the management component.

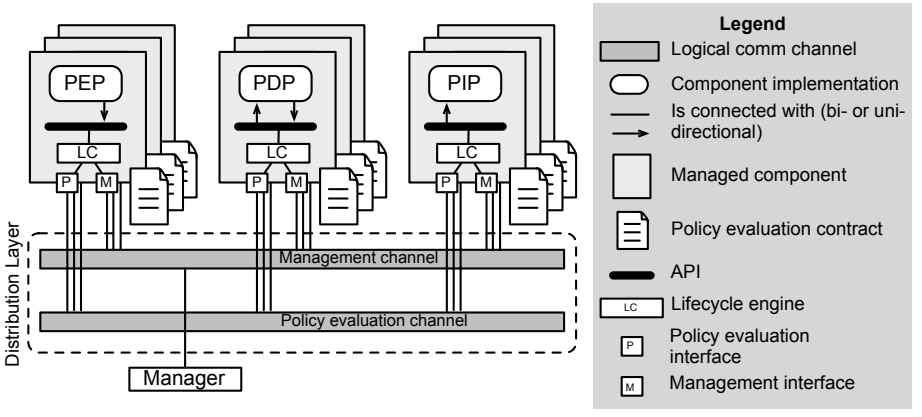



**Fig. 2.** Overview of the authorization middleware

### 3.1 Managed Authorization Components

Authorization components are treated as first-class entities. As shown in Figure 2, an authorization component consists of several elements.

Each component exposes two interfaces towards the rest of the authorization middleware. The *policy evaluation interface* is used during policy evaluation and implements generic and stable protocols that define the interactions between PEPs, PIPs and PDPs: requesting and returning authorization decisions and missing attribute values. This interface is refined for each individual authorization component by means of a *policy evaluation contract* (further abbreviated as contract). The contract lists the dependencies and capabilities of each authorization component in terms of which authorization and attribute requests are required or provided. We refer to [4] for a detailed description of these contracts. Secondly, the *management interface* is used by the centralized manager to inspect and modify the states and to control the incoming and outgoing interactions of the authorization components. The management interface also has operations to distribute policies to PDPs.

In addition, the middleware uses a lifecycle model for authorization components, which is shown in Figure 3 and is implemented by the lifecycle engine (see Figure 2). The lifecycle model is similar to dynamic module systems such as OSGi[9] and Java Business Integration (JBI)[10] and consists of four states. Initially, each component publishes its *capability contract* to the manager, which can then then deploy the component under a stricter version of its contract (with less provided elements), that is called the *deployed contract*. A component is only allowed to participate in policy evaluation once it has been activated by the manager.

The middleware also provides an API for implementing authorization components that maps the underlying message-based interface to an object-oriented interface and that makes the entire middleware appear as a single PDP or PIP.

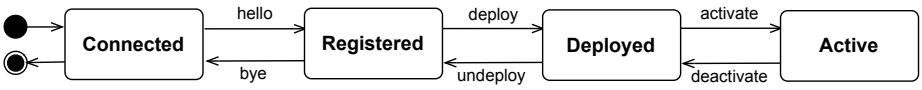



**Fig. 3.** State machine diagram of the authorization component life cycle model

### 3.2 Distribution Layer

The distribution layer mediates the remote interactions between authorization components and the manager. Figure 2 shows that the distribution layer provides two logical channels, one for policy evaluation and one for management.

Typically, the interactions between authorization components are implemented using remote method invocations. However, this model tightly couples the authorization components in space: PEPs need to know which PDPs to contact and PDPs need to know which PIPs to contact. This makes it hard to dynamically adapt the interactions. Therefore, we have opted for a messaging-based distribution layer that loosely couples authorization components. Authorization and attribute requests are sent to the distribution layer as destination-less messages. The distribution layer automatically routes requests to compatible components. Routing is configured using component-side *subscriptions filters* that indicate which resource or attribute types the component is interested in. The subscription filters are completely shielded from the component implementation and can only be influenced by the lifecycle engine (and, thus, indirectly by the manager). Filters are derived from the policy evaluation contracts and can be updated dynamically.

### 3.3 Manager

The flexibility of the routing of authorization and attribute requests provided by the distribution layer ensures that any component can be dynamically connected to any other component at runtime. The main responsibility of the manager component is to deploy and activate authorization components and ensure that (re)configurations are carefully controlled such that the infrastructure is and remains able to enforce all deployed policies. At its core, the manager consists of a set of primitives that allow administrators to safely configure the system, load policies to remote PDPs and perform efficient reconfigurations in response to policy and application changes. The effective component-side (re)configurations are implemented as state changes and their execution is delegated to the lifecycle engines of the deployed authorization components. In the rest of this section, we discuss each of the management primitives in detail.

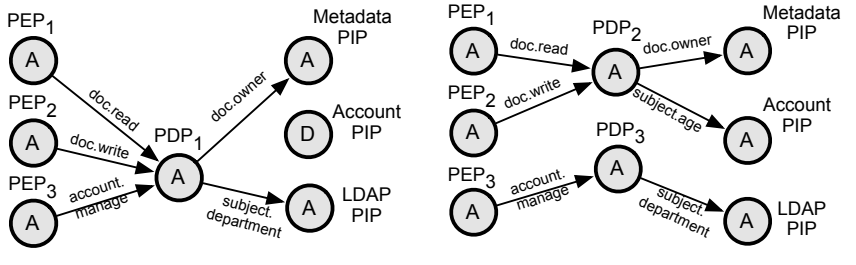
*Deployment/Undeployment.* The deployment primitive assigns a deployed contract to a component. The deployed contract can provide less attributes or resource types in order to resolve conflicts (see further). Likewise, undeployment removes the contract associated with a given component.

*Cascading Activation/Deactivation.* A deployed component needs to be activated before it can communicate with other components. The manager guarantees the following invariant for the set of services that are active: *each possible authorization or attribute request made by a component is handled by exactly one other component.* The manager uses the algorithm described in [4] to compute a dependency graph of all deployed components, based on the dependencies described in the contracts. When activating one component, the manager uses the dependency graph to obtain all components that are directly or indirectly required by the given component. Each of those components that is not already active, is also activated. In the case that it is impossible to satisfy all dependencies, the activation request is refused. If there is a conflict because multiple components handle the same requests (eg. two PIPs provide the same attribute), a selection must be made of one of the providers.

Figure 4(a) shows a dependency graph of a part of the PCM system. In case no single component is activated yet, the activation of $PEP_2$ will trigger the activation of the PDP and its two required PIPs. Deactivation occurs similarly, except that a cascading deactivation is performed of all components that directly or indirectly require the given component. Thus, deactivating the LDAP PIP also deactivates the PDP and all PEPs.

It can be the case that it is impossible to find a proper selection because of overlaps. For instance, suppose that the PDP to activate requires the attributes `age`, `location` and `presence` and that one PIP provides `age` and `location` and another PIP provides `location` and `presence`). Such conflicts must first be resolved by restricting the deployed contract of one of the two PIPs.

*Contract and Policy Updates.* We reconsider the example shown in Figure 4(a) in case of a policy update. Suppose that the document policy is updated with a new rule that is based on the `subject.age` attribute. The policy update is reflected in a contract update that makes the PDP dependent on that attribute. Suppose that the Account PIP deployed and provides `subject.age`. With only the basic management primitives, implementing such an update would require




(a) Basic example topology. (b) Desired topology after migration.

**Fig. 4.** Dependency graphs of example topologies. $PDP_1$ is the Authorization Server PDP, $PEP_1$ and $PEP_2$ are third party service PEPs and $PEP_3$ is the PCM Controller PEP.

the deactivation the PDP, including all depending PEPs, change the contract and then reactivate the PDP and PEPs. To limit disruption in such cases, the manager implements an optimized `updateContract` primitive. Because changes that trigger contract updates originate in the components, the components can trigger the contract update themselves.

We only discuss the non-trivial case in which the involved component is active. The `updateContract` primitive takes the affected component (with old capability and deployed contracts $c_c$ and $c_d$) and the new capability contract $c'_c$ as parameters and consists of the following steps:

1. Generate a new deployed contract $c'_d$ that requires the same elements as $c'_c$ and that provides the same elements as $c'_c$, except for the provided elements that $c_d$ also omitted from $c_c$.
2. Remove all provided elements from $c'_d$ that are already provided by other active components.
3. Try to replace $c_d$ with $c'_d$ in the dependency graph of active components.
   - If this maintains the invariant (one unique provider for every required element), the update is allowed and $c'_d$ is loaded to the component.
   - If additional elements are required that are uniquely provided by deployed components, those deployed components are activated and then $c'_d$ is loaded to the component.
   - If the new contract breaks incoming dependencies that can be uniquely satisfied by other deployed components, activate those components and then load $c'_d$ to the component.
   - Otherwise, the update is refused and manual deconfliction is required.

If the algorithm is applied to the example, the PIP that provides `subject.age` would be activated and the PDP is briefly interrupted while committing the updated contract.

*Migration.* Consider the basic topology after the update (the topology of Figure 4(a) with an added dependency of the PDP on the Account PIP for `subject.age`). Suppose that $\text{PDP}_1$ has become a performance bottleneck and that we want to move its policies to two new PDPs, called $\text{PDP}_2$ and $\text{PDP}_3$. The desired topology after the migration is shown in Figure 4(b). The implementation of such a reconfiguration would again require the de- and reactivation of all PDPs and PEPs. The manager provides a `migrate` primitive to deal with migrations like these efficiently.

The migration algorithm migrates from a set of active components $C$ to a set of deployed components $C'$ and goes as follows:

1. Check for internal violations within $C'$. No element required by components in $C'$ may be provided by more than one component in $C'$.
2. Check for external violations by verifying that the dependency graph for the components after the migration does not invalidate the invariant.
3. If these two conditions hold, deactivate all components of $C$ and activate all components of $C'$. Otherwise refuse the migration and report the violating dependencies.

## 4 Prototype

We have built a prototype of our architecture in Java. The size of the prototype is around 12000 lines of code. The prototype is designed to be extensible with different implementations of the distribution layer. The distribution layer is based on the Java Message Service (JMS) API and Apache's ActiveMQ implementation thereof. We use the topic-based publish/subscribe of the JMS API for message delivery between components. Each JVM maintains one session with ActiveMQ to publish messages and one session per component for receiving messages. Per action on a resource type and per attribute type, there is one JMS topic. Components publish requests to the topic that corresponds to the resource or the attribute. Authorization and attribute requests are represented as objects that are embedded in JMS object messages. Because of the large number of topics, one change was made to the default configuration so that topics are processed by a thread pool instead of a single thread per topic. All JMS messages are sent non-persistently.

We have also developed an XACML-based PDP that is based on Sun's XACML implementation (version 1.2). Attribute retrieval is integrated by means of a custom `AttributeFinder` module that uses our middleware. Furthermore, to study the middleware in isolation from the environments and policy engines it must be integrated with, we have developed synthetic PEP, PDP and PIP components. These synthetic components do not contain actual interception, policy evaluation or attribute retrieval logic, but simulate it. The synthetic components are fully configurable and allow us to instantiate and study the behavior of the middleware for large topologies.

## 5 Evaluation and Discussion

In this section, we evaluate the scalability of the proposed middleware by measuring the impact of the complexity of the instantiated authorization infrastructure on the performance of the supported interactions and reconfigurations. Furthermore, we discuss the applicability of the middleware in terms of the approach-specific development efforts and we indicate how these efforts can be reduced or partially automated.

One of the core goals of the proposed middleware is to scale to large and complex setups that are infeasible to manage manually. We have evaluated the scalability of (re)configuration operations as well as the runtime performance impact of the middleware in a distributed setup that consists of 12 identical Pentium 4 2GHz machines with 512MB RAM, interconnected by a 100Mbps switch and running a vanilla Kubuntu 10.04 install with the Sun Java 1.6.0_20 JVM. One node runs the ActiveMQ server, another node runs the manager component and collects the results, and the other 10 nodes run authorization components. To scale the complexity of the infrastructures, we have used the synthetic authorization components discussed in Section 4. We use the notation $[X, Y, Z]$ to represent an authorization infrastructure consisting of $X$ PEPs, $Y$

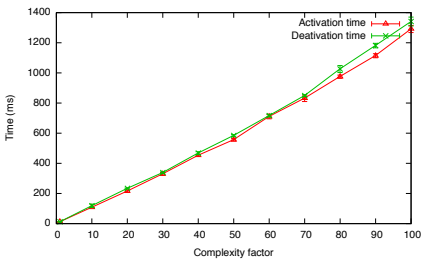
PDPs and $Z$ PIPs. In the rest of this section, we discuss the obtained results in detail in logical chronological order.

*Activation/deactivation.* Before an authorization infrastructure can be used, it must first be activated. Therefore, we first study the scalability of the activation and deactivation times of an entire infrastructure in function of its complexity. We have measured the time it takes to activate and deactivate 10 infrastructures of increasing complexity, where the simplest configuration of $[10, 1, 2]$ is multiplied by a factor of 1 to 100 in steps of 10. The setup has worst-case dependencies (each PEP depends on all PDPs and each PDP depends on all PIPs). Figure 5(a) shows the resulting activation and deactivation times. For the largest setup of $[1000, 100, 200]$, the activation and deactivation times are both approximately 1.2s. The performance increase is polynomial because dependency calculations are affected by both the increase in number of contracts as well as number of dependencies. However, the relative overhead of the dependency calculations is very small in comparison to the communication overhead of the (de)activation.

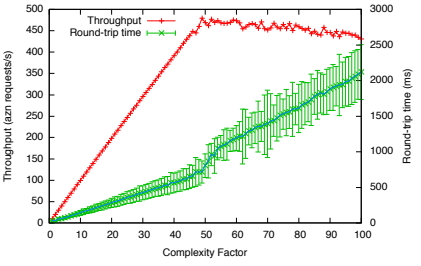
*Performance degradation.* The distribution overhead caused by the middleware once components are collaborating must degrade gracefully with the complexity of the setup. Therefore, we have measured the throughput and mean round-trip times of authorization requests in setups of increasing complexity. The same basic setup of $[10, 1, 2]$ was used as in the previous experiment, but the complexity factor was increased in steps of 1. Each PEP sends one random request per second and each PDP requests two random attributes per authorization request in parallel. From the results in Figure 5(b), we observe that the setup is unsaturated for all infrastructures smaller than $[500, 50, 100]$. From then on, the CPU of the JMS server becomes the bottleneck and the throughput starts to degrade from its maximum of 470 requests/s to 430 requests/s for $[1000, 100, 200]$. As desired, this degradation is graceful. Similarly, the round-trip times degrade linearly but the slope slightly increases after the saturation point.

