Fabio Massacci   Samuel T. Redwine Jr.
Nicola Zannone (Eds.)

Engineering Secure Software and Systems

First International Symposium, ESSoS 2009
Leuven, Belgium, February 4-6, 2009
Proceedings

Springer

Volume Editors

Fabio Massacci
Università di Trento
Dipartimento di Ingegneria e Scienza dell'Informazione
Via Sommarive 14, 38050 Povo (Trento), Italy
E-mail: fabio.massacci@unitn.it

Samuel T. Redwine Jr.
James Madison University
Department of Computer Science
701 Carrier Drive, Harrisonburg, VA 22807, USA
E-mail: redwinst@cisat.jmu.edu

Nicola Zannone
University of Toronto
Department of Computer Science
40 St. George Street, Toronto, ON M5S 2E4, Canada
E-mail: zannone@cs.toronto.edu

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.4.6, H.2.7, J.1, K.6.3, K.6.5

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN	0302-9743
ISBN-10	3-642-00198-X Springer Berlin Heidelberg New York
ISBN-13	978-3-642-00198-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12612079 06/3180 5 4 3 2 1 0

Preface

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

This unique events aims at bringing together researchers from Software Engineering and Security Engineering, helping 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 SIGSOFT (the ACM interest group in software engineering) and the IEEE TCSE is a clear sign of the importance of this inter-disciplinary 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, and less controllable, factors include the complexity of modern networked software systems, the unpredictability of practical development lifecycles, 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. And although some battles have been won, the jury is still out on the final verdict.

The conference program included two major keynotes from Axel Van Lamsweerde (U. Louvain) and Wolfram Schulte (Microsoft Research) and an interesting blend of research, industry and idea papers.

In response to the call for paper, 57 papers were submitted. The Program Committee selected 10 papers as research papers (17.5%), presenting new research results in the realm of engineering secure software and systems. It further selected four industry reports, detailing concrete case studies in industry and the lessons that practitioners and researchers can learn out of this experience. We also included four ideas papers that the Program Committee judged as interesting but not yet mature for a full paper presentation.

The program also featured a slate of tutorials by industry representatives on “Security by Construction” by Rod Champan (Praxis), “Management in Practice – Model – Based Security Risk Analysis with the CORAS Method” by Heidi E. I. Dahl and Mass Soldal Lund (SINTEF), “Inside the Biggest of the OWASP Top-10 Issues” by R. van Wyk (KRvW Associates, LLC), and a research tutorial on “Security – Philosophy, Patterns and Practices” by Munawar Hafiz (University of Illinois at Urbana-Champaign).

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. Nicola Zannone did a great job by assembling the proceedings for Springer.

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

Last but not least, we would like to thank all sponsors, and the DistriNet research group of the K.U.Leuven in particular, for covering the financial aspects of the event.

November 2008

Bart De Win
Fabio Massacci
Sam Redwine

Organization

General Chair

Bart De Win	Katholieke Universiteit Leuven, Belgium

Program Co-chairs

Fabio Massacci	Università di Trento, Italy
Samuel Redwine	James Madison University, USA

Publication Chair

Nicola Zannone	University of Toronto, Canada

Tutorial Chair

Riccardo Scandariato	Katholieke Universiteit Leuven, Belgium

Financial Chair

Katrien Janssens	Katholieke Universiteit Leuven, Belgium

Local Arrangements Chair

Koen Yskout	Katholieke Universiteit Leuven, Belgium

Steering Committee

Gary McGraw	Cigital, USA
Bashar Nuseibeh	The Open University, UK
Jorge Cuellar	Siemens AG, Germany
Fabio Massacci	Università di Trento, Italy
Samuel Redwine	James Madison University, USA
Wouter Joosen	Katholieke Universiteit Leuven, Belgium

Program Committee

Matt Bishop	University of California (Davis), USA
Brian Chess	Fortify Software, USA
Richard Clayton	Cambridge University, UK
Christian Collberg	University of Arizona, USA
Noopur Davis	Davis Systems, USA

Bart De Win	Katholieke Universiteit Leuven, Belgium
Juergen Doser	ETH, Switzerland
Eduardo Fernández-Medina	University of Castilla-La Mancha, Spain
Dieter Gollmann	Hamburg University of Technology, Germany
Michael Howard	Microsoft, USA
Cynthia Irvine	Naval Postgradual School, USA
Jan Jürjens	Open University, UK
Volkmar Lotz	SAP Labs, France
Antonio Maña	University of Malaga, Spain
Robert Martin	MITRE, USA
Fabio Massacci	Università di Trento, Italy
Mira Mezini	Darmstadt University, Germany
Mattia Monga	Milan University, Italy
Andy Ozment	DoD, USA
Gunther Pernul	Universitat Regensburg, Germany
Domenico Presenza	Engineering, Italy
Samuel Redwine	James Madison University, USA
Riccardo Scandariato	Katholieke Universiteit Leuven, Belgium
Ketil Stølen	Sintef, NO
Eric Vetillard	Trusted Logic, France
Jon Whittle	Lancaster University, UK
Mohammad Zulkernine	Queen's University, Canada

External Reviewers

Mohamed Atef
Sean Barnum
Christian Broser
Istehad Chowdhury
Heidi Dahl
Lieven Desmet
Francois Dupressoir
Stefan Durbeck
Joy Forsythe
Christoph Fritsch
Oliver Gmelch
Wolfgang Hennes
Thomas Heyman
Shahriar Hussein
Jan Kolter
Hristo Koshutanski
Edward Lee
Fredrick Lee
Olav Ligaarden
Matias Madou
Lorenzo Martignoni
Chris McLaughlin
Steven Murdoch
Antonio Muñoz
Brian O'Hanion
Yekaterina O'Neil
Aida Omerovic
Gimena Pujol
Atle Refsdal
Alfonso Rodriguez
Rolf Schillinger
Bjørnar Solhaug
Stefan Taubenberger
Hsiao-Ming Tsou
Lingyu Wang
Jacob West
Michael Yang
Yves Younan
Koen Yskout

Table of Contents

Verification of Business Process Entailment Constraints Using SPIN

Christian Wolter[1], Philip Miseldine[1], and Christoph Meinel[2]

[1] SAP Research
Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
{christian.wolter,philip.miseldine}@sap.com
[2] Hasso-Plattner-Institute (HPI) for IT Systems Engineering
University of Potsdam, Germany
meinel@hpi.uni-potsdam.de

Abstract. The verification of access controls is essential for providing secure systems. Model checking is an automated technique used for verifying finite state machines. The properties to be verified are usually expressed as formula in temporal logic. In this paper we present an approach to verify access control security properties of a security annotated business process model. To this end we utilise a security enhanced BPMN notation to define access control properties.

To enhance the usability the complex and technical details are hidden from the process modeller by using an automatic translation of the process model into a process meta language (Promela) based on Coloured Petri net (CPN) semantics.

The model checker SPIN is used for the process model verification and a trace file is written to provide visual feedback to the modeller on the abstraction level of the verified process model. As a proof of concept the described translation methodology is implemented as a plug-in for the free web-based BPMN modelling tool Oryx.

Topics: Business Processes, Access Control, Verification, Model Checking, SPIN.

1 Introduction

Business Process Management systems coordinate activities, resources, and data based on the formal representation of process logic defined by a process model. Different kinds of resources are associated with a process model. Passive resources, such as physical materials or information, are related to the flow of data in a process model, while active resources, such as humans, perform activities at runtime and are related to the flow of control, which deals with the order of activity occurrences in a process model [1].

Traditional access controls for active resources include mandatory access control (MAC), discretionary access control (DAC), and role-based access control (RBAC). To implement access controls, access control lists (ACLs), capability lists, or policy-based mechanisms are often adopted [2].

Several extensions for those access controls have been developed capturing organisational aspects, temporal dependencies, and task and history-based concepts [3,4,5,6].

These extensions allow access control requirements to be specified at a finer-grained level than can be achieved using traditional access control models. Typically, this is realised by enriching access control policies by adding constraints defining rules, obligations, or exceptions [5,7]. The addition of access control constraints however, significantly increases the complexity, so it quickly becomes difficult to understand the policy expressed, such that changes can lead to potential unwanted side-effects, inconsistencies, and security breaches [8,9].

Controlling access in a business process is a recognised fundamental principle of user interaction in computer security. Organisational goals must be complemented by control goals such as realised through access constraints [7]. One of the earliest access constraints is the *four-eyes principle* that first appeared in Saltzer and Schroeder [10]. Later the term *Separation of Duty* was introduced as a principle for preserving integrity and a mechanism for error control and fraud prevention. These concepts are applied to a process by limiting a user's access rights statically at the point of the process definition and dynamically at process runtime. The later concept is further refined by Crampton *et al.* in [11] by *entailment* and *cardinality* constraints that restrict access to process activities depending on the execution history of a user in a running process.

The existing verification methodologies for access control properties in the context of business processes as discussed in [11,12,13] are based on simulations and the identification of an execution trace that does not violate the defined authorisation requirements. At runtime the process must exactly follow this plan, which is unlikely in case execution behaviour is not predictable, due to arbitrary cycles or choice-based branching. Therefore, a complete state space exploration is necessary to guarantee that any possible runtime behaviour violates the authorisation properties. Existing state space exploration approaches for business processes focus on control-flow related property verification, such as deadlocks, and do not cover authorisation related properties [14,15,16]. Accordingly, the contributions of this paper are as follows:

- This paper presents an approach for automated model-checking and analysis of entailment constraints, such as Separation of Duty, in the context of business processes to detect deadlocks and security property violations.
- Authorisation artifacts are added to the process modelling standard BPMN [17] that is widely accepted in the industry and academia to ease the specification of authorisation requirements. A formal semantic for these artifacts is defined by using Coloured Petri nets (CPN) allowing the automated processing of security properties defined in the business processes.
- To verify the security properties, we define a translation of a process model into the the Process Meta Language (Promela) based on the given formal semantics. Promela specficiations can be used as input for the model checker SPIN [18]. Related security properties are verified against a set of linear temporal logic formulae.
- The results appear to be promising enough to develop a model checking plug-in for the free web-based process modelling tool Oryx[1] allowing an automated translation of security annotated process models into Promela, simulation and verification of security properties defined in a simple dialogue, and reporting verification results to the modeller.

[1] See http://oryx-editor.org

The rest of the paper provides some required background about the formal execution semantics of business process models and access control requirements in Section 2 by utilising classical and Coloured Petri nets (CPNs). Section 3 describes the translation of the process model into Promela code for automated verification of reachability and security properties. We also briefly present the model checking plug-in developed for Oryx. In Section 4 we give an overview about related work in the domain of model checking for business processes and security properties. The results of our work along with a conclusion and future research directions are given in Section 5.

2 Formalisation of Access Controls in Process Models

Existing process modelling notations, such as BPMN, UML 2.0 AD, or BPEL provide poor support for expressing access control properties [19,20]. Such properties assert who is allowed or must perform a certain activity in the context of a running business process. Fine-grained specification of authorisation constraints has been investigated for a long time in literature [7,21,11]. Concepts were presented investigating how authorisation, risk assessment, and compliance artefacts can be specified as part of a business process model [22,23]. To verify such properties directly in the context of process models we first provide a formal semantic of a general business process as a Petri net and then extend these nets by the concepts of token colours turning the net into a CPN for formally defining authorisation concepts in a business process.

2.1 Formalising Control Flow Behaviour

BPMN essentially provides a graphical notation for business process modelling, with an emphasis on control-flow. In general a business process diagram consists of nodes and edges. A node can be typed as an activity, an event or a branching node, such as a parallel split, a parallel join, an option split, a merge join, and data or external event-based choices. An edge links two nodes in a diagrams and shows the execution order in terms of a sequence flow or message exchange.

Therefore, a process model can be defined as a tuple (N,F, n_s, n_e) where:

- N is a set of general nodes objects
- $F \subseteq N \times N$, defines the flow relation, i.e. edges connecting nodes,
- n_s is the start node of the process,
- n_e is the end node of the process.

For any node $n \in N$, the input nodes of n are given by $in(n) = \{m \in N | mFn\}$ and the output nodes are given by $out(n) = \{m \in N | nFm\}$. $\forall n \in N, \exists n_s, \exists n_e, n_s F n \wedge n F n_e$. i.e. every node is on a path from the start node s_n to the end node e_n.

We will use this definition to provide a formal definition in terms of a Petri net fragment for a selected set of BPMN nodes. Petri nets were initially proposed by C.A. Petri in 1962 [24] for describing information processing systems, characterised as being concurrent, asynchronous, distributed, parallel, non-deterministic and stochastic. Petri nets are used in many different application areas for modelling and analysis of systems.

A Petri net is a tuple $N = (P, T, F, i, o, \alpha)$, where P = $p_1, p_2, ..., p_n$ is a finite set of places, T = $t_1, t_2, ..., t_m$ is a finite set of transitions, satisfying $P \cap T = \oslash, F \subseteq (P \times T) \cup (T \times P)$ is the flow relation and $m_0 : P \rightarrow N$ is the initial marking. $\alpha : T \rightarrow A$ is a labelling function where A is a transition label. The symbols $\bullet t$,$t\bullet$, $\bullet p$, $p\bullet$ define the pre-set and post-set of every place p or transition t respectively. A *marking* of a Petri net is an assignment of a non-negative integer to each place, called a token. The structure of the Petri net defines a set of firing rules that determine the behaviour of the net. A transition t is enabled when each $p \in \bullet t$ has at least one token. The Petri net moves from one marking to another by firing one of the enabled transitions. When a transition t fires, one token is removed from each place $p \in \bullet t$ and one token is added to each place $p \in t\bullet$. If m_1 and m_2 are markings, we will denote by $m_1 \, [t\rangle \, m_2$ the fact that m_2 is reached from m_1 after transition t being fired. Accordingly, the formal semantics are defined [25]:

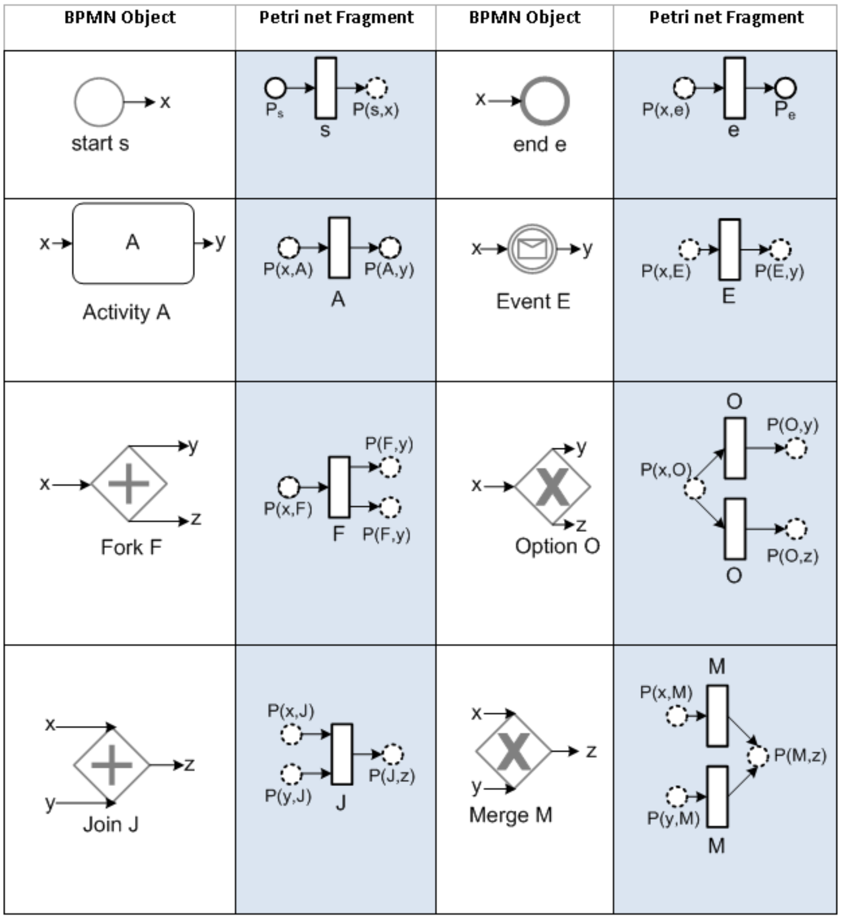


Fig. 1. Mapping BPMN to Petri nets [25]

Generally, a node such as a task or an intermediate event is interpreted as a transition with one input place and one output place and any sequence flow is mapped to a place. Similarly, a start event and an end event is a transition signalling the process's start and end. Branches are mapped to one or two transitions capturing their routing behaviour.

2.2 Expressing Access Control in CPN

A Coloured Petri net (CPN) is a specialised Petri net containing the classical elements, but also adds textual *inscriptions* next to the places, transitions, and arcs. The inscriptions are written in the CPN ML language [14]. CPN ML is a type-safe programming language and can be used for functional programming. A CPN is not bounded, which means each place can be marked with one or more tokens. In contrast to classical Petri nets, CPN tokens are allowed to not only represent a single integer value, but can contain arbitrary complex data types called *token colour*.

A place p in a CPN is a tuple $(i_{colour}, i_{marking})$ where, i_{colour} is an inscription defining the token colour of tokens that are allowed to be put into p and $i_{marking}$ is an inscription defining the initial set of tokens put into p at the beginning. When a transition fires its arc inscriptions determine which coloured tokens are to be removed from input places and which tokens have to be added to the output places.

Arc inscriptions can be also used to express data modifications, such as incrementing a data value of a token when a transition fires or testing the containment of a specific data value to determine which token is to be consumed in case several tokens are available. Compared to classical Petri nets a transition's enablement is determined based on the tokens present in the input places and the evaluation of the arc inscriptions, thus these inscriptions can be considered as transition *guards*.

In order to capture human interaction and related authorisation requirements we have to extend the formal semantic of BPMN. As shown in Figure 2 during the modelling process annotations and additional attribute values have to be defined to specify that a task must be performed by a person acting in a specific functional role or providing a specific capability, such as speaking English. This concept refers to role-based and attribute-based access control [4].

Using CPN we can use an *unbound* place representing the *resource repository* accepting tokens of the colour set *USER*. An unbound place may contain an arbitrary number of tokens. This token repository is an input and output place of every human activity in the process model with an appropriate arc inscription based on the role and attribute requirements expressed in the diagram, such as English speaking clerk. Therefore, the arc inscription represents the authorisation decision. In order to support dynamic Separation of Duty and Binding of Duty requirements the history of the token consumed by the transition is modified to keep track of the participation of the token in a particular transition firing. As depicted in Figure 2, we express Separation and Binding of Duty requirements by defining transition guards that evaluate to true if a specific transition id is present, or absent respectively in a token's history list.

Therefore, the overall colour set *USER* is defined as $USER \subseteq id \times role \times history \times A$, where id is an unique identifier, $role$ represents the assigned functional role at runtime, $history$ is a list of all transitions this token was consumed by, and A be the set of different attribute types defined in the process model.

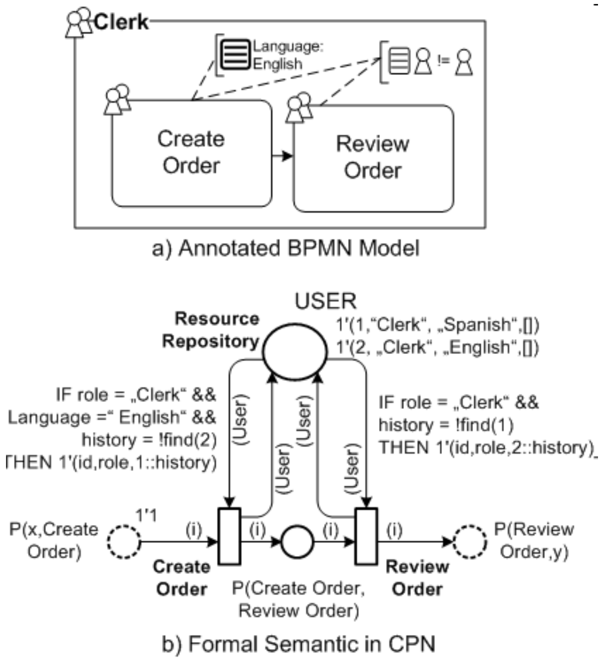

Clerk
Language: English
!=
Create Order
Review Order
a) Annotated BPMN Model
USER
Resource Repository
1'(1,"Clerk", „Spanish",[])
1'(2, „Clerk", „English",[])
IF role = „Clerk" && Language =" English" && history = !find(2) THEN 1'(id,role,1::history)
IF role = „Clerk" && history = !find(1) THEN 1'(id,role,2::history)
(User)
1'1
P(x,Create Order)
(i)
Create Order
P(Create Order, Review Order)
Review Order
P(Review Order,y)
b) Formal Semantic in CPN

Fig. 2. Formal Semantic of Authorisation Annotations

3 Authorisation Verification

The previous section outlined how a business process model extended with support for representing authorisation constraints could be mapped to a CPN model. As this provides a formalised, rigorous model of the process model without ambiguity, assertions can be checked against this model that allow verification of the authorisation constraints. This section details how such analysis can take place by using the model checker SPIN [18] which previous work has shown, allows verification against Petri net models [26].

SPIN, an acronym for *S*imple *P*romela *In*terpreter, is a generic model checking tool to formally analyse the logical consistency of distributed systems, which are defined using the Process Meta Language Promela. The process meta language Promela is an abstract system design specification language. Promela is not an implementation language, but rather focuses on synchronisation and coordination, targeted to the description of concurrent software systems rather than on hardware. The basic building blocks of Promela are asynchronous processes, buffered and unbuffered message channels, synchronisation statements and structured data. There is only a very limited number of computational functions.

3.1 Translation Rules

There exist different approaches in translating Petri nets into a Promela specification. We base our approach on the concepts presented in [26] by Ribeiro and Fernandes. In order to correctly specify a Coloured Petri net transition in Promela, two primary issues must be addressed.

1. First, determine when a transition t is enabled according to $in(t) \wedge out(t)$.
2. Second, for each firing the marking must be modified according to $m_1\ [t\rangle\ m_2$.

In Promela we represent the control-flow related marking of a Petri net as a boolean array $bool\ p[]$. Meaning, either there is a token in place p_i or not. Further, this implies that all control flow places are 1-bounded. Therefore, we represent the enablement of a transition t with $p_i = 1 \forall p_i \in in(t) \wedge p_j = 0 \forall p_j \in out(t)$ in Promela as a propositional symbol with $in(t) = p_1$ and $out(t) = p_2$:

$$ready\_t\ p[1]\ \&\&\ !p[2]$$

Accordingly the firing of t is specified as:

$$fire\_t\ atomic\{p[1] = 0; p[2] = 1; \}$$

The firing of a transition results in the modification of several data values and the movement of several tokens to create the new marking m_2. Because each token movement would lead to a new state in SPIN all movements must be performed in an *atomic* block to represent the state transition from $m_1\ [t\rangle\ m_2$ properly.

The coloured tokens representing human resources are defined as a complex data structure *TOKEN* in Promela. Each user attribute is a member of that structure. In addition a global process history is defined as a buffered channel acting as a data storage. In Figure 3 the history list and the resource repository are defined as channels holding up to *size* entries of TOKENs or a tuple $(transition\_id, token\_id)$:

```
chan resources = [size] of { TOKEN };
chan history = [size] of { int, byte };
/* generic colour list */
typedef TOKEN {byte id;mtype role;mtype language;}
```

Fig. 3. TOKEN type definition

By utilising Promela channels we are able to randomly select or query the existence of a specific token based on its data values. In Promela we can determine the existence of a token with a specific colour (i.e. data values) of the colour set $\{id, role, language\}$ by using a channel poll:

$$resources\ ??\ [\_, \text{Clerk}, \_]$$

The underscore represents the *don't care* variable acting as a wild card, while the term *Clerk* is a constant. This expression evaluates *true* in case a token with any id, the role attribute value equals *Clerk*, and any language is present in the channel. Accordingly, we can extend the enablement of a human task t requiring a *clerk* by:

$$ready\_t\ p[1]\ \&\&\ !p[2]\ \&\&\ \text{resources}\ ??\ [\_, \text{Clerk}, \_]$$

In case we want to modify the data values of a token stored in a channel we have to remove the token from the channel, modify its data values and put it back into the channel. To be side-effect free this set of operations must be wrapped into an *atomic* block:

$$atomic\{\text{ resources? token; token.id+1; resources!token}\}$$

In order to specify a Separation of Duty a simple channel poll is not enough to find a token fulfilling all attribute-based properties as well as not violating the Separation of Duty constraint. That means for each token that satisfies the attribute constraints, we also have to inspect its history data. Therefore, for each enabled transition that must satisfy an entailment constraint we must iterate over all tokens that potentially satisfy the attribute requirements and poll if their history channel contains the excluded activity or not. Accordingly, the first token satisfying the attribute-based requirements and not violating the Separation of Duty constraint will be selected (cf. Figure 4):

```
inline checkHistory(transition_id,oldtransition_id){ atomic{
do
::oldtransition_id > 0 && nempty(history) ->
do /* get excluded token */
  ::history ?? [eval(oldtransition_id), tmp] ->
    history ? eval(oldtransition_id), tmp;
    /* remove token from potential owner list */
    resourceplaces ??
    eval(tmp),excludedToken.role, excludedToken.language;
    excluded !  tmp, excludedToken.role, excludedToken.language;
  ::else -> break;od;
::empty(history) -> break;od;
do /* get abac satisfying token */
::resourceplaces ??
    [_,eval(currentToken.role),eval(currentToken.language] ->
    resourceplaces  ?? <currentToken.id,eval(currentToken.role),
    eval(currentToken.language)>;
    logHistory(currentToken.id,transition_id);
    t[transition_id]= currentToken.id;break;
::else -> break;od;
do /* restore the original resource */
::nempty(excluded) ->
  excluded ? currentToken; resourceplaces ! currentToken;
  history ! oldtransition_id, currentToken.id;
::empty(excluded) -> break;od;}}
```

Fig. 4. Potential Owner Determination for Entailment Constraints

3.2 Property Verification in Oryx

By applying formal semantics to the annotated process model it is possible to translate a business process and its authorisation requirements into a Promela specification to verify its security properties. As an example consider the process given in Figure 5.

Assume this process diagram has been modelled by a business process expert. It defines the process of handling a purchase order in a company's British branch. In case the order value is above 5000 it must be reviewed, e.g. to prevent wrong order placement. In addition, the branch manager must approve the order and it must be verified that the ordered goods are available before the process ends properly.

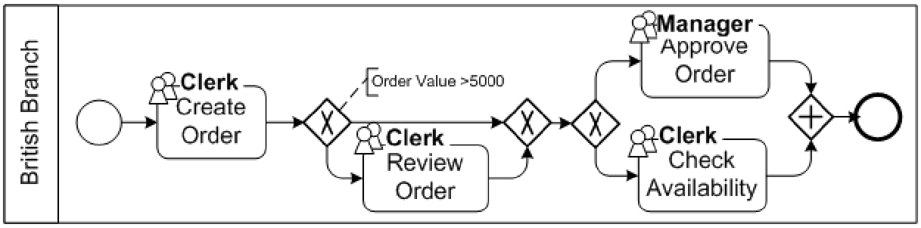


Fig. 5. Erroneous Purchase Order

Further, we assume in this company exists a document defining corporate security and governance guidelines that the company and all its processes must comply to. Such guidelines are typical defined by security experts and mandated by law regulations. By using a formal verification method it is possible to verify a process model against such guidelines. We assume, the corporate guidelines demand that every order must be reviewed. In addition, a creator of an order can not review his own order.

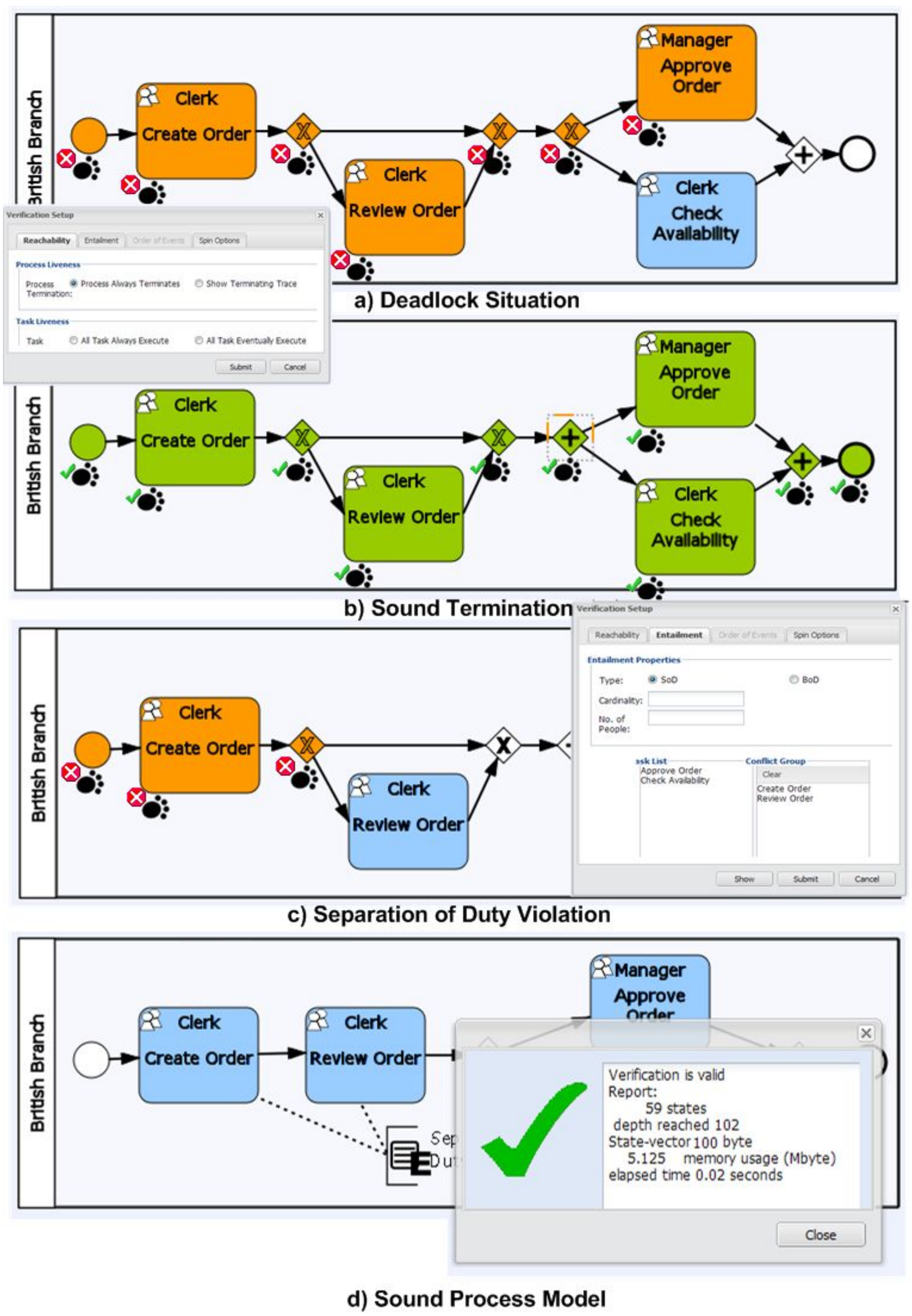
Given the example process we can use the translation rules described in the previous section to verify the Promela specification against general reachability and deadlock properties, such as:

- Does the process always terminate: []<>endstate
- Find at least one path that reaches the end: <> endstate
- All tasks are eventually enabled: $\forall t \in T$ ([]<> rd_t)

This list can be extended by security and governance based state and temporal properties:

- Compliance Violation: Every order must be reviewed:
  [](rd_create_order $\rightarrow$ <> rd_review_order) $\wedge$ <> rd_create_order
- Attribute Violation: A *clerk* must execute the task *Review Order* (id=2):
  [] <> abac_create_order, with *abac_create_order* being a propositional symbol:
  abac_create_order history?? [2,t_id] && resources ?? [eval(t_id),Clerk,_])
- SoD Violation: *Create Order* (id =1) and *Review Order* (id=2) must be performed by different clerks:[] ! *sod_create_review*,
  with *sod_create_review* history??[1,t_id] && hist??[2,eval(t_id)]

The described formalisation and translation is implemented for a process modeller. Authorisation requirements are captured either by using swimlanes representing an organisational role, which contains tasks assigned to that role or by adding authorisation annotations to one or more tasks in the model, for instance to represent an attribute-based



Verification Setup

Reachability | Entailment | Order of Events | Spin Options

Process Liveness

Process Termination: [✓ Process Always Terminates] Show Terminating Trace

Task Liveness

Task: All Task Always Execute, All Task Eventually Execute

Submit Cancel

Verification Setup

Reachability | Entailment | Order of Events | Spin Options

Entailment Properties

Type: [✓ SoD] BoD

Cardinality:

No. of People:

sk List: Approve Order, Check Availability

Conflict Group: Clear; Create Order, Review Order

Show Submit Cancel

```
Verification is valid
Report:
      59 states
 depth reached 102
State-vector 100 byte
   5.125    memory usage (Mbyte)
elapsed time 0.02 seconds
```

Fig. 6. Verification with Oryx

requirement or a Separation of Duty annotation. These visual elements are translated into rules when generating the Promela specification. The previously described security properties that a process model must comply to are defined in a simple user dialogue in terms of natural language representation of the formal properties. In case of entailment constraints the modeller defines a group of conflicting tasks that shall or shall not be executed by the same user. The input is translated into a LTL formula and is fed along the Promela code to SPIN. The Promela code is enriched with C-like *printf* statements that are printed whenever a user is assigned to a manual activity or a transition fires. The *printf* traces are send back to the browser and rendered on top of the process diagram. Several verification results visualised in Oryx based on the discussed process example are shown in Figure 6. In test case *a* a deadlock is detected due to the wrong usage of an *exclusive* split and a *parallel* join. In test case *b* the process terminates and the potential sound execution path is visualised if desired. In case *c* a Separation of Duty violation is detected between the tasks *Create Order* and *Review Order*, because a related security annotation is missing preventing the assignment of the same user token to both tasks. The last case *d* shows a proper annotated process model with regards of the discussed security requirements.

4 Related Work

The verification of access controls is essential for verifying security properties of a system. Model checking is an automatic technique used for state space exploration and analysis. The basic idea is to compute all reachable states and state changes.

The computation and verification of reachable states of business process control-flow behaviour has been actively discussed. In [27], Aalst *et al.* provide a mapping of all control-flow constructs of BPEL into Petri nets. Later a similar mapping of BPMN to Petri nets was discussed in [25] on which our approach is based. To verify the behaviour of business processes a number of researchers have developed a mapping from BPEL to Promela [28,29]. Next to the control-flow behaviour, Fu, Bulant and Su [30,31] also provide a mapping to formal XML schemas, effectively allowing the verification of data manipulation. Their work resulted in the Web Service Analysis Tool (SWAT). In [32], a tool called VERBUS, an acronym for verification for business processes, is presented supporting the translation of BPEL into Promela and SMV the input language for NuSMV. A major difference compared to our approach is that they all focus on control-flow related property verification and therefore do not address any kind of authorisation verification.

Some work exists considering the verification of authorisations in the context of business processes, but most of them only address role-based access control requirements, without any consideration of attributes and entailment properties. In [11], the verification of access control properties for general workflows by proving the absence of security implied deadlocks is presented. Similarly, in [33] the SAL model checker is used to verify role-based authorisations. They derive security configurations from a RBAC enhanced BPEL process description and assert the existence of an execution path that does not lead to a policy implied deadlock. Slightly different, in [34] X-GTRBAC policies are used to generate a behavioural model that is translated into a finite state machine to verify the access control policies. The model checker NuSMV in combination with LTL formulae is used in [7] and [16] to verify a process model by translating it into a finite state machine.

The existing fine-grained authorisation verification approaches do not explore the full state space, and are satisfied with the detection of a single execution trail that successfully reaches the end state. In [35,36] a formal model and an algorithm that calculates a single process execution path that is deadlock free is presented. Wang and Li discussed a role-relation-based access control model for workflow systems in [12]. They examined the workflow resiliency problem and outlined an approach to determine a valid set of users that must execute activities in a process to successfully reach the end state of the process. All these approaches are able to assert that there exists at least one security violation free execution path and therefore the behaviour at runtime must exactly follow this path to avoid violations, which is infeasible in most of today's dynamic and event-driven business scenarios.

5 Conclusion

In this paper we presented an approach to automatically verify access control properties defined on the abstract level of business processes. We discussed that this is helpful in order to detect potential deadlocks, due to modelling errors, as well as to verify security properties, that could be derived from corporate governance documents and general law regulations.

Existing modelling notations lack the support of role-, attribute and entailment-based authorisation requirements. We developed an extension [9] for BPMN to capture such requirements on a very abstract level suitable for business analysts. To verify such annotated models, we formally specified these extensions by using CPN ML to capture user roles, capabilities in terms of attributes and define entailment rules, such as Separation of Duty. This allows us to automatically analyse the process model and directly translate it into a Promela specification. The results of the model checking are traced and visual feedback is provided on the process diagram hiding the complexity of the model checking output from the modeller. A prototype has been implemented for the Oryx modelling tool demonstrating the applicability of the proposed formal semantics and translation into Promela. The following Table 1 provides some insight into the computation time and state vector size for the four test cases shown in Figure 6.

The results indicate SPIN's state space optimisation depending on the property to be verified. To conduct a first evaluation of the scalability of our approach we modelled a

Table 1. Model Checking Experiments

No.	States	Computation Time (Sec)	State Vector (Byte)
a)	15	0.01	88
b)	59	0.01	88
c)	64	0.01	92
d)	59	0.02	100
e)	45177	0.33	168

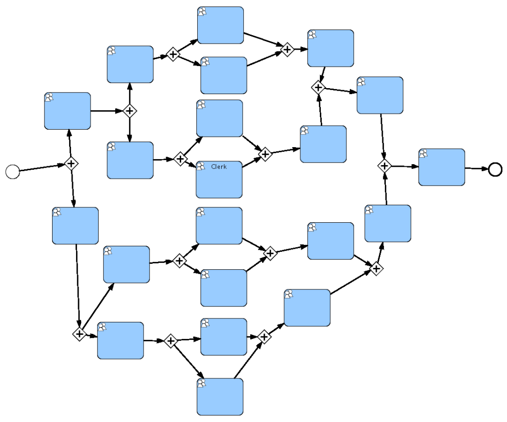


Fig. 7. Simple Scalability Test

process with several concurrent human tasks (cf. Figure 7). The results listed in the table show that while the state space grows significantly, state vector and time consumption are still very small. Therefore, our approach is able to deal with a process size well suited for the domain of authorisation verification. The experiment was conducted using the latest version of the SPIN tool executed under a Debian operating system on a recent computer.

Explicit state space verification becomes difficult when dealing with potentially unbounded states spaces, for instance because of arbitrary cycles. In future work we plan to apply unrolling techniques to support the verification of annotated cycles. We also plan to investigate into alternative model checking techniques such as bounded model checking. At the moment the resource repository is initialised with a fixed set of user tokens calculated based on the Cartesian product of all defined roles and attribute types in a process model. We plan to automatically adjust the number of generated tokens in order to identify a minimal set of user tokens that are necessary to execute the process without implying a deadlock with respect to defined security properties. As demonstrated in [31], Information flow is an essential part of common process modelling notations, thus we plan to integrate the concept of data flow into the proposed Coloured Petri net semantics to detect integrity and confidentiality violations. In the end we will integrate the model checking techniques described in this paper with a model-driven approach to generate platform-specific security policies to provide a tool chain supporting security specification, security verification, security policy deployment and security policy enforcement.

References

1. Zur Muehlen, M.: Organizational Management in Workflow Applications – Issues and Perspectives. Inf. Technol. and Management 5(3-4), 271–291 (2004)
2. Cao, X., Iverson, L.: Intentional Access Management: Making Access Control Usable for End-Users. In: SOUPS 2006: Proceedings of the second symposium on Usable privacy and security, vol. 2, pp. 20–31. ACM Press, New York (2006)

3. Alotaiby, F.T., Chen, J.X.: A model for team-based access control (tmac 2004). In: ITCC 2004: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004), Washington, DC, USA, p. 450. IEEE Computer Society, Los Alamitos (2004)
4. Oh, S., Park, S.: Task-role-based access control model. Inf. Syst. 28(6), 533–562 (2003)
5. Wang, L., Wijesekera, D., Jajodia, S.: A logic-based framework for attribute based access control. In: FMSE 2004: Proceedings of the 2004 ACM workshop on Formal methods in security engineering, pp. 45–55. ACM, New York (2004)
6. Thomas, R.K.: Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management. pp. 166–181 (1997)
7. Schaad, A., Lotz, V., Sohr, K.: A model-checking approach to analysing organisational controls in a loan origination process. In: SACMAT 2006: ACM symposium on Access control models and technologies, pp. 139–149. ACM, New York (2006)
8. Jeager, T.: Managing access control complexity using metrics. In: SACMAT 2001: Proceedings of the sixth ACM symposium on Access control models and technologies, pp. 131–139. ACM Press, New York (2001)
9. Wolter, C., Schaad, A., Meinel, C.: Task-based entailment constraints for basic workflow patterns. In: SACMAT 2008: Proceedings of the 13th ACM symposium on Access control models and technologies, pp. 51–60. ACM, New York (2008)
10. Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. In: Proc. IEEE, vol. 63, pp. 1278–1308. IEEE Computer Society Press, Los Alamitos (1975)
11. Tan, K., Crampton, J., Gunter, C.A.: The Consistency of Task-Based Authorization Constraints in Workflow Systems. In: CSFW, p. 155- (2004)
12. Wang, Q., Li, N.: Satisfiability and Resiliency in Workflow Systems. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 90–105. Springer, Heidelberg (2007)
13. 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)
14. Jensen, K., Kristensen, L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer (STTT) 9(3), 213–254 (2007)
15. Liu, Y., Mueller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Syst. J. 46(2), 335–361 (2007)
16. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
17. Object Management Group. Business Process Modeling Notation Specification (2006), `http://www.bpmn.org`
18. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
19. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Workflow Resource Patterns: Identification, Representation and Tool Support. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 216–232. Springer, Heidelberg (2005)
20. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N.: On the Suitability of BPMN for Business Process Modelling. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 161–176. Springer, Heidelberg (2006)
21. Botha, R.A., Eloff, J.H.P.: Separation of Duties for Access Control Enforcement in Workflow Environments. IBM System Journal 40(3), 666–682 (2001)
22. Wolter, C., Schaad, A.: Modelling of Task-Based Authorization Constraints in BPMN. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 64–79. Springer, Heidelberg (2007)

23. Sadiq, W.S., Governatori, G., Namiri, K.: Modeling Control Objectives for Business Process Compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 149–164. Springer, Heidelberg (2007)
24. Desel, J., Reisig, W., Rozenberg, G. (eds.): Lectures on Concurrency and Petri Nets. LNCS, vol. 3098. Springer, Heidelberg (2004)
25. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of bpmn process models. Technical report, Queensland University of Technology (2007)
26. Ribeiro, O.R., Fernandes, J.M.: Translating Synchronous Petri Nets into PROMELA for Verifying Behavioural Properties. In: International Symposium on Industrial Embedded Systems, SIES 2007 (2007)
27. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in ws-bpel. Sci. Comput. Program. 67(2-3), 162–198 (2007)
28. Yang, Y., Tan, Q., Xiao, Y., Yu, J., Liu, F.: Exploiting Hierarchical CP-Nets to Increase the Reliability of Web Services Workflow. In: SAINT 2006: Proceedings of the International Symposium on Applications on Internet, pp. 116–122. IEEE Computer Society Press, Los Alamitos (2006)
29. Nakajima, Shin: Lightweight formal analysis of Web service flows. Progress in informatics: PI 2, 57–76 (2005)
30. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: WWW 2004: Proceedings of the 13th international conference on World Wide Web, pp. 621–630. ACM Press, New York (2004)
31. Fu, X., Bultan, T., Su, J.: Model checking XML manipulating software. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, pp. 252–262. ACM, New York (2004)
32. Fisteus, J.A., Fernández, L.S., Kloos, C.D.: Applying model checking to BPEL4WS business collaborations. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 826–830. Springer, Heidelberg (2006)
33. Xiangpeng, Z., Cerone, A., Krishnan, P.: Verifying BPEL Workflows Under Authorisation Constraints. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 439–444. Springer, Heidelberg (2006)
34. Masood, A., Bhatti, R., Ghafoor, A., Mathur, A.: Model-based Testing of Access Control Systems that Employ RBAC Policies. In: BPM 2006. LNCS, pp. 439–444. Springer, Heidelberg (2006)
35. Huang, W.-k., Atluri, V.: SecureFlow: A Secure Web-Enabled Workflow Management System. In: ACM Workshop on Role-Based Access Control, pp. 83–94 (1999)
36. Crampton, J.: A Reference Monitor for Workflow Systems with Constrained Task Execution. In: SACMAT 2005: Proceedings of the tenth ACM Symposium on Access Control Models and Technologies, pp. 38–47. ACM, New York (2005)

From Formal Access Control Policies to Runtime Enforcement Aspects

Slim Kallel[1,2], Anis Charfi[3], Mira Mezini[1], Mohamed Jmaiel[2], and Karl Klose[4]

[1] Software Technology Group, Darmstadt University of Technology, Germany
[2] ReDCAD Laboratory, National Engineering School of Sfax, Tunisia
[3] SAP Research CEC Darmstadt, Germany
[4] Department of Computer Science, University of Aarhus, Denmark

Abstract. We present an approach that addresses both formal specification and verification as well as runtime enforcement of RBAC access control policies including application specific constraints such as separation of duties (SoD). We introduce *Temporal$\mathcal{Z}$*, a formal language based on Z and temporal logic, which provides domain specific predicates for expressing RBAC and SoD constraints. An aspect-oriented language with domain specific concepts for RBAC and SoD constraints is used for the runtime enforcement of policies. Enforcement aspects are automatically generated from *Temporal$\mathcal{Z}$* specifications hence avoiding the possibility of errors and inconsistencies that may be introduced when enforcement code is written manually. Furthermore, the use of aspects ensures the modularity of the enforcement code and its separation from the business logic.

1 Introduction

Ensuring that software exhibits certain security properties is crucial in many application domains. Static verification, while the most reliable means for this purpose, may not always be feasible. Runtime verification is an alternative in such cases. There are two aspects of runtime verification: (a) a formal specification of the properties to be verified, and (b) mechanisms to enforce the formally specified properties on particular program runs (aka runtime monitoring). We propose an approach to runtime verification of security properties that addresses both aspects and supports automatic generation of enforcement code from formal specifications.