*Contract Update.* Next, we consider the scalability of a dynamic policy (and, by consequence, contract) update in a running infrastructure with 1 PDP that depends on 50 of 100 PIPs. Such an update briefly disrupts the PDP and this disruption time must scale with the number of PEPs that depend on the PDP. To evaluate this, we have changed the number of depending PEPs from 1 to 1000 in steps of 50. For each infrastructure, the PDP contract is updated 20 times to depend on a random subset of 50 PIPs. The mean of the resulting disruption times are reported in Figure 5(c). We observe that the disruption times are always lower than 50ms and that they increase linearly with the number of depending PEPs. The disruption time in case of 1000 depending PEPs is only 25% higher than for one depending PEP.

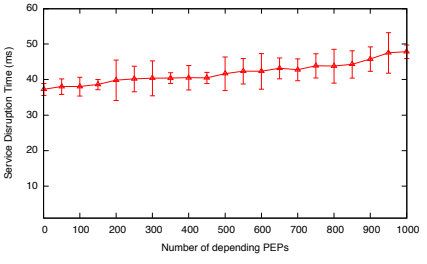
*Migration.* In the second reconfiguration experiment, we study the scalability of component migration. The base setup consists of one PDP that depends on 50 of 100 PIPs and that is migrated to a second PDP that depends on a different subset



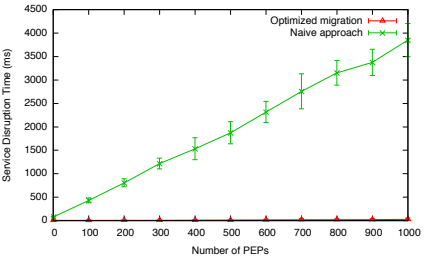

(a) Activation and deactivation time vs. infrastructure complexity.




(b) Round-trip times and throughput vs. infrastructure complexity.




(c) Service disruption time for a contract update of one PDP in function of the number of depending PEPs.




(d) Service disruption time for migrating from one PDP to another in function of number of depending PEPs.

**Fig. 5.** Scalability results

of 50 PIPs, which disrupts all depending PEPs. The performance of the migration must scale with the number of PEPs that depend on the PDPs, which is varied from 1 to 1000 in steps of 100. We compare the optimized primitive versus a naive approach in which the setup is first completely deactivated and then reactivated. Figure 5(d) shows the resulting disruption times. The disruption time caused by the naive approach scales linearly with respect to the number of PEPs and takes almost 4 seconds for a 1000 depending PEPs. The optimized migration algorithm drastically reduces the disruption time to a maximum of 30.9 ms by avoiding cascading deactivation. The optimized version is still scales linearly, but the slope is so small that it is not visible in the plot.

The integration of a distributed application with the proposed authorization middleware requires some additional effort. First, custom PEPs and PIPs must be implemented. However, since this is not specific to our approach, we like to refer to existing work that addresses this problem [11,1]. Secondly, each authorization component needs to be enhanced with a contract. This effort can be reduced because PDP contracts can be derived automatically from the policies based on syntactic analysis and because PEP/PIP contracts can be codeveloped with the components [11].

## 6 Related Work

Because SOAs are typically built using Enterprise Service Buses (ESBs), policies are often enforced at the the ESB level [3]. Such approaches work well for protocol-level policies but are not suited for enforcing more expressive higher level policies. Therefore, we see both approaches as complementary enforcement strategies for different types of policies.

To the best of our knowledge, there are two authors that have proposed to use messaging for interactions between security components. McDaniel [7] proposes a flexible security policy enforcement architecture for the Antigone group communication system in which security mechanisms are composed over an event bus. Wei [12] proposes the publish-subscribe model between remote PEPs and PDPs and confirms that this improves flexibility and availability. These approaches do not consider the consistency aspect. In our opinion, the flexibility gains of messaging must be carefully controlled by higher order management functionality that preserves consistency.

The deployment model of the Ponder policy language [2] manages policy deployment in distributed settings. Policies are translated into native access control configurations and are enforced local to the protected resources. This reduces the runtime overhead in comparison to or approach, but makes updates more complex: each policy update or new enforcement component requires policy redeployment. Moreover, although remote attribute retrieval could be implemented, it must be mapped to all native mechanisms, which is not always feasible. Additionally, if policies are heavily dependent on remote attributes, native enforcement might not be the best choice from a performance standpoint.

The run-time reconfiguration process discussed in this paper only ensures that a correct composition is achieved at the end of the reconfiguration process. In contrast and as a complement, safe reconfiguration approaches for distributed services (such as [6]) can guarantee that no single request is lost during the reconfiguration process or is processed by an incorrect composition.

## 7 Conclusion

We have presented a distributed authorization middleware for ABAC policy evaluation. The middleware supports the interactions between remotely deployed PEPs, PDPs and PIPs and manages their policy- and application-specific dependencies. By guaranteeing the consistency of the configuration with respect to the deployed policies, the system supports dynamic (re)configuration of the authorization infrastructure in response to policy or application changes. We have shown that the distribution and reconfiguration overhead scale to hundreds of authorization components and thousands of dependencies. In future work, we will improve the flexibility of the selection of required components and we will extend the system with more automated management processes such as PDP load balancing or performance optimization through automatic policy redistribution.

**Acknowledgements.** This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT and the Research Fund K.U. Leuven.

## References

1. Beznosov, K.: Object security attributes: Enabling application-specific access control in middleware. In: On the Move to Meaningful Internet Systems - DOA/-CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, London, UK, pp. 693–710. Springer, Heidelberg (2002)
2. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. In: IEEE/IFIP International Symposium on Integrated Network Management Proceedings, pp. 529–543 (2001)
3. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: Nishigaki, M., Jøsang, A., Murayama, Y., Marsh, S. (eds.) IFIPTM 2010. LNCS, vol. 321, pp. 63–78. Springer, Heidelberg (2010)
4. Goovaerts, T., De Win, B., Joosen, W.: Policy Evaluation Contracts. Technical report, Department of Computer Science, Katholieke Universiteit Leuven (2009)
5. IBBT. PeCMan project (Personal Content MANagement) (2007), `http://projects.ibbt.be/pecman`
6. Janssens, N., Joosen, W., Verbaeten, P.: Necoman: middleware for safe distributed-service adaptation in programmable networks. IEEE Distributed Systems Online 6(7) (2005)
7. McDaniel, P., Prakash, A.: A flexible architecture for security policy enforcement. In: Proceedings of DARPA Information Survivability Conference and Exposition, vol. 2 (2003)
8. OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0 (December 2005)
9. OSGi Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.1 (May 2007)
10. Ten-Hove, R., Walker, P.: Java Business Integration (JBI) 1.0 Final Release (August 2005)
11. Verhanneman, T., Piessens, F., De Win, B., Truyen, E., Joosen, W.: A modular access control service for supporting application-specific policies. IEEE Distributed Systems Online 7 (2006)
12. Wei, Q., Ripeanu, M., Beznosov, K.: Authorization using the publish-subscribe model. In: Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, Washington, DC, USA, pp. 53–62. IEEE Computer Society, Los Alamitos (2008)
13. Yuan, E., Tong, J.: Attributed based access control (ABAC) for web services. In: IEEE International Conference on Web Services, pp. 561–569 (2005)

# Adaptable Authentication Model: Exploring Security with Weaker Attacker Models

Naveed Ahmed and Christian D. Jensen

Informatics and Mathematical Modelling
Technical University of Denmark
{naah,Christian.Jensen}@imm.dtu.dk

**Abstract.** Most methods for protocol analysis classify protocols as "broken" if they are vulnerable to attacks from a strong attacker, e.g., assuming the Dolev-Yao attacker model. In many cases, however, exploitation of existing vulnerabilities may not be practical and, moreover, not all applications may suffer because of the identified vulnerabilities. Therefore, we may need to analyze a protocol for weaker notions of security. In this paper, we present a security model that supports such weaker notions. In this model, the overall goals of an authentication protocol are broken into a finer granularity; for each fine level authentication goal, we determine the "least strongest-attacker" for which the authentication goal can be satisfied. We demonstrate that this model can be used to reason about the security of supposedly insecure protocols. Such adaptability is particularly useful in those applications where one may need to trade-off security relaxations against resource requirements.

## 1 Introduction

Authentication protocols are used to verify the 'identity' of a far-end entity. This may appear to be a simple goal, but in reality it entails establishing the *identification*, *existence*, *operativeness* and *willingness* of an entity.

*Identification* refers to selecting a particular identity within a group. *Existence* refers to the assurance that a particular entity once existed in the group. *Operativeness* is achieved if the entity is currently active in the group. *Willingness* is the assurance that the entity is aware of being authenticated and that it is willing to communicate. Combining these goals in different ways leads to what are commonly referred to as unilateral and mutual authentication [3].

A large number of (entity) authentication protocols have been proposed in the literature; new protocols are often introduced to address vulnerabilities identified in existing protocols. Traditionally, the objective of an authentication protocol designer is to provide the strongest possible security. So-called "standard authentication models" normally incorporate a quite powerful adversary and then the security analysis is aimed at verifying whether a given protocol is secure or

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 234–247, 2011.

not. The most common example is the use of Dolev-Yao attacker [11][1] in the security models used in most of the tools based on formal analysis [1,6].

We introduce the notion of an *adaptable authentication model* for the type of reasoning which estimates the actual level of security provided by an authentication protocol, i.e., we infer suitable values for different parameters in the authentication model such that the protocol can be shown secure; these different parameters may include the attacker capabilities and the assumptions about the operating environment. This methodology is fundamentally different from the traditional one in which these parameters are fixed in the security model. For example, it is not clear how to use a formal analysis tool [6] if we assume that the attacker cannot delete messages, without redesigning the tool itself.

The adaptable authentication model makes it possible to reason about the weaker notions of security (e.g., where an attacker cannot delete messages); not doing so may result in using inefficient protocols and discarding slightly insecure protocols. Not all real-world applications require the highest level of security (e.g., see Ksiezopolski et al. [15]); and some of the rest cannot afford to implement the required level of security due to resource constraints (e.g., see Burmester et al. [8], Lindskog's PhD thesis [16]).

Although entity authentication is a pivotal part of all communication security protocols, there are many applications where mere authentication — without subsequent communication — is required. One such domain, which we also consider in this paper, is Radio Frequency Identification (RFID). One motivation of doing so is to convey the core idea behind the *adaptable authentication model* in its simplest form.

The design of RFID based security protocols is a relatively challenging task because RFID tags are resource constrained devices with only a little memory and limited computational and communication capabilities. These constrains mandate the use of light-weight cryptography. This is not the only challenge however. The generic attacker for typical RFID system is very powerful, as the tags are not assumed to be tamper-proof. Moreover, privacy also needs to be considered in the design of RFID protocols — the use of RFID based identification in wearable items can lead to serious *privacy* concerns, e.g., some people may not want to be publicly identifiable if they wear clothes that have embedded RFID tags.

We believe that our *adaptable authentication model* can be useful in such applications where it is not feasible to implement the highest level of security and where the need of a trade-off, between security and resource constraints, exists. We do not, however, consider any of the RFID domain specific problems, e.g., the efficiency of reader side of a protocol, side channel attacks and relay attacks, which may also be crucial to the feasibility of an authentication protocol.

We start by briefly considering the related work in § 2. In § 3 we describe the theoretical foundation of the model. This is followed by the concrete definitions

[1] Informally, such an attacker runs the communication network, and therefore can insert, delete, modify, delay and replay any message. The attacker, however, cannot break standard cryptographic schemes, such as encryption and signature.

of the model in § 4. We consider a simple RFID system and reason about the security of a generic authentication protocol in § 5. In § 6 we discuss the usefulness of our contribution and at last, in § 7, we conclude the work. All the proofs of security can be found in our technical report (TR) [2].

## 2 Related Work

We focus on the work directly related to the flexibility in security models. Various other styles of formal definitions for authentication can be found, e.g., in static analysis [6] and type theory [1]. The readers are referred to Avoine's thesis [5] for a general introduction and prior art of RFID security and privacy at that time. Outside the RFID community, the book of Boyd et al. [7] contains a large number of authentication protocols and the list of reported attacks against them.

Security and performance trade-offs in client-server environments are studied in Authenticast [21], which is a dynamic authentication protocol. The adaptation is due to flexible selection of key length, algorithm and the percentage of total packets that are authenticated. The term Quality of Protection (QoP) is also used to describe adaptable security models. Ong, et al. [19], address the problems introduced by the traditional view of security — a system is either secure or insecure— by defining different security levels based on key size, block size, type of data and interval of security[2].

Hager [14] considers the trade-offs of security protocols in wireless networks; but, the security adaption is only on the basis of performance, energy, and resource consumption. Covington, et al. [9], propose Parameterized Authentication, in which quality of authentication is described in terms of sensor trustworthiness and the accuracy of the measurements.

Lindskog [16] develops some solutions in his thesis, for tuning security for networked applications. The proposed methods are however limited to confidentiality. Instead of using just one instance of authentication protocol, in some approaches, e.g., by Ganger [13], over a period of time a system can fuse observations about the entities into a kind of probabilistic authentication.

Ksiezopolski et al. [15] describe the problem of an unnecessarily high level of security that may have impacts on dependability; they present an informal model of adaptable security, which, however, is difficult to justify for the soundness of results. Sun et al. [22] propose an evaluation method for QoP, based on a normalized weighted tree. Most of the existing QoP based approaches cannot be justified on concrete basis of modern cryptography. Interestingly, most of the foundational work for adaptable authentication is in the domain of RFID.

Many of the proposed RFID protocols [17] are too heavy for low cost tags, and not supported by EPCGen2 [12]. Burmester et al. [8] report that five different proposals that are compliant to EPCGen2 but have some security vulnerabilities. Currently, there are many parallel efforts going on to develop (adaptable) privacy models that can be used to capture the security and privacy requirements in order to optimize the resource requirement for the protocols.

[2] This implies to the time period in which data protection is meaningful.

Damgård et al. [10] study the trade-offs between complexity and security using secret key cryptography. They propose a weaker but more practical notion of privacy; strong privacy requires a separate and independent key for each of the RFID tags. Vaudenay [23] uses eight different attacker models to reason about the security of RFID identification protocols. The strongest notion in Vaudenay's model is shown to be impossible to achieve; even the two other strong models mandate the use of public key cryptography. This model also serves as an inspiration for much of the following work where the authors model the adversary as a class of attacker models, e.g., Paise et al. [20] and Yu Ng et al. [18].

The proposal in this paper is radical in the sense that we define adaptability over the definition of authentication. In all previous work, adaptability is defined over the attacker's capabilities, strength of cryptographic algorithms, trustworthiness of credentials or use of multiple channels (e.g., multi-factor authentication and context-aware authentication). Some of our previous work [3,4] can be considered as a pre-cursor to this work; the major developments in this paper are the proposal of general operational definitions for authentication, use of probabilistic arguments for LSAs (least strongest-attackers [2]) and the evaluation of our model in the RFID domain.

## 3 Overview of the Approach

Traditionally, protocols are analyzed for security using a fixed security model that typically incorporates an all powerful attacker. The end-result of such an analysis is the answer to whether or not the given protocol is secure or not. This approach is intuitively described in the following allegorical example.

Imagine a system where the only feasible operation is multiplication and our task is to establish the truth value of expressions of type $a \times x \geq b$, where $a, b, x \in D$ and $D$ is some finite set of natural numbers. Let, $a$ and $b$ be the given values in the specification of expression, while $x$ is an unknown value. Of course, the natural approach for solving this problem is to assume $x = x^{\perp}$, where $x^{\perp}$ is the smallest number in $D$ and then evaluate the expression, $a \times x^{\top} \geq b$. This approach is similar to how security is analyzed traditionally — in which the evaluation of the expression is replaced by the security analysis, $a$ corresponds to a given protocol, $b$ is the required security goal and $x^{\perp}$ is the security model that incorporates the all powerful attacker.

On the other hand, by initially assuming a given protocol as a secure protocol, we may need to find out the "strongest" security model for which the initial assumption remains valid. Allegorically, we assume $a \times x \geq b$ to be true and try to find the smallest $x$ in $D$ such that this assumption remains valid. For this purpose, a naive (brute-force) approach is to iterate the evaluation of expression: start with the largest $x$; if the expression is valid then decrement $x$ and repeat the process, otherwise, the value of $x$ in the previous iteration is the solution. Similarly, such a naive approach for security requires iterating the security analysis over the possible types of security models.

In our proposal, we present a single framework that can be used to find out such a "strongest security model". In the example, this means solving $a \times x \geq b$ as an inverse problem, even though $x = a \div b$ is not feasible to compute directly. The precise description of our main idea and its formal construction is presented in the following.

Let $\Theta$ be a background theory[3], $\alpha$ be an attacker model, $\Pi$ be an authentication protocol and $\mathcal{G}$ be one of the authentication goals[4]. The standard approach towards security analysis is in the following general form.

$$\Theta, \Pi, \alpha \models \mathcal{G} \tag{1}$$

The above argument represents the security analysis as a process in which one tries to show, under the security model $\{\Theta, \alpha\}$, that $\mathcal{G}$ can be achieved by running an instance of $\Pi$. It must be noted that the above argument is not confined to classic mathematical logic; the meaning of '$\models$' depends on the type of analysis, e.g., in complexity theoretic cryptography $\models$ stands for reductionist type proofs, in formal analysis it may stand for static analysis [6].

We aim to formulate the above argument as an inverse problem, in order to determine certain parameters of the security model such that $\mathcal{G}$ can be inferred. In place of Equation 1, we use the following two abductive style arguments in the adaptable authentication model.

$$\Theta, \Pi, \beta \models \mathcal{G} \tag{2}$$

$$\Theta, \Pi, \alpha \models \beta \tag{3}$$

We refer to Equation 2 as the *Beta argument* and Equation 3 as the *Alpha argument*. These two arguments are proposed to divide the classic authentication problem (Equation 1) in two, by introducing an intermediate statement $\beta$. The concrete form of $\beta$ is defined in § 4. For now, $\beta$ may be considered as a special security property over the protocol messages.

Two types of processes are involved in both of the arguments above. The first one is an *abduction process*, which refers to hypothesizing $\beta$ in the Beta argument and $\alpha$ in the Alpha argument. The second process is called *validation process*, which deals with the validity of the arguments itself. So, in total we need to consider the following four processes.

1. Abduction process in the Beta argument, i.e., hypothesizing $\beta$
2. Validation process in the Beta argument, i.e., validating $\mathcal{G}$
3. Abduction process in the Alpha argument, i.e., hypothesizing $\alpha$
4. Validation process in the Alpha argument, i.e., validating $\beta$

The process of hypothesizing $\beta$ is trivial, as $\Pi$ (in the class of authentication protocols) typically contains a relatively small number of base terms. So, we can

[3] A security model may consists of an environment model, a system model, an attacker model $\alpha$, semantics of terms, etc; $\Theta$ refers to all these components except $\alpha$.

[4] For more details see Definition of Authentication [3].

exhaustively search the possible space of $\beta$. For example, if $\Pi$ is a challenge response protocol then there are only two messages, a challenge $m_1$ and a response $m_2$; the possibilities for which $\beta$ may be satisfied are only three: $\{m_1, m_2\}$, $\{m_1\}$ and $\{m_2\}$.

The *validation process* for $\mathcal{G}$ in the Beta argument is defined in § 4 under the names of *operational definitions*. A valid $\beta$ in the Beta argument is *just an hypothesis* in the complete model and thus a separate validation argument for $\beta$ is required, which is the Alpha argument. If the Alpha argument can be validated, for some $\alpha$, then $\beta$ becomes a valid statement, but apparently at the cost of $\alpha$ being a hypothesis.

Traditionally, the issue '$\alpha$ as a hypothesis' — i.e., $\alpha$ really models the attacker for a specific domain? — is not so important because $\alpha$ usually models an 'all powerful attacker'. For example, the Dolev-Yao attacker [11] is believed to subsume all conceivable attackers in most computer networks. So, one can safely assume the validity of the hypothesis (Dolev-Yao attacker) while, e.g., deploying a 'secure' authentication protocol in a cooperate network. In our proposal, however, the hypothesis $\alpha$ may correspond to a weaker attacker. Therefore, $\alpha$ obtained in the *adaptable* authentication model should not be assumed valid by default and a decision, whether $\alpha$ is reasonable to assume or not, must be made in an application specific manner.

For the Alpha argument, we outline a Hybrid process that incorporates both the abduction process (i.e., hypothesizing $\alpha$ for a given $\beta$) and a validation process (i.e., validating $\beta$ using $\Theta$, $\Pi$ and $\alpha$). The Hybrid process is overall sound[5] if the validation process is sound.

**The Hybrid process:** The Hybrid process starts with the validation process of the Alpha argument by assuming the capabilities of most powerful attacker, say $\alpha_i$, relevant to the given system model. If the validation process fails then depending on the cause of failure we weaken $\alpha_i$ to $\alpha_{i+1}$ and continue with the validation process once again[6]; the loop continues until the validation process succeed. The $\alpha$ for which the validation process succeed is the *least strongest-attacker* (LSA).

Consequently, the resultant LSA is not necessarily correspond to some standard attacker model, such as the Dolev-Yao attacker [11]. As long as the *validity process* is sound, $\alpha_i$ at *ith* iteration is rejected if it is not a valid hypothesis. The existence of any undiscovered 'stronger' LSA does not invalidate the earlier results, as any weaker attacker model is always a subset of the stronger model. Thus, the soundness of the Hybrid process is not a defeasible guarantee[7], as long as the background theory $\Theta$ remains the same.

[5] But, the process may not be complete or optimum.

[6] In general, the weakening process is not deterministic, as there could be more than one option for weakening $\alpha_i$ to $\alpha_{i+1}$. Therefore, the hybrid process is non-deterministic, however, every solution is valid (but may not be optimum).

[7] In classic reasoning, a valid but defeasible argument has the possibility of invalidation when more premises are added.

The Alpha and the Beta arguments could be probabilistic,[8] i.e., there is some probability $p < 1$ that a goal $\mathcal{G}$ is valid assuming the validity of $\Theta$, $\Pi$ and $\beta$. Similarly, there is some probability $q < 1$ that a $\beta$ is valid assuming the validity of the premises $\Theta$, $\Pi$ and $\alpha$. In the complete model, $\mathcal{G}$ is achieved with a probability that is a function of $p$ and $q$. The probabilistic concerns do not arise until we actually start validating the alpha and beta arguments.

To sum up, we highlight the two aspects that form the basis of the adaptability of *adaptable authentication model*. The first aspect is the way we formulated the Alpha and the Beta arguments; the Beta argument is *attacker-independent*; the Alpha argument is *goal-independent*. The second aspect is the abductive style treatment of the two arguments, namely the traditional security problem (whether $\mathcal{G}$ can be satisfied under a standard security model) is formulated as an inverse problem by breaking it in to four smaller problems: *abductive* and *validation* process for both Alpha and Beta arguments. This allows us to infer an actual security model for which a given authentication protocol is secure.

## 4 Adaptable Authentication Model

In this section, we present the concrete forms of $\beta$, the validation process of the Beta argument and a restricted form of the Hybrid process. The validation of the Beta argument means how to infer an arbitrary authentication goal $\mathcal{G}$ from $\Theta$, $\Pi$ and $\beta$; we define this process with the operational definitions of $\mathcal{G}$ that are in terms of $\beta$. On the other hand, we define the Hybrid process at an abstract level, i.e., for arbitrary form of $\alpha$, $\Pi$ and $\Theta$; an example of $\beta$ with a concrete Hybrid process (i.e., for the specific choices of $\alpha$, $\Pi$ and $\Theta$) can be found in §5.

Consider a network of entities who communicate with each other through message passing protocols; the protocols under consideration are of entity authentication. A protocol $\Pi$ may be executed among two or more entities. An instance of $\Pi$ is denoted by $\Pi(i)$, where $i$ is an index. We propose the following definition of *binding sequence* as a concrete form of $\beta$.

**Binding Sequence:** A sequence of protocol messages is called a binding sequence $\beta_X$ if the violation of any of the following properties generate an efficiently detectable event for an entity $X$.

1. Deletion/Insertion/Modification of a message in $\beta_X$.
2. Reordering of messages in $\beta_X$.

Intuitively, a binding sequence is a list of selected messages that preserves their "integrity", as all unauthorized changes in $\beta_X$ should be detectable for $X$; of course, $\beta_X$ may be replayed. Note that such an integrity property of $\beta_X$ is different from the integrity of the messages it contains. A $\beta_X$ can be constructed from completely unauthenticated messages.

Next, we propose *operational definitions* as a validation process of the Beta argument, namely we define the authentication goals for $X$ in terms of $\beta_X$. Our

[8] For example, in complexity theoretic proofs of security (e.g., see §2.6 [7]).

formulation is for a two-party case, but it is trivial to extend them to multi-party case[9]. In the following $A$ and $B$ denote specific network entities, while $X$ represents an arbitrary network entity. These operational definitions are also illustrated in Figure 1.

**Existence:** Let $\beta_A(i)$ and $\beta_A(j)$ be generated when $A$ executes the protocol with $B$ and $X$ respectively. If $A$ can efficiently distinguish between $\beta_A(i)$ and $\beta_A(j)$ (for all choices of $X$) then $A$ is said to achieve the goal *existence* of $B$ from $\beta_A(i)$. This is abbreviated as $\text{EXST}(A \rightarrow B, \beta_A(i))$.

**Operativeness:** Let $\beta_A(i)$ and $\beta_A(j)$ be generated when $A$ executes the protocol twice with $X$ . If $A$ can efficiently distinguish between $\beta_A(i)$ and $\beta_A(j)$ (for all choices of $X$) then $A$ is said to achieve the goal *operativeness* for $X$. This is abbreviated as $\text{OPER}(A \rightarrow B, \beta_A(i))$.

**Willingness:** Let $\beta_B(i)$ and $\beta_B(j)$ be generated when $A$ and $X$ execute the protocol with $B$. If $B$ can efficiently distinguish between $\beta_B(i)$ and $\beta_B(j)$ for all choices of $X$ then $A$ is said to achieve the goal *willingness* for $B$, from the corresponding binding sequence $\beta_A(i)$. This is abbreviated as $\text{WLNG}(A \rightarrow B, \beta_A(i))$.

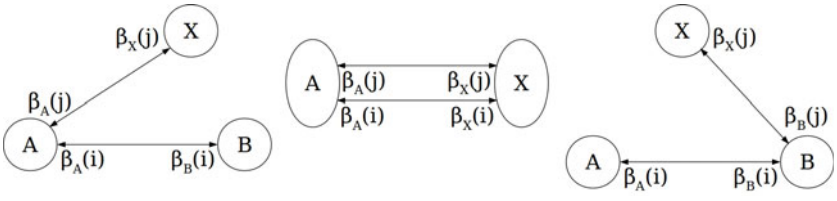



**Fig. 1.** Operational Definitions: (a) Existence (b) Operativeness (c) Willingness

**Single-sided Authentication:** If an entity $A$ achieves $\text{OPER}(A \rightarrow B, \beta_A(i))$, $\text{EXST}(A \rightarrow B, \beta_A(i))$, and $\text{WLNG}(A \rightarrow B, \beta_A(i))$ then $A$ is said to achieve single sided authentication, abbreviated as $\text{SATH}(A \rightarrow B, \beta_A(i))$.

In the above definition, the goals (existence, operativeness and willingness) are achieved in the same instance $\beta_A(i)$. We skip the operational definitions for rest of the authentication goals due to space constraints (see TR [2] for details). We turn to the privacy related goals. We introduce a notion of *controllability* as follows. A $\Pi$ is *controllable* by peer entities if the union of their binding sequences, $\bigcup_Y \beta_Y$, contains all the messages of $\Pi$, where $Y$ is some peer entity in $\Pi$. So, controllability implies that each protocol message is part of at least one binding sequence belonging to the peer entities. Consequently, we have the following relation for an instance $\Pi(i)$ of a *controllable* protocol: $\Pi(i) = \bigcup_Y \beta_Y(i)$.

[9] For instance, if $A$ is supposed to achieve $\mathcal{G}$ for both $B$ and $C$ then the operational definition of $\mathcal{G}$ for $A$ to $B$ and for $A$ to $C$ must be satisfiable using a common $\beta_X$.

**Anonymity:** Let $\Pi(m)$ and $\Pi(n)$ be the two instances of a controllable $\Pi$, such that $\Pi(m)$ involves an entity $X$ and $\Pi(n)$ does not involve $X$, as an initiator or responder. The goal *Anonymity*, abbreviated as ANMT$(X, \Pi)$, is achieved if an adversary $\mathcal{A}$ cannot distinguish between $\Pi(m)$ and $\Pi(n)$.

**Untraceability:** Let $\Pi(m)$, $\Pi(n)$ and $\Pi(o)$ be the three instances of a controllable $\Pi$, such that $\Pi(m)$ and $\Pi(n)$ involve $X$ as an initiator or responder, while $\Pi(o)$ does not involve $X$. The gaol *untraceability*, or UNTC$(X, \Pi)$, is achieved if $\mathcal{A}$ cannot pick the pair $(\Pi(m), \Pi(n))$ with significantly different probability than either of the rest of two pairs, namely $(\Pi(m), \Pi(o))$ and $(\Pi(n), \Pi(o))$.

At last, we define an abstract form of the Hybrid process, which is used to hypothesize $\alpha$ and to validate $\beta$ at the same time. For simplicity, we restrict the abstract form to those protocols in which the number of messages are fixed. Moreover, we assume that all messages are well typed. With these restrictions, the only property that needs to be checked (as per definition of binding sequence) is *modification*.

Consider an arbitrary binding sequence: $\beta_X = [m_1, ..., m_n]$, where $n \geq 1$. Let $m_i$, with $1 \leq i < n$, be an *unmodified* message in $\beta_X$. Let $m'_i$ represents a message obtained by modifying $m_i$. Let ACCEPT$(\beta_X)$ be an event that $X$ accepts[10] $\beta_X$. Let $M_{\beta_X}$ be a set that contains all choices of modified $\beta_X$, e.g., if $\beta_A = [m_1, m_2]$ then $M_{\beta_A} = \{[m'_1, m_2], [m_1, m'_2], [m'_1, m'_2]\}$. In general, $M_{\beta_X}$ contains $2^n - 1$ elements.