Schneider [1] uses Büchi automata to characterize the class of security properties enforceable by runtime monitoring. Schneider's automata are also called truncation automata [2], since they only support the truncation of execution action sequences that violate security properties. Our approach supports all policies modeled by security automata [1]. However, to keep the discussion focused, this paper specifically addresses role-based access control (RBAC for short) [3] with application specific constraints over execution action sequences known as separation of duty properties (SoD for short) [4].

There are three ingredients in our proposal. First, we present *Temporal$\mathcal{Z}$*, a formal specification language combining elements of Z [5] and temporal logic (LTL) [6]. To enable a direct and declarative specification of security policies, *Temporal$\mathcal{Z}$* embeds domain-specific formal predicates. Second, we propose to encode runtime enforcement logic in aspect modules in the aspect-oriented language ALPHA [7], which has expressive logic-based means to reason about the program execution. ALPHA is also extended with domain-specific language constructs to enable more direct and declarative encoding of runtime monitoring of security policies. Third, an automatic mapping of formal specifications to enforcement aspects is described as a mapping of domain-specific specification language constructs to domain-specific programming language constructs.

The proposal makes several contributions to the state of the art of runtime verification of security properties.

First, the proposed specification language is well suited for several reasons. In fact, set theoretic and predicate logic foundations of Z directly support the expression of concepts such as roles, rights, and related constraints. Moreover, the temporal operators of LTL are important for declaratively expressing temporal constraints on system behavior. In addition, complementing the general-purpose means of Z and LTL with domain-specific predicates increases the level of abstraction and declarativeness of the specifications. Finally, tools for verification and theorem proving such as Z-eves [8] can be used for verifying the consistency of formally specified security policies.

Second, the use of aspects ensures that the enforcement logic for a particular property is encapsulated in one module, separated from the functional code and from the enforcement code for other properties. Compared to many other AO languages, ALPHA is particularly well-suited for implementing security monitors. First, it exposes a rich model of the execution. Second, reasoning about execution action traces is expressed as logic queries. These features enable declarative and concise expression of constraints over execution history and facilitate the construction of security-specific predicates. Together the AO modularity and the specific features of ALPHA make it easy to map between specifications and respective enforcement code. As a result, enforcement code can be maintained incrementally when the formal specifications change. Further, it is possible to dynamically switch certain security policies on/off.

Finally, the automatic generation of enforcement aspects from formal specifications avoids the risk that faults and inconsistencies are introduced when enforcement logic for a formally specified policy is integrated manually with the application code. Other approaches that generate enforcement code from models [9,10] do not maintain the modularity of the specification, i.e., code responsible for enforcing different policies is not encapsulated in separated modules.

The remainder of the paper is organized as follows. Sec. 2 presents a running example and some prerequisites. Sec. 3 shows how *Temporal$\mathcal{Z}$* is used for the formal specification of RBAC policies and SoD. Sec. 4 explains how formal specifications are automatically mapped to ALPHA aspects. Sec. 5 reports on related work. Sec. 6 concludes the paper and discusses areas of future work.

2 Example and Prerequisites

We introduce a simple loan approval process (LAP) (inspired from [11]) and then give brief introductions to RBAC and to SoD. After that, we introduce *Temporal$\mathcal{Z}$*and ALPHA, which will be used respectively for formal specification and enforcement of RBAC and stateful access control policies.

2.1 Loan Approval Process

Initial application: A *teller* discusses with the *customer* to identify an appropriate loan program and verifies the application documents.

Rating process: Two types of rating are performed. An internal rating performed by a *financial clerk* verifies that the customer situation fits with the loan program being applied for. An external rating performed by a *teller* involves an external agency to check whether the customer has financial problems with other companies. Finally, a *supervisor* verifies both ratings to reduce the risk of error.

Bank decision: Based on the internal and external ratings, a *supervisor* takes a decision on the application. If the latter is approved, he gives the order to create the contract; otherwise, the process is canceled.

Sign contract: After the loan has been approved, a contract is sent to the *manager* and to the *customer* for signature.

Loan closing: The *teller* who entered the application data transfers the money to the customer's account. To avoid errors, a *supervisor* verifies this transfer step.

2.2 RBAC and Separation of Duties

RBAC is an authorization mechanism in which access decisions are based on the roles that users hold within an organization. The permissions to execute a set of operations are grouped in roles and users are assigned to one or more roles. SoD policies reduce the risk of frauds in business processes by a fine-grained control over the privileges for workflow tasks. Two categories are distinguished: static and dynamic SoD [4].

Static SoD (SSoD): Specifies that two mutually exclusive roles must never be assigned to the same user simultaneously, e.g., a bank employee cannot be assigned the roles *Supervisor* and *Manager*.

Dynamic SoD (DSoD): Describes a class of policies that take the dynamics of the system execution into account for managing roles and role membership. Four variations are found in the literature:

Simple Dynamic SoD (SDSoD): Two mutually exclusive roles must never be activated by a user at the same time, e.g., a bank employee can be statically assigned both roles *Teller* and *FinacialClerk*, but cannot activate them simultaneously.

Object-Based SoD (ObjDSoD): A user can activate two exclusive roles at the same time, but he cannot act upon the same object via both roles. E.g., if a user in LAP can be *Supervisor* and *Teller*, then he can execute the operations of the role *Teller* and verify his own work using his *Supervisor* role. E.g., an ObjDSoD constraint can be defined stating that the user can activate the role *Supervisor* and *Teller* at the same time as long as he does not act on the same loan application object.

Operational SoD (OpDSoD): A user can activate two exclusive roles at the same time, but cannot have all required authorizations to execute all tasks in a workflow. E.g., the critical sub-processes (e.g., *enterApplicationData*, *verifyRating*, *verifyTransfer*) cannot be executed by the same employee.

Operational Object-Based SoD (OpObjDSoD): Combines operational and object-based SoD; a user can activate two exclusive roles at the same time and can have the authorizations to execute all tasks in a workflow process, as long as these tasks do not act upon the same object.

2.3 *Temporal$\mathcal{Z}$*

In the following, we define the syntax and semantics of Temporal$\mathcal{Z}$ [12], which is a formal language that integrates linear temporal logic into the Z framework. A Temporal$\mathcal{Z}$ formula is either (a) a *domain specific predicate*, or (b) constructed from other formulas using the *temporal operators*: $\Box$(always), $\Diamond$(eventually), $\bigcirc$(next), $\boxminus$(up till now), $\diamondminus$(previously), $\ominus$(atlast), $\mathcal{U}$ (until), and $\mathcal{S}$ (since).

The semantics of *Temporal$\mathcal{Z}$* is defined in Fig. 1 as an evaluation according to a temporal model - a function from a time point *t* to the set of predicates that hold at *t*.

Three axiomatic functions are defined to evaluate temporal formulas at different abstraction levels. *EvalMT* interprets each formula at a given time point t using the temporal model m. *EvalMT* evaluates domain-specific predicates without temporal operators to *True* (E1), if the predicate holds at time point *t* according to *m*. For composed formulas, a variant of *EvalMT* is defined for each temporal operator; for brevity, only the evaluation of formulas composed with *always* $\Box$ (E2) and *until* $\mathcal{U}$ (E3) is shown in Fig. 1. The function *EvalM* (E4) generalizes the evaluation by abstracting away the time parameter. *Eval* (E5) also abstracts away the model parameter. We also define equivalence relationships between each temporal operator and its evaluation (Fig. 1 (E6) shows these relations exemplarily for *Always* and *Until*). This allows specifying temporal constraints in a convenient way and facilitates their translation to code.

$$Time == x : \mathbb{N}$$
$$Model == Time \rightarrow \mathbb{F}\ Formula$$

E1:
$$\begin{array}{|l} EvalMT\_ : \mathbb{P}(Formula \times Model \times Time) \\ \hline \forall p : DSpredicate;\ m : Model;\ t : Time \\ \bullet\ EvalMT(p, m, t) \Leftrightarrow p \in m(t) \end{array}$$

E2:
$$\begin{array}{|l} \forall f : Formula;\ m : Model;\ t : Time \\ \bullet\ EvalMT((\Box f), m, t) \\ \Leftrightarrow (\forall t1 : Time \mid t1 \geq t \bullet EvalMT(f, m, t1)) \end{array}$$

E3:
$$\begin{array}{|l} \forall f1, f2 : Formula;\ m : Model;\ t : Time \\ \bullet\ EvalMT((\mathcal{U}\ (f1, f2)), m, t) \\ \Leftrightarrow (\forall t1 : Time \bullet \exists t2 : Time \mid t2 \geq t1 \geq t \\ \bullet\ EvalMT(f1, m, t1) \land EvalMT(f2, m, t2)) \\ \ldots \end{array}$$

E4:
$$\begin{array}{|l} EvalM\_ : \mathbb{P}(Formula \times Model) \\ \hline \forall f : Formula;\ m : Model \bullet EvalM(f, m) \\ \quad \Leftrightarrow (\forall t : Time \bullet EvalMT(f, m, t)) \end{array}$$

E5:
$$\begin{array}{|l} Eval\_ : \mathbb{P}\ Formula \\ \hline \forall f : Formula \bullet Eval(f) \Leftrightarrow (\forall m : Model \bullet EvalM(f, m)) \end{array}$$

E6:
$$\begin{array}{|l} \forall f : Formula \bullet Eval(\Box f) \Leftrightarrow \Box f \\ \forall f, g : Formula \bullet Eval(\mathcal{U}\ (f, g)) \Leftrightarrow f\ \mathcal{U}\ g \\ \ldots \end{array}$$

Fig. 1. Evaluation of *Temporal$\mathcal{Z}$* Formulas

2.4 AOP and ALPHA

Aspect-oriented programming (AOP) [13] allows to modularize *crosscutting concerns*. The enforcement of security policies is an example of such concerns. Important AOP concepts are *pointcut*, *join point model*, and *advice*. *Pointcuts* are predicates over program execution actions – called *join points*. That is, a pointcut defines a set of join points related by some property; a pointcut is said to be *triggered* or to *match* at a join point, if the join point is in that set. An *advice* is a piece of code associated with a pointcut – it is executed whenever the pointcut is triggered, thus implementing crosscutting functionality. There are three types of advice, *before*, *after*, and *around*, relating the execution of advice to that of the action that triggered the pointcut the advice is associated with.

ALPHA[1] [7] belongs to the family of AO languages that use logic queries (a subset of valid Prolog queries) as pointcuts to reason about related actions in the program execution. ALPHA has a rich join point model: its pointcut queries are run against several models of the program execution including the static program structure (AST), the complete history of the execution , and the object store at each moment in execution.

The availability of the execution trace is one of the features that distinguishes ALPHA from other AO languages. The execution trace can be thought of as a list containing *representations* of all join points that occurred in the execution of the program so far. In ALPHA, this list structure is only implicit. Instead, all join point representations carry a time stamp which corresponds to their position in that list. There are different types of join points corresponding to different types of execution events of an object-oriented program, such as method calls, or field accesses. For each type of join point, there is a relation which has a time stamp as its first argument, the join point specific information as further arguments. Method calls, e.g., are stored in the database as pairs of `calls/5`[2] and `endCall/3` facts denoting the beginning resp. the end of a method call. For example, `calls(ID, ExpID, Receiver, Method, Arg)` describes a method call execution action that occurred at time stamp `ID`, the method `Method` is called on the receiver `Receiver` with `Arg` as the input argument; `ExpID` identifies the lexical position of the expression that caused the action. The unary relation `now/1` has – at every point in the execution – only one fact in the database that contains the current time stamp as its argument.

The following properties of ALPHA prove to be useful in the context of encoding stateful access control policies into aspects: *First*, the availability of the execution trace gives access to all relevant execution actions and associated data in a natural way. Furthermore, the ability to add or remove facts during runtime allows a flexible binding between the data objects and the elements in our domain. *Second*, the pointcut language allows to *precisely* and *declaratively* specify temporal relations between actions and relations between objects that are part of or related to these actions. This avoids the need for manually implementing the bookkeeping logic for relating this data. *Third,* ALPHA allows to define new predicates/relations, which enables building up an abstraction of predicates specific for our domain – access control policies.

[1] http://www.st.informatik.tu-darmstadt.de/pages/projects/alpha

[2] |`/n`| specifies the arity of the relation.

3 Formal Specification

In this section, we apply *Temporal$\mathcal{Z}$* in the specification of RBAC models and stateful access control policies exemplified by SoD. We extend the syntax and semantics definition that was presented in Section 2 with domain-specific security predicates. The syntax definition of these predicates is shown in Fig. 2 whereas their semantics is defined using the evaluation function *EvalMT* as shown in Fig. 3.

The predicate *include(u,r)* (Fig. 3, E7) tells whether *u* is one of the users of the RBAC system assigned to the role *r*. The predicate *active(u,r)* (Fig. 3, E8), verifies that *u* has activated the role *r*, i.e., the user should be already in the role *r* and should have connected to that role within a session s. Fig. 3, E9 and E10, define the semantics of *execOp*, resp. *execOpObj*. The predicate *execOp(u,op)* tells whether the user *u* has already executed the operation *op* by verifying whether *u* has activated a role that includes *op*.

$$\begin{aligned}
SDPredicate ::= {} & include\langle\langle USER \times ROLE\rangle\rangle \mid active\langle\langle USER \times ROLE\rangle\rangle \\
& \mid execOp\langle\langle USER \times ROLE \times OPERATION\rangle\rangle \\
& \mid execOpObj\langle\langle USER \times ROLE \times OPERATION \times OBJECT\rangle\rangle \\
& \mid execSeqOp\langle\langle USER \times \mathrm{seq}\ OPERATION\rangle\rangle \\
& \mid execSeqOpObj\langle\langle USER \times \mathrm{seq}\ OPERATION \times OBJECT\rangle\rangle
\end{aligned}$$

Fig. 2. Syntax of Security Predicates

E7:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ r : ROLE \bullet EvalMT((include(u,r)), m, t) \\
& \Leftrightarrow (\forall\, system : RBAC \bullet u \in system.users \wedge r \in system.roles \wedge u \in system.RoleUser\llparenthesis\ \{r\}\ \rrparenthesis)
\end{aligned}$$

E8:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ r : ROLE \bullet \exists\, s : SESSION \bullet EvalMT((active(u,r)), m, t) \\
& \Leftrightarrow (\forall\, system : RBAC \bullet EvalMT((include(u,r)), m, t) \wedge s \in system.sessions \\
& \wedge s \in system.UserSession\llparenthesis\ \{u\}\ \rrparenthesis \wedge s \in system.RoleSession\llparenthesis\ \{r\}\ \rrparenthesis)
\end{aligned}$$

E9:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ r : ROLE;\ op : OPERATION \bullet EvalMT((execOp(u,r,op)), m, t) \\
& \Leftrightarrow (\forall\, system : RBAC \bullet EvalMT((active(u,r)), m, t) \\
& \wedge op \in system.operations \wedge op \in system.RoleOperation\llparenthesis\ \{r\}\ \rrparenthesis)
\end{aligned}$$

E10:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ r : ROLE;\ op : OPERATION;\ obj : OBJECT \\
& \bullet EvalMT((execOpObj(u,r,op,obj)), m, t) \\
& \Leftrightarrow (\forall\, system : RBAC \bullet EvalMT((execOp(u,r,op)), m, t) \\
& \wedge obj \in system.objects \wedge obj \in system.OperationObject\llparenthesis\ \{op\}\ \rrparenthesis)
\end{aligned}$$

E11:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ Sop : \mathrm{seq}\ OPERATION \\
& \bullet EvalMT((execSeqOp(u, Sop)), m, t) \\
& \Leftrightarrow (\forall\, i : \mathbb{N} \mid 1 \leq i \leq \#Sop \bullet (\exists\, r : ROLE \bullet EvalMT((\Diamond(execOp(u,r,(Sop(i))))), m, t))) \\
& \wedge (\forall\, i : \mathbb{N} \mid 1 < i \leq \#Sop \bullet ((\exists\, r1 : ROLE \bullet EvalMT((execOp(u,r1,(Sop(i)))), m, t)) \\
& \Rightarrow (\exists\, r2 : ROLE \bullet EvalMT((\Diamond(execOp(u,r2,(Sop(i-1))))), m, t))))
\end{aligned}$$

E12:

$$\begin{aligned}
& \forall\, m : Model;\ t : Time \bullet \forall\, u : USER;\ Sop : \mathrm{seq}\ OPERATION;\ obj : OBJECT \\
& \bullet EvalMT((execSeqOpObj(u, Sop, obj)), m, t) \\
& \Leftrightarrow (\forall\, i : \mathbb{N} \mid 1 \leq i \leq \#Sop \bullet (\exists\, r : ROLE \bullet EvalMT((\Diamond(execOpObj(u,r,(Sop(i)),obj))), m, t))) \\
& \wedge (\forall\, i : \mathbb{N} \mid 1 < i \leq \#Sop \bullet ((\exists\, r1 : ROLE \bullet EvalMT((execOpObj(u,r1,(Sop(i)),obj)), m, t)) \\
& \Rightarrow (\exists\, r2 : ROLE \bullet EvalMT((\Diamond(execOpObj(u,r2,(Sop(i-1)),obj))), m, t))))
\end{aligned}$$

Fig. 3. Semantics of Security Predicates

The predicate *execOpObj* refines *execOp* by checking whether *u* has already executed *op* with *obj* as parameter. The predicates *execSeqOp* and *execSeqOpObj* (Fig. 3, E11 and E12), introduce the concept of an ordered operation sequence. The predicate *execSeqOp(u,sop)* tells whether the user *u* executes the sequence of operations *sop* in the order defined by the sequence. The predicate *execSeqOpObj(u,sop,obj)* refines *execSeqOp(u,sop)* to express that all operations are executed on the same object *obj*.

3.1 RBAC and Administrative Operations

The specification of the base RBAC system in *Temporal$\mathcal{Z}$* is shown in the Z Schema *RBAC* in Fig. 4. The upper part defines sets of roles, operations, and objects, as well as relations between them. Each role has a set of operations associated with it, defined by *RoleOperation* [D1]. A user should have the role, as defined by the relation *RoleUser* [D2], to be allowed to execute the operations of that role. A session is a mapping of a user to a subset of his roles as defined by ([D3], [D4], [C2], and [C3]).

The relation *OperationObject* [D5] maps operations to objects they operate on. The lower part of Fig. 4 defines constraints on the range and domains of the relations defined in the upper part [C1] and RBAC constraints ([C2], [C3]). Further, constraints on RBAC concepts and their relationships are specified in terms of first-order predicates.

For example, the *SystemConstraints* schema in Fig. 5 defines constraints on LAP that restrict the number of (a) roles [C4], (b) users assigned to the role *Teller* [C5], and (c) operations a role can have [C6]. We can also define other types of constraints, e.g., an operation cannot be included in more than one role [C7].

Each administrative operation, e.g., assigning a role to a user, is specified by an operation schema, which defines its input parameters and its pre- and post-conditions. For example, in LAP, we can specify that an employee cannot be a member in any role unless he had activated the role *Teller* some time in the past [C8], as shown in Fig. 6.

RBAC

$roles : \mathbb{F}\, ROLE;\ users : \mathbb{F}\, USER;\ sessions : \mathbb{P}\, SESSION$

$operations : \mathbb{F}\, OPERATION;\ objects : \mathbb{F}\, OBJECT$

$RoleOperation : ROLE \leftrightarrow OPERATION$ [D1]

$RoleUser : ROLE \leftrightarrow USER$ [D2]

$RoleSession : ROLE \leftrightarrow SESSION$ [D3]

$UserSession : USER \leftrightarrow SESSION$ [D4]

$OperationObject : OPERATION \leftrightarrow OBJECT$ [D5]

$\mathrm{dom}\ RoleOperation \subseteq roles \land \mathrm{ran}\ RoleOperation \subseteq operations$ [C1]

$\forall u : users, r : roles \bullet (r, u) \in RoleUser$

$\quad \Leftrightarrow \exists s : sessions \bullet (r, s) \in RoleSession \land (u, s) \in UserSession$ [C2]

$\forall u1, u2 : users \mid u1 \neq u2 \bullet \exists s : sessions$ [C3]

$\quad \bullet (u1, s) \notin UserSession \lor (u2, s) \notin UserSession$

...

Fig. 4. RBAC Model

$$\begin{array}{|l}
\textit{SystemConstraints} \\
\hline
\textit{RBAC} \\
\hline
\#\textit{roles} < 10 \quad \text{[C4]} \\
\#(\textit{RoleUser}(\!|\ \{\textit{Teller}\}\ |\!)) \leq 5 \quad \text{[C5]} \\
\forall r : \textit{roles} \bullet \#(\textit{RoleOperation}(\!|\ \{r\}\ |\!)) \leq 7 \text{[C6]} \\
\forall p : \textit{operations};\ r1, r2 : \textit{roles} \mid r1 \neq r2 \\
\bullet\ p \notin \textit{RoleOperation}(r1) \\
\vee\ p \notin \textit{RoleOperation}(r2) \quad \text{[C7]} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{assignUserToRole} \\
\hline
\Delta \textit{RBAC} \\
u? : \textit{USER};\ r? : \textit{ROLE} \\
\hline
u? \notin \textit{users} \\
\Diamond(\textit{include}(u?, \textit{Teller})) \quad \text{[C8]} \\
\textit{users}' = \textit{users} \cup \{u?\} \\
\textit{RoleUser}' = \textit{RoleUser} \cup \{r?, u?\} \\
\ldots \\
\hline
\end{array}$$

Fig. 5. System Constraints

Fig. 6. Prerequisite Constraints

3.2 SoD Properties

In the following, we present examples of formal specification of SoD for LAP (cf. Section 2) using *Temporal$\mathcal{Z}$*. Using the predicate *include* (Fig. 3, E7), the specification *StaticSoD* states that an employee in LAP cannot be a *Manager* and a *Supervisor* at the same time.

$$\begin{array}{|l}
\textit{StaticSoD} \\
\hline
\textit{SystemConstraints} \\
\hline
\textit{ExclusiveRole}(\textit{Supervisor}, \textit{Manager}) \Leftrightarrow \forall u : \textit{USER} \\
\qquad \bullet \neg\ \textit{include}(u, \textit{Supervisor}) \vee \neg\ \textit{include}(u, \textit{Manager}) \\
\hline
\end{array}$$

The specification *DynamicSoD* uses the predicate *active* (Fig. 3, E8) to state that an employee can be a *Teller* and a *FinancialClerk*, but not simultaneously.

$$\begin{array}{|l}
\textit{DynamicSoD} \\
\hline
\textit{SystemConstraints} \\
\hline
\forall u : \textit{USER} \bullet \Box(\neg\ (\textit{active}(u, \textit{teller}) \wedge \textit{active}(u, \textit{financialClerk}))) \\
\hline
\end{array}$$

Using the predicates *execOp* and *execOpObj* (Fig. 3, E9 and E10), the specification *ObjectBasedSoD* states that a user who has executed *checkInternalRating* in the *FinancialClerk* role can never in the future execute *verifyRating* in the *Supervisor* role for the same customer *cst*.

$$\begin{array}{|l}
\textit{ObjectBasedSoD} \\
\hline
\textit{SystemConstraints} \\
\hline
\forall u : \textit{USER};\ \textit{cst} : \textit{Customer} \bullet \Box((\textit{execOpObj}(u, \textit{financialClerk}, \textit{checkInternalRating}, \textit{cst})) \\
\Rightarrow (\neg\ (\Diamond(\textit{execOpObj}(u, \textit{supervisor}, \textit{verifyRating}, \textit{cst}))))) \\
\hline
\end{array}$$

The specification *OperationalSoD* uses *execSeqOp* (Fig. 3, E11) to disallow an employee from executing the critical sequence (*enterApplicationData*, *verifyRating*, *verifyTransfer*). The specification *OperationalObjectBasedSoD* uses *execSeqOpObj* (Fig. 3, E12) to disallow executing the operations of the rating subprocess (*checkInternalRating*, *checkExternalRating*, *verifyRating*) on the same customer application *ctr*.

$$\begin{array}{|l}
\textit{OperationalSoD} \\
\hline
\textit{SystemConstraints} \\
\textit{SeqOp} : \mathrm{seq}\ \textit{OPERATION} \\
\hline
\forall u : \textit{USER} \\
\mid \textit{SeqOp} = \langle \textit{enterApplicationData}, \\
\textit{verifyRating}, \textit{verifyTransfer} \rangle \\
\qquad \bullet \Box (\neg\ (\textit{execSeqOp}(u, \textit{SeqOp}))) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{OperationalObjectBasedSoD} \\
\hline
\textit{SystemConstraints} \\
\textit{SqOO} : \mathrm{seq}\ \textit{OPERATION} \\
\hline
\forall u : \textit{USER};\ \textit{ctr} : \textit{Customer} \\
\mid \textit{SqOO} = \langle \textit{checkInternalRating}, \\
\quad \textit{checkExternalRating}, \textit{verifyRating} \rangle \\
\qquad \bullet \Box (\neg\ (\textit{execSeqOpObj}(u, \textit{SqOO}, \textit{ctr}))) \\
\hline
\end{array}$$

3.3 Ensuring the Consistency of the Specification

To ensure consistency, we verify that the specification of the base RBAC system and its constraints does not contain any contradiction by proving with Z-EVES an initialization theorem, which states that there exists a state of the RBCA system that fulfills all constraints. One may still argue that SoD constraints can introduce other inconsistencies. However, as we follow a prohibition-based approach[3] [14], SoD constraints, which are safety properties, cannot produce any inconsistencies in the already consistent RBAC system. If a SoD constraint is evaluated to $\textit{True}$, the user is prohibited from executing the respective operation. Another reason for choosing the prohibition-based approach is that we cannot force the users to do certain actions to comply with the constraints; we can only prohibit them from executing operations that may break the constraints.

4 Aspect-Based Security Monitors

In this section, we present the mapping of the formally specified security policies into enforcement aspects. First, we present the mapping of Temporal$\mathcal{Z}$ predicates into domain-specific ALPHA predicates used in the pointcuts of enforcement aspects. Then, we outline the generation schema of aspects that enforce pre-conditions and RBAC constraints related to administrative operations and also of aspects that enforce SoD policies. We developed a prototype implementation[4] that incorporates the presented mapping and generates enforcement aspects from the formal specification and deploys them into the ALPHA runtime.

4.1 Mapping Temporal$\mathcal{Z}$ Predicates to Domain Specific ALPHA Predicates

A library of domain specific predicates has been defined in ALPHA to facilitate the process of mapping formal specifications to enforcement aspects. These predicates are used in the pointcuts of these aspects. The library includes a Prolog predicate for each temporal operator; RBAC and SoD specific predicates are also part of the library.

[3] Defined by the axioms $\neg\ \varphi$ and $\varphi \Rightarrow \neg\ \psi \ldots$

[4] The prototype is a proof-of-concept. Efficiency issues will be addressed in future work.

The temporal operator $\diamondminus$ is mapped to the Prolog predicate `before/2`. The operator $\Diamond$ is mapped to the Prolog predicate `eventually/2` which verifies that the first time stamp corresponds to an action after the action corresponding to the second time stamp. Predicates `justBefore/2` and `next/2` correspond to $\ominus$ and $\bigcirc$, respectively. The latter are restricted versions of `before/2`, resp. `eventually/2`: not only does the first time stamp correspond to an action before, resp. after, the second time stamp; there is also no action in-between. The operators $\mathcal{U}$ and $\mathcal{S}$ are mapped to `until/2` and `since/2` predicates, respectively. There is no need for a predicate corresponding to $\square$; the use of time stamp variables in Prolog predicates states that a property should hold at any time, i.e., always.

There is also a mapping of the logical conjunctives, e.g., the formal conjunction operation ($\wedge$) is mapped to the *and* operator (`,`) of Prolog. Any parameter declared in the formal specification with the quantifier for all ($\forall$) is translated to a Prolog variable.

4.2 Aspects for Administrative Operations

For each administrative operation we generate an aspect for enforcing the corresponding pre-conditions and system constraints. The pseudo-code in List. 1 shows the overall structure of such an aspect in a generic way. The pointcut (line 2) uses the ALPHA predicate `calls` to intercept calls to the administrative operation at hand[5]. If one of the pre-conditions of the administrative operation uses past temporal operators, the pointcut would have a second part (denoted by `temporalConstraintsQuery`) that uses domain specific ALPHA temporal predicates. E.g., the pointcut of the aspect enforcing the pre-condition of `assignUserToRole` in Fig. 6 would have a `temporalConstraintsQuery` part that uses the predicate `before/2`.

```
1  class AspectForAdministrativeOperation {
2   void around calls(N,_,_,administrativeOperation,_),temporalConstraintsQueries(){
3     if (zOperatorMethod(parameters, systemState)) { ... }
4     if (constraint_i()) { ... }
5     if (allConstraints) {
6          proceed();
7        updateSystemState();
8     }
9     else { ... }
10  }
11  public boolean constraint_i() { ... }
12  public void updateSystemState() { ... }
13 }
```

Listing 1. Template of Aspect Generation

The advice of this aspect (lines 3–10) has an *around* advice, that is executed instead of the method calls to `administrativeOperation` captured by the pointcut. The advice verifies the Z predicates specified for the administrative operation and a conditional statement is generated for each Z predicate (lines 3 and 4). If all Z predicates are fulfilled (`allConstraints` is true), the administrative operation will be executed

[5] *administrativeOperation* is a placeholder for a concrete administrative operation.

using `proceed()` in line 6 and the state of the RBAC system is updated by calling `updateSystemState` (line 7); otherwise, the aspect truncates the execution.

To allow the mapping of the Z parts of the formal specifications to Java, we have implemented a meta-model of Z as a Java library: Each operator, mathematical object, etc., has a corresponding class and/or methods in the library. Formal constraints without quantification are checked by directly calling the respective method in the Java Z library. The `zOperatorMethod` (lines 3) is a representative of library methods for such constraints. Formal constraints with quantification operators are mapped to helper methods, denotated by `constraint_i` (line 11).

4.3 Aspects for SoD Properties

For each formally specified SoD property, an ALPHA aspect is generated. This aspects uses an `around` advice to stop the execution if the property does not hold.

Simple Static SoD. The generation of aspects for SSoD properties maps each pair of predicates *ExclusiveRole(r1,r2)*[6] and *include(user,r1)* to the following conjunction of ALPHA predicates: `now(T), calls(T,_,_,assignUserToRole,<User,r1>), roleUser(r2,User)`. For illustration, the aspect generated from the specification *StaticSoD* in Sec. 3.2 is shown in List. 2. The pointcut matches in two cases (note the use of the Prolog operator or). First (lines 3–4), `assignUserToRole` is called to assign an arbitrary user (the parameter `User`) the role `manager`, while that user has the role `supervisor` (see the query `roleUser` in line 4). Second (lines 5–6), `assignUserToRole` is called to assign an arbitrary user the role `supervisor`, while that user is a member of the role `manager` (see the query `roleUser` in line 6). If any of these execution states is reached, the SSoD property is violated; hence, the around advice displays an error and stops the execution (no call to `proceed`).

```
1  class SimpleStaticSoD {
2   void around
3      now(T), calls(T,_,_,assignUserToRole,<User,manager>),
4      roleUser(supervisor,User);
5      now(T), calls(T,_,_,assignUserToRole,<User,supervisor>),
6      roleUser(manager,User).
7   {
8     System.out.print("Violation of a SSoD property");
9    }
10 }
```

Listing 2. Example Aspect for SSoD

Simple Dynamic SoD. The generation of aspects for SDSoD properties maps the formal predicate $\mathit{active}(u, r)$ to the following conjunction of ALPHA predicates:

`calls(T,_,_,Op,_),callingUser(T,u), roleOperation(r,Op), roleUser(r,u).`

[6] The predicate *ExclusiveRole* is used only in Static SoD to ease the translation of the SoD propety to Alpha code.

To verify a SDSoD property for two exclusive roles `r1`, `r2` for any user, the pointcut of the generated aspect should be triggered whenever there is a pair of time stamps (`T1`,`T2`), such that operations `Op1` and `Op2` are called by `User` at `T1`, resp. `T2`, whereby `Op1` and `Op2` are associated with roles `r1` and `r2` resp. If such a pair is found, and `T1` or `T2` denotes the current time stamp `(now(T1);now(T2))`[7], a violation of the SDSoD constraint is found. For illustration, List. 3 shows the aspect generated from the specification *DynamicSoD* in Sec. 3.2, which states that any user that is a member of the two exclusive roles *Teller* and *FinancialClerk* cannot activate them at the same time[8].

```
class SimpleDynamicSoD {
 void around
%One of timestamps is the current time
   (now(T1);now(T2)),
%Translate from the predicate active(User, teller)
    calls(T1,_,_,Op1,_), callingUser(T1,User),
    roleOperation(teller,Op1),
    roleUser(teller,User),
%Translate from active(User,financialclerk)
    calls(T2,_,_,Op2,_), callingUser(T2,User),
    roleOperation(financialClerk,Op2),
    roleUser(financialClerk,User).
  {
     System.out.print("Violation of a SDSoD property");
  }
}
```

Listing 3. Example Aspect for SDSoD

Object-Based SoD. The specifications of ObjSoD properties use the formal predicate $\mathit{execOpObj}(u, r, \mathit{op}, \mathit{obj})$. In Sec. 3.2, this predicate is used twice in *ObjectBased-SoD* (with concrete bindings of *op* to *checkInternalRating*, resp. *verifyRating* and of *r* to *financialClerk*, resp. *supervisor*). The evaluation of $\mathit{execOpObj}(u, r, \mathit{op}, \mathit{obj})$ to true (E10 in Sec. 3) requires that $\mathit{execOp}(u, r, \mathit{op})$ evaluates to true for any *obj* in the set of objects on which *op* operates ($\mathit{obj} \in \mathit{OperationObject}(\mathit{op})$); The predicate $\mathit{execOp}(u, r, \mathit{op})$, in turn, is true if $\mathit{active}(u, r)$ is true and *op* is one of the operations assigned to *r* (E9 in Sec. 3). Following this reduction of the evaluation of $\mathit{execOpObj}$ to active, the two uses of $\mathit{execOpObj}$ in *ObjectBasedSoD* are mapped to the conjunction of ALPHA predicates shown e.g., in lines 3–5 and 7–9 of List. 4. The condition $\mathit{obj} \in \mathit{OperationObject}(\mathit{op})$ used in $\mathit{execOpObj}$ is mapped to having the `calls` predicates in the pointcut explicitly relate *op* and obj.

This listing shows the aspect generated for the $\mathit{ObjectBasedSoD}$ property specified in Sec. 3.2: a user who has executed $\mathit{checkInternalRating}$ as a $\mathit{FinancialClerk}$ can never in the future execute $\mathit{verifyRating}$ for the same customer as a $\mathit{Supervisor}$. The pointcut is triggered whenever a user in a supervisor role is about to verify the rating (at present time `T2`) and the same user has checked the internal rating of the same `Customer` as a financial clerk at `T1` in the past (`eventually(T2,T1)`); in this case, the advice signals a violation and truncates the execution (no *proceed*).

[7] If none of them denotes the current time, the violation must have been found earlier.

[8] With the Prolog variable `User` in List. 3 we state that the property should hold for all users.

```
class ObjDynamicSoD {
 void around
  calls(T1,_,_,checkInternalRating,Customer),
  callingUser(T1,User), roleUser(financialClerk,User),
  roleOperation(financialClerk,checkInternalRating),

  now(T2), calls(T2,_,_,verifyRating,Customer),
  callingUser(T2,User), roleUser(supervisor,User),
  roleOperation(supervisor,verifyRating),

  eventually(T2,T1). %checkInternalRating is called before verifyRating
 {
  System.out.print("Violation of a ObjDSoD property");
 }}
```

Listing 4. Example Aspect for ObjDSoD

Unlike the conjunction generated for *active* in List. 3, which uses variables `Op1` and `Op2` to express that the corresponding SDSoD property holds for **any** pair `Op1`, `Op2`, concrete operation names are used in the conjunction sequences in List. 4 to reflect the fact that *execOpObj* in *ObjectBasedSoD* refers to concrete operations. Also, in addition to the `User` variable, the `Customer` variable is used in the `calls` predicate in List. 4 to express that the property should hold for **any** customer object.

Operational SoD. The generated aspect has a pointcut that verifies whether the user is about to execute the last of a critical sequence of operations. E.g., given the sequence (`enterApplicationData`, `verifyRating`, `verifyTransfer`) passed to the formal predicate *execSeqOp* in *OperationalSoD* in Sec. 3.2, the pointcut[9] shown in List. 5 is generated. It is triggered on a call to the last operation of the sequence and checks if the user has called the other operations of that sequence in the respective order. In this case, a violation is signaled and the last operation is not executed.

```
class OperatSoD {
 void around
  calls(T1,_,_,enterApplicationData,_),
  callingUser(T1,User),
  verifyAccess(User,enterApplicationData),
  calls(T2,_,_,verifyRating,_),
  callingUser(T2,User),
  verifyAccess(User,verifyRating), eventually(T2,T1),
  now(T3), calls(T3,_,_,verifyTransfer,_),
  callingUser(T3,User),
  verifyAccess(User,verifyTransfer), eventually(T3,T2)
 {
 System.out.print("Violation of a OpDSoD property");
 }}
```

Listing 5. Example Aspect for OpDSoD

Operational Object-Based SoD. The generation of aspects for OpObjDSoD properties is similar to that for operational SoDs except that the last parameter of the generated `calls` predicates is a variable rather than unbound. This is to reflect that object-based

[9] `verifyAccess` checks if a user is allowed to execute an operation independent of his role.

operational SoD properties are specified for **any** $obj \in OBJECT$. By using the same variable in all generated `calls` predicates, Prolog unification ensures that the pointcut is triggered only when the critical sequence is executed on the same object.

5 Related Work

In [15], we presented an approach for specifying and enforcing architectural constraints in distributed object-oriented applications. In that work, we used Z and Petri nets for formal specification and we generated AspectJ aspects for enforcement. In [16], we presented an extension of the approach shown in the current paper supporting delegation models and their characteristics such as permanence and monotonicity.

In works like [17,18], informal methods such as UML are used to specify RBAC systems. E.g., Song et al. [17] propose an aspect-oriented modeling approach based on UML and OCL, which addresses only basic RBAC system constraints but not temporal constraints. A major advantage of our approach over works that use informal languages is that *TemporalZ* is formal, i.e., unambiguous, precise, and verifiable.

Others used several formal methods to specify RBAC systems and their constraints such as temporal logic [11], automata [2], Z, Petri Nets, and process algebra. In [19], Z is extended with temporal logic operators. However, the temporal operators are used only to reason about sequences of Z schema states, where two consecutive states correspond to "before and after" states of an already specified Z schema operation. In our work, we extend the Z mathematical notation with temporal operators. Thus, temporal operators can be applied to any Z formula, which is not necessarily about Z states. Moreover, the semantics that we provide for temporal formulas within the framework of Z, enables us to use the existing Z tools for syntax checking and theorem proving.

None of the works mentioned so far addresses runtime enforcement, unlike our approach. Next, we discuss works on enforcing security policies.

Works such as [9,10] address the translation of access control properties to enforcement code. In [9], a UML-based security modeling language is presented and transformation functions are defined to map such models to particular technologies for access control in application platforms.Temporal constraints are not supported because of OCL and the generated code is not modular and incomplete (only skeletons).

In [20], an approach to enforce trace properties by program transformation is presented. A transformation takes the original program and the properties in input and generates an equivalent program that enforces the specified properties. That approach is based on graphs and automata transformation. Like our approach, the properties to be enforced are specified formally and declaratively in this work. However, the generated programs are not modular since the code enforcing the formal properties is mixed with the functional code.

Stolz et al. [21] propose a runtime verification tool for Java based on LTL and AOP. LTL over joinpoints is defined to specify the properties to verify at runtime. AspectJ is used as the target language for translating LTL specifications. The lack of a rich joinpoint model and the restricted expressiveness of AspectJ's pointcut language lead to complex aspects, since logic for simulating the automata corresponding to LTL formulas has to be encoded in the aspect.

In [22,23] AO techniques are used to model and enforce access control policies. As a representative, we discuss the work presented in [23]. This approach only supports simple access control constraints; SoD properties and administrative operations are not supported. In addition, compared to ALPHA, the aspect languages used for implementing the policies are based on less powerful pointcut languages. This leads to less modular aspects especially when temporal constraints are to be enforced [7].

6 Conclusion and Future Work

In this paper, we presented an approach that covers the formal specification and enforcement of RBAC models and their constraints including separation of duties. We also described the translation of these specifications into the AOP language ALPHA and developed a supporting tool. The use of aspects makes the policy enforcement code well-modularized and separated from the business logic. The automatic generation of the enforcement aspects brings major benefits. First, programmers do not have to do the translation manually. Second, as the translation is automated no human errors may be introduced. Third, the enforcement code is in sync with the specification as this code will be simply regenerated whenever the specification is changed. As future work, we will investigate the extension of our approach to support edit automata [2] and we intend to cover other access control policies such as Chinese wall.

References

1. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3 (2000)
2. Ligatti, J., Bauer, L., Walker, D.W.: Enforcing non-safety security policies with program monitors. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 355–373. Springer, Heidelberg (2005)
3. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST model for Role-based Access Control: Towards a Unified Standard. In: Proc. of RBAC. ACM, New York (2000)
4. Gligor, V.D., Gavrila, S.I., Ferraiolo, D.F.: On the formal definition of separation-of-duty policies and their composition. In: Proc. of Symposium on Security and Privacy. IEEE, Los Alamitos (1998)
5. Spivey, M.: The Z notation: a reference manual. Prentice Hall International Ltd., Englewood Cliffs (1992)
6. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer, Heidelberg (1992)
7. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)
8. Meisels, I., Saaltink, M.: The Z/EVES reference manual (v 1.5) (1997)
9. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. 15 (2006)
10. Neumann, G., Strembeck, M.: An approach to engineer and enforce context constraints in an RBAC environment. In: Proc. of SACMAT. ACM Press, New York (2003)
11. Schaad, A., Lotz, V., Sohr, K.: A model-checking approach to analysing organisational controls in a loan origination process. In: Proc. of SACMAT. ACM, New York (2006)
12. Regayeg, A., Kacem, A.H., Jmaiel, M.: Towards a formal methodology for designing multi-agent applications. In: Eymann, T., Klügl, F., Lamersdorf, W., Klusch, M., Huhns, M.N. (eds.) MATES 2005. LNCS, vol. 3550, pp. 153–164. Springer, Heidelberg (2005)

13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
14. Ahn, G.J.: Specification and Classification of Role-based Authorization Policies. In: Proc. of WETICE. IEEE, Los Alamitos (2003)
15. Kallel, S., Charfi, A., Mezini, M., Jmaiel, M.: Combining formal methods and aspects for specifying and enforcing architectural invariants. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 211–230. Springer, Heidelberg (2007)
16. Kallel, S., Charfi, A., Mezini, M., Jmaiel, M.: Aspect-based enforcement of formal delegation policies. In: Proc. of CRISIS. IEEE, Los Alamitos (2008)
17. Song, E., Reddy, R., France, R., Ray, I., Georg, G., Alexander, R.: Verifiable composition of access control and application features. In: Proc. of SACMAT. ACM, New York (2005)
18. Ray, I., Li, N., France, R., Kim, D.K.: Using UML to visualize role-based access control constraints. In: Proc. of SACMAT. ACM Press, New York (2004)
19. Duke, R., Smith, G.: Temporal logic and Z specifications. Australian Computer Journal 21, 62–66 (1989)
20. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proc. of POPL. ACM Press, New York (2000)
21. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Proc. of 5th Workshop on Runtime Verification. ENTCS (2005)
22. Chen, K., Lin, C.-W.: An aspect-oriented approach to declarative access control for web applications. In: Zhou, X., Li, J., Shen, H.T., Kitsuregawa, M., Zhang, Y. (eds.) APWeb 2006. LNCS, vol. 3841, pp. 176–188. Springer, Heidelberg (2006)
23. Verhanneman, T., Piessens, F., Win, B.D., et al.: Implementing a modular access control service to support application-specific policies in caesarJ. In: Proc. of AOMD. ACM Press, New York (2005)

Idea: Trusted Emergency Management

Timothy E. Levin[1], Cynthia E. Irvine[1], Terry V. Benzel[2], Thuy D. Nguyen[1], Paul C. Clark[1], and Ganesha Bhaskara[2]

[1] Naval Postgraduate School, Monterey, CA 93943, USA
{levin,irvine,tdnguyen,pcclark}@nps.edu
[2] USC Information Sciences Institute, Marina Del Rey, CA 90292
{tbenzel,bhaskara}@isi.edu

Abstract. Through first-responder access to sensitive information for which they have not been pre-vetted, lives and property can be saved. We describe enhancements to a trusted emergency information management (EIM) system that securely allows for extraordinary access to sensitive information during a crisis. A major component of the architecture is the end-user device, the security of which is enhanced with processor-level encryption of memory. This paper introduces an approach to more efficiently use the processor-encryption feature for secure data storage, as well as ISA instructions for the management of emergency state.

1 Introduction

During crises, first-responders can more effectively save lives and property if given access to certain restricted information relating to physical security (e.g., blueprints); individual privacy (e.g., medical records); and classified information (e.g., *continuity of government plans*), etc. However, the large number of potential first responders makes it infeasible to pre-screen them all, e.g. via national security clearances. Yet, if sensitive information is made available but not protected adequately, extensive damage could result. We describe a policy and operational model for emergency information management (EIM), and an architectural foundation for the realization of EIM in a modern IT environment where transient trust is possible [1]. We introduce a technique for leveraging processor-level encryption of data in memory to protect data in persistent storage; and we describe a set of new processor features for secure distributed state management, and describe how they are used to in the EIM context [2]. The target platform for this research is a dual-use hand-held computer, the *E-device.*

2 Model for Emergency Information Management (EIM)

In the emergency-response milieu we consider the following roles, each of which has a direct stake in effectiveness of the E-device. *First responders*, are, e.g., members of medical, police, fire, transportation, communication, construction, maintenance and other organizations, who may be called on at the scene of a disaster. *The Authority* is an organization that coordinates emergency response in a given context, such as the Department of Homeland Security, non-governmental organization, or a selected enterprise department. The *Third Party* is one or more data providers that supply emergency information. As a simplification for our initial work, we consider Third Parties to be mutually trusting and are represented simply as a single Third Party. *Emergency information* is information designated to be available to emergency first responders, which they may not have been vetted or cleared to see.

Emergency information is *owned* by third parties, who may not wish it to be generally distributed or shared, or may be constrained from doing so (currently) due to regulatory hurdles.The E-device is initialized with certain emergency information, which can be updated in the field. The collection of E-devices, and the Authority's and the Third Parties' trusted systems, comprise the Emergency Network. For simplicity, we characterize the emergency state of the Emergency Network as either *on* or *off*. The Authority manages the emergency state, and communicates state changes to E-devices and Third Parties.

3 Transient Trust Policy and Stakeholder Trust Model

We assume a strict policy regarding the authorized accesses of users to data objects that is consistent across the emergency network. The Authority defines an emergency policy that allows additional, *extraordinary accesses* by end users to emergency information, which may occur only (transiently) during an emergency. While such accesses do not violate the security policy, per se, they are beyond the pale of usual MAC and DAC controls [3]. Together the strict policy and the emergency policy describe the complete emergency network security policy. The temporal constraint on extraordinary accesses is a key element of this policy, as it reduces the window of opportunity for inappropriate use of information resulting from adverse security events outside of the control of the trusted computing base, e.g., inadvertent password disclosure, or malicious insider behavior. The natural variability of mobile device connectivity means that the emergency network stakeholders must agree on a revocation policy should an E-device lose connectivity during an emergency, e.g., an upper bound on delayed revocation [4].

Operational agreements define the information sharing policies and levels of protection to be afforded to shared information, including the level of assurance provided by the E-device. The agreements may provide for Third Party *trusted* applications to be hosted in the E-device's Trusted Partition (see below). The Third Parties rely on the Authority to declare the start and end of the emergencies, and to correctly configure the E-devices, including communication keys.

4 Security Architecture for EIM

The foundation of our solution is the SecureCore security architecture [5]. Its Trusted Management Layer (TML) comprises the Least Privilege Separation Kernel [6][7] and the Trusted Services Layer (TSL) layer [8]. The LPSK partitions the platform's physical resources and controls interactions between the partitions in the form of a lattice-flow policy; controlling access to data while it is in transit; at rest (on disk); and when processed (in both on-chip and on-board memory). The TSL layer virtualizes certain LPSK resources for the use of applications (e.g., commercial OSs), and associates MLS security labels with the kernel's exported resources.

Users have interactive sessions with one partition at a time. The Trusted Partition provides high integrity services, such as logon-on and partition selection provided by the Trusted Path Application (TPA), and other services that may be provided by Third Parties, such as for the secure sealing of documents. Normal Partitions host a commercial OS and typical office applications, providing familiar and functionally rich user interfaces. Emergency information is stored and accessed in the Emergency Partition. The TML ensures that the only way a user can ever access emergency information is through an Emergency Partition during an emergency. When an emergency ends, the TML renders the Emergency Partition inaccessible to the E-device user; and it can transmit updated emergency data to the Third Party. Now to our hardware protection scheme.

We utilize several hardware cryptographic primitives like those postulated for the Secret Protected (SP) processor [9][1]. The processor provides a special "crypto mode", in which access to both the cryptographic primitives and several non-volatile processor registers (DRK, SRK and CEM) are available to software. Three crypto-transform functions are provided: `sp_derive` hashes two words with the DRK; `secure_store` marks a cache line for transformation (i.e., hash and encryption with the DRK) upon eviction from the processor cache; and `secure_load` decrypts memory as it is loaded into the processor cache and validates its hash. To this, we introduce the *code integrity check* (CIC) processor mode, which supports privileged *supervisor* program protection: at compile time, the TML code is hashed with the DRK, and when this code is executed the inline hash values are validated, and execution halts if the validation fails. As a result of using CIC, TML code can only be executed on the intended device, and any unintended changes to TML code are immediately detected, thus adding to the TML's high assurance self-protection mechanisms.

5 Secure Storage Solution

The `sp_secure_store` instruction is intended to be used for transient memory encryption. Yet, we also need to efficiently encrypt data as it is transferred between memory and the disk, and the built-in hardware encryption function appears ideal for this purpose. However, the `secure_load` instruction decrypts data as it is loaded into the processor. The solution presented here addresses this by marking data for encryption with `secure_load`, pushing it out of cache (e.g., with *clflush* with the x86 ISA), and then using device DMA to move it, encrypted, onto the disk. Alternatively, if it is desired to use programmed I/O

[1] While we characterize these features as part of the hardware instruction set, some may be suitably instantiated through an off-chip device [10].

to write to disk, data previously marked with `secure_store` can be moved from memory into the processor with a normal *load* instruction, which will cause it to arrive in the processor register encrypted. On re-accessing the encrypted data from disk, it is decrypted using `secure_load`, and its integrity can be validated relative to memory-segment and disk-volume seals generated upon storage.

6 Emergency State Management Solution

The security of the Emergency Network depends on how securely the emergency state is managed. Although TML functions to receive, store, and respond to emergency status updates could be implemented in software, to make emergency management more robust, we extend the processor ISA with key *state management* primitives: two new instructions, `hw_update_state` and `hw_get_state`; a local state counter: `e_counter`; and a state bit: `e_state`, described next [5].

The Authority associates a sequential number with each emergency state change. To announce a change to the emergency state, the Authority generates a point-to-point message for each E-device and sends them over a trusted channel. The message contains the new emergency state and the corresponding state-change number, as well as the hash of this payload. Each message is also encrypted with the target E-device's DRK (secure broadcast is left for future work). When the TML receives an emergency message it submits it to the processor using `hw_update_state`. The processor validates the message hash against the payload and decrypts it with the DRK; it checks that the new counter value is greater than the previous value, to prevent message replay; and finally writes the payload state to the hardware e-state register. If the `hw_update_state` operation is successful, the TML sends an acknowledgement to the Authority and uses `hw_get_state` to retrieve the new e-state value. The TML then either announces a new emergency to the user and makes the Emergency Partition available, or terminates the existing emergency.

7 Related Work

Our work utilizes currently-proposed concepts for CPU-based cryptographic support. Although mechanisms for cryptographic support using coprocessors are available, e.g. the IBM 4758 co-processor [11] and the TCG Trusted Platform Module [10], these off-chip schemes are more vulnerable to internal attack by elements within the platform architecture.

Anderson proposed a model for patient medical information protection, which accommodates *extraordinary* access to information under certain conditions [12]. In this model, access control lists restrict access to patient records. Usually, changes to the lists require user consent; if an emergency results in an over-ride of the access list mechanism, the user is "notified." In contrast, our approach provides the ability to control when emergency overrides may occur, control the extent of emergency override, and revoke permissions to information in real time.

8 Conclusions

We have described a system for secure dissemination and control of sensitive information during crises, and two significant enhancements to emergency information management: transient-memory encryption for secure data storage, and new hardware instructions to support distributed emergency state management.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0430566 and CNS-0430598 with support from DARPA ATO. This paper does not necessarily reflect the views of the National Science Foundation or of DARPA ATO.

References

1. Irvine, C.E., Levin, T.E., Clark, P.C., Nguyen, T.D.: A security architecture for transient trust. In: Proc. of Computer Security Architecture Workshop, Fairfax, Virginia, USA. ACM, New York (2008)
2. Levin, T.E., Irvine, C.E., Benzel, T.V., Nguyen, T.D., Clark, P.C., Bhaskara, G.: Trusted emergency management. Technical Report NPS-CS-09-001, Naval Postgraduate School, Monterey, CA (Naval Postgraduate School)
3. McCollum, C.J., Messing, J.R., Notargiacomo, L.: Beyond the pale of MAC and DAC: defining new forms of access control. In: Proc. of Symposium on Security and Privacy, Oakland, CA, pp. 190–200. IEEE Computer Society, Los Alamitos (1990)
4. Grossman, G.: Immediacy in distributed trusted systems. In: Proc. of Annual Computer Security Applications Conference, New Orleans, Louisiana. IEEE Computer Societ, Los Alamitos (1995)
5. Levin, T., Bhaskara, G., Nguyen, T.D., Clark, P.C., Benzel, T.V., Irvine, C.E.: Securecore security architecture: Authority mode and emergency management. Technical Report NPS-CS-07-012 and ISI-TR-647, Naval Postgraduate School and USC Information Science Institute, Monterey, CA (October 2007)
6. NSA: U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.03. National Security Agency (June 2007)
7. Levin, T.E., Irvine, C.E., Weissman, C., Nguyen, T.D.: Analysis of three multilevel security architectures. In: Proc. of Computer Security Architecture Workshop, Fairfax, Virginia, USA, pp. 37–46. ACM, New York (2007)
8. Clark, P.C., Irvine, C.E., Levin, T.E., Nguyen, T.D., Vidas, T.M.: Securecore software architecture: Trusted path application (TPA) requirements. Technical Report NPS-CS-07-001, Naval Postgraduate School, Monterey, CA (December 2007)
9. Dwoskin, J.S., Lee, R.B.: Hardware-rooted trust for secure key management and transient trust. In: Proc. of 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, pp. 389–400. ACM, New York (2007)
10. TCG: TCG specification architecture overview. Technical Report 1.2, Trusted Computing Group (April 2004)
11. Smith, S., Weingart, S.: Building a high-performance, programmable secure coprocessor. Computer Networks 31, 831–860 (1999)
12. Anderson, R.J.: A security policy model for clinical information systems. In: IEEE Symposium on Security and Privacy, Oakland, CA, May 1996, pp. 30–43 (1996)

Idea: Action Refinement for Security Properties Enforcement[⋆]

Fabio Martinelli[1] and Ilaria Matteucci[2]

[1] IIT CNR, Pisa, via Moruzzi, 1 - 56125 Pisa, Italy
[2] CREATE-NET, Trento, Via alla cascata, 56D -38100 Povo, Italy
fabio.martinelli@iit.cnr.it, matteucci.ilaria@gmail.com

Abstract. In this paper we propose an application of *action refinement* theory for enforcing security policies at different levels of abstraction. Indeed we assume to have a (high level) specification of a secure system with a possible untrusted component. It is controlled by a controller program, in such a way the system is secure. We show that it is possible to guarantee that the refinement of this system at a lower level of abstraction is still secure, regardless the behavior of the implementation of the untrusted component.

1 Introduction

In the development of software components, it is quite often required to pass from the specification level (high level of abstraction) to the implementation level (low level of abstraction). This passage is referred as *refinement*.

Here we consider the theory on *action refinement* given in [3]. The action refinement function converts a specification of an action on a system into an implementable program (*e.g.*, a procedure). Our goal is to present how the action refinement technique combined with our framework on process algebra controller operators for enforcing secure system(see, *e.g.*, [7]), allows to enforce security properties at different levels of abstraction.

In particular, we consider a controller operator, denoted by $\triangleright_T$, that works by monitoring a possible untrusted component of the system, called *target system*, and terminating its execution that is about to violate the security policy being enforced.

In our scenario, let S be a system that cooperate with a possible untrusted component (the target) X. Exploiting $\triangleright_T$, we are able to control each action of the target before executing it, *i.e.*, there exists a controller operator Y that controls X according to the semantics rule of $\triangleright_T$, $Y \triangleright_T X$ for short.

Let us consider a security policy P, represented by a process describing the "correct" (expected) behavior of the considered system, *i.e.*, a system is secure if the behavior of the system S is compliant w.r.t. the behavior described by P. To establish if S behaves as P requires, we exploit the weak simulation behavioral relation, denoted by $\preceq$, (see [8]) for comparing the behavior of the system and the policy. According to [4], the simulation relation guarantees that the system does not perform actions that are not

⋆ Work partially supported by EU-funded project "Software Engineering for Service-Oriented Overlay Computers"(SENSORIA) and by EU-funded project "CONSEQUENCE".

allowed by the policy. Hence, at high level of abstraction, the problem of enforcing security policy by $\triangleright_T$ is specified as follows:

$$\forall X \in \mathcal{E}_{Act_A} \quad S\|_{\mathbf{A}}(Y \triangleright_T X) \preceq P \tag{1}$$

where $\mathcal{E}_{Act_A}$ represents the set of high level processes and $\|_{\mathbf{A}}$ is the CSP process algebras parallel operator on a set of actions $\mathbf{A}$ (*e.g.*, see [5]).

Starting form the secure system in Statement 1, we show a method, based on action refinement, that provides a secure system also at a lower level of abstraction, regardless the behavior of the implementation of the possible malicious components. To do this, first we show that the refinement function is a congruence w.r.t. the parallel operator and the controller operator $\triangleright_T$, thus we conclude that a possible implementation of $S\|_{\mathbf{A}}(Y \triangleright_T X)$ is $r(S)\|_{\mathbf{A}}r(Y) \triangleright_T r(X)$ where $r(S)$, $r(Y)$ and $r(X)$ are the implementation of S, Y and X respectively, through a *refinement function* r. Successively, we show that the high level system and the low level one are *vertical bisimilar up to* r (see [3]), where r is a refinement function. This permits us to conclude that:

$$\forall r(X) \quad r(S)\|_{\mathbf{A}}(r(Y) \triangleright_T r(X)) \preceq r(P)$$

and more generally that

$$\forall X \in \mathcal{E}_{Act_C} \quad r(S)\|_{\mathbf{A}}(r(Y) \triangleright_T X) \preceq r(P)$$

where $\mathcal{E}_{Act_C}$ represents the set of lower level processes. Thus, since the implementation of each component, S, Y and X, of the system is independent one each other, the refined systems can be made secure by the refinement of the controller operator $r(Y)$ regardless the behavior of the implementation of the target, that has to be at the same level of abstraction of $r(Y)$.

2 The Action Refinement Theory

Referring to [3], we assume a set of action names Act, ranged over by $a, b, c\ldots$ and an invisible action τ that models the internal (silent) action. The set of actions Act has as subsets the set of *abstract* actions, denoted by Act_A, and the set of *concrete* actions, denoted by Act_C. For the sake of simplicity we consider that $Act_A \cap Act_C = \emptyset$. In addition we use a termination predicate $\checkmark$ and a set of processes $\mathcal{E}$ ranged over by $P, Q, E, F\ldots$. We denote with $\mathcal{E}_{Act_A}$ ($\mathcal{E}_{Act_C}$) the set of high (low) level processes, whose actions are in Act_A (Act_C). The syntax of the considered process algebra is the following:

$$P ::= \mathbf{0} \mid a.P \mid P;P \mid P+P \mid P\|_A P \mid P[f] \mid P/A$$

where $A \subseteq Act$ and the relabelling function $f : Act \mapsto Act$ must be such that $f(\tau) = \tau$. The semantics of this operators is the standard semantics of the CSP process algebra (see [5]). Informally, $\mathbf{0}$ is the term that does nothing; a (closed) term $a.P$ represents a process that performs an action a and then behaves as P. The term $P;P$ states that two processes are performed in sequence, one after the other. The term $P + P$ states that choosing the action of one of the two components the other is dropped; the term

$P\|_A P$ is the *synchronous parallel operator* on the set of action A. Any action in A is performed when all component processes perform it. The process P/A behaves like P but the actions in A are forbidden and replaced by τ; the process $P[f]$ behaves like P, but its actions are renamed through relabelling function f.

The usual concept of derivation in one step, $P \xrightarrow{a} P'$, means that process P evolves in one step into process P' by executing action $a \in Act$. Given a process P, $Sort(P)$ is the set of names of actions that syntactically appear in the process P.

The basic idea of action refinement is that, given a *refinement function* r mapping abstract actions to concrete processes.

In order to avoid unnecessary complications, we single out the fragment of refinement terms, **R**, made of **0** and the prefix, choice and sequence operators. The refinement function r is of the form $r : Act_A \rightarrow \mathbf{R}_C$ where $\mathbf{R}_C$ is the set of term on concrete actions and it is ranged over by $P_C, Q_C, \ldots$[1]. We overload notation by using $r(P)$ to denote a refined process P, *i.e.*, given a process P, the process $r(P)$ is the result of the application of the function r to each action in $Sort(P)$.

2.1 Vertical Bisimulation

One of the central concept of the refinement procedure is the notion of *vertical implementation relation* parameterized w.r.t. a refinement function r. It describes the relation that exists between a process at high specification level and its refinement at a lower specification level. The notion of vertical implementation was introduced in [3,10] in combination with a more standard "horizontal" implementation relation sometimes referred to as the *basis* of the vertical one. In this paper we are interested to a particular vertical implementation relation that has the weak simulation relation [9] as basis: The *vertical bisimulation*. It is formed of three components: a *down-simulation*, in which each abstract move must be matched by a sequence in the implementation, an *up-simulation* in which each move of the implementation should find a justification either as the initial action of a new refined action or as a continuation of a pending refinement, and a *residual simulation* requiring that each move of the pending refinements must be present in the implementation.

According to [3], the vertical bisimulation is a congruence w.r.t. the operators of the considered language.

3 Application of Refinement to Enforcement

In this section we show how the action refinement theory is suitable, when combined with our framework on controller operators, for enforcing security properties at lower level of abstraction.

Let us consider the controller operator $\triangleright_T$ whose semantics definition, according to [7], is the following:

$$\frac{E \xrightarrow{a} E' \quad F \xrightarrow{a} F'}{E \triangleright_T F \xrightarrow{a} E' \triangleright_T F'}$$

[1] We denote by $P_A, Q_A, \ldots$ terms on the abstract actions.

Informally, its semantics rule states that if E and F perform the same action, thus such action is allowed. Hence, the controlled process $E \triangleright_T F$ performs the action a, otherwise it halts.

Let us consider a systems S made secure by a controller program Y that controls possible unknown malicious component of the system, said X, trough a controller operator $\triangleright_T$. Formally, the (high level) specification of the scenario we are going to study, is the following:

$$\forall X \quad S \|_{\mathbf{A}} (Y \triangleright_T X) \preceq P \tag{2}$$

where P is the policy that describes the correct behavior of the considered system and $\preceq$ is the *weak simulation* relation (see [9]). According to [4], the simulation relation guarantees that all action executed by the system is also executed by the policies, *i.e.*, the system does not perform actions that are not allowed by the policy. This guarantees that the system is secure.

Our goal is to present a framework that guarantees that whether a system is secure at the high specification level, after the application of an action refinement function, we are able to guarantee, without check it again, that also the system at a lower specification level is secure regardless the behavior of the implementation of the target, under the assumption that we are always speaking at the same level of abstraction.

Let us consider an action refinement function r. We aim that, after the application of r to all components of the system, S, Y and X, and also to the policy P, the relation between this components is the same that we have at the specification level.

Theorem 1. *Let $S \|_{\mathbf{A}} (Y \triangleright_T X)$ be a secure system w.r.t. a policy P described at a high specification level. Let r be the refinement function w.r.t. the implementations are vertical bisimilar to the respective specification then:*

$$\forall X \in \mathcal{E}_{Act_C} \; r(S) \|_{\mathbf{A}} (r(Y) \triangleright_T X) \preceq r(P)$$

To do this we note that, under particular assumption (see [3]), the refinement function is a congruence w.r.t. $\triangleright_T$. As a matter of fact, according to [3], the following rule holds:

$$\frac{P_A \preccurlyeq^r P_C \quad Q_A \preccurlyeq^r Q_C \quad r \text{ preserves and is distinct on } \mathbf{A}}{P_A \|_{\mathbf{A}} Q_A \preccurlyeq^r P_C \|_{\tilde{r}(\mathbf{A})} Q_C} \tag{3}$$

where $\tilde{r}(\mathbf{A})$ is the set of refined actions obtained by applying r to each action in $\mathbf{A}$. Assume that r is distinct on $\mathbf{A}$ means assuming that the refinement of two distinct action cannot lead to the same resulting process. Referring to them semantics definition we know $\|_{\mathbf{A}}$ is semantically equivalent to $\triangleright_T$ when $\mathbf{A} = Act$. Hence the implementation of the controller program Y, obtained by using a refinement function r, does not depend on the behavior of the implementation of the target X. Hence $r(Y)$ makes the system secure for all possible $r(X)$ an, in particular for all processes defined at the same level of abstraction of $r(Y)$. Successively, it is possible to note that proving that the weak simulation is preserved by the vertical bisimulation, we have the Theorem 1.

4 Conclusion and Related Work

In this paper we have presented a way for applying action *refinement* theory for the enforcement of security properties at different specification level. By starting from our theory on process algebra operators (see [7]), we have shown how it is possible to refine a system, including the existing controller programs, by guarantying that the resulting one is yet secure (w.r.t. to the refined security policy).

Much work has been done in developing process refinement theories.

In [11] the authors present a case study on analysis of refinement. Indeed, using Event-B they refine controller for a security property along the different network layer of the TCP/IP stack and prove that such refinement is valid. Our approach allows to treat several scenarios, from mobile phone applications to security web services composition.

Referring to the framework of policies refinement, in [2] is introduced KAOS, that is a goal oriented methodology to analyze and refine requirements, especially, security requirements. An advantages of our work w.r.t. KAOS is that we are able to refine enforcement mechanisms. In KAOS the enforcement is not treated specifically. Also in [1] the authors refer to the policy refinement problem as the passage from the high specification level of a system to the implementation of the system itself guaranteeing that the goals are achieved. There are no considerations neither investigation about security aspects. In [6] a slightly different notion of refinement of policies is given. It is a projection of a big policy into small ones whose composition gives the big one.

Acknowledgments. We would like to thank the anonymous reviewers for their helpful comments.

References

1. Bandara, A.K., Lupu, E., Moffett, J.D., Russo, A.: A goal-based approach to policy refinement. In: POLICY, pp. 229–239. IEEE Computer Society Press, Los Alamitos (2004)
2. Dardenne, A., Lamsweerde, A.V., Fickas, S.: Goal-directed requirements acquisition. In: Science of Computer Programming, pp. 3–50 (1993)
3. Gorrieri, R., Rensink, A., Zamboni, M.A.: Action refinement. In: Handbook of Proacess Algebra, pp. 1047–1147. Elsevier, Amsterdam (2001)
4. Greci, P., Martinelli, F., Matteucci, I.: A framework for contract-policy matching based on symbolic simulations for securing mobile device application. In: Margaria, T., Steffen, B. (eds.) ISoLA. Communications in Computer and Information Science, vol. 17, pp. 221–236. Springer, Heidelberg (2008)
5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
6. Carey, V.W.K., Lewis, D.: Automated policy-refinement for managing composite services. In: M-Zones White Paper June 2004, whitepaper 06/04, Ireland (June 2004)
7. Martinelli, F., Matteucci, I.: An approach for the specification, verification and synthesis of secure systems. Electr. Notes Theor. Comput. Sci. 168, 29–43 (2007)
8. Milner, R.: Operational and algebraic semantics of concurrent processes. In: van Leewen, J. (ed.) Handbook of Theoretical Computer Science, ch. 19. Formal Models and Semantics, vol. B, pp. 1201–1242. The MIT Press, New York (1990)

9. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press, Cambridge (1999)
10. Rensink, A., Gorrieri, R.: Action refinement as an implementation relations. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 772–786. Springer, Heidelberg (1997)
11. Stouls, N., Potet, M.L.: Security policy enforcement through refinement process. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 216–231. Springer, Heidelberg (2006)

Pattern-Based Confidentiality-Preserving Refinement

Holger Schmidt

University Duisburg-Essen, Germany, Faculty of Engineering,
Department of Computer Science and Applied Cognitive Science,
Workgroup Software Engineering
holger.schmidt@uni-duisburg-essen.de

Abstract. We present an approach to security requirements engineering, which makes use of special kinds of problem frames that serve to structure, characterize, analyze, and solve software development problems in the area of software and system security.

In this paper, we focus on confidentiality problems. We enhance previously published work by formal behavioral frame descriptions, which enable software engineers to unambiguously specify security requirements. Consequently, software engineers can prove that the envisaged solutions provide functional correctness and that the solutions fulfill the specified security requirements.

1 Introduction

As a consequence of an increasing demand for *security*, software engineers are not only confronted with functional requirements, but also with security requirements, although they are not experts in *security engineering*. In the early phases of software development, functional as well as security requirements have to be elicited and analyzed. This task alone is difficult enough, but the software engineers are then faced with realizing the requirements. Clearly, they need methods and techniques that help them to elicit, analyze, specify and finally realize security requirements in a *feasible* and *correct* way.

In earlier publications (cf. [4, 5, 6, 7]), we introduced a security engineering process that focuses on the early phases of software development. The basic idea is to make use of special patterns defined for structuring, characterizing, and analyzing *problems* that occur frequently in security engineering. Similar patterns for functional requirements have been proposed by Jackson [11]. They are called *problem frames*. Accordingly, our patterns are named *security problem frames*. Furthermore, for each of these frames, we define a set of *concretized security problem frames* that take into account generic security mechanisms to prepare the ground for solving a given security problem.

In this paper, we concentrate on the (concretized) security problem frames that deal with confidentiality. We present the following enhancements of our security requirements engineering approach:

– We underlay the (concretized) security problem frames with a *formal behavior description* to gain an unambiguous comprehension of the frames, and to clarify their semantics.
– As a prerequisite for software development based on *stepwise refinement*, we prove that the step from security problem frames to concretized security problem frames is a functionally correct refinement, which preserves the confidentiality requirement.
– We provide a point of contact to the *formal probabilistic (and possibilistic) security requirement descriptions* by Santen [18]. This allows software engineers to express security requirements in a well-defined way.

Furthermore, the work in this paper constitutes a basis to analyze the instantiation process of the (concretized) security problem frames and to investigate necessary applicability conditions for the frames.

The bottom line of these enhancements is a security engineering approach that focuses on the early phases of secure software development. Furthermore, it yields a formal specification of the software to be built, which constitutes a starting point for software design and implementation.

In the following, we first present Jackson's problem frames as well as security problem frames and concretized security problem frames in Sects. 2 and 3. In Sect. 4, we briefly introduce the formal specification language CSP (Communicating Sequential Processes) [9], which we subsequently use to create formal behavior descriptions of (concretized) security problem frames for confidential data transmission (using encryption). Furthermore, we analyze these formal models with respect to confidentiality-preserving refinement in Sect. 5. Section 6 discusses related work, and the paper closes with a summary and perspectives in Sect. 7.

2 Problem Frames

Problem frames are a means to analyze and classify software development problems. Jackson [11] describes them as follows: "A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement." Problem frames are described by *frame diagrams*, which basically consist of rectangles and links between these (see frame diagrams in Figs. 1 and 2). The task is to construct a *machine* that improves the behavior of the environment it is integrated in.

Plain rectangles denote *domains* (that already exist), a rectangle with a single vertical stripe denotes a *designed domain* physically representing some information, and a rectangle with a double vertical stripe denotes the machine to be developed. *Requirements* are denoted with a dashed oval. The connecting lines represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain. For example, if a user types a password to log into an IT-system, this is a phenomenon shared by the user and the system, which is controlled by the user. A dashed line represents a

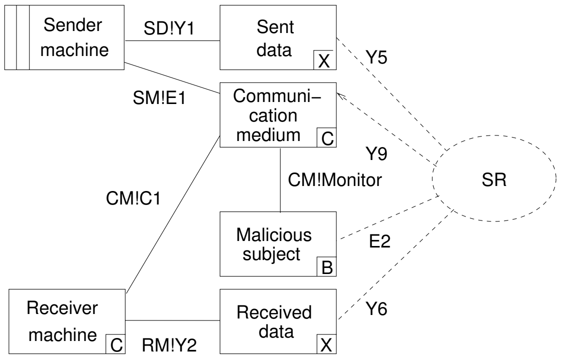


Fig. 1. Security problem frame for confidential data transmission

requirements reference, and the arrow shows that it is a *constraining* reference. Furthermore, Jackson distinguishes *causal* domains that comply with some physical laws, *lexical* domains that are data representations, and *biddable* domains that usually are people.

In the frame diagram depicted in Fig. 1, a marker "X" indicates a lexical domain, "B" indicates a biddable domain, and "C" indicates a causal domain. The notation "SM!E1" means that the phenomena of interface E1 are controlled by the machine domain Sender machine.

Problem frames greatly support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*.

The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the requirements are realizable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment regardless of how the machine is built.

Requirements describe the environment, the way it should be, after the machine is integrated. In contrast to the requirements, the *specification* of the machine gives an answer to the question: "How should the machine act, so that the system, i.e., the machine together with the environment, fulfills the requirements?" Specifications are descriptions that are sufficient for building the machine. They are implementable requirements.

3 (Concretized) Security Problem Frames

To meet the special demands of software development problems occurring in the area of security engineering, we introduced security problem frames (SPF) [4, 5]. SPFs are a special kind of problem frames, which consider *security requirements*.

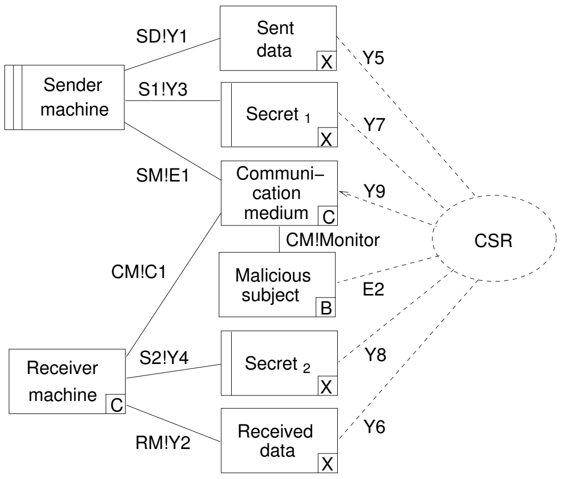


Fig. 2. Concretized security problem frame for confidential data transmission using encryption

The SPFs we have developed strictly refer to the *problems* concerning security. They do not anticipate a solution. For example, we may require the confidential transmission of data without mentioning encryption, which is a means to achieve confidentiality.

Solving a security problem is achieved by applying generic security mechanisms (e.g., encryption to keep data confidential), thereby transforming security requirements into *concretized security requirements*. The generic security mechanisms are represented by concretized security problem frames (CSPF). The benefit of considering security requirements without reference to potential solutions is the clear separation of problems from their solutions, which leads to a better understanding of the problems and enhances the re-usability of the problem descriptions, since they are independent of solution technologies.

Figure 1 shows the frame diagram of the SPF for confidential data transmission. The domain Sent data denotes the data that is sent by a sender, represented by the machine domain Sender machine. Analogously, the domain Received data denotes the data that is received by the domain Receiver machine. The data is transmitted over some network, which is represented by the domain Communication medium. Informally speaking, the sender machine generates the transmitted data from the sent data, and the receiver machine generates the received data from the communication medium. In this scenario, a potential attacker represented by the domain Malicious subject can eavesdrop on the Communication medium. The informal security requirement SR is described as follows:

> Malicious subject should not be able to infer Sent data and Received data except for their length by eavesdropping on Communication medium.

One of the CSPFs for confidential data transmission considers (symmetric and asymmetric) encryption. Its frame diagram is shown in Fig. 2. In transforming the security requirement for confidential data transmission into a concretized security requirement CSR, the domains Secret_1 and Secret_2 are introduced for the encryption mechanism. The informal concretized security requirement CSR is described as follows:

> Malicious subject should not be able to infer Sent data and Received data except for their length without Secret_1 and Secret_2 by eavesdropping on Communication medium. Malicious subject should not be able to obtain Secret_1 and Secret_2.

In the subsequent Sects. 4 and 5, we first equip the frame diagrams of the (C)SPFs depicted in Figs. 1 and 2 with formal behavior descriptions, and second we analyze these formal descriptions with respect to confidentiality-preserving refinement.

4 Formal Foundation of (C)SPFs

The software development principle of *stepwise refinement* is popular in software engineering, and is also well supported by *formal methods*. When performing stepwise refinement, a software engineer develops software by creating intermediate levels of abstraction. Starting with the requirements, an abstract *specification* is constructed, which is refined by a more concrete *implementation*. Then, the implementation must be verified against the specification, and further refinement steps are accomplished until the desired level of abstraction is achieved.

Refinement is traditionally either data-refinement or behavior-refinement. Since the (C)SPFs deal with interfaces and communicating domains rather than with states, we decided to describe them using CSP (Communicating Sequential Processes) [9]. CSP is a model-based formal method to describe parallel processes that communicate synchronously via message passing. Furthermore, with the model-checker FDR2 (Failure-Divergence Refinement) [14] sophisticated tool support is available for CSP.

In Sect. 4.1, we present a general procedure to create a formal CSP model for a given (C)SPF. In Sect. 4.2, we apply the procedure described in Sect. 4.1 to create CSP models for the (C)SPFs shown in Figs. 1 and 2. Furthermore, we formalize the (concretized) security requirements of the (C)SPFs based on the functional CSP models in Sect. 4.3.

In Sect. 5, we show that the CSPF model in Fig. 7 is a refinement of the SPF model in Fig. 5, and we include the confidentiality requirements presented in Sect. 4.3 in our analysis in order to show a *confidentiality-preserving refinement* (CPR). Figure 3 describes that CPR is not only of interest on the pattern level, but also on the instance level. It is desirable that once we have shown the functional and confidentiality-preserving refinements on the pattern level, they also apply (conditionally or not) to the instance level. We call the latter *confidentiality-preserving instantiation*.

Applying CSP and stepwise refinement to the (C)SPF approach has several benefits:

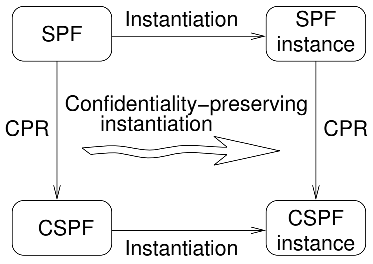


Fig. 3. Confidentiality-preserving instantiation

- Enable a developer to *unambiguously express security requirements* captured by (C)SPFs.
- Since problem frames and (C)SPFs as such only provide a static view of a system [1], we obtain an understanding of the dynamic aspects of (C)SPFs.
- Allow one to verify that the functional and the security requirements of a SPF are *correctly* implemented by an associated CSPF, i.e., that the functionality and the security requirement are preserved.
- Verification is tool-supported by the model-checker FDR2.
- The CSP models provide a point of contact to the formal probabilistic (and possibilistic) security requirement descriptions by Santen [18].

Note that the enhancements described in this paper are not restricted to (C)SPFs concerning confidentiality. In fact, also frames concerning integrity and authentication problems can be translated into CSP models to formally express the security requirements they capture.

4.1 Formal (C)SPF Models in CSP

We make use of the CSP ASCII notation named CSP_M since this is a prerequisite for formal verification using the model-checker FDR2.

Using CSP_M notation, we define *processes* that interact only by communicating. Communication takes the form of visible *events* or *actions*. A sequence of events produced by a process is called a *trace*. The set of all traces that can be produced by a process P are denoted $traces(P)$. Let a be an action and P be a process; then $a- > P$ is the process that performs a and behaves like P afterwards. This is called *prefixing*. A process can have a name, e.g., $Q = a- > P$. *Recursion* makes it possible to repeat processes and to construct processes that go on indefinitely, e.g., $Q = a- > Q$.

We can make use of input and output data: the expression $in?x$ binds the identifier x to whatever value is chosen by the environment, where x ranges over the type of *channel in*. The expression $out!y$ binds an output action to the identifier y, where y ranges over the type of channel out. The variables x and y

[1] A formal metamodel covering the static nature of problem frames is already developed (cf. [8]).

can then be used in the process following the prefix. By convention, ? denotes input data and ! denotes output data.

A process acts in a *nondeterministic* way when its behavior is unpredictable because it is allowed to make internal decisions that affect its behavior as observed from outside. The *replicated internal choice operator* $|\sim|$ models these internal decisions: let P be a process and X a finite and non-empty data type, then $|\sim|\ a : X \bullet P(a)$ behaves according to the selected a. This operator gives the environment no control over which data item is chosen. In contrast, the *replicated external choice operator* [] models external decisions: let P be a process and X a non-empty data type, then $[]\ a : X \bullet P(a)$ behaves according to the a selected by the environment.

To formalize a given (C)SPF, we describe each of its domains as a recursive CSP process. The interfaces and the control direction of the shared phenomena (control flow) of a domain are translated into CSP channels as well as input and output events. For lexical shared phenomena, we define data types and declare the corresponding channels to be of one of these data types.

Note that when using a model-checker such as FDR2 to analyze real-world problems, we have to address the state explosion problem. A common approach to keep the model-checking effort manageable is to simplify the system to be analyzed. For that reason, we usually must define simplified data types.

We describe a (C)SPF as a CSP process consisting of the CSP processes of all of its domains. The processes are combined using *synchronized parallel communication* denoted by [| |]. The synchronization is accomplished over the channels modelling the interfaces that connect the domains.

The described procedure can be applied to express any (C)SPF as a formal CSP model.

4.2 CSP Models of (C)SPFs Confidential Data Transmission (Using Encryption)

As examples, we present in Figs. 5 and 7 the CSP models of the (C)SPFs confidential data transmission (using encryption) shown in Figs. 1 and 2.

Figure 4 shows type and function definitions as well as channel declarations for the CSP model in Fig. 5. We define a simple data type named *Plaintext* with four values $p1, p2, p3, p4$. Then, we declare the channels SD_Y1, RM_Y2, SM_E1_S, and CM_C1_S (see Fig. 1) to be of this data type, i.e., all events communicated over these channels are $p1, p2, p3$, or $p4$.

We represent the interface between the Malicious subject domain and the Communication medium domain by a channel $CM\_monitor\_S$ of the data type *Length*. The data items leaked over this channel are defined by a *leakage function f*. As an example, the leaked data items are *short* and *long*, and they correspond to the lengths of the plaintexts sent over the channels of the process $CM\_monitor\_S$ (see definition of function f in Fig. 4).

We describe each domain of the SPF in Fig. 1 as a recursive CSP process, e.g., the process SM_S in Fig. 5 is a formal representation of the domain Sender machine of the SPF confidential data transmission. The process SD_S in Fig. 5 offers

```
-- Data type definitions
datatype Plaintext = p1 | p2 | p3 | p4
datatype Length = short | long

-- Channel declarations
channel SD_Y1, RM_Y2, SM_E1_S, CM_C1_S: Plaintext
channel CM_monitor_S : Length

-- Function definition
f(p) = if p==p1 or p==p2 then short else long
```

Fig. 4. Type and function definitions as well as channel declarations for the CSP model in Fig. 5

```
-- Process for domain Sent data
SD_S = |~| pt : Plaintext @ SD_Y1!pt -> SD_S

-- Process for domain Sender machine
SM_S = SD_Y1?pt -> SM_E1_S!pt -> SM_S

-- Process for domain Communication medium
CM_S = SM_E1_S?pt -> CM_monitor_S!f(pt) -> CM_C1_S!pt -> CM_S

-- Process for domain Malicious subject
MS_S = CM_monitor_S?l -> MS_S

-- Process for domain Receiver machine
RM_S = CM_C1_S?pt -> RM_Y2!pt -> RM_S

-- Process for domain Received data
RD_S = RM_Y2?pt -> RD_S

-- Process for SPF Confidential Data Transmission
SPF_CONF = ((((SD_S [| {|SD_Y1|} |] SM_S)
       [| {|SM_E1_S|} |] CM_S) [| {|CM_monitor_S|} |] MS_S)
            [| {|CM_C1_S|} |] RM_S) [| {|RM_Y2|} |] RD_S
```

Fig. 5. CSP model of SPF depicted in Fig. 1

a data item *pt* that might be internally processed (expressed using the replicated internal choice operator) to the environment using the channel SD_Y1. Then, the process SM_S begins with reading in this data item *pt* over channel SD_Y1, and *pt* might be passed over to the environment using the channel SM_E1_S.

We specify the SPF in Fig. 1 as a process SPF_CONF in Fig. 5 that combines all formalized domains of the SPF confidential data transmission. For example, the process SM_S synchronizes over the channel SD_Y1 with the process SD_S, or, informally speaking, the domain Sender machine reads data from the domain Sent data.

```
-- Data type definitions
datatype Ciphertext = c1 | c2 | c3 | c4
datatype Secret = s1 | s2 | s3 | s4

-- Channel declaration
channel S1_Y3, S2_Y4 : Secret
channel SM_E1_I, CM_C1_I : Ciphertext.Secret
channel CM_monitor_I : Ciphertext.Length

-- Function definition
encr(p1,s1) = c1   encr(p1,s2) = c2   encr(p1,s3) = c1   encr(p1,s4) = c2
encr(p2,s3) = c2   encr(p2,s4) = c1   encr(p2,s1) = c2   encr(p2,s2) = c1
encr(p3,s1) = c3   encr(p3,s2) = c4   encr(p3,s3) = c3   encr(p3,s4) = c4
encr(p4,s3) = c4   encr(p4,s4) = c3   encr(p4,s1) = c4   encr(p4,s2) = c3

decr(c1,s1) = p1   decr(c1,s2) = p2   decr(c1,s3) = p1   decr(c1,s4) = p2
decr(c2,s1) = p2   decr(c2,s2) = p1   decr(c2,s3) = p2   decr(c2,s4) = p1
decr(c3,s1) = p3   decr(c3,s2) = p4   decr(c3,s3) = p3   decr(c3,s4) = p4
decr(c4,s1) = p4   decr(c4,s2) = p3   decr(c4,s3) = p4   decr(c4,s4) = p3
```

Fig. 6. Type and function definitions as well as channel declarations for the CSP model in Fig. 7

Figure 6 shows type and function definitions as well as channel declarations for the CSP model in Fig. 7. We introduce data types *Secret* and *Ciphertext*, and the functions *encr* and *decr* in Fig. 6 to model that encryption is used in the CSPF confidential data transmission using encryption. The functions *encr* and *decr* model a length-preserving cryptographic mechanism. Furthermore, we declare the channels $S1\_Y3$ and $S1\_Y4$ to be of type *Secret* and the channels SM_E1_I and CM_C1_I to be of the composed type *Ciphertext.Secret*. The role of the Malicious subject's monitoring channel has changed: the channel *CM_monitor_I* not only leaks the lengths of the transferred data items to the environment, but also the complete ciphertexts. For that reason, the channel *CM_monitor_I* is of the composed type *Ciphertext.Length*.

We introduce two new processes $S1\_I(s)$ and $S2\_I(t)$ in the CSP model in Fig. 7. They stand for the domains Secret_1 and Secret_2 of the CSPF confidential data transmission using encryption. Both processes are equipped with parameters that represent the secrets chosen by the environment. The process *SM_I* corresponds to the process *SM_S* of the CSP model in Fig. 5, and is extended by reading in a secret s over the channel $S1\_Y3$. Proceeding with the events of the process *SM_I*, the function *encr* is applied to a plaintext *pt* using a secret s, and the result as well as the secret s is passed over to *CM_I* using the channel SM_E1_I. Afterwards, the ciphertext *ct* as well as the secret s are passed over to the environment using the channel CM_C1_I. In a similar way, the function *decr* is used when receiving encrypted data (see process *RM_I*). The process *MS_I* corresponds to the process *MS_S* of the CSP model in Fig. 5, and is changed to be able to receive the ciphertexts and their lengths over channel *CM_monitor_I*.

```
-- Process for domain Sent data
SD_I = |~| pt : Plaintext @ SD_Y1!pt -> SD_I

-- Process for domain Secret_1
S1_I(s) = S1_Y3!s -> S1_I(s)

-- Process for domain Sender machine
SM_I = SD_Y1?pt -> S1_Y3?s -> SM_E1_I!encr(pt,s).s -> SM_I

-- Process for domain Communication medium
CM_I = SM_E1_I?ct.s -> CM_monitor_I!ct.f(decr(ct,s))
          -> CM_C1_I!ct.s -> CM_I

-- Process for domain Malicious subject
MS_I = CM_monitor_I?ct.l -> MS_I

-- Process for domain Secret_2
S2_I(t) = S2_Y4!t -> S2_I(t)

-- Process for domain Receiver machine
RM_I = CM_C1_I?ct.s -> S2_Y4?t -> RM_Y2!decr(ct,t) -> RM_I

-- Process for domain Received data
RD_I = RM_Y2?pt -> RD_I

-- Process for CSPF Confidential Data Transmission using Encryption
CSPF_CONF_ENCRYPTION(s, t) = ((((
       (SD_I [| {|SD_Y1|} |] SM_I) [| {|S1_Y3|} |] S1_I(s))
            [| {|SM_E1_I|} |] CM_I) [| {|CM_monitor_I|} |] MS_I)
                  [| {|CM_C1_I|} |] (RM_I [| {|S2_Y4|} |] S2_I(t)))
                     [| {|RM_Y2|} |] RD_I

-- initialization: choosing secrets
INIT_CSPF_CONF_ENCRYPTION =
       ([] x : Secret, y : Secret, x == y @ CSPF_CONF_ENCRYPTION(x, y))
```

Fig. 7. CSP model of CSPF depicted in Fig. 2

We specify the CSPF in Fig. 2 as a process *CSPF_CONF_ENCRYPTION* (s, t) in Fig. 7 that combines all formalized domains of the CSPF confidential data transmission using encryption.

The parameters of the processes $S1\_I(s)$ and $S2\_I(t)$ provide a point of contact to the *(C)SPFs Distributing Secrets* [5], which consider the problem to communicate *matching* secrets to those subjects who are privileged to receive them (cf. [4]). Since no CSPF Distributing Secrets is considered in this paper, we simulate the mechanism using the replicated external choice operator to define the process *INIT_CSPF_CONF_ENCRYPTION*. Hence, the secrets are chosen by the environment, and as an example for modelling a *symmetric* encryption mechanism, the constraint $x == y$ requires both secrets to be equal.

Using FDR2, we successfully verified that the CSP models in Figs. 5 and 7 are deadlock-free and livelock-free.

4.3 Formal Description of Confidentiality Requirements

Confidentiality requirements can be expressed as *information flow properties* of two flavors:

Possibilistic. Based on the fact that an IT system has a system behavior, which produces observations visible to the environment, there must exist at least one alternative possible system behavior that produces the same observation.
Probabilistic. Stochastic system behavior is taken into account.

In this section, we consider *possibilistic* information flow properties, and we apply the framework for the specification of confidentiality requirements by Santen [18] to capture the confidentiality requirement of the (C)SPF confidential data transmission (using encryption).

In general, we call the formal description of a confidentiality requirement a *confidentiality property* (cf. Definition 9 in [18]). Since confidentiality properties are predicates on *sets* of traces, they cannot be modelled in CSP, and thus cannot be verified using FDR2. Nevertheless, we can specify a confidentiality property "on paper" and prove that a given machine and environment satisfy the property.

There does not exist *the* confidentiality property that allows us to express every (informal) confidentiality requirement. Instead, an adequate confidentiality property depends on the confidentiality requirement that it formalizes (cf. [15] for a comprehensive overview of possibilistic information flow properties).

The concept of indistinguishable traces (cf. [18, p. 223]) is the foundation for defining confidentiality properties. Given a set of channels W, two traces s, $t \in \mathit{traces}(P)$ of a process P are *indistinguishable* by W (denoted $s \equiv_W t$) if their projections to W are equal: $s \equiv_W t \Leftrightarrow s \upharpoonright W = t \upharpoonright W$, where $s \upharpoonright W$ is the projection of the trace s to the sequence of events on W. The *indistinguishability class* $J_W^{P,k}(o)$ contains the traces of P with a length of at most k that produce the observation o on W.