**The generic Hybrid process:** We consider each element of $M_{\beta_X}$ and calculate $Pr(\text{ACCEPT}(\beta^i_X))$, where $\beta^i_X \in M_{\beta_X}$ and $1 \leq i \leq (2^n - 1)$. If $Pr(\text{ACCEPT}(\beta^i_X))$ is negligible then we consider another element in $M_{\beta_X}$, until no element is left in $M_{\beta_X}$. If $Pr(\text{ACCEPT}(\beta^i_X))$ is not negligible, we include reasonable assumptions so that $Pr(\text{ACCEPT}(\beta^i_X))$ is negligible under the new assumptions. These additional assumption could be related to $\alpha$ or the environment in which $\Pi$ operates. In the worst case, $\alpha$ is empty.[11]

In the next section, we demonstrate the utility of the adaptable authentication model by applying the approach to a relatively simple, but realistic, system, namely RFID authentication. The purpose of the paper, however, is not to address the general RFID authentication and privacy problems.

## 5 Case Study: A Simple RFID System

All RFID tags that we consider here are passive transponders identified by an ID; the ID is not necessarily unique. In practice, this ID may correspond to the item to which the tag is attached.

[10] $X$ accepts $\beta_X$ as per the definition of binding sequence.

[11] This corresponds to an attacker with no capabilities besides what is assumed in the ideal execution of $\Pi$.

We define four classes of attacker capabilities: Destructive, $\alpha^D$; Forward, $\alpha^F$; Weak, $\alpha^W$; and Coward, $\alpha^C$. An attacker $\mathcal{A}$ may belong to one of these classes, which are modeled over the following set of oracles.

- CreateTag($k$): $\mathcal{A}$ can create a tag with a key $k$ stored in it.
- Launch($\Pi(i)$,ID): $\mathcal{A}$ can interact with a tag ID of his choice.
- Respond($\Pi(i)$): $\mathcal{A}$ can respond to a RFID reader's interrogation.
- Corrupt(ID): $\mathcal{A}$ can read all memory contents of any tag ID.

$\mathcal{A}$ in Destructive class can access all of these oracles, but, if $\mathcal{A}$ access Corrupt oracle then the tag ID is destroyed as the tampering is assumed to be detectable. Forward class is similar to Destructive one, except no other type of oracle can be accessed after an access to Corrupt oracle is made; additional Corrupt queries are allowed. $\mathcal{A}$ in Weak class cannot access Corrupt oracle. $\mathcal{A}$ in Coward class cannot invoke Corrupt and Respond oracles.

We consider a relatively simple system model, where there is one reader $R$ and $n$ number of RFID tags[12]: $\mathrm{ID}_i$, $1 \leq i \leq n$; each tag is attached to an item in *Warehouse*. There is a single entry point *Entry* into Warehouse and a single exit point *Exit* from Warehouse. Whenever an item passes through Entry, it gets attached with a tag $\mathrm{ID}_i$ and a relevant entry $(\mathrm{ID}_i, K_i)$ is stored in the reader's database. When an item arrives at Exit, the tag is identified as $\mathrm{ID}_i$ by executing a protocol that is derived from the generic RFID authentication protocol, which we specify later. When a tag $\mathrm{ID}_i$ leaves Exit, it is killed (i.e., no more radio communication is possible with $\mathrm{ID}_i$) by $R$ and database entry $(\mathrm{ID}_i, K_i)$ is removed.

The reader $R$ may have multiple front-ends at Entry and Exit but has one common back-end. In reality, the back-end could be a distributed system in which the database of tags is synchronized in real-time. Whenever a tag is in Exit, the authentication protocol is automatically executed.

An attacker $\mathcal{A}$, in some attacker's class, exists inside the warehouse from where he can invoke the oracle queries following the rules of the attacker class. In addition to that, $\mathcal{A}$ also has access to customer records. Privacy is defined as protecting the relation between the item bought and the customer. We assume that an attacker is not capable to 'physically observe' the item all the way to a customer, but, if an attacker can link communication of a tag ID to a customer at Exit then he can violate the privacy.

Usually, the concrete definitions of *privacy* and *security* are system dependent. In our case, *anonymity* is not a concern as attacker is already in Warehouse. Similarly, the reader at Exit is assumed honest and therefore *willingness* is not a concern for security. We use the following definitions for our RFID system.

**Privacy:** A protocol $\Pi$, involving $R$ as an initiator and ID as a responder, is *private* if untraceability, $\mathrm{UNTC}(\mathrm{ID}, \Pi)$, can be achieved.

[12] The assumption of single reader is prevalent in the literature, e.g., see Damgård [10] and Vaudenay [23].

**Security:** A protocol $\Pi$ , involving $R$ as an initiator and ID as a responder, is *secure* if existence, EXST($R \rightarrow \text{ID}, \beta_R$), and operativeness, OPER($R \rightarrow \text{ID}, \beta_R$), can be achieved.

We consider the following generic RFID protocol, $\Pi$. Note that many of the existing protocols follow this generic form, e.g., weak-private RFID schemes [23].

1. $R \rightarrow \text{ID} : \text{challenge} = V_R$
2. $\text{ID} \rightarrow R : \text{response} = F_K(V_R, V_{\text{ID}})$

In this generic form, $F_K(..)$ is a PRF (pseudo-random function), $V_R$ is the value generated by the reader and $V_{\text{ID}}$ is the value generated by the tag. The reader $R$ can find (ID,K) in the database by the searching the value of $K$ from database, such that the following predicate is true: $p(\text{challenge}, \text{response}, K)$ (see TR [2]). For instance, in $\Pi_3$, with $V_{\text{ID}} = b$, we have,
$p(N_R, \text{response}, k) ::= (\text{response} = F_k(N_R, b))?\ \mathit{true} : \mathit{false}$.

As shown in Table 1, we can construct eight different concrete protocols, say $\Pi_1, \Pi_1, ..., \Pi_8$, over the following three parameters: the challenge $V_R$ is random or not; the value $V_{\text{ID}}$ is random or not; and the key $K$ is different or not for each tag.

Next, we analyze the eight concrete protocols for security and privacy, using the adaptable authentication model. As we may recall, there are three processes involved in the analysis, i.e., *abduction* and *validation* process for Beta argument and the Hybrid process for Alpha argument. The *abduction* process is trivial, namely $\beta_X = [V_R, F_K(V_R, V_{\text{ID}})]$ for the generic protocol. The *validation* process corroborates the *operational definitions* of relevant $\mathcal{G}$s, namely EXST($R \rightarrow \text{ID}, \beta_R$) and OPER($R \rightarrow \text{ID}, \beta_R$) for the *security*, and UNTC(ID, $\Pi_i$) (where $1 \leq i \leq 8$) for the *privacy*.

A concrete Hybrid process can be derived from the generic Hybrid process described in §4. In all of the protocols, *deletion*, *insertion* and *reordering* is not possible, which justifies the restrictions of the abstract Hybrid process. So, we only analyze for any undetectable modifications in $\beta_R$. Let $\mathcal{A}$ be an attacker from one of the attack classes. Let ACCEPT($\beta_R$) be an event that the reader accepts $\beta_R$. The generic, but concrete, Hybrid process is as follows.

1. $Pr(\text{ACCEPT}([V'_R, F_K(V_R, V_{\text{ID}})]))$:
   This case analyzes the probability of the event that $\mathcal{A}$ select $V'_R$, where $V'_R \neq V_R$, in such a way that $F_K(V_R, V_{\text{ID}})$ can be returned to the reader.
2. $Pr(\text{ACCEPT}([V_R, F_K(V_R, V_{\text{ID}})']))$:
   This case analyzes the probability of the event that $\mathcal{A}$ compute the response $F_K(V_R, V_{\text{ID}})'$, where $F_K(V_R, V_{\text{ID}})' \neq F_K(V_R, V_{\text{ID}})$, in such a way that probability of ACCEPT is close to 1.
3. $Pr(\text{ACCEPT}([V'_R, F_K(V_R, V_{\text{ID}})']))$:
   This case analyzes the probability of the event that $\mathcal{A}$ select $V'_R$, where $V'_R \neq V_R$, and compute the response $F_K(V_R, V_{\text{ID}})'$, where $F_K(V_R, V_{\text{ID}})' \neq F_K(V_R, V_{\text{ID}})$, in such a way that probability of ACCEPT is close to 1.

The actual security proofs for the eight concrete protocols can be found in the technical report [2]; the result of the analysis is summarized in Table 1. Each row in the table corresponds to one of the concrete protocols; the specific choices made for $V_R$, $V_{\mathrm{ID}}$ and $K$ are mentioned in the corresponding columns. The last column summarizes the result of our analysis, in form of assumptions required to justify the security and privacy of these protocols. The details of these assumptions are presented in the following list.

(a) The *existence* is not achieved, so, e.g., same type of items should be in Warehouse, or there should be some auxiliary (e.g., physical) mechanism to distinguish between items if they are different.
(b) Only one item is presented to the reader $R$ at a time.
(c) The *operativeness* is not achieved, so, e.g., visual inspection should be done at Exit to make sure the item with ID is currently there.
(d) The reader $R$ should query a tag more than once to detect collisions in the values of $V_{\mathrm{ID}}$.
(e) $F_k(...)$ is a PRP (pseudo-random permutation) with an efficiently computable inverse function $F_K^{-1}(...)$ (e.g., AES encryption).
(h) Privacy is not possible to achieve within the model, so, privacy should not be a concern for the items in Warehouse.

**Table 1.** Concrete Forms of the Generic Protocol

| Protocol | $K$ is different | $V_{ID}$ is random | $V_R$ is random | Results |
|---|---|---|---|---|
| $\Pi_1$ | No | No | No | {a,b,c,Destructive} |
| $\Pi_2$ | No | No | Yes | {a,b,Weak} |
| $\Pi_3$ | No | Yes | No | {a,b,d,e,Coward} |
| $\Pi_4$ | No | Yes | Yes | {a,b,e,Coward} |
| $\Pi_5$ | Yes | No | No | {c,h,Destructive } |
| $\Pi_6$ | Yes | No | Yes | {Destructive } |
| $\Pi_7$ | Yes | Yes | No | {d,e,Destructive} |
| $\Pi_8$ | Yes | Yes | Yes | {e,Destructive} |

Let us consider, for instance, the case $\Pi_2$ in the table. The protocol $\Pi_2$ corresponds to a system where all RFID tags share a common key and there is no pseudo-random generator implemented on the tags. The $i^{th}$ interrogation of the reader consists of a random challenge $N_R(i)$. As shown in the last column that the protocol is secure and private against Weak class of attackers, as long as the assumption, (a) and (b), are satisfied. This example illustrates the types of results that we obtain in adaptable authentication model, i.e., the appropriate parameters of authentication model required to justify security and privacy[13].

[13] Intuitively, the results may be traced back to $\alpha_i$ in § 3; one option is to let $\alpha_1$ be the Destructive class, $\alpha_2$ be the attacker class in the result column and $\alpha_3$ be $\alpha_2$ constrained by the corresponding assumptions.

## 6 Discussion

In traditional security analysis, most of the concrete protocols in Table 1 are 'insecure', with the possible exceptions of $\Pi_6$ and $\Pi_8$ (e.g., see the case of strong privacy [10]). This is due to a strict deductive style interpretation of classic security argument (Eq. 1) and use of a strict notion of attacker in the corresponding security model. For example, if $V_R$ is not random in a concrete protocol (i.e., one of $\Pi_1$, $\Pi_3$ or $\Pi_5$) then the protocol is simply considered 'insecure' under Dolev-Yao attacker [11] — as it is prone to replay attacks.

On the other hand, we consider security and privacy as an a-priori assumption, even for the weakest protocol $\Pi_1$. We infer an attacker model and a set of assumptions about the environment such that a weaker notion of security can be justified. It is not always possible to prove a security goal within the model, but this does not necessarily mean a dead-end to the analysis; Since our security and privacy goals are formulated at primitive level (compared to a single high level formulation, e.g., matching conversation [7]), impossibility of one fine level goal does not imply such impossibility for all other goals.

In the long run, the proposed model needs to be evaluated on a broader scale, with more authentication goals and for more general RFID system, albeit, the results in Table 1 are promising for the simple RFID system. In particular, there are two open questions that we like to address in future work. First, we need to determine whether or not the validity process of $\beta$ , namely the Alpha argument, is at most as hard as that of standard security analysis (Eq. 1). Secondly, whether or not the operational goals in terms of binding sequences are equivalent to some of other standard formulation, e.g., matching conversation [7], Blinders [23].

## 7 Conclusion

The adaptable authentication model can be used to reason about authentication protocols that are not normally considered secure as per "standard authentication models". We propose the adaptable authentication model as a step towards exploring a rather unexplored area of weak security. The results are particularly useful when one needs to optimize security for resource constrained systems, e.g., the protocols $\Pi_1$ and $\Pi_2$, which only require a hash function on the tags, may suffice for the requirements of certain applications.

It must be noted that the adaptable authentication model does not change the actual security of a protocol; the model just capture the weak form of security which otherwise labeled as insecurity under standard model. System designers, therefore, must be cautious while interpreting the results obtained in the adaptable model; security guarantees are accompanied by extra assumptions and typically a weaker attacker model, which may not be justifiable for every conceivable environment.