Applied to the CSP models presented in Sect. 4.2 this means that any distinction (e.g., data item length is *short* or *long*) the malicious subject can make about the internal communication of the system (e.g., sending different plaintexts and ciphertexts) based on the observations on *CM_monitor_S* and *CM_monitor_I* is information revealed by the system. Conversely, any communication that cannot be distinguished by observing *CM_monitor_S* and *CM_monitor_I* is concealed by the system. We can determine two indistinguishability classes, one that contains those traces that produce the observation *short* on the monitoring channel, and another one that contains those traces that produce the observation *long* on the monitoring channel.

An *adversary model* (cf. [18, p. 222]) is a system model that consists of the machine to be developed, the honest user environment, the adversary enviroment, and their interfaces. The CSP models presented in Sect. 4.2 constitute valid adversary models.

As defined by Santen (cf. Definition 11 in [18]), a *mask* $\mathcal{M}$ for an adversary model is a set of subsets of the traces over the alphabets of the processes modelling the machine to be developed, the honest user environment, and the adversary enviroment such that the members of each set are indistinguishable by observing the monitoring channel of the adversary enviroment W: $\forall M : \mathcal{M};\ t_1, t_2 : M \bullet t_1 \equiv_W t_2$.

If Malicious subject observes the single event $CM\_monitor\_S.l$ (where $l \in Length$), then s/he knows that exactly one data transfer has taken place. All traces of the form

$$t_0(pt) = \begin{cases} \langle SM\_E1\_S.pt, CM\_monitor\_S.short\rangle & \text{if } pt \in \{p1, p2\}, \\ \langle SM\_E1\_S.pt, CM\_monitor\_S.long\rangle & \text{else}, \end{cases}$$

where $pt \in Plaintext$, produce the observation $CM\_monitor\_S.l$ for Malicious subject. According to the informal confidentiality requirement as it has been stated in Sect. 3, this observation should not allow Malicious subject to infer the transferred plaintext.

Note: the leakage function f must not be injective. If the function f were injective, i.e., f assigns exactly one plaintext to each length, the confidentiality requirement could not be achieved.

A mask $\mathcal{M}_0$ supporting the confidentiality requirement needs to require that for a given length l all variations of plaintexts pt in the parameter list of the trace t_0 are possible causes of the observation $CM\_monitor\_S.l$. Therefore, the sets $M_0 = \{t_0(p1), t_0(p2)\}$ and $M_1 = \{t_0(p3), t_0(p4)\}$ should be members of $\mathcal{M}_0$.

If the traces in a set $M \in \mathcal{M}$ are indistinguishable by observing the monitoring channel, then the differences between these traces are kept confidential. This confidentiality property is named *concealed behavior* (cf. [18, p. 228]). It is formalized based on a set inclusion $M \subseteq J_W^{QE,k}(o)$, where the process QE is a *variant* (i.e., a purely deterministic process, cf. [18, p. 228]) of the adversary model. It is required that members of $\mathcal{M}$ are either completely contained in an indistinguishability class, or not at all. One says that the set of indistinguishability classes $\mathcal{I}$ *covers* $\mathcal{M}$.

In general, a given adversary model satisfies a confidentiality property, which is defined based on a *basic confidentiality property* (cf. [18, p. 225]), if there exists a probabilistic deterministic realization of a machine that satisfies the basic confidentiality property in all admissible environments. In the case of concealed behavior, the question is if there is an adversary model that covers a given mask.

To show that the adversary model represented by the CSP model in Fig. 5 conceals the mask $\mathcal{M}_0$, a deterministic machine realization must be found such that its composition with all realizations of the environment covers $\mathcal{M}_0$.

We choose the implementation of the CSP model in Fig. 5 that resolves the nondeterministic choice of the process SM_S in Fig. 5 by a probabilistic choice with equal probabilities for all alternatives.

The admissible environments consist of realizations that deterministically produce traces according to the pattern $t_0(pt)$, where $pt \in Plaintext$, i.e., data transmissions from Sender machine to Receiver machine.

The members M_0 and M_1 of $\mathcal{M}_0$ are covered by the indistinguishability classes of all resulting variants of *SPF_CONF*, because the chosen machine realization does not exclude any of the traces $t_0(pt)$, where $pt \in Plaintext$.

In summary, we formally described the (C)SPFs confidential data transmission (using encryption) by CSP models. Furthermore, we introduced a formal description of a sample confidentiality requirement. The presented approach is not limited to express an informal confidentiality requirement only by the confidentiality property concealed behavior. In contrast, other confidentiality properties (e.g., *ensured entropy* [18, p. 229]) can be used. In the next sections, we verify that the CSPF model is a correct refinement of the SPF model, and that the confidentiality requirement is preserved under refinement.

5 Confidentiality-Preserving Refinement

Refinement is the transformation of an abstract specification into a concrete specification (implementation). CSP supports three types of process refinements:

Trace refinement. A process Q trace-refines a process P, if all the possible sequences of communications, which Q can perform, are also possible in P.
Failure refinement. Trace refinement extended by consideration of deadlocks.
Failure-divergence refinement. Failure refinement extended by consideration of livelocks.

We first prove on a functional level that the CSPF confidential data transmission using encryption failure-divergence refines the SPF confidential data transmission (Sect. 5.1). Second, we show that the confidentiality requirement is preserved in the CSPF confidential data transmission using encryption (Sect. 5.2).

5.1 (C)SPF Functional Refinement

To show that a CSPF refines a SPF, we make use of the failure-divergence refinement. Since all structural elements of a SPF are preserved in an associated CSPF, we can show a failure-divergence refinement after we reduce the structural additions of the CSPF to the SPF structure:

– We hide events that can only be communicated in the CSPF model, i.e., all events communicated over $S1\_Y3$ and $S2\_Y4$.
– We map those events that have a more concrete structure in the CSPF model to events that are compatible with events of the SPF model, e.g., events passed over SM_E1_I are substituted by events passed over SM_E1_S. This mapping constitutes a data refinement.
– The data refinement is characterized by the fact that a plaintext is refined by a pair consisting of a ciphertext and a secret. In the implementation, two such pairs are indistinguishable if the ciphertexts are equal, because the malicious subject can observe the ciphertexts and their lengths using channel *CM_monitor_I*.

We construct a CSP process *ABS_CSPF_CONF_ENCRYPTION* using *relational renaming* $[[<-]]$ and *hiding* $\backslash$ (the corresponding CSP artifacts are not shown in this paper because of space limitations). This process failure-divergence refines the CSP process *SPF_CONF*, which we successfully verified using FDR2[2]. This kind of refinement is called *behavior refinement of adversary models* (cf. [18, p. 232]).

5.2 (C)SPF Refinement of Confidentiality Requirements

After we have shown that the CSPF model in Fig. 7 is a functionally correct refinement of the SPF model in Fig. 5, we include the confidentiality requirement presented in Sect. 4.3 in our analysis in order to show that it is a confidentiality-preserving refinement.

In the CSP model of the CSPF confidential data transmission using encryption depicted in Fig. 7, the monitoring channel *CM_monitor_I* has changed compared to its specification *CM_monitor_S*. This harbors the danger that leaks are introduced in the implementation through stepwise refinement.

To show that the confidentiality property concealed behavior, i.e., that the CSP model of the SPF confidential data transmission in Fig. 5 conceals $\mathcal{M}_0$, is preserved (and no further leaks are introduced), we must show that a similar property applies for the CSP model of the CSPF confidential data transmission using encryption in Fig. 7. Figure 8 informally describes the approach.

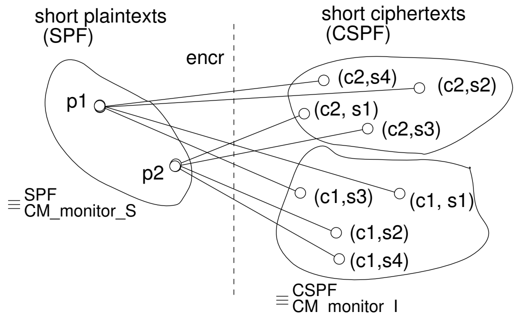


Fig. 8. Concretization and indistinguishability

Let $p1$ and $p2$ be indistinguishable plaintexts with respect to the channel *CM_monitor_S* and the SPF model, i.e., $p1 \equiv^{SPF}_{CM\_monitor\_S} p2$. According to the function *encr*, the plaintext $p1$ is represented by the pairs $(c1, s1)$, $(c1, s3)$, $(c2, s2)$, and $(c2, s4)$, and the plaintext $p2$ is represented by the pairs $(c1, s2)$, $(c1, s4)$, $(c2, s1)$, and $(c2, s3)$. Thereby, the pairs containing $c1$ as well as the pairs containing $c2$ are indistinguishable with respect to the channel

[2] FDR2 has to check 3.732.737 states with 10.323.200 transitions, which takes ~ 3 minutes on a dual-core machine with 2×2 GHz and 2GB RAM.

CM_monitor_I and the CSPF model. Similar facts apply to the *long* plaintexts and ciphertexts.

In the concrete CSPF model, all traces of the form

$$\langle SM\_E1\_I.ct.s, CM\_monitor\_I.ct.l, CM\_C1\_I.ct.s \rangle$$

where $ct \in Ciphertext$, and $s \in Secret$ produce the observation $CM\_monitor\_I.ct$ for Malicious subject. In Sect. 5.1, the behavior refinement changed the monitoring channel and refined the data communicated by the processes. Since the confidentiality property concealed behavior refers to both, the monitoring channel and the data, we must relate the concrete monitoring channel and the data back to the abstract ones originally referred to by the confidentiality property. Applied to concealed behavior, this general concept provides a basis for defining *refined concealed behavior* (cf. [18, p. 239]).

After the re-abstraction, we must check if the re-abstracted traces are members of M_0: the re-abstracted traces are the same traces as the abstract ones. For that reason, the CSP model of the CSPF confidential data transmission using encryption in Fig. 7 conceals $\mathcal{M}_0$, and the confidentiality property concealed behavior is preserved in the CSPF confidential data transmission using encryption.

6 Related Work

In this section, we discuss our work in connection with other formal approaches to security requirements engineering, as well as with other approaches to formalize problem frames. Note that not all relevant publications are mentioned because of space limitations.

Li et al. [13] use an extended CSP version [12] to systematically derive a specification from requirements. Their work does not consider non-functional requirements such as security requirements. Furthermore, biddable domains are not formalized. Since biddable domains are used to model unpredictable parts of the environment (such as honest and malicious users), we believe that this is a key feature to security requirements engineering.

Nelson et al. [17] describe problem frames as well as requirements using Alloy [10]. Compared to our work, security requirements and their refinement to specifications are not considered. Additionally, Alloy does not allow to express security requirements in terms of information flow properties.

KAOS – Keep All Objectives Satisfied[3] is a goal-driven requirements engineering approach that can also be used to address security requirements by means of anti-goals [19]. A linear real-time temporal logic is used to formalize goals. The goals and further ingredients such as domain properties as well as pre- and postconditions form patterns that can be instantiated and negated to describe anti-goals. This formal approach is also adopted by Secure Tropos [16].

SREF – Security Requirements Engineering Framework [2] is a framework that defines the notion of security requirements, considers security requirements

[3] C.f. `http://www.info.ucl.ac.be/~avl/ReqEng.html`

in an application context, and helps answering the question whether the system can satisfy the security requirements. Haley et al. [1] introduce the notion of a *trust assumption*, which is "an assumption by an analyst that the specification of a domain can depend on certain properties of some other domain in order to satisfy a security requirement". To decide whether a system can satisfy the security requirements, Haley et al. make use of structured informal and formal argumentation [3]. A two-part argument structure for security requirement satisfaction arguments consisting of an informal and a formal argument is proposed. In combination with trust assumptions, satisfaction arguments facilitate showing that a system can meet its security requirements.

In contrast to our work, the approaches by van Lamsweerde (including Secure Tropos) and SREF do not allow to express security requirements in terms of information flow properties. Moreover, the refinement of security requirements to specifications is not covered.

7 Conclusion and Future Work

The paper at hand constitutes an extension of the (C)SPF approach by a formal foundation and a pattern-based refinement analysis, which is heavily based on Santen's work on the preservation of security requirements under refinement [18]. In fact, we combined his proposals for a formal security requirements analysis approach with our hitherto informal (C)SPF approach.

The main benefits of the extension are the clear and unambiguous security requirements descriptions and the support of verifiably correct and confidentiality-preserving refinement steps.

In the future, we would like to consider probabilistic confidentiality properties, applicability conditions and environment patterns for (C)SPFs, and the compositionality of confidentiality-preserving refinement.

Acknowledgments. My sincere gratitude belongs to Thomas Santen for his great support on this work. Thomas gave feedback on the CSP artifacts, and helped to improve my understanding of confidentiality requirements as well as their formal specification. Furthermore, I thank Maritta Heisel for her extensive and valuable feedback on my work.

References

[1] Haley, C., Laney, R., Moffett, J., Nuseibeh, B.: Picking battles: The impact of trust assumptions on the elaboration of security requirements. In: Jensen, C., Poslad, S., Dimitrakos, T. (eds.) iTrust 2004. LNCS, vol. 2995, pp. 347–354. Springer, Heidelberg (2004)

[2] Haley, C.B., Laney, R., Moffett, J., Nuseibeh, B.: Security requirements engineering: A framework for representation and analysis. IEEE Transactions on Software Engineering 34(1), 133–153 (2008)

[3] Haley, C.B., Moffett, J.D., Laney, R., Nuseibeh, B.: Arguing security: Validating security requirements using structured argumentation. In: Proceedings of the 3rd Symposium on Requirements Engineering for Information Security (SREIS 2005) held in conjunction with the 13th International Requirements Engineering Conference (RE 2005) (2005)

[4] Hatebur, D., Heisel, M., Schmidt, H.: Security engineering using problem frames. In: Müller, G. (ed.) ETRICS 2006. LNCS, vol. 3995, pp. 238–253. Springer, Heidelberg (2006)

[5] Hatebur, D., Heisel, M., Schmidt, H.: A pattern system for security requirements engineering. In: Proceedings of the International Conference on Availability, Reliability and Security (AReS), pp. 356–365. IEEE, Los Alamitos (2007)

[6] Hatebur, D., Heisel, M., Schmidt, H.: A security engineering process based on patterns. In: Proceedings of the International Workshop on Secure Systems Methodologies using Patterns (SPatterns), pp. 734–738. IEEE, Los Alamitos (2007)

[7] Hatebur, D., Heisel, M., Schmidt, H.: Analysis and component-based realization of security requirements. In: Proceedings of the International Conference on Availability, Reliability and Security (AReS). IEEE Transactions, pp. 195–203. IEEE, Los Alamitos (2008)

[8] Hatebur, D., Heisel, M., Schmidt, H.: A formal metamodel for problem frames. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 68–82. Springer, Heidelberg (2008)

[9] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1986)

[10] Jackson, D.: Micromodels of software: Lightweight modelling and analysis with Alloy, `http://softwareabstractions.org/`

[11] Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)

[12] Lai, L., Lai, L., Sanders, J.W.: A refinement calculus for communicating processes with state. In: 1st Irish Workshop on Formal Methods: Proceedings, Electronic Workshops in Computing. Springer, Heidelberg (1997)

[13] Li, Z., Hall, J.G., Rapanotti, L.: From requirements to specifications: a formal approach. In: Proceedings of the International Workshop on Advances and Applications of Problem Frames (IWAAPF 2006), pp. 65–70. ACM, New York (2006)

[14] F. S. E. Limited. Failures-divergence refinement, FDR2 (2008)

[15] Mantel, H.: A Uniform Framework for the Formal Specification and Verification of Information Flow Security. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany (July 2003)

[16] Mouratidis, H., Giorgini, P.: Secure Tropos: A security-oriented extension of the Tropos methodology. International Journal of Software Engineering and Knowledge Engineering 17(2), 285–309 (2007)

[17] Nelson, M., Nelson, T., Alencar, P., Cowan, D.: Exploring problem-frame concerns using formal analysis. In: Proceedings of the International Workshop on Advances and Applications of Problem Frames (IWAAPF 2004), Edinburgh, Scotland, pp. 61–68. IET (2004)

[18] Santen, T.: Preservation of probabilistic information flow under refinement. Information and Computation 206(2-4), 213–249 (2008)

[19] van Lamsweerde, A.: Elaborating security requirements by construction of intentional anti-models. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), pp. 148–157. IEEE Computer Society Press, Los Alamitos (2004)

Architectural Refinement and Notions of Intransitive Noninterference[⋆]

Ron van der Meyden

School of Computer Science and Engineering,
University of New South Wales
meyden@cse.unsw.edu.au

Abstract. This paper deals with architectural designs that specify components of a system and the permitted flows of information between them. In the process of systems development, one might refine such a design by viewing a component as being composed of subcomponents, and specifying permitted flows of information between these subcomponents and others in the design. The paper studies the soundness of such refinements with respect to a spectrum of different semantics for information flow policies. These include Goguen and Meseguer's purge-based definition, Haigh and Young's intransitive purge-based definition, and some more recent notions TA-security, TO-security and ITO-security defined by van der Meyden. It is also shown that refinement preserves weak access control structure, an implementation mechanism that ensures TA-security.

1 Introduction

Architectural design is a high level of systems specification, concerned with identifying the components of a system and the patterns of their interaction. In this paper, we consider the relationship between information flow security policies and the architecture development process. We use ideas from the literature on information flow security to give semantics to architectures, and study how such semantics support a systems development process that refines high level architectural designs to more detailed architectures.

The type of information flow security policies that form the basis of this work place constraints on the permitted flows of information, or causal effects, between system components, and are referred to in the literature as *non-interference* policies. These policies can be represented as a binary relation on the set of components. For classical multi-level security policies, this relation is transitive. It has been proposed that extensions to multi-level security, such as downgraders, require that the policy be intransitive [21]. An architectural interpretation of intransitive noninterference policies is gaining increased prominence through such efforts as the MILS (Multiple Independent Levels of Security and Safety) approach to high-assurance systems design [1,24], which envisages the utilization

[⋆] Work supported by Australian Research Council Discovery grant DP0451529.

of recent advances in the efficiency of separation kernels to increase the degree of componentization of systems, enabling secure systems to be built from a mix of small, trusted and more complex, untrusted components [22], with global security properties assured from the separation property and a verification effort focussed on the trusted components.

During the process of system design, one may *refine* an architectural diagram by specifying internal structure for some of the systems components, breaking them down into sub-components, and specifying the permitted interferences of these sub-components with each other and with other components in the design. This leads to the following question: if one now builds a system according to the refined architecture, is it guaranteed to be compliant to the original architectural diagram? This property needs to hold in order for the process of architectural refinement to be sound for designs of information flow secure systems.

In this paper, we answer this question for architectural designs interpreted using several different semantics for intransitive noninterference policies. We give a formal definition of architectures and architectural refinement in Section 2. In Section 3, we recall Goguen and Meseguer's original semantics [6] for transitive policies, Haigh and Young's [8,21] semantics for intransitive noninterference policies and three alternate semantics for noninterference policies — TO-security, ITO-security and TA-security - introduced in [13]. In Section 4, these definitions are used to give semantics to architectures, enabling us to state precisely the result that architectural refinement is sound with respect to *all* these semantics for intransitive noninterference policies. Moreover, we consider access control as a mechanism for the implementation of information flow policies, in Section 5, and show that access control structure is also preserved under architectural refinement. Section 6 discusses related work on refinement for notions of security.

2 Architectural Refinement

We begin by formalising the notion of architecture, and defining a relation of refinement between architectures. In this section, we leave the question of the formal semantics of architectures open. In the following sections, we will show that architectural refinement is sound with respect to several different semantic interpretations of architectures.

The notion of architecture that we consider expresses only the highest levels of a system design. Intuitively, an architecture specifies that the system is comprised of a set of components that generate and hold information, and constrains the permitted flows of information between these components. Using terminology from the security literature, we refer to these components as domains.

Define an *architecture* to be a pair $\mathcal{A} = (D, \rightarrowtail)$, where D is a set of domains and $\rightarrowtail$ is a reflexive binary relation over D. We call the relation $\rightarrowtail$ the *information flow policy* of the architecture. Several related intuitions may be associated with this relation. One is that information is permitted to flow from a domain u to a domain v only if $u \rightarrowtail v$. Another is that actions in domain u may have directly observable effects in domain v only if $u \rightarrowtail v$. The relation is assumed to

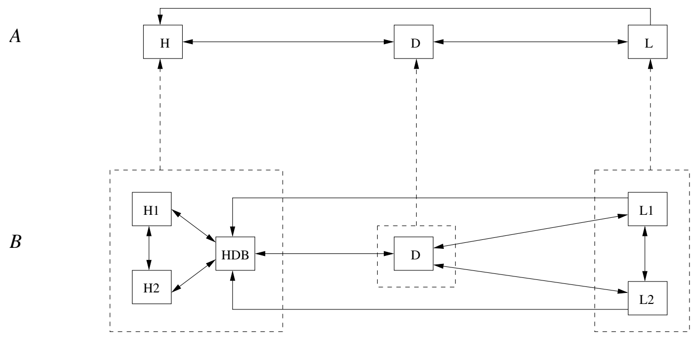


Fig. 1. Refinement of a downgrader architecture

be reflexive because information flow from a domain to itself can never be prevented. The security literature has frequently assumed information flow policies to be transitive. The following example illustrates a case where this assumption is not desirable.

Example 1. Consider the architecture $\mathcal{A} = (\{H, D, L\}, \rightarrowtail)$ where $\rightarrowtail$ is the downgrader policy depicted in the upper part of Figure 1 (we omit reflexive edges). Here H represents the high security domain, L the low security domain and D the downgrader. Information may flow from L to H, but any flow in the other direction needs to be mediated by the downgrader (so we would not want transtivity of $\rightarrowtail$). Information flow from L to D is permitted, to allow for requests by L to D for H information, e.g., freedom-of-information requests. □

The process of systems design may take a component in a high level design, and specify that it is to be implemented as the composition of a set of lower level components. In this design step, the permitted flows of information between the lower level components, and others in the design, should also be specified. One way to understand this process is to view the design step as establishing a relationship of *refinement* between low level and high level architectures.

Example 2. The architecture $\mathcal{B} = (\{H_1, H_2, HDB, D, L_1, L_2\}, \rightarrowtail)$ (where the policy $\rightarrowtail$ is depicted in the lower part of the diagram) represents a refinement of $\mathcal{A}$. We refine H into three components, two high level users H_1 and H_2 and a database HDB. Similarly, we refine L into two low level users L_1 and L_2. We assume that these may transmit information to each other, the high database, and D, but that all information flow from H_1 and H_2 to L_1 and L_2 is mediated by HDB and D. □

To formalise the notion of architectural refinement, let $\mathcal{A}_1 = (D_1, \rightarrowtail_1)$ and $\mathcal{A}_2 = (D_2, \rightarrowtail_2)$ be architectures. A *refinement mapping* from $\mathcal{A}_1$ to $\mathcal{A}_2$ is a function $r : D_1 \rightarrow D_2$ such that

1. r is *onto* D_2, and
2. for all $u, v \in D_1$, if $u \rightarrowtail_1 v$ then $r(u) \rightarrowtail_2 r(v)$.

In this case, we write $\mathcal{A}_1 \leq_r \mathcal{A}_2$, and say that $\mathcal{A}_1$ is a *refinement* of $\mathcal{A}_2$.

The requirement that r be surjective captures that intuition that if a design calls for the existence of a component, then a more detailed design should include an implementation of that component. The second condition expresses that the lower level policy should not permit information flow between two subdomains that was prohibited between their superdomains. That is, if in a high level design, information were not permitted to flow directly from component U to a component V, it should be incorrect to implement U and V using lower level components u and v, respectively, with information permitted to flow from u directly to v. (Note that the refinement mapping in Figure 1 satisfies this condition.)

We note that the definition of refinement is transitive in the following sense: if $\mathcal{A}_1 \leq_r \mathcal{A}_2$ and $\mathcal{A}_2 \leq_s \mathcal{A}_3$, then $\mathcal{A}_1 \leq_{s o r} \mathcal{A}_3$. This permits the development of an architectural design to proceed in a number of stages, each of which refines the previous, by guaranteeing that the final design is a refinement of the original design.

Our main result in this paper is that the process of replacing a high level architecture by a lower level refinement is sound design step, in the sense that any concrete system that implements the refined architecture also implements the higher level architecture. In order to make this claim formally precise, we first need to define what counts as a concrete implementation of an architecture. As a step towards this, we first consider a range of possible semantic interpretations of information flow policies. We use these in the semantics of architectures in Section 4.

3 Semantics of Information Flow Policies

To give semantics to architectures, we recall in this section several classical semantics for information flow policies [6,8,21], and several new definitions proposed by van der Meyden [13]. These definitions can be given for both state- and action-observed machines. We consider here the state-observed versions. The content of this section is largely definitional, and drawn from [13].

The *state-observed* machine model [21] for these definitions consists of deterministic machines of the form $\langle S, s_0, A, \texttt{step}, \texttt{obs}, \texttt{dom}\rangle$, where S is a set of states, $s_0 \in S$ is the *initial state*, A is a set of actions, $\texttt{dom} : A \rightarrow D$ associates each action to an element of the set D of security domains, $\texttt{step} : S \times A \rightarrow S$ is a deterministic transition function, and $\texttt{obs} : S \times D \rightarrow O$ maps states to an observation in some set O, for each security domain. We write $s \cdot \alpha$ for the state reached by performing the sequence of actions $\alpha \in A^*$ from state s, defined inductively by $s \cdot \epsilon = s$, and $s \cdot \alpha a = \texttt{step}(s \cdot \alpha, a)$ for $\alpha \in A^*$ and $a \in A$. Here ϵ denotes the empty sequence.

Transitive information flow policies have been given a formal semantics using a definition based on a "purge" function [6]. Given a set $E \subseteq D$ of domains and a sequence $\alpha \in A^*$, we write $\alpha \upharpoonright E$ for the subsequence of all actions a in α with $\texttt{dom}(a) \in E$. Given a policy $\rightarrowtail$, define the function $\texttt{purge} : A^* \times D \rightarrow A^*$ by

$$\texttt{purge}(\alpha, u) = \alpha \upharpoonright \{v \in D \mid v \rightarrowtail u\}.$$

For clarity, we may use subscripting of domain arguments of functions, writing e.g., $\texttt{purge}(\alpha, u)$ as $\texttt{purge}_u(\alpha)$.

Definition 1. *A system M is* P-secure *with respect to a policy $\rightarrowtail$ if for all sequences $\alpha, \alpha' \in A^*$ such that* $\texttt{purge}_u(\alpha) = \texttt{purge}_u(\alpha')$, *we have* $\texttt{obs}_u(s_0 \cdot \alpha) = \texttt{obs}_u(s_0 \cdot \alpha')$.

While P-security is well accepted for transitive policies, it has been felt to be inappropriate for intransitive policies, since it does not permit L to *ever* learn about H actions if the policy is $H \rightarrowtail D \rightarrowtail L$, even if D is intended to be a trusted downgrader of H information. To address this deficiency, Haigh and Young [8] generalised the definition of the purge function to intransitive policies as follows. Intuitively, the intransitive purge of a sequence of actions with respect to a domain u is the largest subsequence of actions that could form part of a causal chain of effects (permitted by the policy) ending with an effect on domain u. More formally, the definition makes use of a function $\texttt{sources} : A^* \times D \Rightarrow \mathcal{P}(D)$ defined inductively by $\texttt{sources}(\epsilon, u) = \{u\}$ and

$$\texttt{sources}(a\alpha, u) = \texttt{sources}(\alpha, u) \cup \{\texttt{dom}(a) \mid \exists v \in \texttt{sources}(\alpha, u)(\texttt{dom}(a) \rightarrowtail v)\}$$

for $a \in A$ and $\alpha \in A^*$. Intuitively, $\texttt{sources}(\alpha, u)$ is the set of domains v such that there exists a sequence of permitted interferences from v to u within α. The *intransitive purge* function $\texttt{ipurge} : A^* \times D \to A^*$ is then defined inductively by $\texttt{ipurge}(\epsilon, u) = \epsilon$ and

$$\texttt{ipurge}(a\alpha, u) = \begin{cases} a \cdot \texttt{ipurge}(\alpha, u) & \text{if } \texttt{dom}(a) \in \texttt{sources}(a\alpha, u) \\ \texttt{ipurge}(\alpha, u) & \text{otherwise} \end{cases}$$

for $a \in A$ and $\alpha \in A^*$. The intransitive purge function is then used in place of the purge function in Haigh and Young's definition:

Definition 2. *A system M is IP-secure with respect to a (possibly intransitive) policy $\rightarrowtail$ if for all sequences $\alpha \in A^*$, and $u \in D$, we have* $\texttt{obs}_u(s_0 \cdot \alpha) = \texttt{obs}_u(s_0 \cdot \texttt{ipurge}_u(\alpha))$.

These definitions are critiqued in [13], where it is shown that IP-security sometimes allows quite unintuitive flows of information. In response, several alternative definitions are proposed. Each is based on a concrete model of the maximal amount of information that a domain may have after some sequence of actions has been performed, and states that a domain's observation may not give it more than this maximal amount of information. The definitions differ in the modelling of the maximal information, and take the view that a domain increases its information either by performing an action or by receiving information transmitted by another domain.

In the first model of the maximal information, what is transmitted when an domain performs an action is information about the actions performed by other domains. The following definition expresses this in a weaker way than the ipurge function.

Given sets X and A, let the set $\mathcal{T}(X, A)$ be the smallest set $\mathcal{T}$ containing X and such that if $x, y \in \mathcal{T}$ and $z \in A$ then $(x, y, z) \in \mathcal{T}$. Intuitively, the elements of $\mathcal{T}(X, A)$ are are binary trees with leaves labelled from X and interior nodes labelled from A.

Given a policy $\rightarrowtail$, define, for each domain $u \in D$, the function $\mathtt{ta}_u : A^* \rightarrow \mathcal{T}(\{\epsilon\}, A)$ inductively by $\mathtt{ta}_u(\epsilon) = \epsilon$, and, for $\alpha \in A^*$ and $a \in A$,

1. if $\mathtt{dom}(a) \not\rightarrowtail u$, then $\mathtt{ta}_u(\alpha a) = \mathtt{ta}_u(\alpha)$,
2. if $\mathtt{dom}(a) \rightarrowtail u$, then $\mathtt{ta}_u(\alpha a) = (\mathtt{ta}_u(\alpha), \mathtt{ta}_{\mathtt{dom}(a)}(\alpha), a)$.

Intuitively, $\mathtt{ta}_u(\alpha)$ captures the maximal information that domain u may, consistently with the policy $\rightarrowtail$, have about the past actions of other domains. (The nomenclature is intended to be suggestive of *transmission* of information about *actions.*) Initially, a domain has information about what actions have been performed. The recursive clause describes how the maximal information $\mathtt{ta}_u(\alpha)$ permitted to u after the performance of α changes when the next action a is performed. If a may not interfere with u, then there is no change, otherwise, u's maximal permitted information is increased by adding the maximal information permitted to $\mathtt{dom}(a)$ at the time a is performed (represented by $\mathtt{ta}_{\mathtt{dom}(a)}(\alpha)$), as well the fact that a has been performed. Thus, this definition captures the intuition that a domain may only transmit information that it is permitted to have, and then only to domains with which it is permitted to interfere.

Definition 3. *A system M is TA-secure with respect to a policy $\rightarrowtail$ if for all domains u and all $\alpha, \alpha' \in A^*$ such that $\mathtt{ta}_u(\alpha) = \mathtt{ta}_u(\alpha')$, we have $\mathtt{obs}_u(s_0 \cdot \alpha) = \mathtt{obs}_u(s_0 \cdot \alpha')$.*

Intuitively, this says that each domain's observations provide the domain with no more than the maximal amount of information that may have been transmitted to it, as expressed by the functions $\mathtt{ta}$.

The notion of TA-security can be shown to be a better fit to the intended applications and theory of IP-security. On the other hand, it may still be too weak for some applications. For example, it considers to be secure a system where a downgrader transmits to L an email attachment that it received from H, without opening the attachment first (so that it does not know what information it is transmitting!) The second of van der Meyden's definitions is intended to address this potential deficiency.

The definition uses the following notion of *view*. The definition uses an absorptive concatenation function $\circ$, defined over a set X by, for $s \in X^*$ and $x \in X$, by $s \circ x = s$ if $s \neq \epsilon$ and x is equal to the final element of s, and $s \circ x = s \cdot x$ (ordinary concatenation) otherwise. Define the view of domain u with respect to a sequence $\alpha \in A^*$ using the function $\mathtt{view}_u : A^* \rightarrow O(A \cup O)^*$ (where O is the set of observations in the system), defined by

$$\mathtt{view}_u(\epsilon) = \mathtt{obs}_u(s_0), \text{ and}$$
$$\mathtt{view}_u(\alpha a) = \begin{cases} \mathtt{view}_u(\alpha)\, a\, \mathtt{obs}_u(s_0 \cdot \alpha) \text{ if } \mathtt{dom}(a) = u \\ \mathtt{view}_u(\alpha) \circ \mathtt{obs}_u(s_0 \cdot \alpha) \text{ otherwise} \end{cases}$$

That is, $\mathtt{view}_u(\alpha)$ is the sequence of all observations and actions of domain u in the run generated by α, compressed by the elimination of stuttering observations. Intuitively, $\mathtt{view}_u(\alpha)$ is the complete record of information available to domain u in the run generated by the sequence of actions α. The absorptive concatenation is intended to capture that the system is asynchronous, with domains not having access to a global clock. Thus, two periods of different length during which a particular observation obtains are not distinguishable to the domain.

Given a policy $\rightarrowtail$, for each domain $u \in D$, define the function $\mathtt{to}_u : A^* \rightarrow \mathcal{T}(O(A \cup O)^*, A)$ by $\mathtt{to}_u(\epsilon) = \mathtt{obs}_u(s_0)$ and

$$\mathtt{to}_u(\alpha a) = \begin{cases} \mathtt{to}_u(\alpha) & \text{when } \mathtt{dom}(a) \not\rightarrowtail u, \\ (\mathtt{to}_u(\alpha), \mathtt{view}_{\mathtt{dom}(a)}(\alpha), a) & \text{otherwise.} \end{cases}$$

Intuitively, this definition takes the model of the maximal information that an action a may transmit after the sequence α to be the fact that a has occurred, together with the information that $\mathtt{dom}(a)$ *actually* has, as represented by its view $\mathtt{view}_{\mathtt{dom}(a)}(\alpha)$. By contrast, TA-security uses in place of this the maximal information that $\mathtt{dom}(a)$ *may* have. (The nomenclature '`to`' is intended to be suggestive of *transmission* of information about *observations*.)

We will also consider a slight variant of this definition. Given a policy $\rightarrowtail$, for each domain $u \in D$, define the function $\mathtt{ito}_u : A^* \rightarrow \mathcal{T}(O(A \cup O)^*, A)$ by $\mathtt{ito}_u(\epsilon) = \mathtt{obs}_u(s_0)$ and

$$\mathtt{ito}_u(\alpha a) = \begin{cases} \mathtt{ito}_u(\alpha) & \text{when } \mathtt{dom}(a) \not\rightarrowtail u, \\ (\mathtt{ito}_u(\alpha), \mathtt{view}_{\mathtt{dom}(a)}(\alpha), a) & \text{if } \mathtt{dom}(a) = u. \\ (\mathtt{ito}_u(\alpha), \mathtt{view}_{\mathtt{dom}(a)}(\alpha a), a) & \text{otherwise.} \end{cases}$$

Intuitively, the definition of security based on this notion will allow that the action a transmits not just the information observable to $\mathtt{dom}(a)$ at the time that it is invoked, but also the new information that it computes and makes observable in $\mathtt{dom}(a)$. This information is not included in the value $\mathtt{ito}_{\mathtt{dom}(a)}(\alpha)$ itself, since the definition of security will state that the new observation may depend only on this value. The nomenclature in this case is intended to be suggestive of *immediate* transmission of information about observations.

We may now base the definition of security on either the function `to` or `ito` rather than `ta`.

Definition 4. *The system M is TO-secure with respect to $\rightarrowtail$ if for all domains $u \in D$ and all $\alpha, \alpha' \in A^*$ with $\mathtt{to}_u(\alpha) = \mathtt{to}_u(\alpha')$, we have $\mathtt{obs}_u(s_0 \cdot \alpha) = \mathtt{obs}_u(s_0 \cdot \alpha')$.*

The system M is ITO-secure with respect to $\rightarrowtail$ if for all domains $u \in D$ and all $\alpha, \alpha' \in A^*$ with $\mathtt{ito}_u(\alpha) = \mathtt{ito}_u(\alpha')$, we have $\mathtt{obs}_u(s_0 \cdot \alpha) = \mathtt{obs}_u(s_0 \cdot \alpha')$.

We remark that under certain circumstances, the definitions given by Roscoe and Goldsmith [19] correspond either to P-security or to ITO-security— see [12] for details. The following result shows how these definitions are related:

Theorem 1 ([13]). *With respect to a given policy $\rightarrowtail$, P-security implies TO-security implies ITO-security implies TA-security implies IP-security.*

Examples showing that all these notions are distinct are presented in [13].

4 Soundness of Architectural Refinement

We are now in a position to assign a formal meaning to architectures, and to state precisely the main result of the paper.

Let X-security be a notion of security for information flow policies, like those discussed in the previous section. We say that a system M is *X-compliant* with an architecture $\mathcal{A} = (D, \rightarrowtail)$, where X is a definition of security, if

1. D is the set of domains of M, and
2. M is X-secure with respect to $\rightarrowtail$.

Intuitively, M is X-compliant with an architecture if it has an implementation for each of the components required by the architecture, and the information flows between these implementation components is consistent with the architecture's policy (with consistency defined by X-security.)

Consider now the definition of architectural refinement introduced in Section 2. In a system design process we may have started with a high level architecture $\mathcal{A}$, refined this to a lower level architecture $\mathcal{B}$, and then constructed a system that implements $\mathcal{B}$. In what sense have we then implemented the architecture $\mathcal{A}$ that formed the highest level specification of the system? For this, we need to be able to view the system from the perspective of the set of domains of the higher level architecture $\mathcal{A}$ rather than those of $\mathcal{B}$. This is the intent of the following definition.

Let $M = \langle S, s_0, A, \mathtt{step}, \mathtt{obs}^1, \mathtt{dom}^1\rangle$ be a system with set of domains D_1, and suppose $r : D_1 \rightarrow D_2$ is surjective. Then we may construct a system $r(M) = \langle S, s_0, A, \mathtt{step}, \mathtt{obs}^2, \mathtt{dom}^2\rangle$ as follows:

1. the actions A, set of states S, initial state s_0, and transition function $\mathtt{step}$ of $r(M)$ are exactly as in M,
2. the set of domains of $r(M)$ is D_2
3. $\mathtt{dom}^2(a) = r(\mathtt{dom}^1(a))$ for each $a \in A$,
4. for $u \in D_2$ and $s \in S$, the observation $\mathtt{obs}^2_u(s)$ is the function $f : r^{-1}(u) \rightarrow O$, given by $f(v) = \mathtt{obs}^1_v(s)$ for $v \in r^{-1}(u)$.

Intuitively, each domain $u \in D_2$ is viewed by the refinement mapping as being broken down into the collection of subdomains $r^{-1}(u)$. An action of a subdomain in M is interpreted in $r(M)$ as belonging to its superdomain. The observation of a superdomain in $r(M)$ is taken to be the collection of all observations of its subdomains.

We can now give a formal meaning to soundness of architectural refinement. Say that a notion of compliance X is *preserved under architectural refinement* if whenever $r : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ is a refinement mapping, if M is a system that is X-compliant with $\mathcal{A}_1$, then $r(M)$ is X-compliant with $\mathcal{A}_2$. That, is, if the goal of design was to construct a system that complies with architecture $\mathcal{A}_2$, we may do so by building a system M that complies with architecture $\mathcal{A}_1$ and viewing this system from the perspective of $\mathcal{A}_2$ through the mapping r.

Our main result is the following.

Theorem 2. *The following notions are all preserved under architectural refinement: P-compliance, TO-compliance, ITO-compliance, TA-compliance and IP-compliance.*

This result means that any of these semantics can be adopted as the meaning of architectures, while retaining the ability to use architectural refinement as a sound design transformation.

5 Access Control

The interpretations of policies discussed above are somewhat abstract. We now consider a further, more concrete interpretation, that is closer to the level of *mechanisms* for control of information flow. One of the key mechanisms for implementation of secure systems is access control. Rushby [21] defined the notion of access control system and showed that access control systems are IP-secure with respect to a policy $\rightarrowtail$ if they are consistent with the policy in an appropriate sense. We present here a variant of Rushby's definitions due to van der Meyden, that strengthens Rushby's result (see [13] for an explanation of how this variant improves on Rushby's.)

Define an *access control structure* for a machine $\langle S, s_0, A, \texttt{step}, \texttt{obs}, \texttt{dom}\rangle$ with domains D to be a tuple $\texttt{AC} = (N, V, \texttt{contents}, \texttt{alter}, \texttt{observe})$, where

1. N is a set, of *names*,
2. V is a set, of *values*,
3. $\texttt{contents} : S \times N \rightarrow V$, with $\texttt{contents}(s, n)$ interpreted as the value of object n in state s,
4. $\texttt{observe} : D \rightarrow \mathcal{P}(N)$, with $\texttt{observe}(u)$ interpreted as the set of objects that domain u may observe, and
5. $\texttt{alter} : D \rightarrow \mathcal{P}(N)$, with $\texttt{alter}(u)$ interpreted as the set of objects whose values domain u is permitted to alter.

We call the pair $(M, \texttt{AC})$ a system with structured state. For a system with structured state, define for each domain $u \in D$, an equivalence relation on the states S of M, of *observable content equivalence*, by $s \sim^{\text{oc}}_u t$ if $\texttt{contents}(s, n) = \texttt{contents}(t, n)$ for all $n \in \texttt{observe}(u)$. That is, two states are related for u if they are identical with respect to the values of objects that u may observe.

The following conditions are a variant of Rushby's "Reference Monitor" conditions.

WAC1. If $s \sim^{\mathtt{oc}}_u t$ then $\mathtt{obs}_u(s) = \mathtt{obs}_u(t)$.
WAC2. For all actions a, states s, t and objects $n \in \mathtt{alter}(dom(a))$, if $s \sim^{\mathtt{oc}}_{\mathtt{dom}(a)} t$ and $\mathtt{contents}(s, n) = \mathtt{contents}(t, n)$ then $\mathtt{contents}(s \cdot a, n) = \mathtt{contents}(t \cdot a, n)$.
WAC3. If $\mathtt{contents}(s \cdot a, n) \neq \mathtt{contents}(s, n)$ then $n \in \mathtt{alter}(dom(a))$.

Intuitively, WAC1-WAC3 capture the conditions under which the machine operates in accordance with the intuitive interpretations of the structure $\mathtt{AC}$. WAC1 and WAC3 are identical to Rushby's RM1 and RM3, respectively. WAC1 says that a domain's observation depends only on the values of the objects observable to it. WAC2 (a modification of Rushby's condition RM2) says that if an action in domain u is permitted to change the value of an object n, then the new value of n depends only on its old value and the values of objects of domain u. WAC3 says that if an action can change the value of an object, then the domain of that action is in fact permitted to alter that object. We call the pair $(M, \mathtt{AC})$ a *weak access control system* if it satisfies WAC1-WAC3. We say that M *admits a weak access control interpretation* if there exists an access control structure $\mathtt{AC}$ such that $(M, \mathtt{AC})$ is a weak access control system.

Plainly, if there is an object that domain u may alter and domain v may observe, then information flow from domain u to v cannot be prevented. We say that a policy $\rightarrowtail$ is *consistent* with a weak access control system if the following condition is satisfied:

AOI. If $\mathtt{alter}(u) \cap \mathtt{observe}(v) \neq \emptyset$ then $u \rightarrowtail v$.

Write $\mathtt{uf}(M)$ for the *unfolding* of a system M: this is a bisimilar system in which states record the entire history, but which otherwise behaves exactly like M — we refer to [13] for details. The following result is shown in [13].

Theorem 3. *The following are equivalent*

1. *M is TA-secure with respect to $\rightarrowtail$,*
2. $\mathtt{uf}(M)$ *admits a weak access control interpretation consistent with $\rightarrowtail$.*

Moreover, if M admits a weak access control interpretation consistent with $\rightarrowtail$, then so does $\mathtt{uf}(M)$ *(hence M is TA-secure).*

This result shows that TA-security captures, in a precise sense, the notion of information flow security that may be enforced by access control mechanisms.

We now consider the interaction of refinement and access control structure. Suppose that M is a system for the set of domains D_1, with access control structure $\mathtt{AC}_1 = (N_1, V_1, contents_1, \mathtt{observe}_1, \mathtt{alter}_1)$ with respect to which M is a weak access control system. Let $r : (D_1, \rightarrowtail_1) \to (D_2, \rightarrowtail_2)$ be a refinement mapping. Define $r(\mathtt{AC}_1) = (N_2, V_2, \mathtt{contents}_2, \mathtt{observe}_2, \mathtt{alter}_2)$ to be the access control structure given by

1. $N_2 = N_1$ and $V_2 = V_1$ and $\mathtt{contents}_2 = \mathtt{contents}_1$,
2. $\mathtt{observe}_2(u) = \cup_{v \in r^{-1}(u)} \mathtt{observe}_1(v)$,
3. $\mathtt{alter}_2(u) = \cup_{v \in r^{-1}(u)} \mathtt{alter}_1(v)$.

Intuitively, $\mathtt{AC}_2$ has the same set of objects, with the same contents at each state of M as in $\mathtt{AC}_1$, and each domain of D_2 observes and alters the objects in all of its subdomains in D_1. The following result shows that weak access control structure is preserved under architectural refinement.

Theorem 4. *Let $r : (D_1, \rightarrowtail_1) \rightarrow (D_2, \rightarrowtail_2)$ be a refinement mapping.*

1. *If $(M, \mathtt{AC}_1)$ is a weak access control system then $(r(M), r(\mathtt{AC}_1))$ is a weak access control system.*
2. *If $\mathtt{AC}_1$ is consistent with $\rightarrowtail_1$ then $r(\mathtt{AC}_1)$ is consistent with $\rightarrowtail_2$.*

It is worth noting that we could also work with a more refined notion of access control structure in which the functions `alter` and `observe` take as inputs the *actions* of a system M (rather than the domains.) Intuitively, this amounts to specifying constraints on the objects that each action is permitted to read and write. We will call an access control structure of this type an *action-based* access control structure, and refer to the former type as a *domain-based* access control structure.