## References

1. Abadi, M.: Secrecy by typing in security protocols. J. ACM 46, 749–786 (1999)
2. Ahmed, N., Jensen, C.D.: Adaptable authentication model. Tech. Rep. IMM-Technical Report-2010-17, DTU Informatics, Lyngby, Denmark (2010)

3. Ahmed, N., Jensen, C.D.: Definition of entity authentication. In: 2nd International Workshop on Security and Communication Networks, pp. 1–7 (May 2010)
4. Ahmed, N., Jensen, C.D.: Entity authentication:analysis using structured intuition. In: Technical Report of NODES 2010 (2010)
5. Avoine, G.: Cryptography in Radio Frequency Identification and Fair Exchange Protocols. Ph.D. thesis, EPFL, Lausanne, Switzerland (2005)
6. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Riis Nielson, H.: Automatic validation of protocol narration. In: 16th CSFW, pp. 126–140 (2003)
7. Boyd, C., Mathuria, A.: Protocols for Authentication and Key Establishment. Springer Book, Heidelberg (2003)
8. Burmester, M., Munilla, J.: A flyweight RFID authentication protocol (2009), http://eprint.iacr.org/2009/212
9. Covington, M.J., Ahamad, M., Essa, I., Venkateswaran, H.: Parameterized authentication. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) ESORICS 2004. LNCS, vol. 3193, pp. 276–292. Springer, Heidelberg (2004)
10. Damgård, I., Pedersen, M.Ø.: RFID security: Tradeoffs between security and efficiency. In: Malkin, T.G. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 318–332. Springer, Heidelberg (2008)
11. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
12. EPC-Global: Epcglobal tag data standards version 1.3, ratified specification (2006), http://www.epcglobalus.org
13. Ganger, G.R.: Authentication confidences. Tech. Rep. CMU-CS-01-123, Carnegie Mellon University School of Computer Science (2001)
14. Hager, C.T.: Context Aware and Adaptive Security for Wireless Networks. Ph.D. thesis, Virginia Polytechnic Institute and State University (2004)
15. Ksiezopolski, B., Kotulski, Z.: Adaptable security mechanism for dynamic environments. Computers & Security 26(3), 246–255 (2007)
16. Lindskog, S.: Modeling and Tuning Security from a Quality of Service Perspective. Ph.D. thesis, Chalmers University of Technology, Sweden (2005)
17. Molnar, D., Soppera, A., Wagner, D.: A scalable, delegatable pseudonym protocol enabling ownership transfer of RFID tags. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 276–290. Springer, Heidelberg (2006)
18. Ng, C., Susilo, W., Mu, Y., Safavi-Naini, R.: RFID privacy models revisited. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 251–266. Springer, Heidelberg (2008)
19. Ong, C.S., Nahrstedt, K., Yuan, W.: Quality of protection for mobile multimedia applications. In: International Conference on Multimedia and Expo. (ICME), vol. 2, pp. II-137–II-140 (2003)
20. Paise, R.I., Vaudenay, S.: Mutual authentication in RFID: security and privacy. In: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, pp. 292–299. ACM, New York (2008)
21. Schneck, P.A., Schwan, K.: Dynamic authentication for high-performance networked applications. In: Sixth IWQoS, pp. 127–136 (May 1998)
22. Sun, Y., Kumar, A.: Quality-of-protection (QoP): A quantitative methodology to grade security services. In: 28th International Conference on Distributed Computing Systems Workshops (ICDCS), pp. 394–399 (2008)
23. Vaudenay, S.: On privacy models for RFID. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 68–87. Springer, Heidelberg (2007)

# Idea: Interactive Support for Secure Software Development

Jing Xie, Bill Chu, and Heather Richter Lipford

Department of Software and Information Systems
Center for Cyber Defense and Network Assurance
University of North Carolina at Charlotte
Charlotte, NC, USA
{jxie2,billchu,heather.lipford}@uncc.edu

**Abstract.** Security breaches are often caused by software bugs, which may frequently be due to developers' memory lapses, lack of attention/focus, and knowledge gaps. Developers have to contend with heavy cognitive loads to deal with issues such as functional requirements, deadlines, security, and runtime performance. We propose to integrate secure programming support seamlessly into Integrated Development Environments (IDEs) in order to help developers cope with their heavy cognitive load and reduce security errors. As proof of concept, we developed a plug-in for Eclipse's Java development environment. Developers will be alerted to potential secure programming concerns, such as input validation, data encoding, and access control as well as encouraged to comply with secure coding standards.

**Keywords:** security software development, secure programming, code refactoring, code annotation.

## 1 Introduction

Programmer errors, including security ones, are unavoidable even for well-trained programmers. One major cause of programming mistakes is software developers' heavy cognitive loads dealing with a multitude of issues such as functional requirements, runtime performance, deadlines, and security [3]. Consider Donald Knuth's analysis of 867 software errors he made while writing TEX [4]. It is clear from the log that some of these errors could have made TEX vulnerable to security breaches. The following quotes illustrate Knuth's experience of heavy cognitive burden as a major source of software errors:

> "a forgotten function: Here I did not remember to do everything I had intended when I actually got around to writing a particular part of the code. ... A reinforcement of robustness. Whenever I realized that TEX could loop or crash in the presence of certain erroneous input, I tried to make the code bullet-proof." [4]

The primary motivation for our work is provide *interactive* support, through the IDE, to reduce security errors introduced due to memory lapses and/or lack

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 248–255, 2011.

of focus on the part of developers during program construction. Most software security tools, such as widely used static code analyzers, focus on catching security errors after the program is written. These tools work in a similar way as early compilers: developers must first run the analyzer, obtain and analyze results, diagnose problems, and fix the code if necessary.

Our approach is based on the following considerations. First, we apply the design principle of recognizing instead of recalling [6]. Developers are provided with appropriate visual alerts on secure programming issues and offered assistance to practice secure programming. There are no additional steps a developer must remember. The tool acts as a helpful assistant / advisor and does not hinder developers' creativity and productivity by dictating a rigid approach to secure programming. Second, it is easiest and most cost effective to write secure code and document security implementations during program construction. Efforts to catch security errors downstream will be much more costly. Third, we want to support best enterprise secure software development practices [5]. The IDE is an ideal place to enable sharing of security knowledge and standards across all developers. Through logs on how security considerations have been addressed, developers can provide the necessary information needed for more effective code review and auditing. Finally, it serves as a complementary approach that covers cases that cannot be covered by formal training and education [2] by reinforcing security as a priority throughout the development process.

We offer two key mechanisms to provide interactive support for secure coding: *interactive code refactoring* and *interactive code annotation*. We discuss a proof of concept implementation, ASIDE (Assured Software IDE) which is an Eclipse plug-in for Java, and report our preliminary evaluation efforts. We envision ASIDE being used in an organization that has a software security group (SSG) which is responsible for ensuring software security, as identified by best industry practice [5]. The size and structure of SSG vary. For example in a small organization, the SSG may consist of a few senior developers on a part time basis. The SSG will be responsible for configuring ASIDE to support organizational application-specific secure programming standards. Developers are expected to have received elementary secure coding training and be familiar with concepts such as input validation and output encoding.

## 2 Interactive Code Refactoring

Interactive code refactoring works in a manner similar to a word processor correcting spelling and grammatical errors. ASIDE continuously monitors workspace changes in order to respond to newly created projects as well as modifications to existing projects. We discuss an example concerning input validation to illustrate key concepts.

A developer would be alerted, by a red marker and highlighted text in the edit window, when input validation is needed (see Figure 1). We created a rule-based specification language based on XML to specify sources of untrusted inputs.

Currently two types of rules are supported. *Method (API) invocations:* for example, method `getParameter(String parameter)` in class `HttpServletRequest` introduces user inputs from clients into the system; *Parameter input*: for example, the argument of the Java program entrance method `main(String[] args)`.

With a mouse click, the developer has access to a list of possible validation options (e.g. file path, URL, date, or safe text). Upon selection, appropriate input validation code will be inserted. Figure 1 shows a screen shot of ASIDE facilitating a developer to select an appropriate input validation for an untrusted input. The library of input validation options can be easily reconfigured by the SSG.


**Fig. 1.** The developer interactively chooses the type of input to be validated using a white-list approach

Hafiz et. al. used refactoring techniques for secure coding based on program transformation rules [1], which operate on completed programs and thus best suited for legacy code. One recognized limitation of that approach is the lack of knowledge of specific business logic and context [1]. In contrast, our approach based on IDE provides interactive support for secure programming taking full advantage of developers' contextual knowledge.

Two timing strategies for input validation are possible: (a) validate a variable containing untrusted input when it is used in critical operations such as database accesses, or (b) validate as soon as an untrusted input is read into an application variable. A major disadvantage of the first strategy is that it is not always possible to predict what operations are critical, and thus may fail to validate input when the application context evolves. The advantage of the second approach, which is used by ASIDE, is that it makes sure all untrusted input is validated while it has the programmer's attention. A potential disadvantage is that the type of the input may depend on further application context and is thus unknown when it is first read into the program. In this case, a developer can choose to delay validation. ASIDE will perform data flow analysis to track this variable while the program is being constructed and alert the developer when it is used again. This analysis is limited to the declared method in the prototype.

ASIDE supports validation of input of composite data types, such as a list. Input types may be different for each element of the list. ASIDE uses data flow analysis to track an untrusted composite object, and as soon as a developer retrieves a component that is of an elementary data type, ASIDE alerts him/her to the need to perform input validation in the same manner as described above.

Note ASIDE performs static analysis incrementally on a small chunk of code under construction. They are carried out "under the hood" without requiring developers to understand static analysis or remember to perform it later.

ASIDE supports two types of input validation rules. *Syntactic rules* define acceptable input's syntax structure. They are often represented as regular expressions. Examples include valid names, addresses, URLs, filenames, etc. *Semantic rules* depend on application context. For example, restricting the domain of URLs, files under certain directories, date range, or extensions for uploaded files. Semantic rules can also be used to validate inputs of special data types, such as certain properties of a sparse matrix. While validation rules can be declaratively specified by SSG, a developer has the option to address the identified input validation issue by custom routines. Once the alert has been addressed, the corresponding red marker and text highlights will disappear.

Another important benefit of ASIDE is that it encourages developers to follow secure software development standards. For example, a company may deploy an input validation library and define a trust boundary for the purpose of performing input validation. Once appropriately deployed, ASIDE can collect information as to where in the source code untrusted input is brought into the system and what actions developers took to address input validation.

Output encoding (e.g. HTML/URL encoding) is another area where interactive code refactoring can support secure programming. Rules are used to specify API calls requiring variables to be properly encoded. A developer can use code refactoring functions on demand. For example, a developer may highlight a variable and, with a mouse click, ask ASIDE to generate appropriate code to do URL encoding for its content.

## 3 Interactive Code Annotation

Consider an online banking application with four database tables: *user*, *account*, *account_user*, and *transaction*. Let's assume that as part of the set up by SSG, tables *account* and *transaction* are specified as requiring authentication in such a way that the subject must be authenticated by the key of the *user* table, referred to as an *access control table*.

Figure 2 shows a highlighted line of code in the `doGet()` method of the `AccountsServlet.java` servlet which contains a query to table *account*. The IDE requests the developer to identify the authentication logic in the program, again by a meaningful marker and highlighted text in the editor window (bottom of the screenshot). The annotation process is seamlessly integrated into the IDE without requiring the developer to learn new language syntax. In this case, the developer highlights a test condition `request.getsession().getAttribute (''USER'') == null)` as illustrated in Figure 2 (top of the screenshot).

Several benefits can be derived from this annotation. First, the developer is reminded of the need to perform authentication, and the annotation process may help the developer to verify that authentication logic has been included. The developer has an opportunity to add intended access control logic should

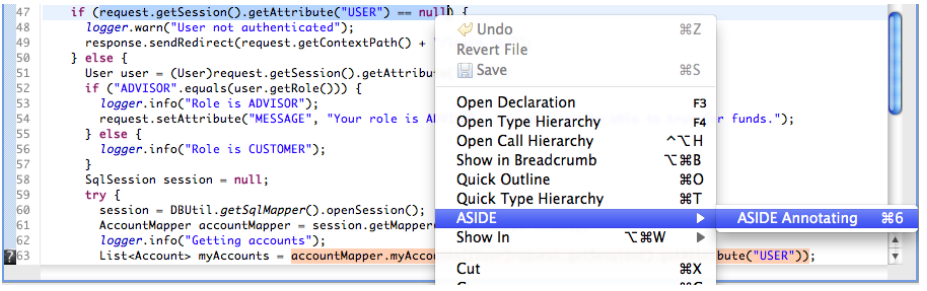



**Fig. 2.** Developer identifies authentication logic (highlighted text on the top) upon request from the IDE (see marker and highlighted text at the bottom)

that be missing. Second, it provides valuable information for code review. Third, heuristics-based static analysis can be performed to provide more in-depth analysis of the authentication logic.

The implementation of interactive code annotation requires a knowledge-based approach, relying on the specific structure of the target technology: for example Java servlet-based applications with JDBC database access for this discussion. The goal is to request developers' annotation on access control logic[1]. Key implementation issues include (a) how to identify the application context to prompt the developer for annotation and (b) what are the forms of acceptable annotations.

Knowing the names of database tables whose access requires authentication, we can identify program statements that access these tables. However, such database access statements may not convey important application context, as they may be part of a utility library used by different web requests. For example, the *account* table may be accessed by the same access routine in multiple web requests (e.g. "customer account balance", and "electronic fund transfer"). Different web requests may require different access control logic (e.g. two-factor authentication for fund transfer).

For Java servlet-based web applications, access control annotation can be requested in the context of a web request: "where is the authentication logic for accessing the account table in the customer account balance request?", and "where is the authorization logic for accessing the account table in the electronic fund transfer request?"

In this example, each web request is implemented as a servlet. Therefore, one can start by tracking `doGet()` and `doPost()` methods within each servlet class (this can be easily extended to include JSP pages). They are referred to hereafter as *entry methods*. Through static analysis techniques one can detect execution paths where an entry method leads to a statement accessing a database table which requires access control. The developer will be requested to provide annotation of access control in the context of the entry method, as illustrated in Figure 2.

[1] Assuming the application is handling access control in application logic, commonly found in web applications, as opposed to configuring access control mechanisms of the underlying database.

Acceptable annotations must satisfy the following: (a) it consists of a set of logical tests (e.g. conditional tests in `if` statements, cases in `switch` statements), and (b) each test must be on at least one execution path, as determined by control flow, from the start of the entry method to the identified table access statement.

Annotations may enable more in-depth static analysis. For example, a broken access control may be detected if there is an execution path in the entry method leading to the database access without any identified access control checks. Furthermore, at least some of the access control tests identified must lead, through static analysis, to the access control tables identified. While these possible analysis heuristics cannot guarantee the correctness of access control logic, they can nevertheless identify some common cases of broken access control.

## 4 Initial Evaluations

### 4.1 Open Source Project Evaluation

We ran ASIDE on two open source Java projects: Apache Roller (a full-featured blog server), and Pebble (a lightweight J2EE blogging software). As a benchmark for secure code review we used default rules in Fortify SCA static analyzer, a tool widely used by industry. To complete this analysis we will, in the near future, perform additional manual audits and compare these with results obtained from ASIDE. Our initial evaluation concentrated only on input validation vulnerabilities, as this is the most mature functionality in ASIDE.

Table 1 summarizes overall results. Default rules in Fortify SCA report an input validation vulnerability whenever an untrusted input is used in a sensitive operation, such as SQL query execution. ASIDE, on the other hand, alerts a developer for input validation whenever untrusted input is read into the system. Therefore, the number of input validation vulnerabilities found by Fortify SCA is larger because the same untrusted value could be used in multiple operations. We found that ASIDE could be used to address all issues identified by Fortify in a manner illustrated in section 2, namely providing real time interactive assistance to developers through code refactoring to address input validation.

**Table 1.** Preliminary analysis of two open source web apps

| Program Name | Executable LoC | Fortify Unvalidated Inputs | ASIDE Untrusted Inputs |
|---|---|---|---|
| Roller 4.0.1 | 65,209 | 160 | 82 |
| Pebble 2.4 | 58,402 | 202 | 184 |

### 4.2 Model-Theoretic Analysis

The objective of this analysis is to understand why a tool like ASIDE might be effective at preventing software security bugs by understanding how such errors may be committed. Ko and Myers have done a comprehensive survey

of previous research on causes of programmer errors [3]. According to them, programmer errors can be traced to three types of cognitive breakdowns: *skill-based breakdown*, *rule-based breakdown*, and *knowledge-based breakdown*.