Given $\mathtt{AC} = (N, V, \mathtt{contents}, \mathtt{observe}, \mathtt{alter})$, an action-based access control structure over actions A, and a domain mapping $\mathtt{dom} : A \rightarrow D$, we may construct $\mathtt{AC}/\mathtt{dom} = (N, V, \mathtt{contents}, \mathtt{observe}', \mathtt{alter}')$, a domain-based access control structure, by defining

$$\mathtt{observe}'(u) = \cup\{\mathtt{observe}(a) \mid a \in A,\ \mathtt{dom}(a) = u\}$$

and

$$\mathtt{alter}'(u) = \cup\{\mathtt{alter}(a) \mid a \in A,\ \mathtt{dom}(a) = u\}.$$

Given a system M, with domain u, let $\sim_u^{\mathtt{oc}}$ be the relation defined above, with respect to $\mathtt{AC}/\mathtt{dom}$, and for an *action* a define the relation $\sim_a^{\mathtt{oc}}$ on the states of M by $s \sim_a^{\mathtt{oc}} t$ if $\mathtt{contents}(s, n) = \mathtt{contents}(t, n)$ for all $n \in \mathtt{observe}(a)$.

The semantic conditions defining an action-based access control system can now be stated as a variant of WAC1-WAC3.

WAC1$_a$. For $u \in D$, if $s \sim_u^{\mathtt{oc}} t$ then $\mathtt{obs}_u(s) = \mathtt{obs}_u(t)$.
WAC2$_a$. For all actions a, states s, t and names $n \in \mathtt{alter}(a)$, if $s \sim_a^{\mathtt{oc}} t$ and $\mathtt{contents}(s, n) = \mathtt{contents}(t, n)$ then $\mathtt{contents}(s{\cdot}a, n) = \mathtt{contents}(t{\cdot}a, n)$.
WAC3$_a$. If $\mathtt{contents}(s \cdot a, n) \neq \mathtt{contents}(s, n)$ then $n \in \mathtt{alter}(a)$.

Intuitively, WAC1$_a$ says that the observations of a domain depend only on the contents of objects that actions in that domain may observe. The remaining conditions are similar to the domain-based versions, except that we work at the level of actions rather than domains.

Proposition 1. *If M satisfies WAC1$_a$-WAC3$_a$ with respect to the action-based access control structure* $\mathtt{AC}$ *and $\rightarrowtail$, then M satisfies WAC1-WAC3 with respect to* $\mathtt{AC}/\mathtt{dom}$ *and $\rightarrowtail$.*

Example 3. To illustrate the interaction of architectural refinement and the implementation of architectures using access control structure, we reconsider the refinement of Example 2. As a further step of refinement towards the implementation of the system, we choose to implement architecture $\mathcal{B}$ using an action-based access control structure $(N, V, \texttt{contents}, \texttt{observe}, \texttt{alter})$. The set N consists of the following objects:

1. local states $l_1, l_2, h_1, h_2, d, hdb$ for $L_1, L_2, H_1, H_2, D, HDB$, respectively,
2. high level files f_1, f_2
3. input buffers hin, din_h, din_l, lin for messages to H, D and L. (In the case of D, we have separate buffers din_h and din_l to receive communications from H and L respectively — this allows the sender to receive acknowledgements without creating a covert channel from H to L.)

The table in Figure 2 gives the actions associated to each domain, and the functions `observe` and `alter`.

Domain	Actions	`observe`	`alter`	Purpose
$L_i,\ i = 1, 2$	request(L_i)	l_i	din_l, l_i	request information from D
	send(L_i, H)	l_i	hin, l_i	send information to H
	get(L_i)	lin	lin, l_i	read L input buffer
	internal(L_i)	l_i	l_i	local computation
D	get$_h(D)$	din_h	din_h, d	read D input buffer from H
	get$_l(D)$	din_l	din_l, d	read D input buffer from L
	query(D)	d	d, hdb	send query to H database
	respond(D)	d	d, lin	send response to L request
	internal(D)	d	d	local computation
$H_i,\ i = 1, 2$	request(H_i)	h_i	h_i, hdb	send query/update to HDB
	internal(H_i)	h_i	h_i	local computation
HDB	get(HDB)	hin	hin, hdb	read H input buffer
	respond(HDB, H_i)	hdb, f_1, f_2	h_i, hdb	respond to H_i request
	respond(HDB, D)	hdb, f_1, f_2	d, hdb	respond to D request
	internal(HDB)	hdb, f_1, f_2	hdb, f_1, f_2	local computation

Fig. 2. Actions of an access control system

Finally, we define observations in the system using the pairs $(L_i, \{l_i\})$, $(D, \{d\})$, $(H_i, \{h_i\})$, $(HDB, \{hdb, f_1\})$. Here the first component of a pair gives u, and $O_u(s)$ is defined to be the sequence of values $\texttt{contents}(s, n)$ where n is an element of the second component. Note that we have not made f_2 observable - this might represent, e.g., that f_2 is used for internal data structures of the database and is not visible at the interface of the database.

It is straightforward to check that this action-based access control structure induces a domain-based access control structure that is consistent with the policy of $\mathcal{B}$. Thus, by Theorem 3, any system M for this access control structure that satisfies the conditions WAC1$_a$-WAC3$_a$ is TA-compliant with $\mathcal{B}$. Further, using either the preservation of TA-compliance under refinement (Theorem 2), or the preservation of access control structure under refinement (Theorem 4) and then Theorem 3, it also follows that $r(M)$ is TA-compliant with $\mathcal{A}$. Here $r(M)$ is the system where the observations are defined by the pairs $(L, \{l_1, l_2\})$, $(D, \{d\})$ and $(H, \{h_1, h_2, hdb, f_1\})$.

Note that this result applies to a *class* of systems, as we have not yet defined a unique system. To do so we would need to also give the values V, and define the effect that actions have on states. Once this is done, one way to ensure WAC2$_a$ and WAC3$_a$ might be by using static analysis to verify that the code implementing an action a reads only from `observe`(a) and writes only to `alter`(a). $\square$

6 Related Work

To the best of our knowledge, our work is the first consideration of the relationship between architectural refinement and intransitive noninterference. However, both formal theories of architecture refinement and refinement of noninterference security properties have been presented in the past.

In general, work on architectural refinement [18,2] is concerned with behavioural notions of refinement, and has not taken security into account. An interesting exception is a sequence of papers by Moriconi *et al.*, [16,14], who develop a very abstract formal account of architecture refinement in which architectural designs are represented as logical theories and refinement is treated as a mapping of the symbols of the abstract theory to those of the concrete theory that must satisfy the logical condition of being a *faithful interpretation*. In order to apply this account to a particular type of architectural design notation, it is necessary to concretize the abstract theory by giving both a syntax for the architectural elements in the notation, and to develop a logical theory that represents the semantics of this notation. (E.g. this is done in [14] for dataflow and shared-memory architectural styles.) In [15], the framework is applied to establish security properties of a number of secure variants of the X/Open Distributed Transaction Processing architecture. The security policy considered here is the Bell La-Padula policy [3], which lacks the kind of information flow semantics that we have studied here, although it can be related to noninterference for transitive policies [21]. It is not clear whether a concretization of the Moriconi *et al.* theory could be developed that would enable it to represent the content of our results, but this would be an interesting topic for further study. Zhou and Alves-Foss [25] have also proposed a number of architecture refinement patterns for Multi-Level Secure systems development, but do not provide any formal semantics for their work.

That we have obtained positive results concerning refinement of security properties may be surprising to those familiar with the literature on formal security properties, where it is folklore that such properties are *not* preserved under refinement [9], a fact known as the “refinement paradox”. However, our notion of refinement differs from the notions of refinement usually studied. Refinement is usually understood as a reduction in the set of possible behaviours of the system, which would be contrary to our assumptions that systems are *input-enabled* (actions always enabled) and *deterministic*.

It is possible to identify conditions under which reduction of the possible behaviours of a system preserves information flow security properties. Jacob [9] presents a method in which an insecure system is first developed using a standard refinement methodology for functional properties, then made secure by a further deletion of behaviors in a fixpoint calculation. It is not guaranteed that this last step terminates, nor that it produces a useful system. Mantel [11] defines refinement operators that take as input a secure system, a set of transitions to be disabled, and a type of unwinding relation on the system that establishes the security property. The operators produce as output a refined system, as well as a new unwinding relation that establishes the security of the refined system. This is achieved by either disabling transitions other than those requested, or by maintaining some transitions whose disablement was requested. He considers a richer notion of information flow policy than we have treated, but with respect to a semantics that seems appropriate only for transitive policies. The practicality of these approaches has not been established.

A number of authors have also identified sufficient conditions under which data-refinement preserves transitive information flow policies [7,17]. Bossi *et al.* [5] develop conditions under which refinement of process algebra terms preserves bisimulation-based information flow security properties using a simulation-based notion of refinement. Only the two-domain policy $L \rightarrowtail H$ is considered in this work. Roscoe [20] defines *Low-determinism*, a very strong notion of security, which is always preserved under refinement, but at the cost of a significantly restricted range of applicability. Some recent works have also sought to overcome the refinement paradox by drawing a distinction between specification-level non-determinism and non-determinism that is inherent in a system, with the latter preserved under refinement [23,10,4]. These works also typically have confined their attention to the two-domain policy.

References

1. Alves-Foss, J., Harrison, W.S., Oman, P., Taylor, C.: The MILS architecture for high-assurance embedded systems. International Journal of Embedded Systems 2(3/4), 239–247 (2006)
2. Barbosa, M.A.: A refinement calculus for software components and architectures. ACM SIGSOFT Software Engineering Notes, 30(5) (September 2005)
3. Bell, D.E., La Padula, L.J.: Secure computer system: unified exposition and multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, M.A. (March 1976)
4. Bibighaus, D.: Applying the doubly labeled transition system to the refinement paradox. PhD thesis, Naval Postgraduate School, Monterey (2006)
5. Bossi, A., Focardi, R., Piazza, C., Rossi, S.: Refinement operators and information flow security. In: Proc. Int. Conf. on Software Engineering and Formal Methods, pp. 44–53 (2003)
6. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. IEEE Symp. on Security and Privacy, Oakland, pp. 11–20 (1982)
7. Graham-Cunning, J., Sanders, J.: On the refinement of noninterference. In: Proc. IEEE Computer Security Foundations Workshop, pp. 35–42 (1991)

8. Haigh, J.T., Young, W.D.: Extending the noninterference version of MLS for SAT. IEEE Trans. on Software Engineering SE-13(2), 141–150 (1987)
9. Jacob, J.: On the derivation of secure components. In: Proc. IEEE Symp. on Security and Privacy, pp. 242–247 (1989)
10. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
11. Mantel, H.: Preserving information flow properties under refinement. In: Proc. IEEE Symp. Security and Privacy, pp. 78–91 (2001)
12. van der Meyden, R.: A comparison of semantic models of intransitive noninterference (submitted for publication) (December 2007), `http://www.cse.unsw.edu.au/~meyden`
13. van der Meyden, R.: What, indeed, is intransitive noninterference? (submitted for publication, copy at `http://www.cse.unsw.edu.au/~meyden` – an extended abstract of this paper appears in Proc. ESORICS 2007) (January 2008)
14. Moriconi, M., Qian, X.: Correctness and composition of software architectures. In: Proc. 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 164–174 (1994)
15. Moriconi, M., Qian, X., Riemenschneider, R.A., Gong, L.: Secure software architectures. In: Proc. IEEE Symp. on Security and Privacy, pp. 884–893 (1997)
16. Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct architecture refinement. IEEE Transactions on Software Engineering 21(4), 356–372 (1995)
17. O'Halloran, C.: Refinement and confidentiality. In: Fifth Refinement Workshop, pp. 119–139. British Computer Society (1992)
18. Philipps, J., Rumpe, B.: Refinement of information flow architectures. In: Proc. 1st IEEE Int. Conf. on Formal Engineering Methods, pp. 203–212 (1997)
19. Roscoe, A.W., Goldsmith, M.H.: What is intransitive noninterference? In: IEEE Computer Security Foundations Workshop, pp. 228–238 (1999)
20. Roscoe, A.W.: CSP and determinism in security modelling. In: Proc. IEEE Symp. on Security and Privacy, pp. 114–221 (1995)
21. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International (December 1992)
22. Rushby, J.M., Randell, R.: A distributed secure system. IEEE Computer 16(7), 55–67 (1983)
23. Seehusen, F., Stolen, K.: Information flow property preserving transformation of UML interaction diagrams. In: Proc. ACM symposium on access control models and technologies, pp. 150–159 (2006)
24. Vanfleet, W.M., Beckworth, R.W., Calloni, B., Luke, J.A., Taylor, C., Uchenick, G.: MILS:architecture for high assurance embedded computing. Crosstalk: The Journal of Defence Engineering, 12–16 (August 2005)
25. Zhou, J., Alves-Foss, J.: Architecture-based refinements for secure computer system design. In: Proc. Policy, Security and Trust (November 2006)

Systematically Eradicating Data Injection Attacks Using Security-Oriented Program Transformations

Munawar Hafiz, Paul Adamczyk, and Ralph Johnson

University of Illinois at Urbana-Champaign
201 N Goodwin Avenue, Urbana, IL 61801, USA
`{mhafiz,padamczy,rjohnson}@illinois.edu`

Abstract. Injection attacks and their defense require a lot of creativity from attackers and secure system developers. Unfortunately, as attackers rely increasingly on systematic approaches to find and exploit a vulnerability, developers follow the traditional way of writing ad hoc checks in source code. This paper shows that security engineering to prevent injection attacks need not be ad hoc. It shows that protection can be introduced at different layers of a system by systematically applying general purpose security-oriented program transformations. These program transformations are automated so that they can be applied to new systems at design and implementation stages, and to existing ones during maintenance.

1 Introduction

Keeping systems secure is a game with a moving target. There are no bounds on either the creativity of attackers in finding new vulnerabilities, or the creativity of secure system developers in writing patches to remove them. Since the same vulnerability affects many types of software, attackers can use various tools and be systematic in their approach. In contrast, writing patches is an ad hoc, manual activity. For example, Bugtraq [5] lists 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors in the first week of September 2008. At least 17 of these incidents occur due to the use of unsafe functions and the common fix is to validate the input or replace the function. Despite the common problem and solution, there are no general purpose tools that these developers can use. Moreover, developers work under time pressure and write patches that are specific to a reported vulnerability. For example, Bugtraq lists 32 instances of data injection vulnerabilities in Microsoft Word 2000. Each time developers created a patch to remove the specific vulnerability, only to let the same vulnerability resurface later in another part of the program. The ad hoc nature of security engineering gives an attacker an advantage.

This paper shows that it is possible to think of applying security in terms of general purpose program transformations. Our previous work [14] introduced the concept of security-oriented program transformations and how using transformations could overcome the problems of traditional approaches of security engineering. This paper describes how to use security-oriented program transformations to protect systems from data injection attacks.

OWASP [28] lists 18 variants of data injection attacks that target different programming environments, e.g. SQL injection attack affects a database access language, LDAP injection attack affects a directory access protocol, cross site scripting (XSS) injects malicious code through an HTTP payload, etc. Buffer overflow attack and format string attack are also variants of injection attacks. Overall, injection attacks are the source of most attacks, posing the most insidious threat to modern software.

Current works on data injection attacks focus on three broad classes of attacks: SQL injection, cross site scripting, and buffer overflow. Many sophisticated static and dynamic analysis tools are available for automatically detecting instances of these data injection vulnerabilities in the source code. Most of these solutions stop one step short of solving the problem, because they still require a programmer to manually add security checks in the program to prevent the attack. People are bad at repetitive manual tasks – computers are much better.

We are interested in automated, general purpose program transformations that remove security threats from programs. This paper introduces eight program transformations to combat data injection attacks and describes three of them in detail. The first one is an architectural transformation that centralizes input checks at the system entry point. The second is a program-level transformation that applies multiple policies to validate data variables. Our third example, following the defense in depth principle [36], replaces the use of unsafe library functions that cause various injection attacks with safe functions. In all these transformations, developers specify the *policy* (e.g. specify where to add an input validation component, which filters to add to validate input, which unsafe functions to replace), while the *mechanism* is executed automatically by the transformation.

This paper makes the following contributions:

- We present a suite of security-oriented program transformations to counter data injection attacks.
- We differentiate the steps of applying a program transformation that are part of an automated tool (mechanism) from the steps that are done manually (policy). This separation of policy from mechanism makes eliminating security threats more systematic.
- We demonstrate these security-oriented program transformations by building proof-of-concept tools and using them to remove injection vulnerabilities from existing programs.

The next section introduces security-oriented program transformations. Then we describe the security-oriented program transformations that add protection against data injection attacks. We discuss three example program transformations in detail. This is followed by a discussion on the diversity of input validation policies supported by these transformations. Finally, we discuss how general purpose tools can be built to automate the program transformations.

2 Security-Oriented Program Transformation

A program transformation is a function that maps programs to programs. A *security-oriented program transformation* maps programs to security-augmented programs, i.e. it introduces a security solution to make programs more secure. We have compiled forty four security-oriented program transformations [14]. For their short description, see `http://netfiles.uiuc.edu/mhafiz/www/sopt.pdf`

Program transformations are "general purpose", but no transformation will work with every program. They usually expect a certain programming language or a certain platform, because the exact details are language-specific. By general purpose transformations, we mean that a transformation for a certain programming language should be applicable to all programs written in that language.

Our security-oriented program transformations make structural changes that do not depend on detailed understanding of the application logic. This property makes them similar to *refactoring* [12], altering the internal structure of code without changing its external behavior. Both refactoring and security-oriented program transformations are types of program transformations. However, security-oriented program transformations are not behavior-preserving the way refactorings are. A security-oriented program transformation preserves the correct behavior and fixes the incorrect behavior caused by a security vulnerability. Our program transformations are behavior-preserving when the system is used correctly; they preserve the good path behavior. Only attackers see change in the behavior, because security-oriented program transformations eliminate the source of vulnerabilities the attackers want to exploit.

Security-oriented program transformations could be automated, because they separate policy from mechanism. To use a security-oriented program transformation, a developer has to follow three steps–1) identify the program points where to apply a transformation, 2) determine which transformation to apply, and 3) use a tool to automatically transform the program. The first two tasks are manual; a developer identifies where to apply a transformation and which transformation to apply. Usually, a developer supplies these parameters to a program transformation with a manual specification. For example, a program transformation that adds checks to validate input variables requires that the developer specify input validation policies. The transformation does not determine which inputs to validate nor which input validation policies to apply. A developer manually specifies these. A developer's specification is the *policy*. The tools implement the *mechanism*; they automatically execute structural changes. With this separation, program transformations can automate new security protections without encoding deep understanding of program behavior in the transformation.

All of our program transformations separate policy from mechanism in this way. For each program transformation, we describe what a developer has to specify and what the tool does to apply the changes.

Each program transformation is different; so is the implementation strategy of a transformation tool. The strategy depends on the context (language and/or platform). This paper focuses on describing the mechanism of each program transformation. We comment on the implementation issues in section 8.

3 Program Transformations to Prevent Injection Attacks

This paper concentrates on eight program transformations (from our list of forty four[1]) that could be used to prevent various types of data injection attacks. Table 1 lists these transformations. Our description of the mechanics of each transformation has two parts: 1) what a developer needs to specify and 2) what structural changes are made by the transformation.

Suppose Alice, a developer who wants to secure a system from SQL injection attacks, has these transformations available. She need not write input validation checks manually. These checks follow some input validation policies to determine if an input is valid; some actively rectify an input to make it valid[2]. With program transformations, Alice can specify what validation policies to apply, while a transformation would automatically add filters that implement the policies.

Suppose Alice wants to factor out validation policies that are common for all inputs and apply them on data as they enter the system. She would specify the system entry points and apply an *Add Perimeter Filter* architectural transformation that would create a policy enforcement point to validate all incoming inputs. Some policies are specific to each input. Both common and input-specific policies would have to be applied to an input variable. A *Decorated Filter* transformation adds a series of filters to an input variable to apply a group of policies.

It might be impossible to completely filter all malicious inputs. In that case, Alice could use safe functions that are designed to robustly handle malicious data. She could apply a *Safe Library Replacement* transformation to replace all instances of unsafe library functions in a program with safe library functions.

These transformations modify user inputs which might lead a program to an invalid state. Alice could apply an *Exception Shielding* transformation to add exceptions to handle the aberrant states. The transformation additionally replaces the error messages with a generic error message to hide internal information.

Other transformations limit the consequences of injection attacks. To prevent data corruption of important files, Alice could apply a *chroot Jail* transformation to run a program in a constrained environment. Some data corruption attacks originate when multiple processes write to a system file with a faulty file locking mechanism. Here, Alice could use an *Unique Location for each Write Request* transformation that modifies a program to use individual files instead of sharing.

Finally, a system should securely keep log and audit data for forensics. Alice could apply two architectural transformations, *Add Audit Interceptor* and *Secure Logger*, to introduce components that perform these tasks securely and reliably.

Some of our transformations are architectural, while some apply at the program level. Some are programming language or platform specific, e.g. an *Exception Shielding* transformation applies to programs written in languages that support exceptions. Some program transformations apply to the source code, while some apply to binaries. Some transformations are less conventional. For example, a *chroot Jail* transformation makes very few changes to a program's

[1] The list is available at `http://netfiles.uiuc.edu/mhafiz/www/sopt.pdf`

[2] We use the term validation to mean both active and passive validation.

Table 1. Security-oriented program transformations to prevent injection attacks

Name	Problem	Mechanics of Transformation
1. Add Audit Interceptor	How can you make it easy to add and change auditing events?	Developer specifies where to intercept data to audit, and how to create audit data. Transformation adds a component that intercepts requests and responses, and creates audit events.
2. Add Perimeter Filter	How can you enforce input validation policies on incoming data?	Developer specifies where the input is checked and what are validation policies. Transformation adds a policy enforcement component and delegates requests to it.
3. chroot Jail	How can you prevent an attacker from corrupting important files ?	Developer specifies the constrained environment for a program. Transformation creates a jail environment for a process and runs it inside a chroot jail.
4. Decorated Filter	How can you apply multiple input validation policies?	Developer specifies the target input and validation policies. Transformation adds a decorator [13] to the input.
5. Exception Shielding	How can you preserve application behavior when rectified user inputs cause an unexpected state?	Developer specifies exception type and insertion point. Transformation inserts exception, and obfuscates the error message produced by the exception.
6. Safe Library Replace- -ment	How can you prevent injection attacks when sanity checks fail to sufficiently validate inputs and the function that uses the inputs are also vulnerable?	Developer specifies the unsafe functions and safe alternatives. Transformation searches and replaces unsafe functions with safe functions.
7. Secure Logger	How can you ensure that system events are logged timely and in a secure manner?	Developer specifies the messages to log, and policies to retain confidentiality and integrity. Transformation adds a logging component that encrypts and signs logged data.
8. Unique Location for each Write Request	How can you prevent data corruption caused by insufficient locking mechanism when multiple processes write to the same file?	Developer specifies the section of a program that writes to a shared file and new file creation policy. Transformation modifies the write request so that a new file is created for each write request.

source code or binary. Instead, it transforms the runtime environment of a program to create a jail environment and then run the program inside the jail.

The next three sections describe three example security-oriented program transformations: *Add Perimeter Filter*, *Decorated Filter* and *Safe Library Replacement.* To describe the transformations, we have augmented Fowler's format [12] for describing refactoring.

4 Add Perimeter Filter Transformation

You want to scan and check incoming data for malicious content before it is used in the system.

Add a policy enforcement point at the system entry point to validate data.

4.1 Motivation

For every application, there are some input validation policies that are common for all inputs. Writing checks for these policies for every input variable duplicates a lot of code. One way of eliminating duplication is to move the checks at the application entry point inside a policy enforcement point component.

However, there are policies that are either input-specific or require knowledge of the business logic; these policies cannot be factored out.

4.2 Pre-condition

A program with the following characteristics benefits from this transformation.

- There are systemwide policies applicable to all incoming data and the policies can be enforced without knowledge of the business logic.
- Program has a few entry points. Before applying this transformation, a developer could apply a *Single Access Point* transformation (part of our larger catalog of forty four transformations) to minimize the number of entry points. A *Single Access Point* transformation makes a Façade [13] in the object-oriented world or introduces a wrapper component with the same API.

4.3 Mechanics

A developer specifies where to insert a perimeter filter and what policies to apply.

The program transformation adds a secure base action component [32] that acts as a policy enforcement point [3]. A secure base action component centralizes authentication, authorization and input validation functionality; here we concentrate on input validation. The transformation creates a policy validation component [32] that contains the filters implementing input validation policies. The program transformation adds a request at the entry point to delegate input validation to the secure base action component which then passes the input through the filters to validate the input.

4.4 Example

Figure 1 shows the transformation applied to a Java program. A developer specifies the insertion point which, in this example, is a JSP or Servlet front controller [1]. A developer also specifies a list of policies.

The transformation creates the `SecureBaseAction` class as a policy enforcement point and the `InterceptingFilter` class that contains the `Filter`s. User inputs are passed to `SecureBaseAction`, which validates by passing inputs through a series of filters.

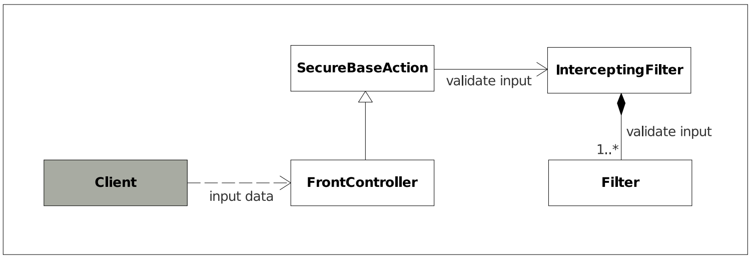


Fig. 1. Applying *Add Perimeter Filter* transformation

4.5 Towards a Perimeter Filter Transformation Tool

A special tool is not needed to perform tasks like adding a new class, creating a superclass and delegating tasks. These are examples of refactorings [12] for which tools are already available in many programming platforms.

The part that is harder to automate is creating filters from user specification. A tool should contain a library of filters that can be used directly or customized using parameters. A developer should also be able to extend the library. Section 7 discusses the various types of policies that could be applied.

5 Decorated Filter Transformation

You want to apply multiple policies to an input variable.

Validate an input variable by decorating it with a series of filters implementing the policies.

5.1 Motivation

Programmers determine a comfort zone [29] of inputs and write checks to prevent inputs outside the zone. Mistakes in writing checks is very common. In the first week of September 2008, Bugtraq lists 34 SQL injection and 21 XSS attack instances that could be solved by more stringent input validation.

Replacing manual checking with automated tools increases programmer efficiency because they can concentrate on policies rather than the mechanism of implementing checks.

5.2 Preconditions

This transformation applies to object-oriented programs. A program with the following characteristics benefits from a *Decorated Filter* transformation.

- The program has injection vulnerabilities originating from unsafe inputs. Inputs are incompletely or incorrectly checked.
- There are attack patterns that can be used as a basis to implement the validation policies.
- Rectification policies do not transform valid inputs.

5.3 Mechanics

A developer specifies the target input variable and the policies to be applied.

The program transformation adds the policies to an input variable by using Decorators [13]. Figure 2 describes how a string variable is decorated with policies that remove SQL injection attack vectors. The string input variable becomes encapsulated in the abstract `AbstractStringContainer` class. Its concrete instance `UnsafeString` becomes the target of input validation policies.

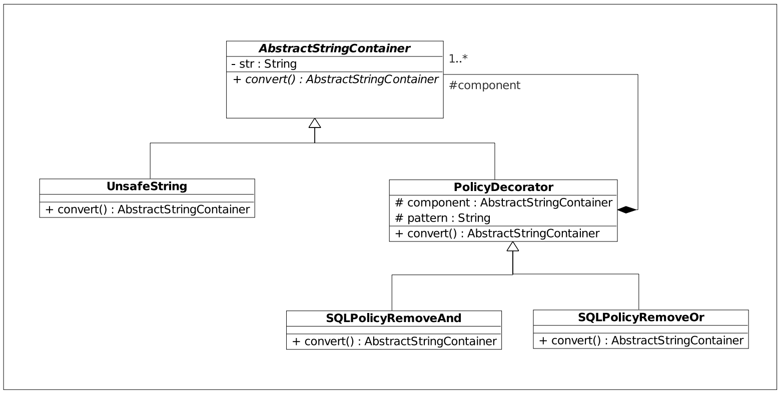

AbstractStringContainer
- str : String
+ convert() : AbstractStringContainer
1..*
#component
UnsafeString
+ convert() : AbstractStringContainer
PolicyDecorator
# component : AbstractStringContainer
# pattern : String
+ convert() : AbstractStringContainer
SQLPolicyRemoveAnd
+ convert() : AbstractStringContainer
SQLPolicyRemoveOr
+ convert() : AbstractStringContainer

Fig. 2. Class diagram describing policies to apply on an unsafe string

5.4 Example

We have written a proof-of-concept Eclipse plugin to apply SQL injection prevention policies to Java programs. A programmer specifies the variable to validate as well as the policies. We illustrate the transformation with an insecure program.

An insecure program instance. Class `DBConnect` contains a method for querying a database and showing the result (`showData`). The `showData` method reads input from standard input (line 7), and prepares the query and executes it (lines 9–16). The database contains a single table named `users`, with three fields for storing user id, user name and password.

```
1   import java.sql.*;
2
3   public class DBConnect {
4     ...
5     public void showData() {
6       ...
7       String username = stdin.readLine();
8       ...
9       try {
10        stmt = connection.createStatement();
11        resultSet = stmt.executeQuery("select * from users " +
12              "where username = '" + username + "'");
13      } catch (SQLException e) {
14        e.printStackTrace();
15        }
16      }
17  }
```

Attacking this program is straightforward. A malicious user enters as input the string “’ or ‘1’=‘1”. The resulting query is,

select * from users where username = ‘’ or ‘1’=‘1’

Applying the transformation. In the example program, we have applied policies to remove AND and OR from user inputs (see figure 2).

The resulting code is shown here with the changes highlighted.

```
import java.sql.*;
import model.PolicyDecorator;
import model.UnsafeString;
import model.sqlpolicy.SQLPolicyRemoveOr;
import model.sqlpolicy.SQLPolicyRemoveAnd;

public class DBConnect {
  ...
  public void showData() {
    ...
    UnsafeString username = new UnsafeString();
    try {
      username.setStr(stdin.readLine());
      } catch (IOException e1) {
    ...
    try {
      stmt = connection.createStatement();
      PolicyDecorator policy = new SQLPolicyRemoveAnd(new SQLPolicyRemoveOr(username));
      resultSet = stmt.executeQuery("select * from users " + "where username = '" +
                         policy.convert().getStr() + "'");
    } catch (SQLException e) {
      e.printStackTrace();
      }
    }
}
```

5.5 Towards a Decorated Filter Tool

Applying filters as decorators is an instance of moving embellishment to decorator [23] refactoring. A tool should additionally contain a library of filters and should be extensible. We discuss the diversity of validation policies in section 7.

6 Safe Library Replacement Transformation

You have a program that uses a function that might cause data injection attacks if it receives an insufficiently validated input. You want to ensure that the program is not vulnerable to injection attacks.

Replace unsafe functions with safe functions that are not vulnerable even if malicious data is injected.

6.1 Motivation

Bugtraq lists 28 buffer overflow vulnerabilities in the first week of September 2008. We could not analyze 9 instances that affect proprietary software. 17 of the remaining 19 instances can be solved by replacing unsafe string functions with safe functions. In the remaining two cases, buffer overflow vulnerabilities originate from direct manipulation of pointers.

Many sophisticated static [35,38] and dynamic [7,18] analysis tools are available for detecting vulnerable functions that lead to buffer overflow attacks. To prevent buffer overflow, programmers write checks or use safe buffer handling functions that check buffer bounds before performing an operation. Some of these functions trim the resultant destination buffer to fit its size [19,25], while other functions dynamically resize the destination buffer [24,21].

Usually, programmers manually search and replace library functions. We asked the developers of the top ten most active projects of all time in sourceforge.net [31] about their development approach. Six projects use C or C++ as one of the languages, three use PHP and one uses Java. In five out of six C/C++ projects, programmers initially used unsafe `strcpy` and `strcat` functions, but manually changed to safer C/C++ string libraries later.

Manual changes are error-prone. This method does not scale for large projects. For example, Ghostscript (about 350 KLOC) is a medium size program, but its programmers have replaced only a small part of its unsafe functions with safe functions. A program transformation automatically replaces all instances of unsafe string functions.

6.2 Preconditions

A program with the following characteristics benefits from a library replacement transformation

- Program uses an unsafe function for which safe alternatives are available.
- Filters in a program fail to sufficiently validate input.

6.3 Mechanics

For each unsafe function, a developer specifies the alternative safe function and the library that includes the function.

The program transformation finds all functions that need to be replaced. It replaces unsafe functions with suitable alternatives in all source files. It adds information about the new library to configuration files so that the new program compiles. Table 2 lists some unsafe functions that cause buffer overflow attacks in C/C++ and their safe alternatives. Safe alternatives exist for other unsafe functions, e.g. `gets`, `memcpy`, `getenv`, `memmove`, `scanf`, `printf`, etc.

Table 2. Some unsafe functions and their safe alternatives

Unsafe string functions	**Safe string functions**

strcpy(3), strncpy (3) - Copy string
`char *strcpy (char *dst, const char *src);`
`char *strncpy (char *dst, const char *src, size_t num);`
	g_strlcpy from glib library [25],
	astrcpy, astrn0cpy from libmib library [11],
	strcpy_s from ISO/IEC 24731 [20]
	`gsize g_strlcpy (gchar *dst, const gchar *src, gsize dst_size);`
	`char *astrcpy (char **dst_address, const char *src);`
	`char *astrn0cpy (char **dst_address, const char *src, int num);`

strcat(3), strncat(3) - Concatenate string
`char *strcat (char *dst, const char *src);`
`char *strncat (char *dst, const char *src, size_t num);`
	g_strlcat from glib library [25],
	astrcat, astrn0cat from libmib library [11],
	strcpy_s from ISO/IEC 24731 [20]
	`gsize g_strlcat (gchar *dst, const gchar *src, gsize dst_size);`
	`char *astrcat (char **dst_address, const char *src);`
	`char *astrn0cat (char **dst_address, const char *src, int num);`

6.4 Example

We have written a proof-of-concept Perl script to search and replace `strcpy` and `strcat` functions and applied it to two open source C programs.

Details of the script. The script transforms three types of files.

- *Source Code.* It replaces `strcpy` and `strcat` functions with `g_strlcpy` and `g_strlcat` functions from the `glib 2.0` library. It uses `malloc_usable_size` and `sizeof` functions to calculate the size of heap buffers and stack buffers. Our code identifies the source and destination parameters by pattern matching. Two patterns are illustrated in Table 3.
- *Makefiles.* The modified program has to compile correctly. The quickest fix is to include the library during link time. The script modifies *Makefile*s by replacing the pattern `gcc` with ``gcc `pkg-config --libs glib-2.0` ``.
- *Configuration Files.* It transforms *config.status* the same as *Makefile*s.

Table 3. Patterns for *strcat* and *strcpy* functions

Patt. Name	Parameters	Search Pattern	Replacement Pattern
1. *Function with two variables*	Two parameters each of which are variables	`strcpy (dst, src);`	`g_strlcpy (dst, src, sizeof(dst));`
2. *Function with pointer arithmetic*	Any of the parameters have pointer arithmetic with integer	`strcpy (*dst + index, src);`	`g_strlcpy (*dst + index, src, malloc_usable_size(*dst) - index);`

Case studies. We have used the Perl script on two open source C programs: a pdf/ps file viewer (`gv`) and a compression library (`zziplib`). These programs have recent buffer overflow exploits [8,9], and the exploit codes are available.

`gv` version 3.6.2 has 46 C files with 27000 lines of code. There are 37 `strcpy` and 51 `strcat` instances in the source code. Our script replaces 86 of these 88 instances. Patterns used in our script are not sophisticated enough to replace the two remaining cases; we manually changed those functions. The script also changes 4 lines in the configuration file.

`zziplib` version 0.13.47 has 7346 lines of code in 33 C files. The script changes all 5 instances of `strcpy` and `strcat`. Also, it changes the *Makefile* in 15 places.

In both cases, the resultant programs do not have buffer overflow vulnerability. They compile correctly, pass all tests, and exhibit the same behavior.

6.5 Towards a Safe Library Replacement Tool

Our proof-of-concept tool performs string matching on source code. A commercial quality tool must use a more traditional design with more robust static and dynamic analysis, perhaps a special purpose transformation language such as TXL [6], Stratego/XT [37] and CIL [27]. A step in this direction is the Gemini tool [10]. Gemini, written in TXL, applies a different transformation than ours. Instead of replacing functions, it transform all stack-allocated buffers in a C program to heap-allocated buffers, because exploiting a heap overflow is more difficult. Gemini does not completely remove the vulnerability, but its use of a special purpose program transformation language is a step in the right direction.

Many safe alternatives exist for each unsafe function. Each safe function in Table 2 uses the same source and destination buffer parameters as the unsafe function; a size parameter is added in most cases. Another approach is to use functions with parameters of a new type [26,15]. The string library in `qmail` uses a new data structure named `stralloc` [15] to keep length information. To use these functions, a program transformation replaces all instances of string with the new data structure in addition to replacing the unsafe functions.

Safe library replacement tools are not limited to preventing buffer overflows. SQL injection attacks can be prevented by replacing all instances of string concatenation based SQL queries with SQL `PreparedStatement` [34]. Another instance of a safe function is Magicquotes or `addslashes` functions in PHP that automatically escape quotes to prevent SQL injection.

7 Policies for Preventing Attacks on Data

Add Perimeter Filter and *Decorated Filter* transformations apply various input validation policies to input variables. The policies differ for different injection attacks. Table 4 lists some attacks and corresponding input validation policies.

The goal of all injections is to run code in place of data; hence most policies remove/replace keywords and special characters. Attackers try to bypass the checks by encoding inputs. In response, there are some checks that decode inputs and convert them to a canonical format. In Table 4, policies 3 and 7 do so.

Another type of policy augments input with metadata. Such policies, applied at the system entry point, mark the input and then use it for taint based injection attack detection. WASP [17] augments a Java string with a `MetaString` class to detect SQL injection; SQLCheck [33] adds marks to the beginning and end of a Java string to detect various types of command injection; Pixy [22] adds metadata to PHP strings to detect cross site scripting. These tools apply policies to data twice–1) they add metadata to an input at system entry point, and 2) they check the metadata to detect a malicious activity before the input is used.

Other policies apply static and dynamic analysis techniques to build a model of good input and match it with the actual input before it is used. AMNESIA [16] statically builds an NFA to model SQL statements, CANDID [2] models SQL statements with a parse tree, while SQLRand [4] randomizes SQL keywords in a query to detect the mixing of control and data channels. These policies decide whether an input is good or malicious.

Not all policies are for input validation. Encryption, encoding, parity checking, etc. can also be thought as special policies applied on data. Program transformation tools that decorate an input with filters could automate these tasks.

Some policies can be odd; they replace valid data with fabricated data. An example is a policy that replaces error messages with a generic error message so as to not reveal any internal information. Another odd thing about this policy: it is applied to output data rather than input.

Table 4. Sample policies for various transformations

Type of Injection Attack	Example Policy	Description of Transformation
SQL Injection	1. Remove SQL keyword	Searches input for SQL keyword, e.g. SELECT, AND, OR, UNION etc. and removes the keyword. Before: ' OR '1'='1 After: ' '1'='1
SQL Injection	2. Escape single quote	Searches input for single quote characters and escapes them. Before: ' OR '1'='1 After: " OR "1"="1
SQL Injection	3. Decode hex encoding	Decodes any part of the input that has been encoded in hex. Before: 0x73656c656374 After: SELECT
Direct Static Code Injection	4. Remove commands	Searches input for system commands such as system, rm etc. Before: rm%20-rf%20/ After: %20-rf%20/
LDAP Injection	5. Remove & and |	Searches and removes LDAP and (&) and or (|) character. Before: Alice)(|(password=")) After: Alice)((password="))
Cross Site Scripting	6. Remove <SCRIPT> tag	Searches for <SCRIPT>...< /SCRIPT> and removes it. Before: <SCRIPT>alert('XSS');</SCRIPT> After: (<SCRIPT> tag removed)

Cross Site Scripting	7. Decode UTF-8 encoding	Decodes UTF-8 encoding in code.
		Before: <IMG SRC=&#106;&#97;&#118;&#97;&#118;
		&#97;&#115;&#99;&#114;&#105;&#112;&#116;
		&#58;&#97;&#108;&#101;&#114;&#116;&#40;
		&#39;&#88;&#83;&#83;&#39;&#41;>
		After: <IMG SRC="javascript:alert('XSS');">

The use of policies is a property of program transformations that distinguishes them from refactoring. Although both take user specification and make structural changes, a refactoring only needs to know the abstract syntax tree of a program to make the change [30]. In contrast, a security-oriented program transformation needs to additionally know the artifacts that make the behavioral change. For example, an *Add Perimeter Filter* transformation needs to know about the library of filters to use to change program behavior.

A *Decorated Filter* transformation might include input validation policies that must be applied in a particular order. The policies that canonicalize input are applied before any other policies. Some other policies might keep the order information. For example, if an input is encrypted before the calculation of its message digest, it is to follow a reverse order when it is decrypted. An automated tool for the transformation can encapsulate these structural details.

As new kinds of security threats continue to appear, even systems that meet high security standards today will eventually require updates; they will need security-oriented program transformations. To cope with requirement changes, organizations will have to modify existing policies or create new ones. Thus policies used by a program transformation should be parameterizable and extensible.

8 Tools for Security-Oriented Program Transformations

None of the security-oriented program transformations described in this paper require deep analysis of the program being transformed. They share many structural similarities with refactorings and can probably be implemented similarly.

Some of the transformations in this paper insert subroutine calls into a program. These could be implemented using an aspect-oriented programming language. Others rearrange the structure of the program. These are more like refactorings, and would be hard to implement using aspects. *Safe Library Replacement* matches a set of code patterns representing calls to an old library, and replaces them with code that calls a new library. It seems unlikely that this could be implemented using an aspect-oriented language. All these transformations can probably be implemented by a code transformation language such as TXL [6] and Stratego/XT [37], or a framework for implementing refactorings.

All the program transformations are platform dependent, at least to the extent that they depend upon the language of the program being transformed, and perhaps on the operating system (*chroot Jail*) or the libraries (*Safe Library Replacement*). Frameworks for refactoring are also platform dependent, as are most of the languages for writing program transformations.

It is unreasonable to expect researchers to build such tools for every platform. Building a tool for a single application is usually not cost effective, so it is unreasonable to expect application developers to do it. This will need to be done by platform and tool vendors, who can amortize the cost over many users.

9 Conclusion

Keeping systems secure requires constantly improving them. Security-oriented program transformations are a repeatable, systematic approach to eradicating system vulnerabilities. This paper focuses on data injection attacks, but the idea can be applied to other areas of vulnerability. Security-oriented program transformations have the potential to enable security engineers to develop policies and let a tool apply those policies. This will blend human creativity with the thoroughness of the computer.

Acknowledgement

Thanks to Baris Aktemur, Farhana Ashraf, Sruthi Bandhakavi, Nicholas Chen, David Garlan, Carl Gunter, Sam Kamin, Darko Marinov, Jeffrey Overbey, Maurice Rabb and Hridesh Rajan for providing valuable feedback. We also thank US Department of Energy for their financial support under the grant number DE-FG02-06ER25752. Finally, we thank the anonymous reviewers for their helpful comments.

References

1. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Pearson Education, London (2001)
2. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.: CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. In: CCS 2007: Proceedings of the 14th ACM conference on Computer and Communications Security, pp. 12–24. ACM Press, New York (2007)

3. Blakley, B., Heath, C.: Security design patterns technical guide - version 1. Open Group (OG), led by Bob Blakley and Craig Heath (2004)
4. Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL injection attacks. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 292–302. Springer, Heidelberg (2004)
5. Bugtraq Vulnerabilities List, `http://www.securityfocus.com/vulnerabilities`
6. Cordy, J., Halpern-Hamu, C., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. In: ICCL, pp. 280–285. IEEE, Los Alamitos (1988)
7. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Seventh USENIX Security Symposium proceedings, San Antonio, Texas (1998)
8. CVE-2006-5864,
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5864`
9. CVE-2007-1614,
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1614`
10. Dahn, C., Mancoridis, S.: Using program transformation to secure C programs against buffer overflows. In: WCRE 2003, Washington DC, USA, p. 323. IEEE Computer Society Press, Los Alamitos (2003)
11. Cavalier III., F.: Libmib allocated string functions,
`http://www.mibsoftware.com/libmib/astring/`
12. Fowler, M.: Refactoring: Improving The Design of Existing Code. Addison-Wesley, Reading (1999)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Reading (1995)
14. Hafiz, M.: Security oriented program transformations (Or how to add security on demand). In: OOPSLA 2008: Companion to the 23rd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, New York (2008)
15. Hafiz, M., Johnson, R.: Evolution of the MTA architecture: The impact of security. Software—Practice and Experience 38(15), 1569–1599 (2008)
16. Halfond, W., Orso, A.: Preventing SQL injection attacks using AMNESIA. In: ICSE 2006: Proceedings of the 28th International Conference on Software Engineering, pp. 795–798. ACM, New York (2006)
17. Halfond, W., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In: SIGSOFT'06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering, pp. 175–185. ACM Press, New York (2006)
18. Haugh, E., Bishop, M.: Testing C programs for buffer overflow vulnerabilities. In: NDSS. The Internet Society (2003)
19. International Organization for Standardization. ISO/IEC 9899:1999: Programming Languages — C (December 1999)
20. International Organization for Standardization. ISO/IEC 24731: Specification For Secure C Library Functions (2004)
21. ISO/IEC 14882. C++ std:string
22. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: PLAS 2006: Proceedings of the 2006 workshop on Programming Languages and Analysis for Security, pp. 27–36. ACM Press, New York (2006)

23. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley, Reading (2004)
24. Messier, M., Viega, J.: Safe C string library v1.0.3
25. Miller, T., de Raadt, T.: strlcpy and strlcat — Consistent, safe, string copy and concatenation. In: 1999 Usenix Annual Technical Conference, Monterey, California, USA (1999)
26. Narayanan, A.: Design of a safe string library for C. Software—Practice and Experience 24(6), 565–578 (1994)
27. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
28. OWASP. Categories of injection attacks (2008)
29. Rinard, M.: Living in the comfort zone. In: OOPSLA 2007: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications, pp. 611–622 (2007)
30. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems 3(4), 253–263 (1997)
31. SourceForge.net. Most active projects - all time (February 2008)
32. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management. Prentice Hall PTR, Englewood Cliffs (2005)
33. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 372–382. ACM Press, New York (2006)
34. Thomas, S., Williams, L.: Using automated fix generation to secure SQL statements. In: ICSEW 2007: Proceedings of the 29th International Conference on Software Engineering Workshops, Washington, DC, USA, p. 54. IEEE Computer Society, Los Alamitos (2007)
35. Viega, J., Bloch, J., Kohno, T., McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code. In: 16th Annual Computer Security Applications Conference. ACM, New York (2000)
36. Viega, J., McGraw, G.: Building Secure Software: How to Avoid Security Problems The Right Way. Addison-Wesley, Reading (2002)
37. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)
38. Wagner, D., Foster, J., Brewer, E., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: NDSS. The Internet Society (2000)

Report: Measuring the Attack Surfaces of Enterprise Software

Pratyusa K. Manadhata[1], Yuecel Karabulut[2], and Jeannette M. Wing[1]

[1] Carnegie Mellon Univeristy, Pittsburgh, PA, USA
[2] SAP Research, Palo Alto, CA, USA

Abstract. Software vendors are increasingly concerned about mitigating the security risk of their software. Code quality improvement is a traditional approach to mitigate security risk; measuring and reducing the *attack surface* of software is a complementary approach. In this paper, we apply a method for measuring attack surfaces to enterprise software written in `Java`. We implement a tool as an Eclipse plugin to measure an SAP software system's attack surface in an automated manner. We demonstrate the feasibility of our approach by measuring the attack surfaces of three versions of an SAP software system. We envision our measurement method and tool to be useful to software developers for improving software security and quality.

1 Introduction

There is a growing demand for secure software as we are increasingly dependent on software in our day-to-day life. Software vendors have traditionally focused on improving code quality for improving software security and quality. The code quality improvement effort aims toward reducing the number of security vulnerabilities in software. In practice, however, building large and complex software devoid of vulnerabilities remains a very difficult task. Software vendors have to embrace the hard fact that their software will ship with both known and future vulnerabilities in them and many of the vulnerabilities will be discovered and exploited. They can, however, minimize the risk associated with the exploitation of these vulnerabilities. One way to minimize the risk is by reducing the attack surfaces of their software. A smaller attack surface makes the exploitation of the vulnerabilities harder and lowers the damage of exploitation, and hence mitigates the security risk. As shown in Figure 1, the code quality effort and the attack surface reduction approach are complementary; a complete risk mitigation strategy requires a combination of both.

Michael Howard of Microsoft introduced the notion of Relative Attack Surface Quotient (RASQ) for the Windows operating system [1]. Pincus and Wing generalized Howard's notion and measured the attack surfaces of seven versions of Windows [2]. Their measurement method, however, was ad-hoc in nature, required a security expert (e.g., Michael Howard for Windows), and was focused on operating systems. Manadhata and Wing of Carnegie Mellon University (CMU) formalized Howard's notion and proposed an abstract but systematic attack

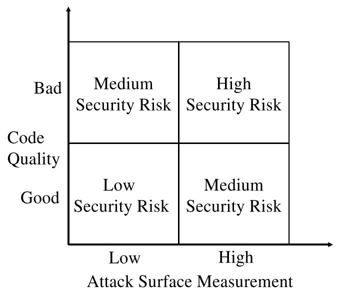


Fig. 1. Attack Surface Reduction and Code Quality Improvement are complementary approaches for improving software security

surface measurement method that does not require a security expert and is applicable to a wide range of software [3].

Intuitively, a system's attack surface is the set of ways in which an adversary can enter the system and potentially cause damage. A larger attack surface measurement indicates that an attacker is likely to exploit the vulnerabilities present in the system with less effort and cause more damage to the system. Since a system's code is likely to contain vulnerabilities, it is prudent to reduce the system's attack surface measurement in order to mitigate the security risk.

To see how well our attack surface method works on enterprise-scale software, SAP and CMU collaborated to apply CMU's attack surface measurement method to SAP's platforms and business applications. Henceforth, we collectively refer to SAP's platforms and business applications as SAP software systems. This collaboration suggested ways to integrate the measurement process with software development process, not just for SAP, which we discuss in Section 6. We describe the collaboration in the rest of this report.

2 Abstract Attack Surface Measurement Method

We briefly describe Manadhata and Wing's abstract measurement method in this section. Please see their technical report for details [3].

We know from the past that many attacks, e.g., exploiting a buffer overflow, on a system take place by sending data from the system's operating environment into the system. Similarly, many other attacks, e.g., symlink attacks, on a system take place because the system sends data into its environment. In both these types of attacks, an attacker connects to a system using the system's *channels* (e.g., sockets), invokes the system's *methods* (e.g., API), and sends *data items* (e.g., input strings) into the system or receives data items from the system. Hence an attacker uses a system's methods, channels, and data items present in the system's environment to attack the system. We collectively refer to a system's methods, channels, and data items as the system's *resources* and thus define a system's attack surface in terms of the system's resources. Not all resources, however, are part of the attack surface. Manadhata and Wing use the entry point and exit point framework to identify the resources that are part of a system's attack surface.

Entry Points. Each system has a set of methods. A method receives arguments as input and returns results as output. Examples of methods are the API of a system. A system's methods that receive data items from the system's environment are the system's entry points. For example, a method that receives input from a user or a method that reads a configuration file is an entry point. A method m of a system s is a *direct entry point* if either (a) a user or a system in s's environment invokes m and passes data items as input to m, or (b) m reads from a persistent data item, or (c) m invokes the API of a system in s's environment and receives data items as the result returned. An *indirect entry point* is a method that receives data from a direct entry point.

Exit Points. A system's methods that send data items to the system's environment are the system's exit points. For example, a method that writes into a log file is an exit point. A method m of a system s is a *direct exit point* if either (a) a user or a system in s's environment invokes m and receives data items as results returns from m, or (b) m writes to a persistent data item, or (c) m invokes the API of a system in s's environment and passes data items as input to the API. An *indirect exit point* is a method that sends data to a direct exit point.

Channels. Each system also has a set of channels; the channels are the means by which users or other systems in the environment communicate with the system. Examples of channels are TCP/UDP sockets, RPC end points, and named pipes. An attacker uses a system's channels to connect to the system and attack the system. Hence a system's channels act as another basis for attacks.

Untrusted Data Items. An attacker uses persistent data items either to send data indirectly into the system or receive data indirectly from the system. Examples of persistent data items are files, cookies, database records, and registry entries. A system might read from a file after an attacker writes into the file. Similarly, the attacker might read from a file after the system writes into the file. Hence the persistent data items act as another basis for attacks on a system. An *untrusted data item* of a system s is a persistent data item d such that a direct entry point of s reads from d or a direct exit point of s writes into d.

Attack Surface Definition. By definition, the set, M, of entry points and exit points, the set, C, of channels, and the set, I, of untrusted data items are the resources that the attacker can use to either send data into the system or receive data from the system and hence attack the system. Hence given a system, s, and its environment, we define s's attack surface as the triple, $\langle M, C, I \rangle$.

Attack Surface Measurement Method. Not all resources contribute equally to a system's attack surface. Manadhata and Wing estimate a resource's contribution to a system's attack surface as a *damage potential-effort ratio* where *damage potential* is the level of harm the attacker can cause to the system in using the resource in an attack and *effort* is the amount of work done by the attacker to acquire the necessary access rights in order to be able to use the resource in an attack.

In practice, we estimate a resource's damage potential and effort in terms of the resource's attributes. Examples of attributes are method privilege, access rights, channel protocol, and data item type. In case of systems implemented in `C`, we estimate a method's damage potential in terms of the method's *privilege*. An attacker gains the same privilege as a method by using a method in an attack. For example, the attacker gains `root` privilege by exploiting a buffer overflow in a method running as `root` and hence causes damage to the system. Similarly, we estimate a channel's damage potential in terms of the channel's *protocol* and a data item's damage potential in terms of the data item's *type*. The attacker can use a resource in an attack if the attacker has the required *access rights*. The attacker spends effort to acquire these access rights. Hence for the three kinds of resources, i.e., method, channel, and data, we estimate attacker effort in terms of the resource's access rights. We assign numeric values to the attributes to compute a numeric damage potential-effort ratio. We describe a specific method of assigning numbers in Section 4.2.

Our abstract measurement method has the following three steps.

1. Given a system, s, and its environment, we identify a set, M, of entry points and exit points, a set, C, of channels, and a set, I, of untrusted data items of s.
2. We estimate the damage potential-effort ratio, $der_m(m)$, of each method $m \in M$, the damage potential-effort ratio, $der_c(c)$, of each channel $c \in C$, and the damage potential-effort ratio, $der_d(d)$, of each data item $d \in I$.
3. The measure of s's attack surface is the triple $\langle \sum_{m \in M} der_m(m), \sum_{c \in C} der_c(c), \sum_{d \in I} der_d(d) \rangle$.

3 Measurement Method for SAP Software Systems

In this section, we walk through the steps of our method for measuring the attack surfaces of software services written in `Java`. We keep our discussion general, bringing in the specifics of the SAP application only where necessary.

In our SAP collaboration, we chose a component of the SAP NetWeaver platform as the system whose attack surface is to be measured [4]. The component is a core building block of the platform; henceforth, we refer to the chosen component as *the service*. The service does not use any persistent data items and opens only one channel, i.e., a `TCP` socket. Hence we only considered the method dimension of the attack surface in our measurement. We would, however, consider the three dimensions of the attack surface for a generic `Java` system.

3.1 Identification of Entry Points and Exit Points

An entry point of a system is a method that receives data items from the system's environment. A method, m, of a system, s, implemented in `Java` can receive data items in three different ways: (a) m is a method in s's public *interface* and receives data items as input, (b) m invokes a method in the interface of a system, s', in the environment and receives data items as result, and (c) m invokes a `Java I/O` library method. For example, a method, m, is an entry point if m invokes the `read` method of the `java.io.DataInputStream` class.

An exit point of a system is a method that sends data items to the system's environment. A method, m, of a system, s, implemented in `Java` can send data items in three different ways: (a) m is a method in s's public interface and sends data items as result, (b) m invokes a method in the interface of a system, s', in the environment and sends data items as input, and (c) m invokes a `Java I/O` library method. For example, a method, m, is an exit point if m invokes the `write` method of the `java.io.DataOutputStream` class.

Given a system, s, we generate s's call graph starting from the methods in s's public interface. From the call graph, we identify all methods of s that invoke either a method in the interface of a system, s', in s's environment or a `Java I/O` library method. These methods are s's entry points and exit points.

3.2 Estimation of the Damage Potential-Effort Ratio

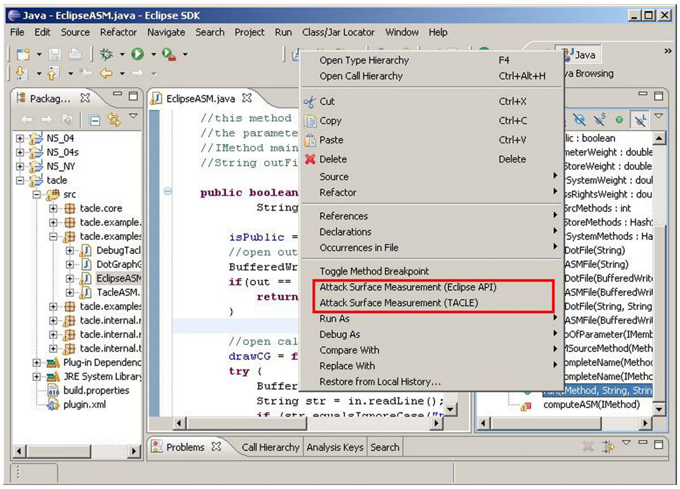
We estimate a method's damage potential using the method's *sources of input data (destinations of output data).* A method can receive (send) data items from (to) three sources: an input parameter, the data store, and other systems present in the environment. For example, a method receives data items from an attacker as an input parameter in case of SQL injection attacks whereas the method receives data items from the data store in case of File Existence Check attacks. We do not use method privilege to estimate damage potential because the entire code of the NetWeaver platform runs with the same privilege and hence we can not make any meaningful suggestions to reduce the attack surface.

Similar to systems implemented in `C`, we use a method's access rights level to estimate the attacker effort. A typical SAP system has two different types of interfaces: (1) public interfaces that can be accessed by all entities belonging to any NetWeaver *role* and (2) internal interfaces that can be accessed by only other components of the NetWeaver platform. Hence the methods in SAP systems can be accessed with two different access rights levels: `public` access rights level for methods in public interfaces and `internal` access rights level for methods in internal interfaces.

We assign numeric values to sources of inputs and access rights levels to compute numeric damage potential-effort ratios. The choice of the numbers depends on a system and its environment. We discuss a specific way of assigning numeric values in case of the service in Section 4.2.

4 Implementation of a Measurement Tool

In this section, we describe a tool we implemented to measure the attack surfaces of software systems implemented in `Java`. We implemented our tool as a *plugin* for the *Eclipse* Integrated Development Environment (IDE) so that software developers can use the tool inside their software development environment [5]. We show a screen shot of our tool in Figure 2.



```
        //this method
        //the paramete
        //IMethod main
        //String outFi

        public boolean
            String

        isPublic =
        //open out
        BufferedWr
        if(out ==
            return
        }

        //open cal
        drawCG = f
        try {
            Buffer
            String str = in.readLine();
```

Fig. 2. Screenshot of the Attack Surface Measurement tool implemented as an Eclipse plugin

4.1 Call Graph Generation

A key component of our tool is the generation of a system's call graph from the system's source code. We use two different techniques to generate the call graph to provide a precision-scalability tradeoff to the software developers: the TACLE Eclipse plugin developed at the Ohio State University, which gives a very precise call graph, but does not scale well to large programs [6]; and an Eclipse API, which gives a less precise call graph, but scales [7].

The TACLE based approach results in a precise attack surface measurement whereas the Eclipse API based approach results in an over-approximation of the measurement. The two approaches are complementary; software developers can use the Eclipse API based approach to produce an imprecise measurement of a large software system, and then use the TACLE based approach to obtain precise measurements of the components that make large contributions to the measurement. The imprecise measurement guides the developers to the relevant components of a system and the precise measurement guides the developers in reducing the attack surface.

Our tool identifies a system's entry points and exit points from the system's call graph. A method of a system is an entry point (exit point) if the method invokes a method in the public interface of another system present in the environment or a `Java I/O` library method. Hence we provide our tool the list of interface methods of other systems and the list of `Java I/O` library methods through two configuration files. Our approach of identifying entry points and exit points is applicable to generic `Java` systems and is independent of SAP software systems.

4.2 Numeric Value Assignment

Another key component of our tool is the estimation of the numeric damage potential-effort ratios of a system's entry points and exit points. The tool determines the sources of input and the access rights levels from the system's call graph; the tool, however, requires the numeric values assigned to the different sources of input and the access rights levels to estimate numeric damage potential-effort ratios. Users of our tool would choose the numbers based on the knowledge of their system and its environment.

The methods of the service have three different sources of input: `parameter`, `data store`, and `other system`. As discussed in Section 3.2, different types of attacks on the service require the methods to have different sources of input. We assign numeric values to the sources of input by correlating the sources with possible attacks on the service. SAP conducted an internal threat modeling process for the service. The process identified possible attacks on the service and assigned severity ratings to the attacks. We correlated the sources of inputs with the possible attacks identified by the threat modeling process. For each source of input, we computed the average severity rating of the attacks that require the source of the input. We show the sources of input in the first column and the average severity ratings in the second column of Table 1.

We assigned numeric values to the sources of input in proportion to the average severity ratings. Manadhata and Wing's parameter sensitivity analysis suggests that the difference between the numeric values assigned to successive damage potential levels should be in the range of 3-14 [8]. Hence we chose the midpoint, 8.5, of the range as the difference. For example, we assigned 1 to the source `other system`, and $1 + (3 - 1) \times 8.5 = 18$ to the source `data store`. We show the numeric values in third column of Table 1.

The methods of the service can be accessed by two different access rights level: `public` and `internal`. We imposed the following total ordering among the access rights level: `internal` $>$ `public`. The parameter sensitivity analysis

Table 1. Numeric values assigned to the sources of input

Source of Input	Average Severity Rating	Value
`other system`	1	1
`data store`	3	18
`parameter`	5	35

Table 2. Numeric values assigned to the access rights levels

Access Rights	Value
`public`	1
`internal`	18

suggests that the difference between the numeric values assigned to successive access rights level should be high (15-20). Hence we chose a difference of 17. We show the numeric values assigned to the access rights level in Table 2.

We use the numeric values shown in Table 1 and Table 2 to compute the numeric damage potential-effort ratios. For example, consider an entry point, m, of a system, s. m is a method in s's public interface and has two input parameters; m also invokes three interface methods of a system, s', in the environment. Then m's damage potential is $2 \times 35 + 3 \times 1 = 73$. If m is accessible with the `public` access rights level, then m's damage potential-effort ratio is $73/1 = 73$. Similarly, if m is accessible with the `internal` access rights level, then m's damage potential-effort ratio is $73/18 = 4.05$.

4.3 Usage of the Tool and Measurements

Software developers can use our tool to measure and reduce a system's attack surface. The tool generates detailed output containing (1) the system's attack surface measurement, (2) a list of the system's entry points and exit points, and (3) for each entry point (exit point), a list of input sources, the access rights level, and its contribution to the attack surface measurement. Software developers can use the detailed output as a guide in reducing the attack surfaces of their software. For example, they can focus on the top x% of the entry points and the exit points to reduce the attack surface. They can also focus on the top contributing interfaces and components instead of considering the entire code base of the system. We are currently working on a visualization tool to guide the developers to the relevant parts of the code base.

The tool also allows the developers to consider many *what-if* scenarios during software development. For example, the developers can easily determine the effect of adding a new feature to the system on the system's attack surface. Similarly, while reducing the attack surface, they can consider the removal of different features and the effect of the removal on the attack surface measurement. They can use the incremental measurements to make an informed decision.

5 Results and Discussion

We measured the attack surfaces of three different of the service included in three different versions of the NetWeaver platform. We identify the three versions of the service as `S1`, `S2`, and `S3`. The `S1` version is the first released version of the service, followed by `S2` and `S3` versions, respectively.

The `S3` version of the service implements 8 public interfaces and 2 internal interfaces. The `S2` and `S1` versions implement 9 and 8 public interfaces, respectively, and no internal interfaces. We show the number of entry points and exit points of the three versions for each access rights level in Table 3. We estimated the damage potential-effort ratio of each entry point (exit point) to compute the attack surface measurements; we show the measurements in Table 4.

The `S2` version is backward compatible with `S1` for the convenience of the customers. Moreover, `S2` added new features to `S1` resulting in an increase in the

Table 3. The number of entry points and exit points

Version	Count	
	Public	Internal
`S3`	71	4
`S2`	67	0
`S1`	63	0

Table 4. Attack surface measurements

Version	Attack Surface Measurement
`S3`	5298.44
`S2`	4687.00
`S1`	4649.00

number of public interfaces. Hence the set of methods of `S2` is a superset of the set of methods of `S1` and as shown in Table 4, the attack surface measurement of `S2` is greater than `S1`.

The `S3` version differs from the `S2` version in two significant ways: (1) `S3` converted a public interface of `S2` to an internal interface to mitigate security risk, and (2) `S3` added new features to the service resulting in an increase in the number of public interfaces and internal interfaces. If no new features were added, the attack surface measurement of `S3` would have been smaller than `S2` due to the conversion of a public interface to an internal interface. The increase in the number of total interfaces due to the addition of new features, however, increases the attack surface measurement of `S3`. Hence as shown in Table 4, the attack surface measurement of `S3` is greater than `S2`.

The measurement results show that addition of new features will increase a software system's attack surface measurement. Software developers should, however, aim to minimize the increase in the attack surface. SAP's developers made a good design decision by introducing internal interfaces that reduced the increase in the attack surface measurement.

6 Potential Usage of Attack Surface Measurements

We envision three potential uses of attack surface measurements in the software development process. First, software developers and architects can use attack surface measurements to prioritize their software testing effort. For example, if a system's attack surface measurement is high, they should invest more in testing efforts to identify and remove vulnerabilities from the system. Similarly, if the measurement is low, they can reduce their testing effort.

Second, software developers can use attack surface measurements as a guide while implementing patches of security vulnerabilities. A good patch should not only remove a vulnerability from a system, but also should not increase the system's attack surface. Software developers can use our tool to ensure that their patches do not increase the attack surface.

Third, software consumers can use attack surface measurements to guide their choice of software configuration. Choosing a suitable configuration, especially for complex enterprise-scale software, is a nontrivial task. Since a system's attack surface measurement is dependent on the system's configuration, software consumers would choose a configuration that results in a smaller attack surface exposure.

7 Summary and Future Work

In summary, we introduced a method to measure the attack surfaces of SAP software systems implemented in `Java` and implemented a tool to measure the attack surface in an automated manner. We demonstrated the use of the method and the tool by measuring and comparing the attack surfaces of three versions of an SAP software system. We also learned important lessons on how to improve the method and the tool to make them more useful in practice.

Based on the feedback received from SAP developers, we identify four possible avenues of future work. First, we plan to extend the tool to measure the attack surfaces of software implemented in other languages such as `JavaScript`. Second, Manadhata and Wing performed three empirical studies to validate the abstract measurement method and the measurement results of systems implemented in `C` [8]. A possible direction of future research is to explore validation ideas in the context of SAP business applications. Third, we plan to extend our work by developing a method to estimate the minimum and the maximum possible attack surface measurements of a system given the system's functionality. The minimum and maximum estimates will be useful in guiding attack surface reduction and test effort prioritization. Fourth, we plan to investigate code analysis techniques to identify a system's indirect entry points and indirect exit points in an automated manner.

References

1. Howard, M.: Fending off future attacks by reducing attack surface (2003), `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp`
2. Howard, M., Pincus, J., Wing, J.M.: Measuring relative attack surfaces. In: Proc. of Workshop on Advanced Developments in Software and Systems Security (2003)
3. Manadhata, P.K., Kaynar, D.K., Wing, J.M.: A formal model for a system's attack surface. Technical Report CMU-CS-07-144, CMU (July 2007)
4. Ag, S.: NetWeaver, `http://www.sap.com/platform/netweaver/index.epx`
5. Eclipse: Eclipse - An open development platform, `http://www.eclipse.org/`
6. Sharp, M., Sawin, J., Rountev, A.: Building a whole-program type analysis in Eclipse. In: Eclipse Technology Exchange Workshop at OOPSLA, pp. 6–10 (2005)
7. Eclipse: Eclipse package org.eclipse.jdt.internal.corext.callhierarchy, `http://mobius.inria.fr/eclipse-doc/org/eclipse/jdt/internal/corext/callhierarchy/package-summary.html`
8. Manadhata, P.K., Tan, K.M., Maxion, R.A., Wing, J.M.: An approach to measuring a system's attack surface. Technical Report CMU-CS-07-146, CMU (2007)

Report: Extensibility and Implementation Independence of the .NET Cryptographic API

Pieter Philippaerts, Cédric Boon, and Frank Piessens

Dept. of Computer Science, K.U.Leuven

Abstract. When a vulnerability is discovered in a cryptographic algorithm, or in a specific implementation of that algorithm, it is important that software using that algorithm or implementation is upgraded quickly. Hence, modern cryptographic libraries such as the .NET crypto libraries are designed to be extensible with new algorithms. In addition, they also support algorithm and implementation independent use. Software written against these libraries can be implemented such that switching to a new crypto algorithm or implementation requires very little effort.

This paper reports on our experiences with the implementation of a number of extensions to the .NET cryptographic framework. The extensions we consider are smart card based implementations of existing algorithms. We evaluate the extensibility of the libraries, and the support for implementation independence. We identify several problems with the libraries that have a negative impact on these properties, and we propose solutions.

The main conclusion of the paper is that extensibility and implementation independence can be substantially improved with only minor changes. These changes maintain backwards compatibility for client code.[1]

1 Introduction

When Microsoft released the first version of the .NET Framework in early 2002, they included a newly designed cryptographic API, built from the ground up. Like with the rest of the .NET runtime, one of the main concerns during the design of the architecture was to develop an object oriented system that is future-proof and ensures extensibility by end users. This is reflected in the class structure by modularizing the different algorithms, and applying standard object oriented design principles.

An additional, crypto-related, requirement was to also ensure algorithmic independence. Cryptography is a fast moving science where implementations of algorithms are regularly broken [1,2,10,11], or where even entire classes of cryptographic algorithms are defeated [9, 13, 14]. This has as a consequence that secure applications must be able to quickly change from one algorithm to another. Having a cryptographic architecture that has been designed to allow this can be beneficial.

[1] This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

Even though the design goes a long way towards achieving these goals, some optimizations can be made. This paper describes some of the shortcomings of the .NET cryptographic architecture and proposes minimal changes to solve these problems. As the .NET Framework is used by numerous applications, a top priority will be to maintain backwards compatibility. Throughout the paper, the example of a smart card library will be used to expose the various problems with the API. It also provides a compelling case study, as smart card usage is increasing worldwide and will eventually be the cornerstone of many secure systems.

This paper is organized as follows: Sections 3 through 5 detail the experiences and problems with extending the framework. Each section explores specific classes of algorithms in the API. A section is composed of a sketch of what the current implementation looks like, a description of the problems that were encountered, and the proposed solution to counter these issues. Finally, section 6 summarizes the observations.

2 Introduction to the .NET Cryptographic API

History and experience have taught us that two key features should be present in modern cryptographic interfaces:

Extensibility. New algorithms are regularly introduced to counter the loss of broken algorithms, so the framework must have hooks to extend the base system to support other algorithms. Moreover, one algorithm can have multiple implementations. When new optimizations are found, for example, older implementation may be replaced with new, faster, implementations.

Algorithmic independence. One certainty in cryptography is that algorithms get broken. When this happens, applications depending on this algorithm must be able to quickly switch to another, more secure algorithm. Hence, some kind of independence of the algorithm must be achieved.

The .NET cryptographic API uses an inheritance-based model to achieve these goals. The architecture consists of three layers:

- **The engine classes:** Engine classes represent a group of algorithms, like symmetric algorithms, asymmetric algorithms, or hash algorithms. These abstract classes feature a *Create* method that returns a default implementation of the default algorithm for the concerning engine class.
- **The algorithm classes:** Algorithm classes represent a specific algorithm. They are abstract and derive from the abstract engine classes. Examples of algorithm classes are *RSA*, *Rijndael*, *SHA1*,... Algorithm classes also have a *Create* method that returns a default implementation of this specific algorithm.
- **The implementation classes:** Implementation classes contain the actual implementation of the algorithms. These classes are concrete, and derive from the algorithm classes.

Implementation classes are either fully managed implementations of the algorithm, or wrappers around the existing *native Windows CryptoAPI*[2] (or short: *CryptoAPI*). The former are denoted with a *Managed* suffix in their class name, the latter use a *CryptoServiceProvider* suffix.

Support for smart cards. Smart cards have been around for a while, mainly used in the banking and telecommunication sector. Only recently, however, the interest in smart cards has increased tremendously, with applications for cell phones [6], e-government [4], authentication [5], ... Smart cards offer a substantial security improvement over traditional magnetic stripe cards, because they contain a small processor that can execute cryptographic operations on data that is contained within the card [12]. This means that secret data, such as the private part of an asymmetric key or a secret symmetric key, doesn't have to leave the card to compute the output of some cryptographic algorithm.

Version 3.5 of the .NET Framework does not provide direct support to perform cryptographic operations on smart cards. There is only limited support to delegate to native smart card based cryptographic service providers (CSP).

To ease the development of .NET applications that rely on smart cards for cryptographic operations, we developed a number of custom CSPs based on the *ISO7816-4* [8] and *ISO7816-8* [7] standards. Care has been taken to fit the smart card CSPs into the current .NET cryptographic model, to simplify the end-user's experience. The problems with the crypto framework reported on in this paper are mainly based on our experience with the integration of these smart card CSPs. Note however that the problems described in this paper are not specific to smart cards, but are indications of imperfections with respect to the cryptographic model's extensibility.

3 Asymmetric Algorithms

3.1 Asymmetric Algorithms in the .NET Framework

The engine class for asymmetric algorithms is the *AsymmetricAlgorithm* class. Version 3.5 of the .NET Framework ships with support for RSA, DSA, and the elliptic curve variants of Diffie-Hellman and DSA. In this paper we will look only at the RSA support. Problems found in the elliptic curve classes are described in [3].

The *RSA* class is located on the second level of the hierarchy. It defines the additional members *EncryptValue* and *DecryptValue*. When overridden in a derived class, these methods perform *raw* RSA encryption and decryption and return the result. In this context, *raw* means that the input data should be encrypted or decrypted without performing any cryptographic padding.

The .NET Framework provides an elegant solution to cope with different padding schemes for RSA, by means of formatters. A formatter accepts an

[2] This is the cryptographic API that's been present in Windows long before .NET existed. It is typically used by so called native (often C/C++) applications.

AsymmetricAlgorithm as a parameter and takes care of padding operations before passing on the modified data for encryption or signing. Specific formatters accept only specific children of *AsymmetricAlgorithm*. This architecture allows for different padding schemes to be used with one *AsymmetricAlgorithm*, and provides an extensible base for future padding schemes.

Concrete implementations for *PKCS#1 1.5* and *OAEP*[3] padding for RSA operations, as well as a formatter and deformatter for DSA operations are available in all versions of the .NET Framework.

As previously mentioned, the potential extension possibilities are one of the big advantages of the use of formatters. When weaknesses in current padding schemes are found, new formatters (e.g. RSA *PSS*[4] signature formatters) could be defined. Furthermore, upgrading an existing application with a new padding scheme is as simple as changing one line of code.

3.2 Identified Problems

Version 3.5 of the .NET Framework relies on the *CryptoAPI* to implement the RSA cryptographic service provider. It does not ship with an RSAManaged class. The *CryptoAPI*, however, does not expose *raw* RSA operations. An invocation of the API has to specify the padding scheme to be used. To fit the .NET RSA CSP in the model of formatters, the formatters handle instances of the *RSA-CryptoServiceProvider* class differently than other subclasses of the *RSA* class.

If the asymmetric algorithm passed to an RSA formatter *is not* an instance of the *RSACryptoServiceProvider* class, padding is performed by the formatter, and the *EncryptValue* and *DecryptValue* methods are invoked. The *EncryptValue* and *DecryptValue* methods always perform raw RSA operations.

The *RSACryptoServiceProvider* does not implement the *EncryptValue* and *DecryptValue* methods. Instead, additional methods are defined that delegate the operations to the *CryptoAPI*. If the asymmetric algorithm passed to an RSA formatter *is* an instance of the *RSACryptoServiceProvider* class, no padding is performed by the formatter. Instead, one of the additional methods of the *RSACryptoServiceProvider* is invoked, and the result is returned as received by the *CryptoAPI*.

While the model retains the elegance of the API, version 3.5 of the .NET Framework incorrectly assumes that the *RSACryptoServiceProvider* class is the only subclass of *RSA* that does not support raw RSA. A class that delegates encryption to a smart card, for example, also has the same constraints. This is because most smart cards do not support raw RSA encryption, for security purposes.

3.3 Suggested Solution

The suggested solution consists of a *pull up* of several methods from the *RSA-CryptoServiceProvider* class to the *RSA* class. The methods of the *RSACrypto-ServiceProvider* that are required by the formatters for *padded* operations are

[3] OAEP: Optimal Asymmetric Encryption Padding.
[4] PSS: Probabilistic Signature Scheme.

Encrypt, *Decrypt*, *SignHash*, and *VerifyHash*. To support the decision process of the formatters, an additional *SupportsRaw* property should also be inserted.

In order to preserve compatibility with existing applications, and to promote extensibility of the proposed solution, a number of changes to the pulled up methods are required:

– The pulled up methods should be marked *virtual*, to allow for dynamic binding.
– An implementation for these methods should be provided, to preserve compatibility with existing subclasses that do not implement these methods. A suggested implementation is throwing a *NotImplementedException*. This is analog to how the current .NET framework solves similar issues.
– The boolean parameter in the *Encrypt/Decrypt* methods should be changed to a more general parameter, like a *String*. This allows extensions with padding types other than *PKCS#1 1.5* and *OAEP*. An Object Identifier (Oid) could also be used, if Oids for padding types are registered.

The implementation of the formatters should be modified to benefit from the new structure, as shown for the *RSAPKCS1KeyExchangeFormatter* in listing 1.

```
if(_rsaKey.SupportsRaw) {
    // Perform Padding
    paddeddata = PerformPadding(data);
    //Encrypt the padded data
    return _rsaKey.EncryptValue(paddeddata)
} else {
    // Encryption and padding the unpadded data
    return _rsaKey.Encrypt(data, "PKCS1")
}
```

Listing 1. Modification of the *RSAPKCS1KeyExchangeFormatter*

It is important to show that the proposed modifications do not break the compatibility with existing applications:

– For users of the *RSACryptoServiceProvider* class, the only change is the addition of three methods, and the addition of an *override* keyword to two existing methods.
– For users of the *RSA* class, the only change is an addition of five methods.
– For classes derived from the *RSA* class, five more methods are available to override. As the *RSA* class provides concrete implementations for these methods, no additional implementation is required at the level of the derived class, which preserves compatibility with existing third party RSA subclasses.
– For third party classes deriving from the *RSA* class and implementing methods with the same signature as the added ones, the existing methods will hide the added methods in the *RSA* base class, leaving the functionality unchanged.

– For users of the formatters, the new implementation would be semantically identical to the existing one for existing classes. As the *SupportsRaw* method would return a default value of *true* and would be overridden in the *RSA-CryptoServiceProvider* class to return *false*, the behavior of the formatters would be the same as provided by version 3.5 of the .NET Framework, for all existing RSA implementations.

Similar issues have been found in the elliptic curve class hierarchy. Section II.C from [3] details the problems and the solution we propose.

4 Symmetric Algorithms

4.1 Symmetric Crypto in the .NET Framework

The engine class for symmetric algorithms in the .NET Framework is the *SymmetricAlgorithm* class. All implementations of symmetric algorithms must inherit from this class. Version 3.5 of the .NET Framework provides cryptographic service providers for the AES, (Triple)DES, RC2 and Rijndael algorithms.

In contrast to the asymmetric algorithms, the class hierarchy of symmetric algorithms is more symmetrical, with most of the operational methods concentrated in the *SymmetricAlgorithm* engine class. Most algorithm and implementation classes do not add additional methods.

The engine class defines a *Key* and *IV* property, which gets or sets a byte array that represents the key or initialization vector for the symmetric algorithm. These byte arrays are transparent, because the interpretation of this data is fixed and known.

One of the objectives of the cryptographic framework is to be able to easily substitute an algorithm by another one in the case that the first one is broken. By using the *Create* method of the *SymmetricAlgorithm* class, a default implementation of a symmetric algorithm will be returned. Version 3.5 of the .NET Framework will return an instance of the *RijndaelManaged* class. However, the framework allows overriding the default cryptographic service providers by means of the cryptographic configuration.

4.2 Identified Problems

One of the advantages of using smart cards for symmetric algorithms is that the key never has to leave the smart card. Consider a system where data has to be encrypted using a symmetric algorithm. A random key could be generated on a smart card, and used in the encryption and decryption process. The key on the smart card could be made accessible to the smart card only, upon entry of a PIN. Once the data is encrypted using the smart card, it will only be decryptable with the same smart card.

As mentioned in section 1, a number of symmetric CSPs (DES, 3DES and Rijndael) for smart cards have been used for the evaluation of the cryptographic framework. No key property has to be given to those cryptographic service providers, as the key is already available on the smart card itself. However, these cryptographic service providers need a number of other parameters, like a PIN (with adequate formatting parameters), an identifier representing the keyslot to be used on the smart card, ...

While it is possible to introduce *ImportParameters* and *ExportParameters* methods that accept opaque key material at the level of the implementation of the CSP itself, this prevents users from exploiting the advantages of the .NET cryptographic architecture, as a cast to the concrete implementation class would always be necessary to set the parameters. Having a mechanism to import opaque key material for symmetric algorithms on a more abstract level would therefore be beneficial to the developers using the framework.

4.3 Suggested Solution

The *AsymmetricAlgorithm* class already offers support for importing and exporting parameters via XML. Through analogy, this support could be added to the *SymmetricAlgorithm* class, by adding *FromXmlString* and *ToXmlString* methods. These methods import and export opaque key material as an XML string. A default implementation in the *SymmetricAlgorithm* class could be used to get or set the key, the initialization vector, the cipher mode and the padding mode.

The modifications to the symmetric algorithms introduced in the previous paragraphs do not break compatibility with existing applications:

- For users of the *SymmetricAlgorithm* class, the only change is an addition of two methods.
- For classes derived from the *SymmetricAlgorithm* class, 2 more methods are available to override. As the *SymmetricAlgorithm* class provides concrete implementations for these methods, no additional implementation is required at the level of the derived class, which preserves compatibility with existing third party symmetric algorithm implementations.
- For third party classes deriving from the *SymmetricAlgorithm* class and implementing methods with the same signature as the added ones, the existing methods will hide the added methods in the *SymmetricAlgorithm* engine class, leaving the functionality unchanged.

5 Hash Message Authentication Codes (HMAC)

5.1 HMACs in the .NET Framework

The HMAC algorithm is represented in .NET by the *HMAC* class. This class is located in the *HashAlgorithm* hierarchy, as a subclass of the *KeyedHashAlgorithm* class. Different subclasses of the *HMAC* class implement the HMAC algorithm for different hash algorithms. Since only the hash function differs from implementation to implementation, the *HMAC* algorithm class contains most of the functionality. Derived classes only have to initialize the correct hash algorithm of their base class.

5.2 Identified Problems

As mentioned in section 1, a SHA1 CSP for smart cards has been used for the evaluation of the cryptographic framework. In the context of HMACs, the CSP has been used to create a smart card HMAC-SHA1 CSP, by deriving from the *HMAC* class. SHA1 hashing operations are delegated to the smart card, whereas the other operations required to calculate an HMAC are performed in software.

The problem with deriving from the *HMAC* class is that the methods used to implement the HMAC algorithm in this base class are marked as *internal.* Thus, deriving from the *HMAC* class does not support reusing the existing implementation in this class.

A number of ways exist to create a CSP that derives from the *HMAC* class. It is possible to re-implement the functionality of the base class in the derived class, by overriding the *HashCore* and *HashFinal* methods. However, this would violate the principles of object oriented design that try to promote the reuse of code. This is also why the additional abstraction of the *HMAC* class was introduced between the *KeyedHashAlgorithm* and the concrete implementations in the first place.

Another way to create a CSP deriving from the *HMAC* class would be to use the *HashName* property to set the required hash algorithms in the constructor of the deriving class. However, this requires that the assembly of the smart card based hash implementation is added in the global assembly cache (GAC)[5], and correctly registered in the cryptographic configuration of .NET. While this is definitely the best way to go using version 3.5 of the .NET framework, it adds a certain complexity to the deployment.

An interesting side note can be made on the *HashName* property. The fact that this property has a public setter, can lead to inconsistencies, as shown by listing 2, where the output of a *HMACSHA1* is in fact an MD5 HMAC.

```
HMACSHA1 hmacSHA1 = new HMACSHA1(key);
hmacSHA1.HashName = "MD5";
hmacSHA1.ComputeHash(message);
byte[] result = hmacSHA1.Hash;
```

Listing 2. Inconsistency induced by an incorrect using of the *HashName* property. The result of the *HMACSHA1* is in fact an MD5 HMAC.

Alternatively, the required internal fields can be set by means of reflection. Using reflection to set internal attributes of core classes of the .NET Framework is however prone to compatibility issues with future versions of the .NET Framework, and is definitely not encouraged.

A similar, though less important problem occurs due to the *InitializeKey* method being marked as internal. A deriving class wanting to set the key in

[5] The global assembly cache is a repository of shared .NET libraries.

the constructor is forced to use the *Key* property, as the *InitializeKey* method is internal. The *Key* property, however, is overridable. It is considered bad practice to call overridable members in constructors, as a deriving class could have its members called before being initialized[6]. For most cases, the class deriving from the *HMAC* class could be marked as sealed (*final* in Java). However, cases could be found where an additional level in the class hierarchy is desirable. An example would be an abstract *smart card HMAC* class that derives from the *HMAC* class and bundles smart card specific functionality (for instance, smart card parameter management). Specific smart card HMAC implementations could then derive from this *smart card HMAC* class.

5.3 Suggested Solution

Using protected methods and fields is a well-established design pattern to allow for a shielded way of code sharing between base classes and their children. The *HMAC* class would benefit in extensibility with a promotion of the internal fields and methods to protected fields and methods. With protected fields, the need for a setter on the *HashName* property disappears. Removing this setter would prevent inconsistencies as illustrated in listing 2.

The change in visibility of the internal fields and methods preserves the backwards compatibility and enables the reusability of the *HMAC* class by third-party developers.

While the setter on the *HashName* property is not used in the .NET Framework itself, its removal could break the backwards compatibility with third party applications and CSPs that rely on this setter. Hence, removing the HashName setter is an optional suggestion.

6 Conclusion

One of the design goals of the .NET cryptographic framework is to provide an extensible platform for developers, that offers algorithmic and implementation independency. While it achieves these goals up to a certain level, in practice a number of problems occur. These problems get bigger when trying to deal with less conventional implementations of cryptographic algorithms, like cryptographic operations on smart cards.

This paper identified a number of general problems, and proposed solutions to alleviate these issues. A common goal of all solutions is to maintain backwards compatibility, and minimize the impact of the changes on the existing cryptographic framework, while improving the extensibility of the API.

The mentioned problems are relatively easy to solve, effectively requiring only minor changes to the existing cryptographic library. These modifications are localized in a minimal set of classes, and preserve backwards compatibility. An overview of additional problems and solutions can be found in [3].

[6] Rule CA2214 in Microsoft FxCop.

References

1. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 629–660. Springer, Heidelberg (1998)
2. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. Journal of Cryptology 14, 101–119 (2001)
3. Boon, C., Philippaerts, P., Piessens, F.: Practical experience with the NET cryptographic API (November 2008), `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW531.abs.html`
4. De Cock, D., Wouters, K., Preneel, B.: Introduction to the Belgian EID Card. In: Katsikas, S.K., Gritzalis, S., López, J. (eds.) EuroPKI 2004. LNCS, vol. 3093, pp. 1–13. Springer, Heidelberg (2004)
5. Gaskell, G., Looi, M.: Integrating smart cards into authentication systems. In: Dawson, E.P., Golić, J.D. (eds.) Cryptography: Policy and Algorithms 1995. LNCS, vol. 1029, pp. 270–281. Springer, Heidelberg (1996)
6. Herzberg, A.: Payments and banking with mobile personal devices. Wireless networking security 46(5), 53–58 (2003)
7. ISO/IEC. ISO/IEC 7816-8 Identification cards - Integrated circuit cards - Commands for security operations, 2nd edn. (2004)
8. ISO/IEC. ISO/IEC 7816-4 Identification cards - Integrated circuit cards - Organization, security and commands for interchange, 2nd edn. (2005)
9. Kelsey, J., Schneier, B., Wagner, D.: Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. Information and Communications Security: First International Conference, 233–246 (1997), `http://www.trusteer.com/dnsopenbsd`
10. Klein, A.: OpenBSD DNS cache poisoning and multiple O/S predictable IP ID vulnerability (2007)
11. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
12. Naccache, D., M'Raihi, D.: Cryptographic smart cards. IEEE Micro 16(3), 14–24 (1996)
13. Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, 2004/199 (2004)
14. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)

Report: CC-Based Design of Secure Application Systems

Robin Sharp

DTU Informatics, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark
robin@imm.dtu.dk

Abstract. This paper describes some experiences with using the *Common Criteria for Information Security Evaluation* as the basis for a design methodology for secure application systems. The examples considered include a Point-of-Sale (POS) system, a wind turbine park monitoring and control system and a secure workflow system, all of them specified to achieve CC assurance level EAL3. The methodology is described and strengths and weaknesses of using the Common Criteria in this way are discussed. In general, the systematic methodology was found to be a good support for the designers, enabling them to produce an effective and secure design, starting with the formulation of a Protection Profile and ending with a concrete design, within the project timeframe.

1 Introduction

The Common Criteria for Information Security Evaluation (CC), of which the latest approved version is v3.1 [1,2,3], are widely accepted as a framework for evaluation of IT products in order to provide assurance that these meet given security requirements. However, since the CC involve the use of security specifications at various levels of detail, and describe how to decide conformance between these specifications, the CC can also be seen as the potential basis for a design method for Systems Engineering, targeted at producing secure systems. In this paper we discuss how readily the CC can be applied to supporting the design task, by considering a series of design cases performed at the Technical University of Denmark: for a Point-of-Sale (POS) system [11,12], a wind turbine park distributed monitoring and control (DMC) system [9] and a secure workflow (SWF) system [6].

The natural choice for a systematic style of development is to base it on CC's concept of *refinement*, i.e. restriction of a set of choices $\mathcal{C}$, which appear as part of a security requirement, to a smaller set, $\mathcal{C}' \subseteq \mathcal{C}$. Refinements may take place at all stages of a development based on CC. Thus the Protection Profile, which is the top-level specification of a class of products, may contain requirements which are refinements of those given in the CC standard, while the Security Target, which specifies a specific product within such a class, will use requirements which are refined from those of the PP.

The structure of this paper reflects the overall structure of the Common Criteria. After a short description of the types of system considered, we first discuss development of the PPs, and then of the STs derived from them. We then present the salient features of the process leading to a final design. At each stage we describe the most important design decisions and relate them to the CC framework. Finally, we discuss lessons which can be learnt from the case studies and point to some areas where attempts to follow the CC framework caused difficulties in the design process.

2 The Application Systems Considered

The systems considered in the case studies are all application systems which require a high degree of security, as opposed to “pure” security components such as firewalls, smart cards or secure OSs. A simple abstract model of such a system is shown in Fig. 1. It has three important properties:

1. The system has a number of *devices* attached to it, which either collect input to or present output from the application.
2. Each device is associated with one or more *data flows* into or out of the application, each associated with a particular *security functional policy (SFP)*.
3. The system maintains an *audit trail* which contains definitive evidence of the transactions performed within the system.

The concept of data flows makes it possible to describe a generic system, where the number and type of devices are not fully specified at the PP level. The idea is to define a set of generic data flows which can be useful for the application, and to describe the desired security properties for these data flows by the SFPs. When a specific system is to be specified by an ST, an appropriate selection of the data flows defined in the PP can be made, and more details provided.

In secure application systems, the basic audit trail for application transactions is typically supplemented by a *security audit trail* containing audit records for all security-related events in the system. In this paper we shall use the unqualified term *audit trail* to mean the application and security audit trails considered together. The main security requirements are to maintain an appropriate degree of integrity and confidentiality of the input/output data flows and the audit trail itself, and an agreed degree of availability of the system’s functionality to the users, when the system operates in a given *operational environment*. The required degree of integrity and confidentiality typically depends on the *role* of the user. Almost all such systems recognise at least the two basic roles:

Operator: A user responsible for operating the application itself. For example in a POS system this will be a salesperson responsible for registering sold goods or services and received payments and for producing evidence for the customer.