Skill-based activities are routine, mechanical activities carried out by developers, such as editing program text [3]. We considered common patterns of skill-based breakdown[3, 4, 7] in a security context. For example, suppose a programmer copies a block of code to reuse in his program. He realizes that input validations need to be modified to suit the new application context. This could be a routine task by changing input validation instances to a different API call. However his task (changing API calls in the edit window) was interrupted, say by an instant message from a colleague. When his attention is brought back to the task, an instance of the old input validation was missed, thus causing a software vulnerability. Attention shifts are the principle cause of skill-based breakdowns [3]. ASIDE could be effective to mitigate such errors by alerting programmers to important security issues, refocusing the developer's attention on security concerns.

Ko and Myers use the term *rule* to refer to a programmer's learned program plan / pattern [3]. We believe ASIDE can also be effective against common patterns of rule-based breakdowns in security contexts. First example relates to the inexperience of a programmer [3]. Suppose a programmer, who has been trained on secure programming practices, invokes an unfamiliar API which brings untrusted input into the system. She may not be aware (i.e. she's missing a "rule") that input validation is needed, leading to a software vulnerability. The second example relates to information overload, which occurs when too many rules are triggered and one of them may be missed [3, 4]. One can envision a situation where the rule requiring access control is not applied due to programmers' information overload. In both cases a tool like ASIDE could be effective in preventing such errors by reminding programmers to apply security related rules.

The third example relates to favoring a previously successful rule without realizing the change of context [3]. For example a block of code is copied and reused. The code being copied may make certain checks on file integrity. However, given the new application context, more types of checks are needed. A developer may not be sufficiently aware of this context switch and mistakenly believe existing integrity checks in the code are sufficient. Again a tool like ASIDE may be helpful to mitigate this instance by alerting programmers of important security considerations and giving them an opportunity to discover that a different rule might be needed. This might be accomplished by requesting an annotation on file integrity check logic.

Knowledge-based activities typically are at a conceptual level, such as requirements specification and algorithm design [3]. For example, it is often impossible to perform an exhaustive search of the problem space. Using black-list input validation instead of white-list input validation is an example of a knowledge-based breakdown leading to security vulnerabilities. A black list is much easier to construct, based on human cognitive heuristics of selectivity, biased reviewing and availability [3]. Writing proper white-list validation requires significant effort,

even for common input types such as a person's last name, address, URL, and filepath, especially when taking internationalization issues into consideration. ASIDE helps to address this issue by making it easier to use white-list input validation, choosing from a predefined list of options.

## 5 Discussions

A main contribution of our work is to design interactive programming support to help developers reduce security errors during program construction. Many types of common software security errors could be addressed: code refactoring can be used to address injection attacks by encouraging input validation and output filtering; code annotation can be used to remind developers of access control, buffering handling, and resource management. Initial evaluations suggest that our approach could be effective to help developers cope with their heavy cognitive burden and help developers to reduce security errors due to memory lapses, lack of attention/focus, and knowledge gaps.

Our approach also offers a tool platform for additional support for secure software development lifecycle in areas of (a) encouraging standard compliance, (b) information collection for secure coding metrics, and (c) capturing developer rational for code review or in depth program analysis. A substantive question we need to address about this approach is the effectiveness of the two techniques in practice. Going forward, we are continuing the development of ASIDE and would like to conduct thorough and comprehensive user studies to examine and evaluate the approach in light of effectiveness and generality.

**Acknowledgement.** This work is supported in part by a grant from the National Science Foundation 0830624, and a research/educational license from Fortify Inc.

## References

[1] Hafiz, M., Adamczyk, P., Johnson, R.: Systematically Eradicating Data Injection Attacks Using Security-Oriented Program Transformations. In: Massacci, F., Redwine Jr., S.T., Zannone, N. (eds.) ESSoS 2009. LNCS, vol. 5429, pp. 75–90. Springer, Heidelberg (2009)

[2] Evans, K., Reeder, F.: A Human Captial Crisis in Cybersecurity. Center for Strategic and International Studies (2010)

[3] Ko, A., Myers, B.: A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages and Computing 16, 41–84 (2005)

[4] Knuth, D.: The errors of TeX–Software: Practice and Experience, vol. 19(7), pp. 607–685 (1989)

[5] McGraw, G., Chess, B., Migues, S.: Building Security in Maturity Model (2009), `http://www.www.bsimm2.com`

[6] Preece, J., Sharp, H., Rogers, Y.: Interaction design: Beyond human-computer interaction. Wiley, Indianapolis (2007)

[7] Reason, J.: Human Error. Cambridge University Press, Cambridge (1990)

# Idea: A Reference Platform for Systematic Information Security Management Tool Support

Ingo Müller[1], Jun Han[1], Jean-Guy Schneider[1], and Steven Versteeg[2]

[1] Swinburne University of Technology, Hawthorn, Victoria 3122, Australia
{imueller,jhan,jschneider}@swin.edu.au
[2] CA Labs, CA Technologies (Pacific), Melbourne, Victoria 3004, Australia
steve.versteeg@ca.com

**Abstract.** The ISO 27001 standard specifies an information security management system (ISMS) as a means to implement security best practices for IT systems. Organisations that implement an ISMS typically experience various challenges such as enforcing a common vocabulary, limiting human errors and integrating existing management tools and security mechanisms. However, ISO 27001 does not provide guidance on these issues because tool support is beyond its scope, leaving organisations to start "from scratch" with manual and usually paper document-driven approaches. We propose a novel reference platform for security management that provides the foundation for systematic and automated ISMS tool support. Our platform consists of a unified information model, an enterprise-level repository and an extensible application and integration platform that aid practitioners in tackling the aforementioned challenges. This paper motivates and outlines the key elements of our approach and presents a first proof-of-concept prototype implementation.

## 1 Introduction

The ISO 27001 standard [1] specifies a well-defined process for implementing and operating an enterprise-level information security management system (ISMS). This ISMS establishes security best practices for an IT system based on (i) systematic risk assessment, (ii) coordinated design and implementation of controls and (iii) ongoing assessment of the effectiveness of these controls. The benefits of an ISMS are increased transparency of security management decisions and their alignment with their system-level implementation. Hence organisations gain better understanding of the security posture of their IT systems enabling them to better protect their information assets and to demonstrate due diligence.

However, as we have learnt from practitioners, reaping these benefits is challenging because of a lack of tool support. ISO 27001 and the related ISO 27003 standard [2] specify a manual document-driven process with tool support out of their scope. Moreover, existing tools such as for governance, risk and compliance (GRC) or risk management focus on specific aspects, to the best of our knowledge, but do not cover the entirety of the implementation and operation of an ISMS. Due to this lack of support, organisations typically start "from scratch" with manually managed paper documents and experience challenges, such as:

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 256–263, 2011.

1. Difficulties to establish and enforce a common vocabulary across the departments and different hierarchy levels of an organisation;
2. Difficulties to limit human impact (errors, inconsistencies and omissions) on security management activities;
3. Lack of standard-compliant support for the integration of "local" security management tools and enforcement mechanisms with the "global" ISMS.

Consequently, an ISMS may not enforce security best practices effectively and additional time, resources and investments may be required to establish ISO 27001 compliance. The goal of our work is to address the above challenges and to support organisations to obtain ISO 27001 certification. We propose a novel security management platform that provides the foundation for systematic and automated ISMS tool support. The platform comprises three key elements: a unified information model, an enterprise-level information repository and an extensible application and integration platform. These elements address the three challenges above. Our platform is defined as a generic and system-independent reference platform that can be applied by any organisation to augment their ISMS with automated tool support, respectively.

In this paper, we motivate and outline the elements of our reference platform in Section 2 and present a proof-of-concept-prototype in Section 3. The paper concludes with a summary and an outlook to future work in Section 4.

## 2 Security Management Reference Platform

Our approach is fully aligned with ISO 27001. Thus, this section motivates and presents the key elements of our approach driven by ISO 27001 features and highlights how these elements help to alleviate the aforementioned challenges.

### 2.1 Unified Information Model

ISO 27001 requires the creation of a common vocabulary that supports the unambiguous definition of security policies and procedures across all units of an organisation. The objective is to unify and align the views of all involved stakeholders. For example, the CIO views access control from the management perspective, considering the number of incidents and their impact on the business, whereas a system administrator is concerned with technical aspects such as password strength and monitoring audit trail records.

The challenge of a manual paper-based approach is twofold. Firstly, it is difficult to establish and synchronise a common vocabulary across documents due to varying authorship and document-specific glossaries. Secondly, it is difficult to enforce that stakeholders adhere to the vocabulary when creating or modifying documents due to lacking means to verify inputs immediately as they are written.

We propose a unified information model to tackle this challenge. An information model specifies concepts, their relationships and operations to define the semantics of a problem domain. We have developed an information model for the information security management (ISM) domain based on ISO 27001

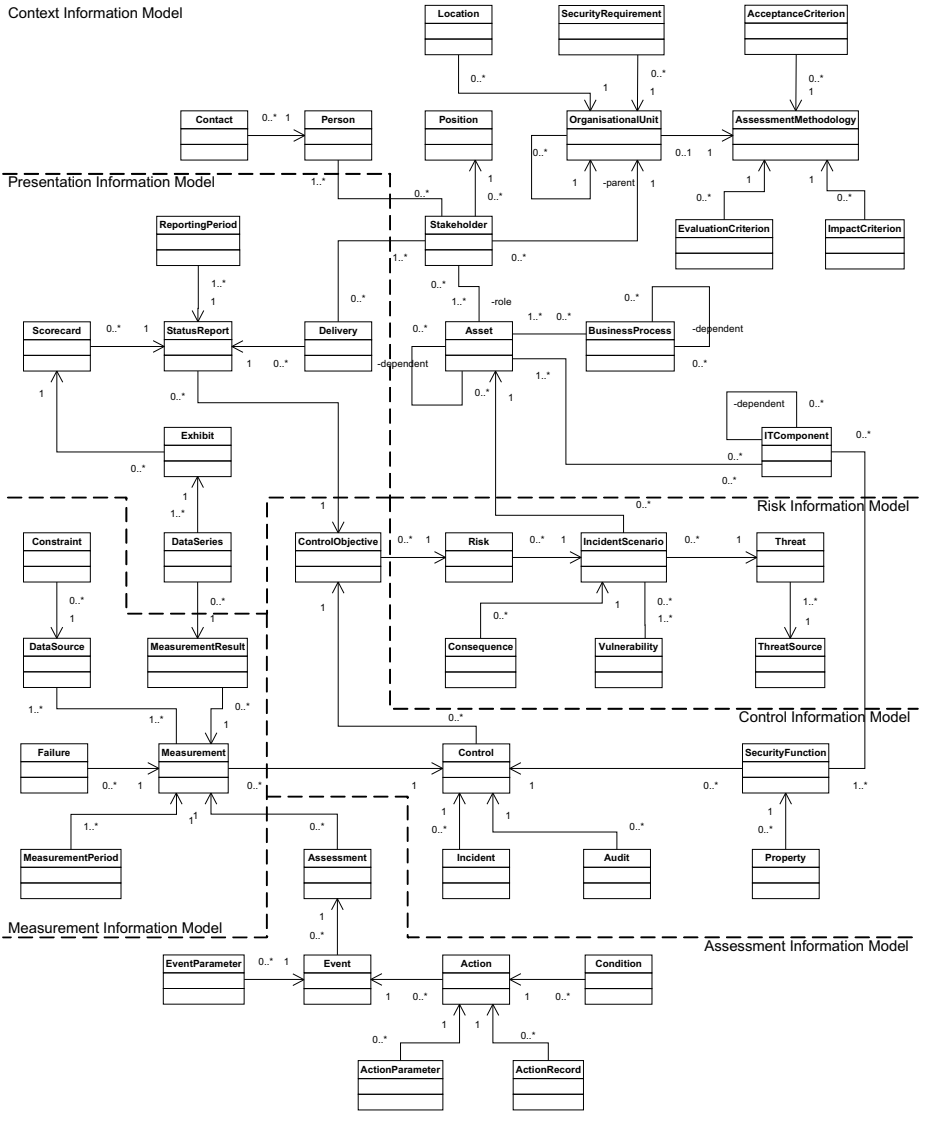



**Fig. 1.** UML class diagram of the information model main concepts

that supports information exchange and management activities across different stakeholders, departments and management applications. Our information model is depicted in Figure 1. It is structured into sub-models, each containing concepts that pertain to a concrete ISM activity. Every concept comprises informally (in natural language) and formally (in formal languages) specified attributes. In this way, management-level requirements and decisions are aligned with system-level configurations, system properties and enforcement mechanisms. The

information model is specified using object-oriented concepts and UML formalisms to enable machine processing of concrete representations and their customisation by taking advantage of inheritance and polymorphism features. The details of the information model are specified in [3].

Our information model mitigates the challenge of managing a common vocabulary. Firstly, it enables organisations to systematically specify a unified vocabulary for all aspects related to their ISMS in a fine-grained and electronic fashion based on *ISM concepts* rather than documents. Secondly, tools can be developed that enforce the correct use of the vocabulary during the creation or modification of concepts based on the information model. For example, Figure 2 depicts the input mask of a graphical user interface (GUI) for the manipulation of information about metrics (*e.g.* a metric for measuring the effectiveness of an access control mechanism). The GUI indicates mandatory information and enforces vocabulary-compliant inputs with dedicated input elements (*e.g.* multiple-choice combo boxes) and input filters (*e.g.* to verify the temporal correctness of *Start Date* and *End Date*). Figure 2 also demonstrates how the information model aligns management-level and system-level views of a concept (*cf. Outline* text field with the prefix algebraic expression of the *Function* text field).

### 2.2 Enterprise-Level Repository

ISO 27001 specifies a document-driven process to ISM. These documents are created by various stakeholders throughout the organisation and modified over time with the objective to reflect the actual state of an ISMS implementation at any time. For example, an organisation documents all access control points including their potentially department-specific configurations and constraints.

The challenge of a manual paper-based approach is twofold. Firstly, it is difficult to keep all documents up to date such that they reflect the actual state of the ISMS. Secondly, it is difficult to ensure that the documents, individually and across different documents, remain coherent, correct and as complete as possible. In other words, it is difficult to limit the human impact on security management activities, such as errors, inconsistencies and omissions.

We propose an enterprise-level information repository to address this challenge. The repository captures, consolidates and correlates security-related information across different stakeholders, departments and management applications. It is structured based on our information model and, thus, ensures correctness and durability of the instances of information model concepts as well as the referential integrity of their associations. The repository also implements capabilities for managing security-related information including unique identification, access control, version control and meta-information management.

The repository enables keeping organisation-wide security-related information coherent, correct and complete. Firstly, all information is consolidated and correlated centrally. In this way, redundancies are avoided and ISO 27001-relevant documents can be generated from the same data. Secondly, the data dictionary of the repository can be used to develop tools to raise alerts in case inputs are incorrect, inconsistent or incomplete at the time of input. For example, see

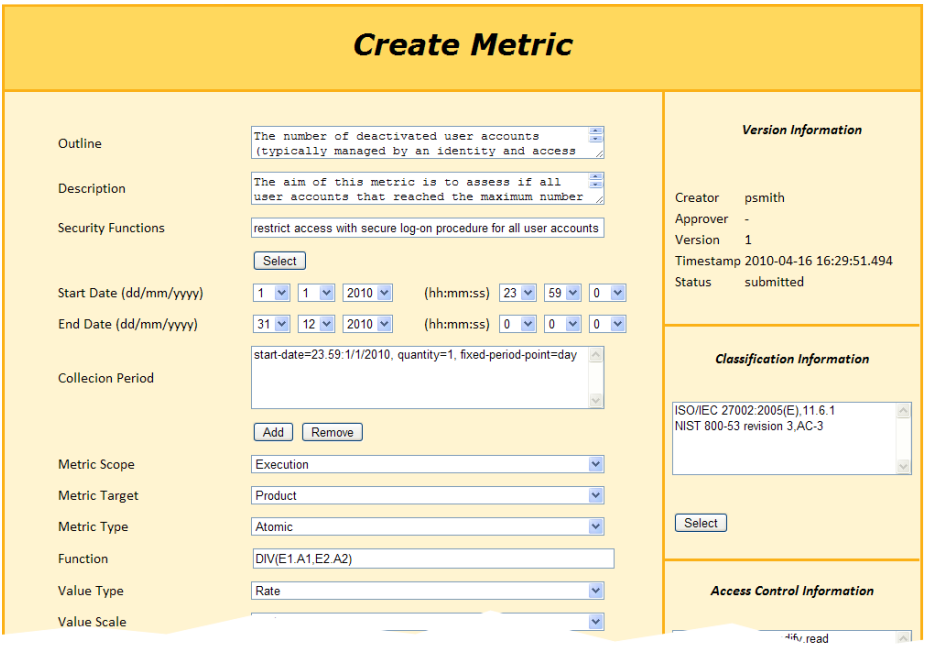



**Fig. 2.** Screenshot of the input mask for metric definitions

Figure 2 for a GUI that enforces the input of all mandatory attributes of the metric concept, thus preventing omissions. It contains input filters (*e.g.* to verify the correctness of the *Collection Period*) and dedicated input elements (*e.g.* multiple-choice combo boxes) to avoid errors and reduce inconsistencies. This GUI can be auto-generated based on the repository's data dictionary.

Additional input checkers are feasible that inspect user inputs to verify syntactic and semantic correctness, *e.g.* for the *Function* text field in Figure 2. Moreover, tools for change management and impact analysis can support the identification of concepts in the repository that are affected by a change by tracing concepts via their associations. Finally, instead of a human reading through a set of documents, the repository supports automated queries for identifying rogue data sets that contain incorrect, incomplete or outdated information.

### 2.3 Extensible Application and Integration Platform

ISO 27001 does not provide explicit guidance for infrastructure and tool support for an ISMS. Hence, organisations need to develop required IT system support from scratch, for example for integrating log files and audit trails of access control points for the sake of measuring their effectiveness on an ongoing basis.

The challenge of a manual document-driven approach is that it is difficult to utilise existing security management tools and security enforcement mechanisms in a systematic and automated fashion such that security best practices are implemented as effectively as possible.

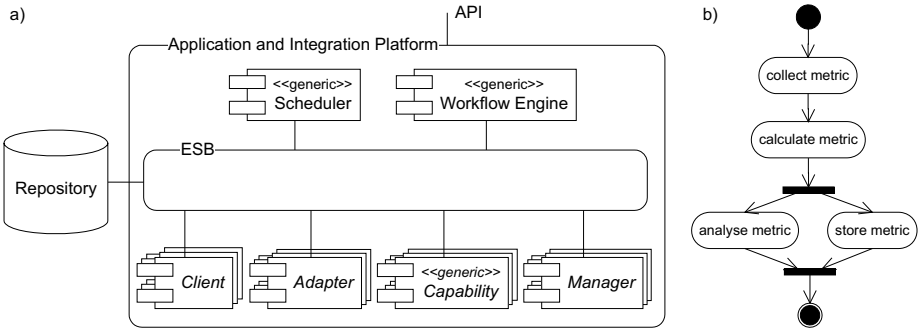



**Fig. 3.** a) conceptual platform architecture and b) metric management workflow

We propose an extensible application and integration platform that tackles this challenge following a generic Service-oriented management framework as outlined in Müller et al. [4]. The conceptual architecture of the platform is presented in Figure 3 a. It exhibits the characteristics of a *hub-spoke architecture*. The centrepiece, or hub, is an Enterprise Service Bus (ESB) that enables uniform message-oriented integration. Hence, the ESB provides the ideal foundation for integrating security management tools, applications, mechanisms and enforcement points. It facilitates interoperability and seamless exchange of data, events between them and supports overarching management activities.

The spokes of the architecture are a set of extension components as outlined below and two default components: Scheduler and Workflow Engine. The *Scheduler* component facilitates the automated scheduling of management workflows on the ESB. The *Workflow Engine* enacts and executes *management workflows*. A management workflow composes the functionality of sets of ESB components. Figure 3 b presents the representation of a generic management workflow for the automated collection, calculation, analysis and storage of metric data.

The platform can be extended with components that implement the following roles. The *Client role* describes the ability to integrate interfaces to humans and systems, *e.g.* a dashboard for administering the integration platform or inspecting the contents of the repository. The *Adapter role* describes the ability to integrate remote data sources and data sinks, management tools and applications. The enterprise-level information repository is integrated with the ESB using a component implementing this role. The *Capability role* describes the ability to integrate generic management functionality locally in the form of plug-ins. Components of this role typically manipulate information in the repository and interact with Adapter and Manager components. The Scheduler and Workflow Engine components embody the Capability role. The *Manager role* describes the ability to integrate remote management functionality that is situated in the managed IT system, as for example policy enforcement points or system monitors. Furthermore, the platform offers an API for programming, integrating and managing components and deploying management workflows on the ESB.

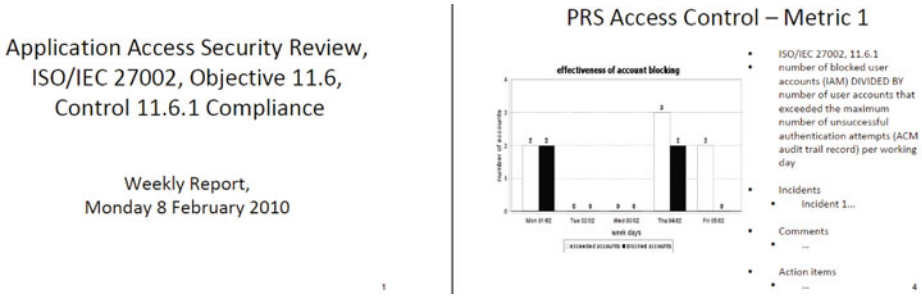



**Fig. 4.** Auto-generated status report with scorecard

The platform enables systematic and automated tool support. It facilitates the integration and interoperation of security management tools and enforcement mechanisms. It fosters information sharing and consistent data management across ISM activities. For example, Figure 4 illustrates a status report that was created based on (i) repository data, (ii) the execution of the management workflow in Figure 3 b and (iii) another generic workflow that automatically generates scorecards and a status report about the effectiveness of a specific access control point, and disseminates this report via e-mail to all involved stakeholders.

## 3 Proof-of-Concept Prototype

We have devised a basic application scenario in order to implemented a prototype of the proposed security management platform. Consider the case where a new law requires hospitals to provide patients with online access to their medical records. Due to the high sensitivity of medical records, various regulations mandate access restrictions to protect the privacy of patients. Thus hospitals must ensure that this online access is secured also and managed with their ISMS.

Figure 5 depicts the design of a Web application that allows patients to interact with the *Patient Record Service* (PRS) in order to access their medical records stored in the *Patient Record Database* (PRDB). Figure 5 also shows the design of a concrete implementation of our reference platform including all components required to implement the conceptual architecture and workflow shown in Figure 3. Moreover, the current prototype implementation contains components that implement another workflow, as outlined in the previous section, that creates status reports and disseminates them to involved stakeholders. The functionality of our approach is illustrated with a basic use case scenario. Figure 2 presents an input screen that allows security experts and other stakeholders to define a metric and setup the automated collection and reporting of metric data. In this use case scenario, a metric is defined for measuring the effectiveness of the ACM component that protects online access to the PRS (*cf.* Figure 5). Figure 4 depicts a screenshot of a corresponding auto-generated status report.

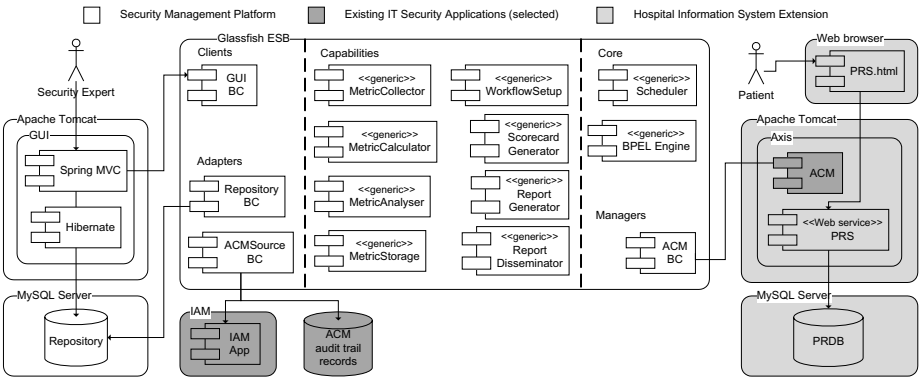



**Fig. 5.** Modified UML deployment diagram of the prototype implementation

## 4 Summary and Outlook

This paper proposed a reference platform that forms the foundation for systematic and automated tool support for the implementation and operation of an ISMS in alignment with the ISO 27001 standard. Our goal is to aid practitioners to overcome challenges of present typically manual and paper document-driven approaches. A proof-of-concept prototype has been developed to demonstrate our approach. Our next steps include the implementation of a realistic industry scenario to gain further insights, refine and validate our approach in collaboration with practitioners. Moreover, we will explore further tool support, *e.g.* how model-driven engineering methods can be utilised to automate the development, configuration and deployment of security mechanisms based on our approach.

**Acknowledgements.** This work is supported by the Australian Research Council and CA Technologies (CA Labs). We also gratefully acknowledge Abhijat Sinha and Swapnil Raverkar who contributed to the implementation of the prototype.

## References

1. ISO/IEC: Information technology — Security techniques — Information security management systems — Requirements. 1st edn. (2005); ISO/IEC 27001:2005(E), International Standard
2. ISO/IEC: Information technology — Security techniques — Information security management system implementation guidance. 1st edn. (2010); ISO/IEC 27003:2010(E), International Standard
3. Müller, I.: The Information Security Management Information Model (ISM-IM). Technical Report. Swinburne University of Technology (2010), `http://www.swinburne.edu.au/ict/research/cs3/rsr/pubs/ISM-IM.pdf`
4. Müller, I., Han, J., Schneider, J.G., Versteeg, S.: A Conceptual Framework for Unified and Comprehensive SOA Management. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 28–40. Springer, Heidelberg (2009)

# Idea: Simulation Based Security Requirement Verification for Transaction Level Models

Johannes Loinig[1], Christian Steger[1],
Reinhold Weiss[1], and Ernst Haselsteiner[2]

[1] Institute for Technical Informatics, Graz University of Technology, Graz, Austria
{johannes.loinig,steger,rweiss}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
ernst.haselsteiner@nxp.com

**Abstract.** Verification of security requirements in embedded systems is a crucial task - especially in very dynamic design processes like a hardware/software codesign flow. In such a case the system's modules and components are continuously modified and refined until all constraints are met and the system design is in a stable state. A transaction level model can be used for such a design space exploration in this phase. It is essential that security requirements are considered from the very first beginning. In this work we[1] demonstrate a novel approach how to use meta-information in transaction level models to verify the consistent application of security requirements in embedded systems.

## 1 Introduction

A lot of modern embedded systems need to provide security functionality. Faults in design and implementation of a system can cause serious security issues. Thus, careful security verification is needed. This is a considerable cost factor. External Common Criteria security evaluation (described later) can cost $100k and more and usually takes months. Finally discovered vulnerabilities in such a late development phase cause serious project delay and cost.

Security has to be considered from the beginning of a development process and in all abstraction levels [6,10]. To support this we propose a methodology to use meta-information in transaction level models (TLMs) for early and continuous security verification in a hardware/software codesign flow. TLMs are abstract functional models of the system. Iterative refinement is used for design space exploration until all system constraints are met. In each iteration our methodology supports security verification appropriate to the model's abstraction level.

The contribution of this work is a novel design and development approach that allows continuous security verification. System designers and developers will gain a better security understanding of the system which reduces the risk for a costly failed Common Criteria security evaluation.

[1] This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

Ú. Erlingsson, R. Wieringa, and N. Zannone (Eds.): ESSoS 2011, LNCS 6542, pp. 264–271, 2011.

## 2 Related Work

Our proposed methodology is based on the Common Criteria (CC) security verification process. It applies the basic CC practices to verify TLMs. Thus, to explain our approach, we first summarize the needed CC and TLM basics. After that we shortly list advantages and disadvantages of formal verification methodologies to motivate our contrary simulation based approach.

### 2.1 The Common Criteria Process

The Common Criteria [3] is *the* de-facto standard for security evaluations of IT products. The entire process is too extensive to be described here. However, some basics should be clarified to understand our proposed approach.

CC is a rather documentation centric approach. Threats against the IT product are analyzed. Security Objectives are defined to counter these threats. Security Functions provide the functionality for them. Notice that Security Functions are rather a concept and not a concrete implementation. Each Security Function is composed of Security Mechanisms implemented in hardware or in software. Security Functional Requirements (SFRs) describe their functional requirements. These SFRs must be provided by the system to achieve the security objectives. The CC Security Target (ST) describes the relations of security threats, objectives, mechanisms, and functions. It is a design document for the security architecture of one certain IT product. In combination with design documentation an external CC evaluator uses the ST to judge if the security architecture is able to counter threats. This is a manual and extensive process.

The authors of [8] mention a lack of a clear relationship between the CC process and a system development approach. However, security engineering should be integrated in the system development process. [8] describes a CC conform UML based approach for security requirement engineering in a software engineering process. In comparison to that, our approach covers system development (hardware *and* software) and is not restricted to UML. Instead, we support implicit security modeling in a TLM of the system.