Administrator: A user authorised to install, configure and maintain system functions, and responsible for system security.

A central task for the designer is to identify the necessary roles and define the rights which they have in relation to the various flows.

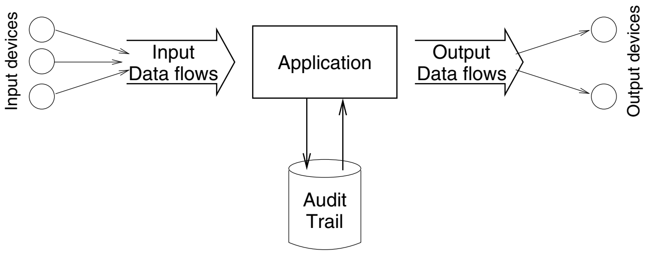


Fig. 1. Application system model with input and output data flows

3 The Protection Profile

The Protection Profile (PP) is used to specify security-related requirements for a general class of system and the environment in which systems of this class are to operate. Thus the PP provides a relatively abstract specification, which can subsequently be *refined* to produce one or more Security Targets (STs) for specific types of system in the class.

A PP has a formalised structure with a number of mandatory elements, starting with a *PP introduction*, which identifies the PP and describes the system under consideration (in CC terminology known as the *Target of Evaluation (TOE)*). The *TOE description* covers the usage and major security features of the TOE, and requirements for additional (non-TOE) hardware or software, if any, needed by the TOE. In terms of traditional software engineering concepts, this is the initial user requirements document.

Security Problem Definition. Given the TOE description, the next step in producing the PP is to define a *Security Problem Definition (SPD)*, specifying the:

Assumptions. About the *environment* in which the TOE will be developed or operated,

Organisational security policies (OSPs). With which the TOE or its operational environment must comply.

Threats. Against the *assets* in the TOE,

In all cases we derived these as directly as possible from the TOE description. Note that the "Threats" in the SPD are *residual threats* which are not adequately countered by any of the mechanisms provided via the Assumptions or OSPs. Thus this approach differs from traditional risk analysis, where all threats are considered, and countermeasures introduced to deal with those which give an unacceptable risk. Obviously there is much room for discussion here, as a choice of fewer or weaker assumptions leads to more residual threats.

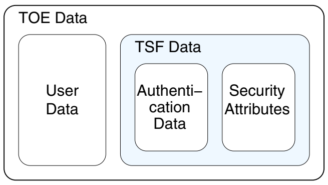


Fig. 2. TOE Data and TSF Data

In secure applications, the assets to be protected in the TOE fall into two classes (see Fig. 2). The *primary assets* are the principal *User Data* for the application (in our case studies the audit trail or similar record of activity). The *secondary assets*, known as *TSF data*, include user names, passwords, cryptographic keys and other *security attributes* needed to provide the *TOE Security Functionality (TSF)*, which is to maintain the security of the application.

Damage to assets arises via the action of *threat agents*. Potential threat agents include users, computer processes, development personnel, physical factors (such as fire or flooding) etc. User threat agents ("*attackers*") may be *authorised* users, who maliciously or accidentally misuse their authorisation, or *unauthorised* malicious outsiders.

As an example, we consider the SPD for the wind turbine DMC system described by Khodaverdi and Vohra [9] (here slightly reformulated for clarity). The assumptions made about the development and operational environments of the TOE were:

A.NO_EVIL. The developers and administrators of the TOE have no evil intentions and are trained to carry out their job correctly.

A.PHYSICAL. The physical security of the TOE is adequate to prevent loss or damage of the TOE due to external factors such as fire, theft or natural disasters.

A.EXTERNAL_PARTY. Any external products which the TOE relies upon are trusted and are installed, configured and managed in a secure manner.

A typical set of Organisational Security Policies (OSPs) within organisations which would use this type of system were found to be:

P.AUTHORISED_USERS. Only authorised users may access the TOE.

P.USER_PRIVILEGES. Authorised users have rights and privileges to access TOE data on a need-to-know basis.

P.TRAIN. Authorised users shall be trained appropriately in operating the TOE.

P.ACCOUNTABILITY. Users that are authorised to access TOE data shall be held accountable for their actions within the TOE.

P.CRYPTOGRAPHY. TOE data shall be encrypted using standard cryptoalgorithms.

With these assumptions and OSPs, the residual threats were identified as:

T.MASQUERADE. An unauthorised user or process may pretend to be another entity in order to gain access to data or other TOE resources.

T.UNAUTHORISED_ACCESS. An unauthorised user or process may gain access to data which they are not allowed to access according to the TOE security policy.

T.MODIFICATION. An attacker may maliciously modify protected data in the TOE.

T.UNATTENDED_SESSION. An attacker may gain unauthorised access to the TOE via a session started by an authorised user who is now absent.

T.USER_ERROR. Users may make accidental errors which could jeopardise the security of the TOE.

T.DATA_TRANSMISSION. An attacker may alter the way in which data are transmitted, thereby compromising the confidentiality and integrity of data in the TOE.

T.CRYPTO_LEAK. An attacker may view, modify or delete keys or code associated with the cryptographic functionality intended to protect the data in the TOE.

Security Objectives. The next step in development of the PP is to select a set of *Security Objectives* which will counter the threats and address the assumptions and OSPs included in the SPD. In most cases, the objectives fall into two classes:

- Objectives for the TOE itself (denoted O.xxx).
- Objectives for the operational environment of the TOE, including both the physical surroundings and the users (denoted OE.xxx).

For example, the objectives for the wind turbine DMC include:

O.UNIQUE_IA. The TOE shall provide means for identifying and authenticating antities before allowing them access to the TOE.

O.ACCOUNTABILITY. The TSF shall provide individual accountability for audited events, so that administrators can hold users accountable for any actions that are relevant to the security of the TOE.

O.SESSION. The TSF shall provide mechanisms that lock sessions automatically when no activity has been seen for a predefined period of time. Users shall be able to lock a session manually to avoid signing out, and to unlock it by re-authentication.

OE.ISOLATION. Those responsible for the TOE shall provide isolation of the parts of the TOE, so they are protected from physical damage, intrusion and theft.

OE.TRAIN. Those responsible for the TOE shall provide training for the administrators, operators and service technicians in the secure use of the TOE.

To meet the assurance requirements, a *security objectives rationale*, which justifies the claim that the objectives cover the OSPs, assumptions and threats, must be produced. The form of the arguments used for this justification reflect the level of assurance required. At assurance levels EAL4 or below, verbal arguments are adequate, while at higher levels the use of formal methods based on formalised models of the system and its environment will be called for. As our case studies aimed for level EAL3, detailed verbal arguments were used; for reasons of space, we omit them here.

Security Requirements. The final step in producing the PP is to derive a set of actual *Security Requirements* for the TOE. These requirements fall into two groups:

Security Functional Requirements (SFRs). Which describe the measures to be taken to ensure security in the TOE. These are used to drive the further design process.

Security Assurance Requirements (SARs). Which describe the requirements for documentation, life-cycle management and testing. These define what the designer and evaluator have to do in order to evaluate the security of the TOE.

The SFRs can be derived systematically from the selected objectives, taking into account the SFPs defined for the data flows in the system. This step resembles a traditional engineering design process, with the slightly unusual feature that the CC provide a catalogue of ready-formulated generic SFRs, usually known as *security components*, grouped into *families* of related requirements. The families are in turn grouped into *classes*, each of which is associated with a particular aspect of security, such as security auditing, communication security, cryptographic support, user data protection and so on. Version 3.1 of the CC contains 268 SFRs, divided into 65 families and 11 classes. This makes it easy in most cases to find suitable SFRs to meet the objectives. If all else fails, the CC permit the PP designer to define *extended components* which lie outside Part 2 of the CC; this was not necessary in any of our case studies. As with the objectives, a *rationale* must be provided which maps the SFRs onto the objectives which they are claimed to meet and provides arguments for why this claim is valid. The form of this rationale is again determined by the desired level of assurance.

Similarly, the CC provide a catalogue of ready-formulated SARs (again known as *components*), grouped into classes. However, the set of SARs to be used is largely determined by the level of assurance to be met, rather than by any particular properties of the design itself. In all the case studies described here, the chosen level was EAL3, *Methodically tested and checked*[1]. The standard SARs for EAL3 are listed in Table 1.

4 The Security Target

The PP specifies a class of system, and any conformant ST derived from it specifies an instance of this class, corresponding to a particular type of POS system, wind turbine DMC system, workflow system etc. In all our case studies, the ST was for a system with a simple distributed architecture:

[1] In the case of the SWF, EAL3+, since ALC_CMS.3 was replaced by one with stricter requirements.

Table 1. Standard Security Assurance Requirements for CC assurance level EAL3

Assurance class	Component
ADV: Development	ADV_ARC.1 Security architecture description ADV_FSP.3 Functional spec. with complete summary ADV_TDS.2 Architectural design
AGD: Guidance documents	AGD_OPE.1 Operational user guidance AGD_PRE.1 Preparative procedures
ALC: Life-cycle support/ Config. management	ALC_CMC.3 CM authorisation controls ALC_CMS.3 Implementation representation CM coverage ALC_DEL.1 Delivery procedures ALC_DVS.1 Identification of security measures ALC_LCD.1 Developer defined life-cycle model
ATE: Tests	ATE_COV.2 Analysis of coverage ATE_DPT.1 Testing: basic design ATE_FUN.1 Functional testing ATE_IND.2 Independent testing – sample
AVA: Vulnerability assessment	ADV_VAN.2 Vulnerability analysis

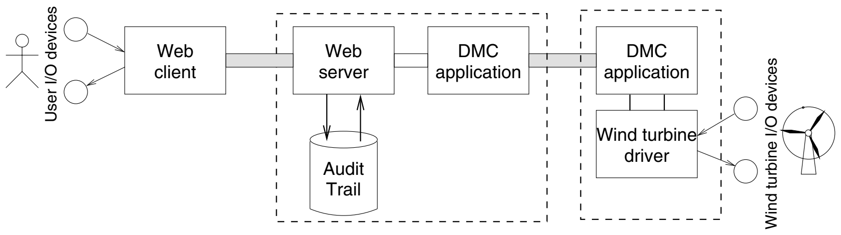


Fig. 3. Architecture of a web-based wind turbine DMC system

– A hosted POS system, with the audit trail on the hosting system and client(s) at the actual point of sale [12].
– A web-based wind turbine DMC system, with clients having access to a central server, to which the wind turbines are connected, as in Fig. 3.
– A single-server workflow system, with the server offering the actual workflow functionality to one or more trusted clients.

In our design methodology, the ST must (in CC terms) be *strictly conformant* to the PP: The concepts used in the ST must be the same as in the PP, and it must fulfil at least as many security requirements as the PP requires, while its operational environment does not offer more functionality than assumed in the PP. To ensure this, the ST is derived by refining the PP security requirements to make the specification less general. This involves reducing the cardinality of the sets of possible choices in the specification:

1. For a *selection* (which specifies a set of possible choices by enumeration), by selecting one or more explicit values.

2. For an *assignment* (which specifies a set of possible choices by set comprehension), by selecting explicit values, by replacing the comprehension by a more restrictive one, or by replacing the assignment by a *selection*.

For example, the SFR FCS_COP.1 appears in the wind turbine DMC PP as:

FCS_COP.1.1 The TSF shall perform [**assignment:** *list of cryptographic operations*] in accordance with a specified cryptographic algorithm [**assignment:** *cryptographic algorithm*] and cryptographic key sizes [**assignment:** *cryptographic key sizes*] that meet the following [**assignment:** *list of standards*].

taken from Part 2 of the CC without modification. In the ST, this is specialised to the following, with chosen values for the assignments:

FCS_COP.1.1(2) The TSF shall perform *encryption of the data flow and mutual authentication between the components of the TOE* in accordance with *any of the following TLS cipher suites: (a) TLS_RSA_WITH_AES_128_CBC_SHA or (b) TLS_RSA_WITH_AES_256_CBC_SHA* and cryptographic key sizes *128 or 256 bits for AES and a minimum of 1024 bits for RSA* that meet the following: *FIPS 140 level 1.*

At this stage, the SFPs for the generic data flows identified in the TOE will be specified in detail for the subjects and devices involved. It may here be necessary to introduce new Threats and OSPs into the SPD, or to extend those in the PP, in order to reflect the specific situation in which the system is to operate. This may give rise to additional Objectives for the TOE or its environment, and thus to additional SFRs. In our cases, for example, the choice of a distributed architecture means the ST must include requirements which ensure secure client/server communication. The effects of these new SFRs are also to reduce the freedom of choice of the designer of the system derived from the ST.

5 The Implementation Representation

In general terms, the final design task consists in selecting a set of system components which have appropriate functionality to satisfy the functional requirements for the system, and which implement the SFRs of the ST and fulfil the SARs for documentation and testing at the chosen level of assurance. Space only allows us to sketch the solution.

Security Functions. In order to satisfy those SFRs for which it is responsible, the TOE must provide suitable IT *security functions*. These can generally be identified directly from the SFRs in the ST. For example, in the case of the wind turbine DMC, the set included functions to provide identification and authentication of users, cryptographic functions needed for secure communication, functions to perform self-tests and to run malware scans on TOE data, functions to set up new users and give them appropriate privileges, depending on their roles, functions to provide backup and recovery of TOE data, functions to support role-based access control, and functions to generate time-stamped records for the audit trails.

Design Documentation. To meet the SARs in the ADV class, the designs must be described in terms of:

- A *functional specification* of the TSF and its external interfaces (TSFIs), including a mapping of the security functions to the SFRs, and a description of the usage, parameters and error messages of the TSFIs (assurance component *ADV_FSP.3*).
- An *architectural design* specifying the subsystems which make up the TSF and their interactions (component *ADV_TDS.2*).
- A *security architecture* describing the security domains of the TSF and how the initialisation process is made secure, and explaining how the TSF protects itself against bypass of the security functions and against tampering (component *ADV_ARC.1*).

Apart from the strong focus on security, this corresponds to standard SWE practice.

Selection of System Components. Given the required set of security functions and a high-level description of the architecture, the next step in the development of the IR is to select or construct concrete system components which can provide these functions. Figure 4 shows the high-level architecture for the DMC system, where most of the components are CC-evaluated standard components.

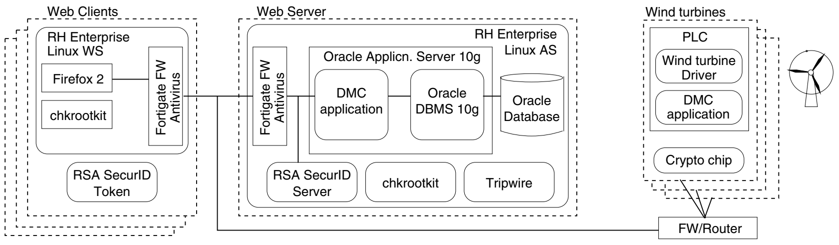


Fig. 4. High level architecture for the wind turbine DMC system

6 Lessons Learnt from the Case Studies

The case studies were performed as a series of M.Sc. thesis projects each lasting about six months. In each project group there were 1–2 participants, who were familiar with standard software engineering techniques and standard security practice, but had previously used *ad hoc* methods for including security in their designs. All had industrial experience through student jobs. Their knowledge of the CC was confined to CC's use for procurement. In the case of the POS system, the participants were familiar with the application domain through contact to an industrial company working in the area. For the wind turbine DMC and workflow systems, the participants accumulated domain knowledge through industrial contacts during the project.

Within their respective project periods, all groups succeeded in developing a PP, ST and IR for their system, using the methodology described here. For many reasons, a complete implementation could not be completed in the time provided. As a consequence, the testing and vulnerabiliy assessment activities required by the ATE and AVA classes of SAR could not be performed. Furthermore, the user guidance and life-cycle support documents were reduced to an absolute minimum.

For practical reasons, the IR was in all cases partly based on COTS components, rather than strictly CC-evaluated components. This leaves open a number of well-known questions with relation to the security properties of COTS components [4,10], but makes it feasible to complete such projects in a reasonable period of time. In a more perfect world, any non-CC components would be designed specifically for the purpose to the highest possible software standards.

The main challenges facing the designers were to understand the CC documents and to produce the PP. The CC documents are complex, and went through a number of radical revisions during the 2-year period over which the case studies were performed. The POS project was based on CC v2.3, while the DMC and SWF projects were based on rev. 1 of CC v3.1. This made it hard to ensure "technology transfer" between projects.

The CC contain some traps for inexperienced users. For example, new assumptions may not be added when deriving the ST from the PP. This is because a prospective purchaser of the TOE must be able, just by looking at the PP, to find out which assumptions the TOE's environment must satisfy. However, in practice it can be hard to foresee which assumptions may be needed in all the STs derived from a given PP. It requires practice to realise in good time that functions such as authentication, encryption and backup can easily be achieved by using standard external components, rather than components which are part of the TOE. It is not possible to make a late decision about where the boundary between the TOE and its environment is to go.

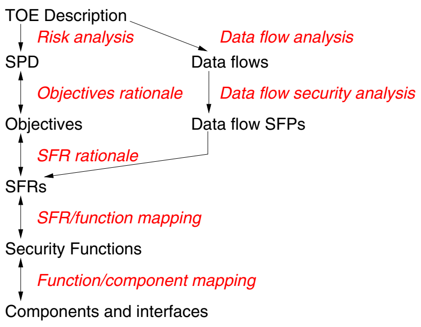


Fig. 5. Steps and documents in the design process

In general, production of the PPs was a challenge because it requires designers to produce an abstract description of a class of systems, which describes the *essential* security properties that are required. In their first attempts to do this, the young engineers typically included a lot of concrete details, which it later turned out possible to reduce to a very small number of basic principles.

The systematic approach based on the CC helped the designers to ensure that nothing was left out of the design – or, if it was, then this was a result of deliberate choices, for example in the selection of assumptions and OSPs. The steps in the design process and the corresponding documents produced are illustrated in Fig. 5. Production of the rationales, in particular, is extremely valuable, as it forces the designers to argue for the correctness of their solution. We are currently working on approaches to formalise some aspects of this in a practical support tool.

7 Concluding Remarks

Other approaches to systematic design of secure products have primarily focused on Object Oriented design techniques, and in particular security-oriented versions of UML such as UMLsec [8], the use of patterns [13,5], and integration of component-oriented techniques with security engineering [7,10]. A notable exception is the approach used in the PalmE project [15], which describes in some detail the steps of the software design process needed to meet the requirements of the CC for EAL2. However, all these approaches focus on the design process from the ST onwards, and systematic design starting from a relatively abstract specification such as a PP is not considered.

In the current state of the art, it appears feasible to incorporate the refinement steps into tools already used for formal development, such as RSL [14]. The final phase in deriving a design, which involves finding a concrete implementation architecture and choosing concrete IT components, could then be performed within the tool's development environment. The challenge here is to integrate the selection of existing (COTS) components into the design environment. We shall continue to work on incorporating appropriate information into support tools.

Acknowledgments

The author would like to thank the many students who have acted as guinea pigs during the development of this methodology, and in particular Rune Friis-Jensen, Anders Hedegaard, Shekoufeh Knodaverdi, Allan Pedersen, and Vikas Vohra, whose experiences and helpful feedback have formed the basis for this paper.

References

1. Common Criteria for Information Technology Security Evaluation, version 3.1, revision 1. Part 1: Introduction and general model, CCMB-2006-09-001 (September 2006)
2. Common Criteria for Information Technology Security Evaluation, version 3.1, revision 2. Part 2: Security functional components, CCMB-2007-09-002 (September 2007)
3. Common Criteria for Information Technology Security Evaluation, version 3.1, revision 2. Part 3: Security assurance components, CCMB-2007-09-003 (September 2007)
4. Bertoa, M.F., Troya, J., Vallecilo, A.: A survey on the quality information provided by software component vendors. In: Proc. 7th. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (July 2003)
5. Blakley, B., Heath, C.: Security design patterns. Technical Report G031, The Open Group, Reading, UK (April 2004)
6. Friis-Jensen, R.: A CC approach to secure workflow systems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark (February 2007)
7. Galitzer, S.: Introducing Engineered Composition (EC). In: ACSA Workshop on the Application of Engineering Principles to System Security Design (WAEPSSD) (November 2002)
8. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
9. Khodaverdi, S., Vohra, V.: A CC approach to windmill control systems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark (February 2007)
10. Lloyd, W.J.: A Common Criteria based approach for COTS component selection. Journal of Object Technology 4(3), 27–34 (2005)
11. Pedersen, A., Hedegaard, A.: Security in POS systems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark (August 2005)

12. Pedersen, A., Hedegaard, A., Sharp, R.: Designing a Secure Point-of-Sale System. In: Proc. 4th IEEE Intl. Workshop on Information Assurance (IWIA 2006), April 2006, pp. 51–65 (2006)
13. Schumacher, M., Roedig, U.: Security engineering with patterns. In: Proc. 8th Conference on Pattern Languages of Programs, Monticello (July 2001)
14. The RAISE Method Group. The RAISE Development Method. BCS Practitioner Series. Prentice Hall (1995)
15. Vetterling, M., Wimmel, G., Wisspeintner, A.: Secure systems development based on the Common Criteria: The PalME project. In: Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, November 2002, pp. 129–138 (2002)

Protection Poker: Structuring Software Security Risk Assessment and Knowledge Transfer

Laurie Williams, Michael Gegick, and Andrew Meneely

Department of Computer Science, North Carolina State University
{lawilli3,mcgegick,apmeneel}@ncsu.edu

Abstract. Discovery of security vulnerabilities is on the rise. As a result, software development teams must place a higher priority on preventing the injection of vulnerabilities in software as it is developed. Because the focus on software security has increased only recently, software development teams often do not have expertise in techniques for identifying security risk, understanding the impact of a vulnerability, or knowing the best mitigation strategy. We propose the Protection Poker activity as a collaborative and informal form of misuse case development and threat modeling that plays off the diversity of knowledge and perspective of the participants. An excellent outcome of Protection Poker is that security knowledge passed around the team. Students in an advanced undergraduate software engineering course at North Carolina State University participated in a Protection Poker session conducted as a laboratory exercise. Students actively shared misuse cases, threat models, and their limited software security expertise as they discussed vulnerabilities in their course project. We observed students relating vulnerabilities to the business impacts of the system. Protection Poker lead to a more effective software security learning experience than in prior semesters. A pilot of the use of Protection Poker with an industrial partner began in October 2008. The first security discussion structured via Protection Poker caused two requirements to be revised for added security fortification; led to the immediate identification of one vulnerability in the system; initiated a meeting on the prioritization of security defects; and instigated a call for an education session on preventing cross site scripting vulnerabilities.

Keywords: Software security, Wideband Delphi, Protection Poker, Planning Poker.

1 Introduction

According to the National Vulnerability Database[1], the number of reported vulnerabilities has increased six-fold (from approximately 1000 to 6000 per year) from 2000 to 2007. As a result, software development teams must place a higher priority on preventing vulnerability injection at each phase of the software life cycle. Because the focus on software security has rapidly risen only in the last decade, software

[1] The Common Vulnerabilities and Exposures (CVE) and Common Configuration Enumeration (CCE) Statistics Query Page (http://web.nvd.nist.gov/view/vuln/statistics) provides the number and percent of a given vulnerability type reported for each year.

development teams often do not have expertise in techniques for identifying security risk, understanding the impact of a vulnerability, or knowing the best mitigation strategy [15]. We propose the Protection Poker activity as a collaborative and informal form of misuse case[2] development and threat modeling[3] that plays off the diversity of knowledge and perspective of the participants. An excellent outcome of Protection Poker is that security knowledge passed around the team. Protection Poker is based upon an interactive effort estimation practice, Planning Poker[4] [8], that is used by many agile software development teams.

More important than the resulting effort estimate of Planning Poker is the discussion that takes place during the practice. Planning Poker provides a structured means for team members to educate each other, uncover hidden assumptions, raise awareness of issues and complications, and expose the range of alternatives of achieving desired goals. Protection Poker, which focuses on security risk assessment rather than effort estimation, provides this same structure for team conversation about security. Protection Poker is conducted in an iteration[5] planning-type meeting to yield a list of the overall relative security risk for each potential requirement for the iteration. After performing Protection Poker, participants can use this relative risk to determine the type and intensity of design and the validation and verification (V&V) effort that needs to be included in the iteration for each requirement. As a result, the necessary effort to implement the requirement securely is factored into the effort estimate.

Students in an advanced undergraduate software engineering course at North Carolina State University participated in a Protection Poker session conducted as a laboratory exercise. Through Protection Poker, students structured their discussion of the security risk of the requirements of their course project, iTrust[6], a role-based healthcare application.

The rest of this paper is structured as follows. Section 2 provides background and related work. We present the Protection Poker practice in Section 3. In Section 4, we discuss our experiences with the use of Protection Poker with advanced undergraduate students. We summarize and discuss future industrial trials of Protection Poker in Section 5.

[2] A misuse case is a use case from the point of view of an actor hostile to the system under design. The goal of the misuse case is not a system function but a threat posed by that hostile actor. [1]

[3] Threat modeling is a method for uncovering design flaws in a software component before the component is built [12]. Threat modeling emphasizes risk management at the architectural/design-level where risks are assessed and mitigation steps are outlined [19].

[4] The "rules" of Planning Poker have no resemblance to the rules of the card game Poker except that each participant hides their cards from the other participants until a designated time. The creator of the Planning Poker likely chose this name to have the catchy effect of alliteration. Co-located teams often do use cards to do their estimation, though the cards must be made for the Planning Poker game to contain only selected values.

[5] An iteration is a term often used by agile software development teams to mean a relatively short period of time (typically two to four weeks) during which a development team produces working code for a predetermined set of requirements [7]. Protection Poker can be used by non-agile teams during release planning.

[6] http://agile.csc.ncsu.edu/iTrust/

2 Background and Related Work

In this section, we provide background on risk assessment in software security, Wideband Delphi estimation, and Planning Poker.

2.1 Risk Assessment in Software Security

The basis of software reliability engineering [17] is the acknowledgement that software is delivered with faults, but it is most important to prevent, find, and remove the faults that will most likely be encountered by users in normal operation. Similarly in software security, we acknowledge that software will be delivered with vulnerabilities, but it is most important to prevent, find, and remove the high risk vulnerabilities which are most likely be exploited by an attacker. If vulnerabilities must remain when the software is deployed, let them remain in low risk areas uninteresting to an attacker and least valuable to businesses.

Effective software engineering security practices include building security into the software product when finding and fixing problems is cheaper and more feasible than if done later in the software process [5]. Limited resources (e.g. time, money, and expertise) preclude software engineers from identifying and fortifying all security risks. Prioritization of software security efforts is most effectively informed by risk assessment of potential security problems [11, 12, 14, 15]. In Section 3, we suggest this risk assessment be structured via the Protection Poker practice.

Through software security [12, 14, 15] development practices, security is built into software rather than added after completion or delivery of the software. Security efforts during development should focus on the areas of a software system that are most likely to be attacked. A vulnerability[7] remains latent until an attacker has exploited the vulnerability to attack a software system. As software reliability engineering [17] places value on areas of code most used by customers, software security efforts should focus on the most risky areas of code that are most attractive to attackers and most valued by businesses. Value-neutral vulnerability-finding techniques, such as conducting the same level of penetration testing on all areas of the code, may cause the development team to expend valuable and limited security resources on low risk areas of the code that may already be adequately fortified, may be uninteresting to an attacker, or contain hard-to-exploit vulnerabilities. Software engineers can prioritize the verification and fortification of components to produce more secure software by considering risk.

2.2 Wideband Delphi Estimation

Wideband Delphi is based upon the Delphi practice [9], developed at The Rand Corporation in the late 1940s for the purpose of making forecasts. With the Delphi practice, participants are asked to make their forecast individually and anonymously in a preliminary round. The first round results are collected, tabulated, and returned to each participant for a second round, during which the participants are again asked to make a new forecast regarding the same issue. This time each participant has

[7] A vulnerability is an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [13].

knowledge of what the other participants forecasted in the first round but not any explanation by the participants of the rationale behind their forecast. The second round typically results in a narrowing of the range in forecasts by the group, pointing to some reasonable middle ground regarding the issue of concern. The original Delphi technique avoided group discussion [4].

Boehm created a variant of this technique called the Wideband Delphi technique [5]. Group discussion occurs between rounds in Wideband Delphi; participants explain why they have chosen their value. Wideband Delphi is a useful technique for coming to some conclusion regarding an issue when the only information available is based more on experience than hard empirical data [4].

2.3 Planning Poker

In recent years, many agile software development [6] teams have estimated the effort needed to complete the requirements chosen to be implemented in an iteration and/or release via a Wideband Delphi practice commonly called Planning Poker [8]. Planning Poker is “played” by the team as a part of the iteration planning meeting.

With Planning Poker, the customer or marketing representative explains each requirement to the extended development team. We use the term *extended development team* (often called the “whole team” [2] by agile software developers) to refer to all those involved in the development of a product, including product managers, project managers, software developers, testers, usability engineers, security engineers, and others. In turn, the team discusses the work involved in fully implementing and testing a requirement until they believe that they have enough information to estimate the effort. Each team member then privately and independently estimates the effort in units of “story[8] points” (discussed more fully below). The team members reveal their estimates simultaneously. Next, the team members with the lowest and highest estimate explain their estimates to the group. Discussion ensues until the group is ready to re-vote on their estimates. More estimation rounds take place until the team can come to a consensus on a quantity of story points for the requirement. Most often, only one or two Wideband Delphi rounds are necessary on a particular requirement before consensus is reached.

Planning Poker is structured such that all team members participate in the estimation process and that everybody’s opinion is heard, regardless of whether they are among the loudest or most influential people in the group [10]. *The diversity of participant opinions about the effort required to implement a requirement drives the Planning Poker discussion.* A dysfunctional Planning Poker session is one in which participants decide to go with or are implicitly or explicitly coerced into agreeing with the estimate given by a person or persons determined to be most important or most respected. As such, a culture in which a diversity of opinions is valued is necessary for Planning Poker to be an effective effort estimation technique.

In Planning Poker, estimation is based upon the notion of story points [7]. Story points are unit-less measures of effort relative to previously-completed requirements. The unit-less story points do not directly correspond to traditional effort estimates

[8] In an agile software development methodology, a “story” is analogous to a functional requirement.

such as person-hours or person-days. As a result, estimation is generally done more quickly because participants focus on relative size and not on thinking about how long the work will take. The latter might depend upon which engineer is assigned the task and what their work schedule might be. The team can focus on the estimation with discussions like the following:

- "<requirement> is similar to <other requirement> which was a 5, so we'll give this a 5"; or
- "<requirement> is likely to take twice as long as <other requirement>"; or
- "<requirement> will take the entire iteration, let's give it an 8"

Team members are constrained to estimating from a set of possible story point values on an exponential scale (most commonly 1, 2, 3, 5, 8, 13, 20, 40 and 100) [7] that are the relative amount of effort necessary for the correct implementation, including software development, usability engineering, testing, and document authoring / updating. There are two reasons behind the use of a limited set of possible values. First, humans are more accurate at estimating small things, hence there are more possible small values than large values [7]. Second, estimation can be done more quickly with a limited set of possible values. For example, why argue over whether the estimate should be 40 or 46 when our ability to estimate such large requirements is most likely inaccurate?

The values are often calibrated such that a very small task is given the value of 1 and a value of 8 indicates that the requirement will take the entire iteration. The values of 2, 3, and 5 are given relative to these endpoints. A requirement which is given an estimate of more than 8 is referred to as an *epic* [7] and can remain an epic for a future iteration. Once an epic is to be implemented in the next iteration, the epic must be broken down into small independent stories with estimates of 1, 2, 3, 5 or 8. A team computes its velocity [2, 7]; *velocity* is a historical number of how many story points the team is able to implement in an iteration. In an iteration planning meeting, the team determines which requirements to implement in the next iteration by choosing the higher priority requirements whose story points fit within the capacity determined by the velocity estimate.

There are three major benefits to using the Planning Poker practice:

1. **Effort Estimate.** The team obtains effort estimates via the expert opinion of all the members of the extended development team. The incorporation of all expert opinions leads to improved estimation accuracy [10, 16], particularly over time as the team becomes experienced with Planning Poker.
2. **Estimate Ownership.** The estimate is developed collaboratively by the extended development team. Therefore, the members will feel the estimate is realistic and will feel more accountability since they own the estimate.
3. **Communication.** The conversations that take place during the process are useful for sharing knowledge and for structuring conversation between those on the extended team with a diversity of perspectives. When one or more team members have a low estimate and others have a high estimate, team members have a very different perception of what is involved in the implementation and verification and/or have a range of technical

knowledge or experience. As such, Planning Poker provides a structured means for:

- obtaining a shared understanding;
- exposing hidden assumptions of the technical aspects of implementation and verification;
- discussing the implications throughout the system for implementing a requirement;
- surfacing and resolving ambiguities realized via divergent perspectives on the requirement; and
- exposing easy and hard alternatives for achieving desired goals.

The first author has participated in more than a dozen Planning Poker sessions conducted with industrial teams. A subjective identification of the value of Planning Poker session would be distributed as follows:

- 20% of the value is the effort estimate obtained;
- 10% of the value is that the team feels ownership of this estimate; and
- 70% of the value is the communication that takes place in the meeting through the structured Wideband Delphi process.

The motivation behind conducting Protection Poker sessions is, likewise, to structure team communication focused on software security.

3 Protection Poker

Interestingly, it may be in this dearth of "qualified" people trained in security that a critical opportunity can be found. Though few practitioners have academic security training, they most assuredly do have academic training in some field of study. That means that as a collective, the computer security field is filled with ***diverse and interesting points of view****. This is exactly the sort of Petri dish of ideas that led to the Renaissance at the end of the Dark Ages...* ***Diversity of ideas is healthy****, and it lends a creativity and drive to the security field that we must take advantage of.* *– Gary McGraw [15]*

We propose a Wideband Delphi, Planning-Poker type practice called Protection Poker that leverages a diversity of ideas, experience, and knowledge related to software security. The dual purpose of a Protection Poker session is (1) to structure a collaborative, interactive, and informal practice for misuse case development and threat modeling; and (2) to spread software security knowledge throughout a team. The output of a Protection Poker session is a list of the overall relative security risk for each potential requirement for the iteration. Protection Poker sets the stage for participants to use this relative risk to determine the type and intensity of design and the V&V effort that needs to be included in the iteration for each requirement. As a result, the necessary effort to implement the requirement securely is properly estimated and an understanding is gained of the steps necessary for this secure implementation. This list of relative risk can be used to guide the prioritization of security engineering resources towards the areas of the software with the highest risk of attack. This prioritization and increased knowledge should lead a team toward the development of more secure software.

In this section, we present instructions for conducting a Protection Poker session. We then discuss how risk assessment and knowledge transfer is achieved through the use of the practice.

3.1 Protection Poker Instructions

Protection Poker is "played" during an iteration planning-type meeting. As with Planning Poker, the extended development team (including the product manager, customer representative(s), requirements analyst(s), usability engineers and marketing representatives) is involved. Iteration planning can be augmented by the participation of a security expert if one is available, though the presence of a security expert is not an absolute necessity. Our initial trial of Protection Poker in an undergraduate class, discussed in Section 4, demonstrated the power of collaborative discussions of security issues by non-security experts. These discussions can also be aided through the use of a checklist of security issues to consider.

With Protection Poker, the customer or marketing representative explains each requirement to the extended development team. Informal discussions of misuse cases and threat models ensue. For example, the discussion might reveal that the role-based access to some functionality needs to be more restrictive.

For Protection Poker, we use unit-less measures to compute risk. Risk is traditionally [3, 19] computed as in Equation 1:

$$\text{Risk} = (\text{probability of loss})\ (\text{impact of loss}) \tag{1}$$

We propose a variation on this general definition and compute the security risk for a requirement as computed in Equation 2:

Security risk = (ease of attack) (the value of asset that could be exploited with a successful attack) (2)

The value of a particular asset does not change based upon the various requirements that use that asset. Therefore, a relative value for an asset can be established for the system based upon unit-less/relative values of *value points*. For example, the team may decide that the password table is 100 times more valuable to an attacker than the table containing statistics of baseball players and the password table would get a value of 100 and the baseball player table a value of 1. A variation of the adage "When everything is high priority, then nothing is high priority" is "When everything is very valuable, then nothing is very valuable." As a result, the team is best served by actually differentiating the value of the assets used by the software system.

In an iteration meeting, the team members provide estimates for:

1. ease of attack (higher value indicates easier to attack) in unit-less/relative values of *ease points*; and
2. the value of the asset (in value points) being accessed by/protected by the program in that requirement. The value of the asset is based upon historical values for the asset or, if a new asset must be created to implement the requirement, based upon the team's discussion/voting.

The estimation can then focus on discussions like:

- "<new requirement> increases the attack surface[9] as much as does <other requirement>"; or
- "Both of these new requirements read and write to a table with credit card information"; or
- "Only the admin can execute this functionality."

Similar to Planning Poker, the team can only choose from a set of values, such as from 1, 5, 8, 13, 20, 40 and 100 (as are used in Planning Poker) for ease points and value points. The objective of limiting the choice of values is to significantly increase the speed of development of the security risk profile such that it can be done as a part of the iteration planning meeting. Our method allows the differentiation of ease points and of value points with seven possible values. NIST advocates this differentiation of risk be limited to High, Medium, and Low [19]. Trials of Protection Poker with industry will indicate whether this additional differentiation provided by the product of ease and value aids in the ranking of security risks and whether these seven possible values are more effective for differentiating risk for prioritization purposes.

Simultaneously, the team members reveal their estimates first for ease points and then for value points for each requirement. A *Protection Poker discussion is driven by the diversity of participant opinions on the ease of attack and the value of the protected asset for a requirement.* McGraw [15] calls this time of discussion ambiguity analysis. *Ambiguity analysis* is the subprocess capturing the creative activity required to discover new risks when one or more participants share their divergent understandings of a system [15]. Disagreements and misunderstandings are often the harbingers of security risk [15]. Through Protection Poker, these disagreements and misunderstandings are surfaced and resolved before a requirement is designed or implemented. As with Planning Poker, the team members with the lowest and highest estimates explain their estimates to the group. Discussion ensues until the group is ready to re-vote on their estimates. More estimation rounds take place until the team can come to a consensus on a quantity of ease and value points for the requirement.

Values for ease points and value points would need to be calibrated at the start of the project such that an implementation of a requirement that would be very difficult to attack because it does not increase the attack surface, is given a value of 1 and the an implementation of a requirement that would be very easy to attack be given a 40 or a 100. All other requirements are estimated relative to these endpoints. The calibration can change through the course of multiple iterations.

3.2 Risk Assessment

The risk profile for a requirement is computed by multiplying ease points by value points, as shown in Table 1. The risk profile can be used to prioritize software security efforts, such as a more formal development of misuse cases or threat models; security inspection; security re-design; the use of static analysis tool(s); and security testing. The relative risk for a requirement and the resulting actions necessary to implement the requirement are factored into the effort estimate for implementing the

[9] The attack surface is the union of code, interfaces, services, and protocols available to all users, especially what is accessible by unauthenticated or remote users [12].

requirement. The necessary software security effort can be factored into the overall effort estimate such that the resources are allocated and realistic completion times are committed for the implementation of a robust, secure, fortified requirement.

Table 1. Prioritizing risk with Protection Poker

Requirement	Ease Points	Value Points	Security Risk
Requirement 1	1	1000	1000
Requirement 2	5	1	5
Requirement 3	5	1	5
Requirement 4	20	5	100
Requirement 5	13	13	169
Requirement 6	1	40	40
Requirement 7	40	20	800

3.3 Knowledge Transfer

As part of the structured Protection Poker conversation, the extended team discusses the ease points and value points for each requirement, in turn. The team can share business details of the proposed requirements; such as the assets which will be utilized (e.g. sensitive, personal information in a database), and about the technical risks that jeopardize the business details. For example, a requirement that requires that 15 user input fields which are used in dynamic SQL queries be filled in by a customer is inherently more risky than a requirement that involves only batch processing. Similarly, high-usage requirements impose a Denial of Service risk. The structured discussion of security issues that occurs as a part of Protection Poker should greatly improve the team members' knowledge and awareness of what is required to build security into the product.

4 Experience with Protection Poker in the Classroom

Protection Poker has had an initial trial with a class of approximately 50 advanced (third and fourth year) undergraduate students at North Carolina State University taking a software engineering class. Prior to coming to a structured laboratory session on software security, the students had one 50-minute lecture that provided an overview of software security. The lecture focused on input validation vulnerabilities including cross-site scripting, SQL injection, and buffer overflows.

In the laboratory, the students were given a vulnerable version of the open source iTrust role-based healthcare application. The students were asked to find vulnerabilities in iTrust, assign ease and value points to each vulnerability, and then enumerate the risk values in descending order to show that the top risks should be addressed first. The students spent 40 minutes searching for and identifying security vulnerabilities in iTrust. After a brief laboratory-wide discussion of the vulnerabilities found, students were given fifteen minutes to assign ease and value points to each vulnerability and then calculate the overall risk for that vulnerability. Two doctoral students studying software security, including the students' regular teaching assistant and his colleague, conducted the laboratory exercise. The primary educational objectives of the exercise were to (a) expose the students to common vulnerabilities in a system they are already familiar with; and to (b) elicit a discussion between the students who are more familiar with security with those who are not. While the hands-on activity of finding vulnerabilities helps students learn about security, both objectives were significantly reinforced by having the Protection Poker activity. As expected, the groups who discussed the relative risk of each vulnerability had disagreed in many cases. The group discussions that started from Protection Poker encouraged students to discover many different ways of exploiting their systems, leading them back to their computers to try to find more problems.

One of the most interesting parts of the activity was how fast general lessons about security were surfaced in the discussions. Independent of each other, many groups came to form general lessons to avoid future vulnerabilities. Among the lessons discussed were:

1. *Security cannot be obtained through obscurity alone.* Students had many discussions about using obscure (but not impossible to guess) information on their system for sensitive features (e.g. auto-incrementing database keys).
2. *Never trust your input.* Before the Protection Poker laboratory, most students never considered that their input could intentionally be malformed for the purpose of launching a security attack.
3. *Know your system.* For the first time in the course, many students felt the need to truly understand their production environment so that they could gauge ease points more accurately.
4. *Know common exploits.* Students realized on their own that as technologies evolve, so do the vulnerabilities.
5. *Know how to test for vulnerabilities.* The students discussed several strategies for adapting their automated tests to reveal possible vulnerabilities.

In addition to the general lessons learned, students discussed methods of discovering and exploiting vulnerabilities. Their discussions had a resemblance to misuse cases and threat modeling even though they were not taught either of these practices.

We observed that the Protection Poker activity provided far more discussion and learning about software security than in previous semesters without Protection Poker. In terms of lessons learned, teaching assistants usually expect one or two of the five previously-discussed lessons to be learned as opposed to the five that consistently came up. Furthermore, students were more likely to listen to each other than listen to the instructors lecturing, since they were all peers in a group. Today's students (from the Millennial Generation[10]) learn better through discovery than by being told, and they prefer learning through participation rather than by learning by being told what to do [18].

The most positive part of the lab was that many students reported the experience as an "eye opener". One student commented, "It really is amazing how such a simple

[10] Students born after 1982.

technique can yield so much." Throughout the rest of the semester, discussions with the teaching assistants regarding major security design flaws of the system took place. Although the activity was simple and natural to the students, it provided a great way to expose students to the nuanced concepts of software security.

5 Experience with Protection Poker with an Industrial Team

Protection Poker was introduced to an industrial organization via a 90-minute tutorial[11]. After an enthusiastic response to Protection Poker based upon the tutorial, plans were made for the research team to conduct Protection Poker sessions with the 11-person team during their iteration planning meetings held every two weeks. At time of writing, one Protection Poker session has been held. During this session, the team had seven new requirements in the iteration. Five of these were discussed in the Protection Poker session. The remaining two were not discussed because the hour set aside for Protection Poker had expired. The Protection Poker session yielded the following results:

- Two requirements were revised for added security fortification. One of the requirements changes involved additional logging so that an audit trail of access could be obtained. The other requirement was revised to explicitly state the need to prevent cross site scripting vulnerabilities;
- During the meeting, an engineer found a cross site scripting vulnerability in the current application at the completion of the first group discussion about cross site scripting;
- The same engineer asked for an education session on preventing cross site scripting vulnerabilities;
- A discussion ensued about the need for the governance process to prioritize the fortification of identified security vulnerabilities over the fixing of non-critical reliability defects; and
- A business analyst suggested that the testing of all new requirement involve additional scripting checks such that cross site scripting and SQL injection vulnerabilities might be discovered. Evidently, malicious scripting had not been included in their testing process to date.

The industrial team deemed the first session a success and agreed to continue to include Protection Poker in future iteration planning meetings.

6 Summary

In this paper, we have proposed the practice of Protection Poker for leveraging the diversity of experience and knowledge about software security in a team. By the use of the practice, teams can collaboratively perform a software risk assessment of the requirements for their product and can share their software security knowledge with their teammates. The result of the use of the practice should be a reduction of vulnerabilities

[11] Tutorial resources can be found here:
http://collaboration.csc.ncsu.edu/laurie/Security/ProtectionPoker/

in the product through an overall increase of software security knowledge in the team. Protection Poker was piloted with a team of undergraduate software engineering students at North Carolina State University. We observed the students discussing misuse cases and threat models and sharing their limited software security expertise as they discussed vulnerabilities in their course project.

A pilot of the use of Protection Poker with an industrial partner began in October 2008. The initial meeting supported the value of Protection Poker for structuring security risk assessment and knowledge transfer.

Acknowledgments. This work is supported in part by the National Science Foundation Grant No. 0716176. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Funding for this work has also been provided by the Center for Advanced Computing and Communications (CACC, http://www.cacc.ncsu.edu/) and the Secure Open Systems Initiative (SOSI, http://www.sosi.ncsu.edu/).