### 2.2 Transaction Level Modeling

Transaction level modeling allows development and analysis of system models [2]. The main concept is the abstraction and separation of the computational part of the system and the communication part of the system. In our approach we extend this by the security part of the system. Meta-information in TLMs allows an early estimation of e.g., the system's performance and power/energy consumption [11]. At the time of writing we were not able to find related work considering security requirements in TLMs.

### 2.3 Formal System Verification

There is no doubt that a formal verification (FV) would be preferable in comparison to our proposed simulation based verification. However, we think that

today one main aspect still acts against FV: FV is very costly [6] if applied on systems with realistic complexity.

As a consequence, FV can be applied on selected parts of a system only or it can be applied on very abstract meta-models of the system. This has been shown for software [4], for hardware [7], and on system level [1]. FV of selected parts only is unacceptable as system security can not be achieved by single separate modules in a system [10]. A meta-model based approach can be difficult to apply in a dynamic design process. The developer would need to work on both, the functional model and the meta-model.

Meta-models are typically written in special modeling languages and can be created manually as a first design step [4,1] or translated from other models by a verification tool [5]. Manual adaptation of the meta-model would be costly and error-prone. FV of a translated model means, that in a strict sense, not the system's model is verified but a model that was generated by the tool. The quality of the verification strongly depends on the translation tool and the capabilities of the modeling language of the meta-model.

The authors of [9] created formal templates for CC SFRs. These templates can be used for FV of the system's security specification. Again, not the system model itself is verified but its specification. In contrast to that our proposed simulation based methodology allows verification of the relationships of Security Mechanisms and SFRs against the functional system model under development.

## 3 Simulation Based Security Requirement Verification

In comparison to the work described in Section 2 our contribution is (1) the consideration and verification of security requirements in early design phases, (2) a verification on the basis on a TLM under development instead of a meta-model, and (3) a fast simulation based verification applicable for iterative design processes instead of a complete but extensive formal verification approach.

### 3.1 Iterative TLM Verification

Figure 1 shows the basic concept of simulation based verification in an iterative refinement process. A pure functional level is the basis for a TLM. In multiple iterations the TLM's modules become refined. During simulation of the TLM, when the test cases are applied, an automated verification generates a functional report and a security report. The reports are the basis for further design decisions. Another refinement cycle is necessary if not all constraints are met.

Different actions can be performed during TLM refinement: modules can be split in several smaller modules, combined to bigger modules, and mapped to software or hardware components of the system. All these actions require consideration of SFRs if the modules are related to Security Functions. To do so, the developer annotates the Security Mechanisms with meta-information about related SFRs. These annotations are evaluated during simulation and verification. If module modifications cause security violations this is reported during the simulation/verification and the developer can react immediately.

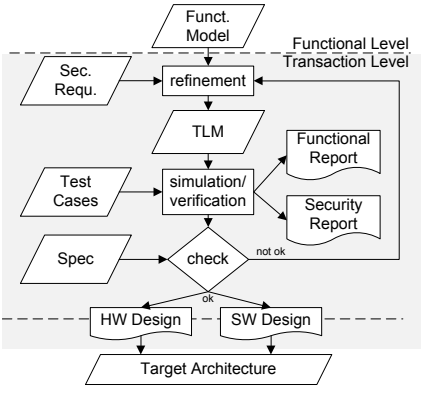



**Fig. 1.** Simulation based verification

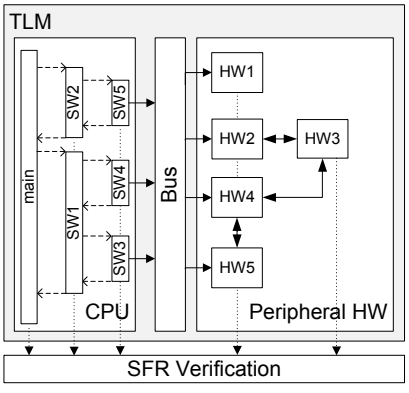



**Fig. 2.** TLM verification

### 3.2 HW/SW Verification Approaches

Figure 2 shows a very simple example of a TLM consisting of software modules (embedded in a CPU module) and peripheral hardware modules. Two things should be noted in this figure: the software is shown as a sequence of function calls whereas the hardware is modeled as a set of concurrent hardware blocks. Having sequential software and concurrent hardware is typical for TLMs. Thus, a verification approach has to be able to handle both concepts.

Requirement verification on sequential software is rather straight forward. A call graph clearly shows the dependencies of the security requirements of the software functions. This is not the case if hardware is modeled. Concurrent hardware processes do not provide a call graph as hardware functions usually never terminate. Additionally, hardware exceptions can interrupt the software at any time. This behavior is very difficult to describe in formal models. This is an essential reason why we chose a simulation based approach for our methodology.

Instead of a call graph, a data flow driven approach can be used for hardware verification. Data is passed from one process to another one. This can be done asynchronously - the data producer often has no influence if the receiver consumes the sent data or decides to ignore it. This might depend on the internal state of the receiving hardware module.

In a data flow driven approach data is annotated with SFRs instead of functions or modules. If this data is consumed, the annotated SFRs have to be evaluated by the simulation environment to verify the security capabilities of the module. As shown in the figure, software and hardware interact naturally (sketched as a bus connecting the CPU with the peripheral hardware blocks). As a consequence, SFRs have to be mapped from the call graph to the data flow graph and vice versa.

In our simulation based approach function calls and data generation respectively data consumption is reported to a security verification module. The reporting includes the annotated SFRs of the acting software or hardware modules.

The verification module evaluates the relationship of the reported SFRs and is able to report occurring security violations.

### 3.3 Verification Rules

Yet we have not discussed the rules our proposed verification module should apply to the reported SFRs. Different sets of rules according to the abstraction level of the TLM are imaginable.

A straight forward approach is a *requires*/*implements* scenario. Modules that require SFRs are annotated with the SFR's identifier (e.g., a unique number) and a *requires*-flag. Modules that provide the functionality described by the SFR are annotated by the SFR's identifier and an *implements*-flag. The verification module checks if every call graph's node with a *requires*-flag leads to nodes with the according *implements*-flag; respectively if data that has a *requires*-flag is handled in modules that provide the *implements*-flag.

A more sophisticated approach is to annotate software and hardware modules that represent a Security Function and also modules implementing SFRs. The ST can be used to extract the relationship of (1) SFRs required by the Security Functions and (2) SFRs that have dependencies on other SFRs to create verification rules. The transaction level model consists of modules assigned to the software and the hardware domain. It represents the functional behavior and the security behavior of the embedded system. Certain modules might be implemented on different abstraction levels. Communication interfaces, for example, can be modeled on very high abstraction levels as they are usually not security relevant. The software model reports information to the verification module to generate call graphs for Security Functions. Similarly, the verification module generates data flow graphs for data in the hardware model. Call graphs and data flow graphs are verified against the rules extracted from the ST. This is shown in Figure 3.

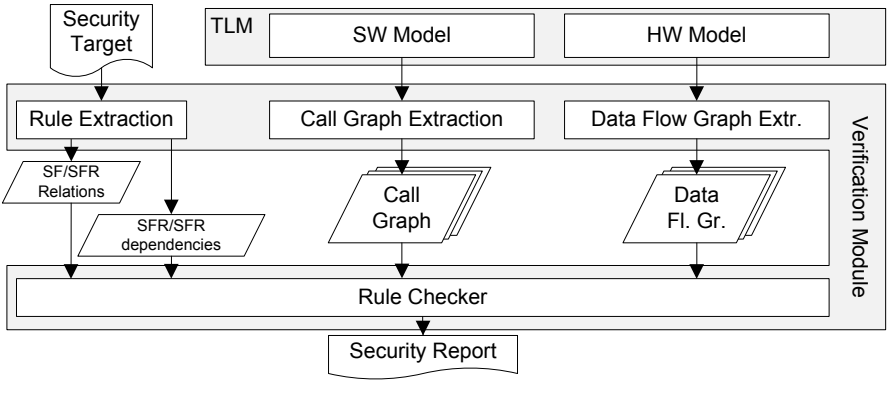



**Fig. 3.** Verification of rules extracted from a Security Target

## 4 Proof of Concept Implementation

We implemented a proof of concept verification library for SystemC[2] TLMs. The library provides (1) macros for annotating functions, data, and SystemC processes with SFRs, and (2) the verification module. The macros cause function calls of the static verification module instance. These function-calls cause the verification module to compute the call graphs and data flow graphs and to store the annotated SFRs. The rule set for verification is static and provides checks for *requires*, *implements*, and *supports* flags for SFRs. Modules that *require* SFRs have to utilize modules that *implement* according SFRs. *Supporting* SFRs can be used if interconnecting modules are needed.

So far, we have not evaluated our approach on a use case with realistic complexity. Instead, we first evaluated the general applicability by verification of different smart card STs from different vendors: STMicrosystems ST23YR80A, NXP P5Cx081V1A, Samsung S3CC91A, Infineon SLE66CX680PE, and Fujitsu MB94RS403. These STs are public[3].

In addition we implemented a very small use case example based on some few hardware modules: a CPU, a memory, a DES crypto co-processor, and a CRC co-processor. The software model reads DES key data from the memory and sends it to the DES co-processor. We chose the FDP_SDI.1.1 SFR from the Common Criteria standard [3] - it basically defines, that the DES key has to be integrity protected. The `setDESKey()` function was annotated to *require* FDP_SDI.1.1. Two different implementations to fulfill the security requirement were evaluated: (1) a software implementation calling `checkIntegrity()` within `setDESKey()` and (2) a hardware implementation connecting the DES co-processor with the CRC co-processor.

## 5 Results and Discussion

Table 1 shows the summarized evaluation results of the STs: the total number of defined Security Functions, the number of selected SFRs, the number of applied SFRs, and the SFRs we think that can be checked automatically by our proposed approach. Notice, that each SFR can be applied on several Security Functions. Thus, the number of applied SFRs can be higher than the number of SFRs. In a strict sense, this number should be even higher than the numbers depicted in Table 1 (third row) because each Security Function consists of several mechanisms. However, the number of Security Mechanisms is depending on the concrete implementation and not given in public STs. Therefore, we took the number of Security Functions as an estimator for our evaluation (we assume that each of them consists of one mechanism in minimum).

26 to 60 SFRs were applied in the selected smart card microprocessors. This should give a feeling about the verification complexity: 26 to 60 applied SFRs means that 26 to 60 times a module has to be verified if it provides the

[2] `www.systemc.org`

[3] `www.commoncriteriaportal.org/products/`

**Table 1.** Security Target evaluation results

| | STMicrosystems | NXP | Samsung | infineon | Fujitsu |
|---|---|---|---|---|---|
| Number of Security Functions | 10 | 9 | 5 | 9 | 10 |
| Number of SFRs | 15 | 16 | 18 | 21 | 19 |
| Number of applied SFRs | 40 | 60 | 26 | 35 | 26 |
| Potentially automated check | 90% | 85% | 73% | 77% | 73% |

appropriate SFR. Notice, that the selected STs describe smart card hardware only. If the software has to be verified as well even more Security Functions and SFRs will emerge and the verification effort will be even higher.

We do not expect that all applied SFRs can be verified with our proposed approach. Thus, we checked which used SFRs can be verified by our method. To do so, we evaluated the meaning of used SFRs in the selected STs and checked if we could define rules for our approach that are able to verify the SFRs automatically. Because of the limited space in this publication we can not explain this process in details here. The results are shown in Table 1 in the last row.

We think that 73% to 90% of the used SFRs could be checked with our proposed approach. Accordingly, the verification effort could be reduced by approximately 80%. Of course this is a very optimistic estimation as verification does not only mean to check if a module provides the right SFRs. However, we think that this clearly shows the potential of an automated verification approach.

From our very small case study we can summarize the following results. Without going into the implementation details we can note that the usage of our annotations is very intuitive and fits very well into the iterative design flow of an hw/sw codesign flow. However, we also have to mention that a puristic *requires*/ *implements* rule-set seems not to be sufficient. At least an additional *ignores*-flag was needed to avoid miss-leading incorrect security violations. However, we think that such an *ignores*-flag can come with a high risk of erroneous miss-usage.

In addition, we have to note that the implemented solutions are not identical from a security point of view. If the integrity check is performed in software, sending the sensitive key material to the DES block is unsecured. On the one hand we think that this could be automatically reflected in our verification result (e.g., as the 'distance' of a module that *requires* the SFR to the module that *implements* the SFR). This could help the developer during the design space exploration phase. On the other hand, if needed, there exist certain SFRs that reflect such requirements. FDP_ITT.1.1, for example, requires system module intercommunication to be protected as well.

As a summary we have to conclude that much more use case studies have to be done to extract a rule set that allows a sufficient verification of the SFRs.

## 6 Conclusion and Future Work

In this work we described a simulation based verification methodology for security requirements in transaction level models of embedded systems. The basic

idea has its roots in the Common Criteria process where Security Functions and according Security Functional Requirements are defined. We showed that our proposed approach is able to verify such requirements by evaluating meta-information in the model during the simulation of its functional behavior. Furthermore, we showed that gained cost savings can be significant.

However, we have to say that our work is in an early stage. More case studies are needed to evaluate the applicability of our methodology. Especially the verification rules have to be refined and their influences on the design have to be evaluated.

## References

1. Balarin, F., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: A formal approach to system level design: metamodels and unified design environments. In: Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design. IEEE, Los Alamitos (2005)
2. Cai, L., Gajski, D.: Transaction level modeling: an overview. In: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM, New York (2003)
3. Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final (July 2009)
4. Deng, Y., Wang, J., Tsai, J.J.P., Beznosov, K.: An approach for modeling and analysis of security system architectures. IEEE Transactions on Knowledge and Data Engineering 15(5), 1099–1119 (2003)
5. Garavel, H., Helmstetter, C., Ponsini, O., Serwe, W.: Verification of an industrial SystemC/TLM model using LOTOS and CADP. In: 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design. IEEE, Los Alamitos (2009)
6. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. In: Proceedings of the 41st Annual Design Automation Conference. ACM, New York (2004)
7. Lotz, V., Kessler, V., Walter, G.H.: A formal security model for microprocessor hardware. IEEE Transactions on Software Engineering 26(8), 702–712 (2000)
8. Mellado, D., Fernández-Medina, E., Piattini, M.: A common criteria based security requirements engineering process for the development of secure information systems. Comput. Stand. Interfaces 29(2), 244–253 (2007)
9. Morimoto, S., Shigematsu, S., Goto, Y., Cheng, J.: Formal verification of security specifications with common criteria. In: Proceedings of the 2007 ACM Symposium on Applied Computing. ACM, New York (2007)
10. Schaumont, P., Verbauwhede, I.: Domain-specific codesign for embedded security. Computer 36(4), 68–74 (2003)
11. Trummer, C., Kirchsteiger, C.M., Steger, C., Weiss, R., Pistauer, M., Dalton, D.: Automated simulation-based verification of power requirements for systems-on-chips. In: 13th International Symposium on Design and Diagnostics of Electronic Circuits and Systems. IEEE, Los Alamitos (2010)

# Author Index