References

[1] Alexander, I.: On Abstraction in Scenarios. Requirements Engineering 6(4), 252–255 (2002)
[2] Beck, K.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Reading (2005)
[3] Boehm, B.: Software Risk Management. IEEE Computer Society Press, Washington (1989)
[4] Boehm, B., Abts, C., Chulani, S.: Software development cost estimation approaches — A survey. Annals of Software Engineering 10(1-4), 177–205 (2000)
[5] Boehm, B.W.: Software Engineering Economics. Prentice-Hall, Inc., Englewood Cliffs (1981)
[6] Cockburn, A.: Agile Software Development. Addison Wesley Longman, Reading (2001)
[7] Cohn, M.: Agile Estimating and Planning. Prentice Hall, Upper Saddle River (2006)
[8] Grenning, J.: Planning Poker or How to avoid analysis paralysis while release planning (2002) (accessed on February 26, 2008), `https://segueuserfiles.middlebury.edu/xp/PlanningPoker-v1.pdf`
[9] Gupta, U.G., Clarke, R.E.: Theory and Applications of the Delphi Technique: A bibliography (1975-1994). Technological Forecasting and Social Change 53, 185–211 (1996)
[10] Haugen, N.C.: An empirical study of using planning poker for user story estimation, in Agile 2006, Minneapolis, MN, 9 pages (electronic proceedings) (2006)
[11] Howard, M., LeBlanc, D.: Writing Secure Code. Microsoft Press, Redmond (2003)
[12] Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft Press, Redmond (2006)
[13] Krsul, I.: Software Vulnerability Analysis, in Computer Science. vol. PhD West Lafayette: Purdue University (1998)
[14] McGraw, G.: Building Secure Software. Addison Wesley, Boston (2002)
[15] McGraw, G.: Software Security: Building Security. Addison-Wesley, Upper Saddle River (2006)

[16] Moløkken-Østvold, K., Haugen, N.C.: Combining Estimates with Planning Poker – An Empirical Study. In: Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, pp. 349–358 (2007)
[17] Musa, J.D.: Software Reliability Engineering: More Reliable Software Faster and Cheaper, 2nd edn. Authorhouse, Indiana (2004)
[18] Oblinger, D., Oblinger, J.: Educating the Net Generation. Educause, Boulder (2005)
[19] Stoneburner, G., Goguen, A., Feringa, A.: NIST Special Publication 800-30: Risk Management Guide for Information Technology Syste (July 2002)

Toward Non-security Failures as a Predictor of Security Faults and Failures

Michael Gegick[1], Pete Rotella[2], and Laurie Williams[1]

[1] Department of Computer Science, North Carolina State University
890 Oval Drive, Raleigh, NC, USA
[2] Cisco Systems, Inc.
7025-6 Kit Creek Rd, Research Triangle Park, NC, USA
{mcgegick,lawilli3}@ncsu.edu, protella@cisco.com

Abstract. In the search for metrics that can predict the presence of vulnerabilities early in the software life cycle, there may be some benefit to choosing metrics from the non-security realm. We analyzed non-security and security failure data reported for the year 2007 of a Cisco software system. We used non-security failure reports as input variables into a classification and regression tree (CART) model to determine the probability that a component will have at least one vulnerability. Using CART, we ranked all of the system components in descending order of their probabilities and found that 57% of the vulnerable components were in the top nine percent of the total component ranking, but with a 48% false positive rate. The results indicate that non-security failures can be used as one of the input variables for security-related prediction models.

Keywords: Attack-prone, classification and regression tree.

1 Introduction

In the search for metrics that can predict the presence of vulnerabilities, there may be some benefit to choosing metrics from the non-security realm. Metrics can be used as input variables in statistical models to identify which software components[1] are most likely to be attacked. Such a statistical model can afford for security engineers to prioritize their efforts to the highest risk components.

According to Viega and McGraw [28] "Reliability problems aren't always security problems, though we should note that reliability problems are security problems a lot more often than one might think [28]." Therefore, when security efforts are performed on a software system, some focus toward failure-prone components may reveal that those components are also likely to be vulnerable. *The objective of this research is to create and evaluate a model that predicts which components are likely to contain security faults based on non-security failures.*

We analyzed pre- and post-release non-security failure data and pre- and post-release security fault and failure data of a Cisco software system[2] to determine if

[1] A component is one of the parts that make up a system [15].
[2] Due to the sensitivity of the security-related data, details of the system and data are omitted.

non-security problems are associated with security problems. Since security faults are typically far fewer in number than non-security faults [1], not all of the failure-prone components will be associated with security faults. To be useful, the statistical model will have to determine which of the failure-prone components are also likely to be vulnerable. We constructed a model where the input variables to the model are non-security failures and the output of the model is a probability that a component in the system has at least one security fault.

The remainder of this paper is organized as follows. In Section 2 we provide background and related work, in Section 3 we detail the industrial case study, in Section 4 we present results, in Section 5 we present the limitations of the study, in Section 6 we provide a discussion, and finally in Section 7 we summarize and provide future work.

2 Background

In this section, we provide definitions of terms used throughout the paper. Seminal sources are used for each definition where possible. We also include prior work that compares security with reliability.

2.1 Definitions

External metrics. "Those metrics that represent the external perspective of software quality when the software is in use…These measures apply in both the testing and operation phases." [14]

Internal metrics. "Those metrics that measure internal attributes of the software related to design and code. These "early" measures are used as indicators to predict what can be expected once the system is in test and operation" [14].

Fault. "An incorrect step, process, or data definition in a computer program. Note: A fault, if encountered, may cause a failure" [15].

Fault-prone component. "A component that will likely contain faults" [4].

Failure. "The inability of a software system or component to perform its required functions within a specified performance requirements [15]."

Failure-prone component. A component that will likely fail due to the execution of faults [26].

Vulnerability. An instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [17].

Vulnerability-prone component. A component that is likely to contain one or more vulnerabilities that may or may not be exploitable [8].

Attack. The inability of a system or component to perform functions without violating an implicit or explicit security policy. We borrow from the ISO/IEC 24765 [15] definition of failure to define attack, but remove the word "required" because attacks can result from functionality that was not stated in the specification.

Attack-prone component. A component that will likely be exploited [8].

An attack-prone component is a component that is likely to be exploited due to the types of vulnerabilities in that component. For example, the vulnerabilities may be easy to find, easy to exploit, or lead to desirable assets. A vulnerability-prone component that is not also attack-prone may contain vulnerabilities that are not easily found, are difficult to exploit, or do not lead to desirable assets. These characteristics represent our initial views of vulnerability- and attack-prone components [12].

In our setting, an attack is the execution of a security fault (vulnerability) that leads to a security failure. We use the context of execution to be consistent with the definitions and distinction between a fault and failure in the general reliability (non-security) context under the assumption that security is, by definition, a subset of reliability. System execution occurs during testing, internal usage, and in the field. In the context of testing, if a tester discovers a buffer overflow, then we say they have attacked the system. Although the tester may not have gone through the trouble of completely exploiting the buffer overflow to cause a denial-of-service or to inject code that escalates their privileges, the failure is a proof of concept that the system can be attacked. A risk value can be assigned to the security fault to describe how detrimental the vulnerability is to the system. Of course, static fault detection techniques can identify vulnerabilities that can be exploited, too.

2.2 Prior Work

The first and third authors performed two case studies on two different large commercial[3] telecommunications systems. The correlation between non-security and security failures was examined [9, 12]. We found a 0.8 ($p<.0001$) Spearman rank correlation between non-security system/feature failures and security failures for the first system and a 0.7 ($p<.0001$) correlation for the second system. For these two case studies the only available data were system/feature testing failures. The high correlations suggest that non-security failures are a good indicator of security problems and that security fortification efforts should be placed in the same areas of the software as reliability efforts. The case study presented in this paper attempts to replicate our previous studies on a different software system from a different vendor to determine if the statistical model yields consistent results.

Research papers comparing security failure data to non-security data are showing that reliability and security models are not dissimilar. Alhazmi et al. [1] compared the cumulative number of vulnerabilities for five different operating systems and found that the plots are analogous to reliability growth plots using logistic and linear models. Mullen et al. [18] have found the occurrence rate of security vulnerabilities follows the Discrete Lognormal distribution, which has also been shown in prior reliability growth, test coverage, defect failure rate, and code execution rates. Condon et al. [3] have found that security incident data can be modeled with Non-Homogenous Poisson Process models as done with reliability failure data. Lastly, Ozment and Schechter [23] found that Musa's Logarithmic model fit their OpenBSD security dataset to predict time-between-security-failures. We continue the examination of potential parallels between non-security and security problems by investigating if the *location* of security faults and failures can be approximated using non-security failure data.

[3] Due to the sensitivity of the data, the name of the vendor is omitted.

2.3 Vulnerability- and Attack-Prone Component Predictions

Neuhaus et al. [21] have also investigated predictive models that identify vulnerability-prone components. They created a software tool, Vulture, that mines a bug database for data including libraries and APIs which components are likely vulnerable. They performed an analysis with Vulture on Bugzilla, the bug database for the Mozilla browser, using imports and function calls as predictors. They were able to identify 45% of all of the vulnerable components in Mozilla. Shin and Williams [27] found a weak correlation (0.2) between complexity and security vulnerabilities in Mozilla, indicating that complexity contributes to security problems, but is not the only factor. We also found a 0.2 correlation between file coupling and vulnerability counts in a large telecommunications system [10]. In that case study, we used a classification and regression trees (CART, as discussed in Section 2.4) model to assign a probability of attack to each file. Upon ranking these probabilities in descending order, we found that 72% of the attack-prone files are in the top 10% of the ranked files and 90% are in the top 20% of the files. The input variables for that study consisted of the count of Klocwork[4] static analysis tools warnings, measure of file coupling, and count of added and changed source lines of code. In our other earlier work [11] we used a CART to predict which components were attack-prone using warnings from the static analysis tool, FlexeLint, and code churn. The model identified all of the attack-prone components, but with an 8% false positive rate. The study in this paper is based on a different type of system than our earlier studies.

2.4 Classification and Regression Trees (CART)

Our predictive model is comprised of a statistical technique and the independent variable non-security failure count. CART is a statistical technique that recursively partitions data according to X and Y values. The result of the partitioning is a tree of groups where the X values of each group best predicts a Y value. The leaves of the tree are determined by the largest likelihood-ratio chi-square statistic. The threshold or split between leaves is chosen by maximizing the difference in the responses between the two leaves [25]. For the case study in this paper, the X values are values from the non-security failures and the Y value is a binary value describing a component as attack-prone or not attack-prone. The CART technique has been shown to be useful for distinguishing failure-prone from not failure-prone components in the reliability realm [29].

3 Cisco Case Study

We analyzed pre- and post-release non-security failure data and pre- and post-release security fault and failure data that were submitted to the Cisco fault-tracking database in 2007 for a typical Cisco software system. The software system was divided into clearly defined components against which failures were reported. Each component consists of multiple files. The count of components was large enough to perform rigorous statistical analyses.

[4] http://www.klocwork.com/

3.1 Non-security External Metrics as Predictors of Security Faults and Failures

The non-security failure reports were obtained from the Cisco fault-tracking database. The reports we used in our study included all severity 1, 2, and 3 non-security failure reports for the software system, where severity 1 is the highest impact to the customer. Most severity 1, 2, and 3 records in the fault-tracking database indicated actual problems in the software. Records with higher severity numbers had a stronger chance of being a feature request. During our failure report analysis, we eliminated duplicate failure records that represented a failure already reported in the system.

The non-security failure reports include failures observed during unit testing, function testing, performance testing, system testing, stress testing, alpha testing, beta testing, automated regression, internal use failures, early field trials, and customer-reported failures. Alpha testing is conducted on a production network within Cisco while beta testing is conducted on the customer site. The number and types of non-security failure reports are not disclosed for confidentiality reasons.

3.2 Security Fault and Failure Data

The security faults and failures are the dependent variables in this industrial case study. We included security faults and failures of all severity levels. We chose to study security faults and failures reported for the duration of a year (2007) to strengthen the goodness-of-fit of the predictive model we will build. Security faults and failures are rare events and difficult to model with small sample sizes. The number and types of security fault and failure reports are not disclosed for confidentiality reasons.

The security faults and failures were provided by the Cisco Security Evaluation Office that handles security data. The security faults in our study were reported during *static* inspections that were performed during the design and development stages of the software life cycle (SLC). The security failures were identified during system *execution* and included problems from pre- and post-release testing and also include those security failures reported in the field.

In our setting, an attack-prone component is a component that contains at least one security fault or a security failure. We use the term attack-prone component instead of vulnerability-prone because most security faults were identified during system *execution*. Additionally, the attack-prone components had at least one security failure identified during pre- or post-release execution. We use the threshold of one security failure because there is little variability in the failure count per component and only one attack is needed to cause substantial business loss. Although some security failures were reported by customers, there was no evidence of successful attacks against the software. A component with no reported security faults or failures will be called a not attack-prone component in this paper.

4 Results

The analysis of the failure reports indicated that only a small percentage of the components consists of at least one security fault or failures. According to Pareto's law, 80% of the outcomes will be derived from 20% of the activities [6]. Although, this observation was originally described in the context of economics, it has also been observed in the context of faults in a software system [22]. The application of the law to the software setting is that software problems will not be evenly distributed across the software system. For example, in a survey of multiple software systems, it was shown that between 60% and 90% of software faults are due to 20% of the modules [2]. We observed Pareto's Law in our setting (see Section 4.2) because the distribution of attacks among the components is not evenly distributed across all components. All results in the sections below are reported on a per component basis.

The first analysis in our case study was to perform correlations between all of our non-security failures types (listed in Section 3.1) and counts of security faults and failures. The correlations with the highest coefficients will aid in independent variable selection during the construction of the model. Our statistical model will be a discriminatory model that classifies a component as attack-prone or not attack-prone. Associated with the classification is a probability of the component being attack-prone. In the event that the models cannot successfully discriminate between attack-prone and not attack-prone components, the correlations may indicate that a statistical technique does not perform well for the given dataset. For example, if we observe a high correlation between non-security failures and security faults and failures, but the discriminatory statistical approach cannot discriminate between attack-prone and not attack-prone components, then we would try a different statistical technique.

4.1 Correlations

We calculated[5] Spearman rank correlations to determine if an increase of non-security failures in a component is followed by an increase in count of security faults and failures for that component. The highest correlation, 0.4 (p<.0001), occurred between customer-reported non-security failures and the sum of security faults and failures as shown in Table 1. The correlations to the security fault and failure counts are low, but they are significant and represent that they have value for indicating the existence of security problems in a statistical model.

Table 1. Correlations between the non-security failures and vulnerabilities

Count of non-security failures	Spearman rank correlation coefficient (p-value)
customer-reported	0.4 (p<.0001)
alpha testing	0.3 (p<.0001)
total non-security	0.3 (p<.0001)
internal use	0.3 (p<.0001)
system testing	0.2 (p<.0001)
performance testing	0.1 (p<.0001)
stress testing	0.1 (p<.0001)
beta testing	0.1 (p<.0001)
function testing	0.1 (p=.04)
early field trial testing	0.1 (p<.0005)

[5] All statistical analyses performed on SAS JMP 7.0.1.

4.2 Classification of System Components

We performed classification analyses to discriminate between attack-prone components and not attack-prone components based on non-security failures. We built models using the discriminant analysis, logistic regression, and CART with the non-security failures enumerated in Section 3.1 as input variables. CART showed better separation between attack-prone and not attack-prone components than discriminant analysis and logistic regression. The non-security failure types that had the most predictive power in the CART model were alpha and beta testing non-security failures and customer-reported non-security failures.

The CART analysis splits the all of the system components into like groups based on the count of non-security alpha and beta testing failures and customer-reported non-security failures. The splits made in CART are shown in Appendix A. The values of *w*, *x*, *y*, and *z* are integer values and are not provided for confidentiality reasons.

In our analysis, the vulnerabilities were most likely to be in components where there are more than *x* customer-reported failures as denoted by the first (top most) split in the tree (see Appendix A). Failures from alpha and beta testing contributed less to the isolation analysis. The other failure types enumerated in Section 3.1 could not split the leaves to achieve separation between attack-prone and not attack-prone components as well as alpha and beta testing failures and customer-reported failures.

The predictive power of the metrics is measured by the likelihood-ratio chi-square, G^2, of each input variable. A larger G^2 value indicates a more optimal split of a leaf in the CART analysis between attack-prone and not attack-prone components. In our model, the customer-reported problems contributed the most fit or separation in the overall model as shown in Table 2.

Table 2. Contribution of metrics to the model

Non-security failure count	**Number of splits in tree**	**G^2**
customer-reported	3	205.7
alpha testing	1	7.4
beta testing	1	4.7
Total	5	217.8

Table 3. Tests for collinearity in the independent variables

Failure type	**Failure type**	**Spearman rank correlation coefficient**
alpha testing	customer-reported	0.2 ($p<.0001$)
beta testing	alpha testing	0.1 ($p<.0001$)
customer-reported	beta testing	0.1 ($p<.0001$)

We tested the input variables of the CART model to test for collinearity. Collinearity is defined as a high degree of correlation between the independent variables of a statistical model [7]. Collinearity occurs when an excessive number of input variables are used to determine an outcome, and the input variables measure the same outcome [7]. The highest correlation between our input variables, 0.2, existed between alpha testing and customer-reported failures and is a low correlation. The low correlations shown in Table 3 indicate that the failures identified by alpha and beta testing and customer-reported failures are measuring different types of failures or failures in different locations in the software. We included these input variables in the model because the correlations between them are low and thus the collinearity among them is small.

The probability of an attack-prone component for a given leaf is given in Table 4. All components in a leaf have the same probability of being attack-prone. For example, in Leaf 1 100% of the components are attack-prone. In this leaf, all components should undergo security analyses. In Leaf 2 64% of the components are attack-prone and all components should go under security analyses, but 36% of the components will either not contain security faults or failures or contain security faults that are difficult to exploit or uninteresting to an attacker. The 36% false positive rate represents that time and effort spent on some components will not contribute greatly to the overall security posture of the software system. As shown in Table 4, there is a general likelihood ranking of attack-prone components. The components in Leaf 1 have the highest rank (probability of being attack-prone) and the components in Leaf 6 have the lowest rank.

In examining the efficacy of the model, the security efforts should be focused to all of the components in the first four leaves of the tree because the true positive rate (probability of finding an attack-prone component) is relatively high. In Leaves 5 and 6, the probability of identifying an attack-prone component is only ten percent and one percent, respectively, representing that most security efforts would be wasted on low risk components. If we accept that the components in Leaves 1-4 are all attack-prone, then the model will have isolated 57% of the attack-prone components in the top nine percent (components in the top nine percent of the leaf-based ranking) of the system component ranking. Leaves 1-4 have a 48% Type I error (false positive) rate where not attack-prone components are interpreted by the model as attack-prone components. The remaining 43% of the vulnerable components are in Leaves 5 and 6 and represent the Type II error (false negative) rate. These attack-prone components would escape security efforts because they are in large groups of components with no reported vulnerabilities. Security engineers would not likely accept a scenario where most of their analyses are spent on components with low risk.

Table 4. Attack-prone probabilities in the leaves of the tree. The non-shaded rows represent the total system components in the top nine percent of the probability ranking. "Customer" represents the count of customer reported problems. "Alpha" represents the count of alpha testing failures. "Beta" represents the count of beta testing failures.

Leaf Number	**Leaf Label**	**Probability not attack-prone**	**Probability attack-prone**
1	customer=x&alpha>=y&beta>=y	0.00	1.00
2	customer>=x&alpha>=1&beta<y	0.36	0.64
3	customer>=x&alpha<1&customer>=z	0.27	0.73
4	customer>=x&alpha<1&customer<z	0.68	0.32
5	customer<x&customer>=w	0.90	0.10
6	customer<x&customer<w	0.99	0.01

The goodness-of-fit of the model can be determined by the receiver operating characteristic (ROC) curve. The ROC graph has the true positive rate on the y-axis and the Type I error rate on the x axis. The larger the area under the ROC curve, the better the goodness-of-fit. In Figure 1, the ROC curve for our model, represented by the thick line, has 88% of the area under the curve indicating that the non-security failures are a good metric for predicting which components are attack-prone. The thin line is a reflection of the solid curve to show the model's ability to classify not attack-prone components. The diagonal line represents the efficacy of the model if the predicted outcomes correctly identified 50% of the attack-prone components.

The R^2 value for the overall model is 36% indicating that not all of the variation in the data can be accounted for by CART. To validate the model, we performed five-fold cross-validation. Five has been shown to be a good value for performing cross-validation [13]. The 36% R^2 value we observed was based on the entire dataset. The cross-validation technique validates the R^2 value by testing the model on data the model has not used before to determine if the model is still effective [30]. The five-fold cross-validation divides ("folds") the total system components into five groups consisting of an approximately equal number of randomly chosen components. One group is used as the test set and the training set consists of the remaining four groups of components. The model is trained on the training set and the analysis is compared to the outcomes of the test set to validate how well the model performs on data that has not been "seen" before by the model. Each of the five groups of components has one turn to be the test set which requires five analyses. After the five analyses are performed, the average error is calculated over the five trials. The cross-validated R^2 value was 34% indicating that the overall model is consistent with the model produced with the entire dataset. Despite the low R^2 values, all of the splits in the CART analysis were performed at or below the .05 significance level representing that each leaf split (separation between attack-prone and not attack-prone components) is statistically significant.

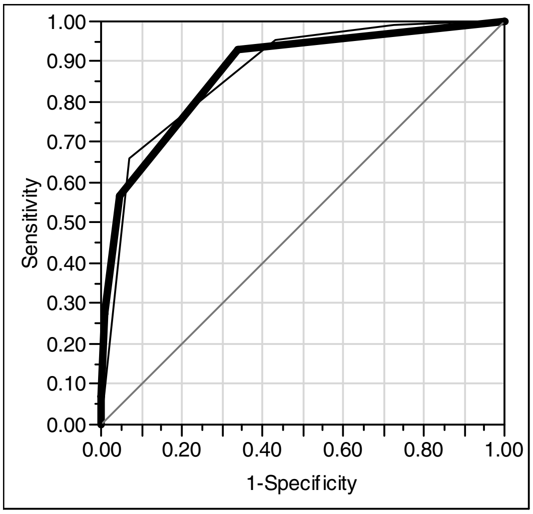


Fig. 1. The ROC curve of the CART analysis

5 Limitations

We cannot claim to have identified all faults in the software based on the failures that have surfaced during testing [5]. Additionally, the customer reported failures do not complete the identification of all non-security faults as predictors or all security faults and failures as dependent variables. Moreover, the testing effort may not have been equal for all components and thus components with fewer failures may appear more reliable or secure. Therefore, our analyses are based on incomplete data. The Type I (48%) and Type II (43%) error rates are high indicating that the model is not precise which, if applied at Cisco, could lead to effort wasted on low security risk components while some attack-prone components are never found. Additional metrics in a statistical model may help identify attack-prone components with lower Type I and Type II error rates. Furthermore, there are few security data making statistical analyses difficult. Lastly, the model presented in this paper is representative of one industrial software system and will not necessarily yield the same results on different software systems.

6 Discussion

In a Mozilla case study [27] and our earlier telecommunications system case study [11] the analyses showed that there is only a 0.2 correlation between complexity measures and security faults and failures. We observed that the 0.4 correlation between non-security failures and security failures in the software system is higher than complexity-related correlations. Further analysis is required to determine how complexity metrics correlate to security faults and failures in the system we studied. The higher the correlation, the more the predictor can contribute in a predictive model. While 0.4 correlation is low, it is significant, as are the complexity correlations. Combining these predictors into one model may build a useful model that has lower Type I and Type II error rates than a model with just one predictor.

According to Table 4, 57% of all attack-prone components were associated with greater than x (a value determined by the CART model) customer-reported non-security failures. Furthermore, Table 2 indicates that customer-reported non-security failures have the most ability to split components into groups of attack-prone and not attack-prone components. We provide three possible explanations for this observation. First, these observations indicate that the customer's operational profile[6] (usage) influences vulnerable execution paths not identified the testing techniques listed in Section 3.1. The more execution in those required features could increase the chance of a deviant operation profile that opens a security hole for an attacker.

The second possible reason why attack-prone components are associated with customer-reported failures is that they are the components that are most important (i.e. the reason the software was built) to the customer and thus those with the largest business risk. Therefore, failures with these components would more likely be considered security problems because they can be exploited by attackers to interrupt the software functions required by the customer. Failures in components that have less importance to a customer may be less likely to be a security problem because the

[6] The complete set of operations (major system logical tasks) with their probabilities of occurrence [19].

impact of the failure does not preclude the customer from performing important tasks. However, the data indicate that 43% of the attack-prone components are associated with components with fewer than x customer-reported failures and show that the system can be exploited via components that are not as frequently as the others. Given the 57% and 43% percentages, security efforts should prioritize against the features that the customers use the most, but not exclude those components that are used less by the customers as they can also impact the customer.

Thirdly, the 0.4 correlation (a weak, but significant correlation) between non-security failures and security faults and failures may indicate that components with the most customer-reported failures are associated with deficiencies in the software process that lead to less reliable code. Gaps in the software process can lead to either the injection of a fault or the failure to remove a fault. The correlation between non-security failures and security faults and failures may indicate that the gaps in the software process lead to both reliability and security failures. The less failure-prone components (i.e. those with fewer or no security faults) indicate that the development groups with a stricter software process mitigate non-security problems at the same time as security problems, perhaps without realizing that some of the risks they encountered were security-related. For example, if architectural risk analyses are not performed during design, then design flaws may not found until late in the software process when it is too late to change the design of the system. The design flaws may lead to unreliable functionality or a vulnerability that an attacker can exploit.

In our earlier work [11][7], we observed a 0.4 correlation between static analysis tool warnings and vulnerabilities found during testing and in the field. We did not observe a correlation between code churn and vulnerability counts. In our other work [10], we observed a 0.2 correlation between static analysis tool warnings and vulnerability counts. We also observed a 0.4 correlation between code churn and vulnerability counts. The vulnerability counts in these two datasets were small and thus may have hindered the identification of a stronger correlation between the predictors and vulnerability counts. Correlations between the same predictors and non-security faults/ failures have been reported to be much stronger than the measurements presented in this paper. For example, Zheng et al. [31] observed a 0.73 correlation between static analysis tool warnings and testing and customer failures. Nagappan and Ball [20] observed correlations as high as 0.883 between code churn measures and general reliability defects/KLOC. If the static analysis tool warnings and code churn are strongly correlated to non-security problems and non-security problems are correlated to security problems, then static analysis tool warnings and code churn in our earlier work ([10, 11]) may have a stronger impact on security problems than what the correlations indicate. If true, the extensive research on reliability statistical models (e.g. [16, 22]) that have been shown to predict fault- and failure-prone components early in the SLC may also be helpful for security prediction models. The models can be modified to isolate security problems, or if we assume that the security faults cluster with the non-security faults, then security engineers can focus their efforts to the components predicted to be the most failure-prone by the reliability-based prediction model.

[7] For the detailed version of this paper, see: M. Gegick, L. Williams, and J. Osborne, "Predicting Attack-prone Components with Internal Metrics," NC State University, Raleigh, TR-2008-08, 25 February 2008.

7 Summary and Future Work

We analyzed a Cisco software system to determine if non-security problems are associated with security problems. We found a 0.4 correlation between security faults and failures and non-security failures suggesting that general reliability of a software component is an indicator of the security posture of that component. Our CART model shows that alpha and beta testing failures and customer-reported failures can discriminate between attack-prone and not attack-prone components. Additionally, the model provides a threshold of non-security failure counts "required" to have a security fault or failure. This threshold is useful for determining which of the failure-prone components should receive security attention in application of the idea suggested by Viega and McGraw [28] in the Introduction. The model correctly identified 57% of the attack-prone components in the top nine percent of the components when ranked by probability of being attack-prone. The CART analysis showed that the best indicator of security faults and failures were those components with the most customer-reported failures. This observation suggests that the customers' operational profiles may influence vulnerable execution flows in the software that were not identified during pre- or post-release testing. We conclude that non-security failures and the predictors of non-security failures are potential metrics security-related predictive models. Given that reliability and security problems exist in the same locations, it may be worthwhile to unify the concepts of "software reliability engineering" and "software security engineering" into a single theme (e.g. *software assurance engineering*) to indicate that security and reliability folks should collaborate in the same sections of the software system and that security should be kept in mind when the system becomes unreliable. Next, we will examine if the non-security failures in our dataset can actually afford an attacker a means to exploit the software system. The initially classified non-security failures may provide opportunities for new types of security attacks. These security holes may result from not abiding by the principle of fail-safe defaults because the failure to perform the required functions within the specified performance requirements opened a security hole. Fail-safe defaults is a security design principle [24].

References

[1] Alhazmi, O.H., Malaiya, Y.K., Ray, I.: Measuring, analyzing and predicting vulnerabilities in software systems. Computers & Security 26(3), 219–228 (2006)

[2] Boehm, B., Basili, V.: Software Defect Reduction Top 10 List. IEEE Computer 34(1), 135–137 (2001)

[3] Codon, E., Cukier, M., He, T.: Applying Software Reliability Models on Security Incidents. In: International Symposium on Software Reliability Engineering, Trollhattan, Sweden (2007)

[4] Denaro, G.: Estimating software fault-proneness for tuning testing activities. In: International Conference on Software Engineering, St. Malo, France, pp. 269–280 (2000)

[5] Dijkstra, E.: Structured Programming, Brussels, Belgium (1970)

[6] Endres, A., Rombach, R.D.: A Handbook of Software and Systems Engineering, Harlow, England. Pearson Education, Limited, London (2003)

[7] Freund, R., Littell, R., Creighton, L.: Regression Using JMP, Cary, NC. SAS Institute, Inc. (2003)
[8] Gegick, M., Williams, L.: Toward the Use of Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components. In: First International Workshop on Systems Vulnerabilities (SYVUL 2007), Santa Clara, CA, July 1-6 (2007)
[9] Gegick, M.: Failure-prone Components are also Attack-prone Components. In: OOPSLA - ACM student research competition, Nashville, Tennessee, October 2008, pp. 917–918 (2008)
[10] Gegick, M., Williams, L.: STUDENT PAPER: Ranking Attack-prone Components with a Predictive Model. In: International Symposium on Software Reliability Engineering, Redmond, WA, November 2008, pp. 315–316 (2008)
[11] Gegick, M., Williams, L., Osborne, J., Vouk, M.: Prioritizing Software Security Fortification through Code-Level Security Metrics. In: Workshop on Quality of Protection, Alexandria, VA, pp. 31–37 (2008)
[12] Gegick, M., Williams, L., Vouk, M.: Predictive Models for Identifying Software Components Prone to Failure During Security Attacks (2008) Build Securit In, https://buildsecurityin.us-cert.gov/daisy/bsi/home.html
[13] Hastie, T., Tibshirani, R., Friedman, J.H.: The Elements of Statistical Learning. Springer, New York (2001)
[14] ISO, ISO/IEC DIS 14598-1 Information Technology - Software Product Evaluation - Part 1: General Overview (October 28, 1996)
[15] ISO/IEC 24765 Software and Systems Engineering Vocabulary (2006)
[16] Khoshgoftaar, T.M., Allen, E.B., Naik, A., Jones, W., Hudepohl, J.P.: Using Classification Trees for Software Quality Models: Lessons Learned. International Journal on Software Engineering and Knowledge Engineering 9(2), 212–231 (1999)
[17] Krsul, I.: Software Vulnerability Analysis, PhD Thesis in Computer Science at Purdue University, West Lafayette (1998)
[18] Mullen, R., Gokhale, S.: A Discrete Lognormal Model for Software Defects Affecting QoP. Quality of Protection, Milan, Italy, September 15 (2005)
[19] Musa, J.D.: Software reliability engineering: More reliable software faster and cheaper, 2nd edn., Bloomington, Indiana, AuthorHouse (2004)
[20] Nagappan, N., Ball, T.: Use of Relative Code Churn Measures to Predict Defect Density. In: International Conference on Software Engineering, St. Louis, MO, May 15-21, 2005, pp. 284–292 (2005)
[21] Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting Vulnerable Software Components. In: Computer and Communications Security, Alexandria, VA, 29 October-2 November 2007, pp. 529–540 (2007)
[22] Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Where the bugs are. In: International Symposium on Software Testing and Analysis, Boston, Massachusetts, pp. 86–96 (2004)
[23] Ozment, A., Schechter, S.: Milk or wine: does software security improve with age? In: 15th Conference on USENIX Security Symposium, July 2006, pp. 93–104 (2006)
[24] Saltzer, J., Schroeder, M.: The Protection of Information in Computer Systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
[25] Institute Inc, S.A.S.: The Partition Platform. SAS Institute, Inc., Cary (2003)
[26] Schroter, A., Zimmermann, T., Zeller, A.: Predicting Component Failures at Design Time. In: International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil, September 21-22, 2006, pp. 18–27 (2006)
[27] Shin, Y., Williams, L.: Is Complexity Really the Enemy of Software Security? In: Workshop on Quality of Protection, Alexandria, VA, pp. 47–50 (2008)

[28] Viega, J., McGraw, G.: Building Secure Software How to Avoid Security Problems the Right Way. Addison-Wesley, Boston (2002)

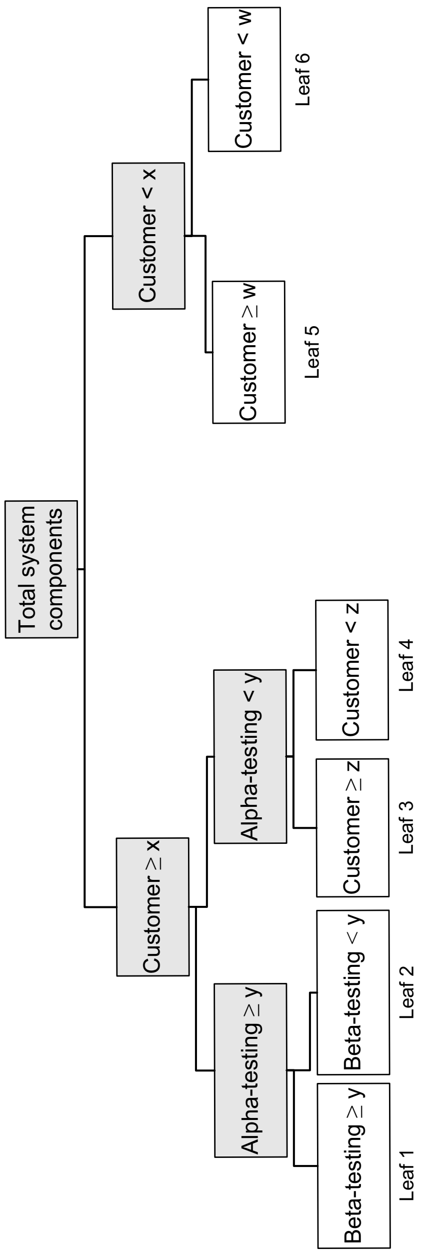
[29] Vouk, M., Tai, K.C.: Some Issues in Multi-Phase Software Reliability Modeling. In: Center for Advanced Studies Conference (CASCON), Toronto, October 1993, pp. 512–523 (1993)

[30] Witten, I., Frank, E.: Data Mining, 4th edn. San Francisco. Elsevier, Amsterdam (2005)

[31] Zheng, J., Williams, L., Snipes, W., Nagappan, N., Hudepohl, J., Vouk, M.: On the Value of Static Analysis Tools for Fault Detection. IEEE Transactions on Software Engineering 32(4), 240–253 (2006)

Appendix A: Classification and Regression Tree

Non-shaded boxes represent leaves of the tree.



A Scalable Approach to Full Attack Graphs Generation

Feng Chen, Jinshu Su, and Yi Zhang

School of Computer National University of Defense Technology,
Changsha, 410073, China
{chenfeng,sjs,zhangyi}@nudt.edu.cn

Abstract. Attack graphs are valuable vulnerabilities analysis tools to network defenders and may be classified to two kinds by application. One is the partial attack graphs which illustrate the potential interrelations among the known vulnerabilities just related to the given attack goal in the targeted network, the other is full attack graphs which evaluate the potential interrelations among all the known vulnerabilities in the targeted network. The previous approaches to generating full attack graphs are suffering from two issues. One is the effective modeling language for full attack graphs generation and the other is the scalability to large enterprise network. In this paper, we firstly present a novel conceptual model for full attack graph generation that introduces attack pattern simplifying the process of modeling the attacker. Secondly, a formal modeling language VAML is proposed to describe the various elements in the conceptual model. Thirdly, based on VAML, a scalable approach to generate full attack graphs is put forward. The prototype system CAVS has been tested on an operational network with over 150 hosts. We have explored the system's scalability by evaluating simulated networks with up to one thousand hosts and various topologies. The experimental result shows the approach could be applied to large networks.

Keywords: Vulnerability, full attack graph, scalable, modeling language.

1 Introduction

With the rapid progress of vulnerability scan techniques, identifying software vulnerabilities in the enterprise network is easier than before. However, evaluating the security threat resulting from the vulnerabilities is still a difficult work. There are many potential interactions among multiple hosts and components in a network, such that the vulnerability of one machine will affect the security of others in the network. To expose the potential interrelations among all the known vulnerabilities in the targeted network, attack graphs that enumerate all possible multi-step, multi-host attack paths are effective tools for security administrator to understand the nature of the threats and decide upon appropriate countermeasures.

Attack graphs may be classified to two kinds by application. One is the *partial attack graphs* which illustrate the potential interrelations among the known vulnerabilities just related to the given attack goal in the targeted network [1,2,3,11], the other is *full attack graphs* which evaluate the potential interrelations among all the known vulnerabilities in the targeted network[5,6,7,8]. *attack graphs* generation has experienced the stage of manual [1] to antomatic [2,3,5,6,7,8,11] generation and the stage of generation for the small-scale [1,2,3,6] to the large-scale [7,8] target network. However, it is still suffering from two key challenges [4]. First, although many of current approaches addressing the scalability problem has made progress, especially the work by Xinming Ou *et al.* in [7,8] with the scalability to large, enterprise-size networks with thousands of hosts, they just could produce the *partial attack graphs* which enumerate all the attack paths to the given attack goal and has no capability of exposing all the security threat raised by vulnerabilities in the target network. There is a lack of effective approaches to generating *full attack graphs* for the enterprise networks with thousands of hosts. Secondly, although some of approaches have made use of the formal language to describe the model involving the target network and the ability of the attacker, e.g. Datalog language in [7,8] and CTL language in [2,3], they only support the partial attack graphs generation, and do not support full attack graphs generation. Thus, it is necessary to design an effective modeling language for the issue of full attack graphs generation.

In this paper, we first present a novel conceptual model for attack graph generation that introduces *attack pattern* simplifying the process of modeling the attacker and formally describe full attack graphs to compactly represent all the attack paths. Secondly, the novel modeling language VAML is proposed, that allows the formal definition of the various elements in the conceptual model in a declarative manner. Thirdly, we put forward a scalable approach to build full attack graphs with the expected worst-case performance $O(MN^3)$, where N is the number of hosts in the network, M is the number of attack patterns.

Furthermore, two optimization techniques are proposed to remarkably improve this approach's performance in the general case. The prototype system CAVS (Comprehensive Vulnerability Analysis System) has been tested on an operational network with over 150 hosts. We have explored our system's scalability by evaluating simulated networks with up to one thousand hosts and various topologies. The experimental result shows the approach could be applied to large enterprise networks.

The rest of this paper is organized as follows. The next section reviews related work. Section 3 presents conceptual model for attack graph generation. Section 4 discusses the modeling language VAML for describing the various elements in the conceptual model. Section 5 proposes the approach to generating attack graphs. Section 6 discusses the performance analysis and experiment result of the proposed method. Finally, Section 7 concludes the paper.

2 Related Work

Phillips and Swiler proposed a concept of attack graphs in [1]. They use the nature language to describe the network and attacker's actions and build attack graphs by hand.

Ritchey and Ammann [2] used model checking techniques to find a counterexample to an asserted security policy. They consider interconnected network of computers with known vulnerabilities that can be combined by hackers in order attack one or more hosts. Every attack goal is encoded as an in computation tree logic (CTL). Sheyner *et al.* modify the model checking NuSMV to produce all the counterexample for generating attack graph[3]. Although model checking is more powerful, it scales very poorly for this application, as noted in [4] and elsewhere.

Ammann *et al.* in [5] introduce the monotonicity assumption, i,e., an attacker never relinquishes any obtained capabilities. Based on the assumption, they developed an algorithm which scales as roughly O(N^6) [4], but is capable of building the full attack graph. Jajodia *et al.* [6] adopt the algorithm and use Nessus scans to identify some vulnerability locations and reachability. Both of them adopt an ad-hoc way to represent input information and output graph data structures and there is no evidence to show they could scales to the network with one thousand hosts. Though our approach is also based on the assumption, it has more efficiency and scalability.

An approach due to Ou et al., called MulVAL [7], uses a monotonic, logic-based approach. The MulVAL reasoning engine uses XSB, a Prolog system developed by Stony Brook, to evaluate the Datalog interaction rules on input facts. They modified the MulVAL engine so that the trace of the evaluation is recorded and sent to the graph builder, where the logical attack graph is output. The results shown in [8] imply a runtime between O(N^2) and O(N^3). Since the input information of the MulVAL must specify the security policy, they can but generate partial attack graph enumerating all possible attack path to the goal in the policy.

Templeton *et al.* present prerequisite/postcondition model for attack components in [9]. Considering the attack process and detection process, Cuppens and Ortalo's LAMBDA language [10] provide detailed models of vulnerability and attacker action. Our modeling language VAML is based on their work, but LAMBDA is so complicated that it is never adapted to generate attack graphs.

The analysis of attack graphs has been used for different purposes in defending against network intrusions. The minimal critical attack set analysis finds a minimal subset of attacks whose removal prevents attackers from reaching the goal state [3, 11]. The minimum-cost hardening solution discovers a minimal set of independent security conditions [12]. In exploit-centric alert correlation [13, 14], attack graphs assist the correlation of isolated intrusion alerts. Finally, Using attack graphs to measure network security seem to be an effective approach in [15].

3 Conceptual Model

Figure 1 presents a high level overview of the various elements involved in attack graph generation and the way they interact with each other. The right part shows the vulnerabilities contained in the target network could be exploited by some *attack patterns*. The attack patterns with variables are instantiated to be the corresponding *atomic attacks*. The left part shows all the atomic attacks used by the attacker compose a full attack graph. In the remaining of this section, we present in more detail some of the components in the figure1.

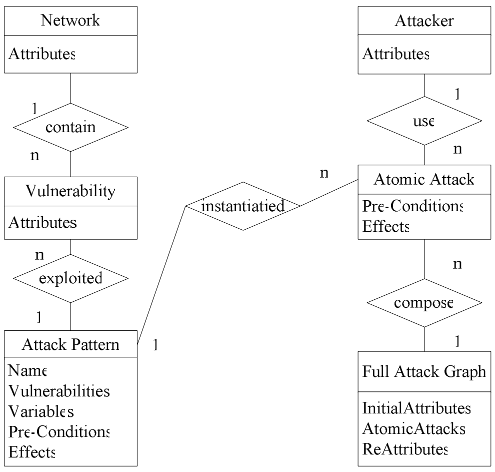


Fig. 1. Conceptual model

3.1 Attack Patterns

Building a database describing the pre and post-conditions of each vulnerability exploited by attackers is crucial to generate attack graphs and many post papers assume the database have been founded. While this assumption is valid for a theoretical graph design, it is unsuitable for implementation for analyzing each of the known vulnerability is labor intensive (e.g. the current CVE has more than ten thousand of vulnerabilities [17]) and difficult (the work requires the expert's knowledge about attacks). In this paper, we introduce an *attack pattern* that is an abstract description of the common approach attackers take to exploit analogous vulnerabilities in a formalized and constructive way. Once the set of *attack patterns* is established, the each vulnerability from various sources (such as CERT, BugTraq CVE etc.) could be mapped to the corresponding attack patterns that are developed by reasoning over large sets of software exploits and attacks. The advantages of introducing *attack patterns* to the model are not only presenting the standard level of abstract description for exploiting the vulnerabilities but also indirectly presenting the model of the target network and the attacker (the reasons are in section 3.2).

The core components of *attack patterns* description are:

– A set of *Pre-Conditions* to be satisfied by the attributes of the target network and the attacker for this attack to succeed.
– The *Effects* of one successful attack is the consequences of its performance in the network (Such effects can be associated with the occurrence of a damage in the system e.g. service disruption) or a gain for the attacker, for instance the acquisition of knowledge concerning the target network.
– A set of *vulnerabilities* from various sources (such as CERT, BugTraq CVE etc.) may be exploited by the attack pattern.
– A set of *variables* includes all the variable appearing in the describing the *Pre-Conditions* and *Effects* of the attack pattern.

To equip software development personnel of all levels to build more secure software, CAPEC [18] are in way to build the knowledge database of attack patterns. The attack patterns in CAPEC focus on illustrating snapshots of attack steps, i.e. "how the attack can happen", whereas our attack patterns focus on causes and consequences of the attacks, i.e. "why and what consequence the attack can happen". They characterize the exploitability of known vulnerabilities utilizing pre-conditions that must exist for successful exploit and post-conditions that result from successful exploit. Though our attack patterns are different from them, we have built our knowledge database of attack patterns based on their work. In CAVS, twelve attack patterns have been developed.

3.2 Modeling Networks and Attackers

An attacker's successful performance of one atomic attack lies on the current attributes of the target network and the attacker satisfying its pre-conditions. At the same time, it will change their attributes according to its effects. However, it is difficult to precisely and succinctly model the target network and attacker. In previous work, though they are prone to model their attributes as attacker access privileges, and network connectivity etc., these simple modeling methods can not determine whether complex atomic attack's pre-conditions can be satisfied by the current state. Thus, more detailed and specific their attributes need to be known.

We adopt *attack pattern-oriented* method to model the network and attacker. The set of predicates leveraged to describe the pre and post-conditions of attack patterns imply that the level of detail about the attributes of the network and attackers is necessary and sufficient to generate attack graphs. As a consequence, these predicates compose the model of the network and attacker. When some new predicates are added to the knowledge database for describing the new attack patterns, the model for the network and attacker will vary along with them.

3.3 Full Attack Graphs Semantic

The state-based attack graphs use nodes for global states, which show all the attack paths explicitly[1-6,11]. The explicit attack graphs face a serious scalability issue because the number of such sequences is exponential in the number of vulnerabilities multiplied by the number of hosts. To avoid such a combinatorial explosion, *full attack graphs* are representation of all the attack paths implicitly. In the remaining of this section, we formally define it.

Definition 1. Let AP be a set of first predicates for describing the attributes of the target network and attacker, and ($InitialAttributes \cup ReAttributes$)$\subset$AP, where *InitialAttibutes* is the set of the initial attributes of the target network and attacker and *ReAttributes* is the set of their reachable attributes. A *full attack graph* AG=($A_0 \cup A_r$,T,E,L), where A_0 is a set of initial attribute nodes, A_r is a set of the reachable attribute nodes, T is a set of atomic attack nodes. L= ($A_0 \rightarrow InitialAttributes$) $\cup$($A_r \rightarrow ReAttributes$)$\cup$($T \rightarrow AtomicAttacks$) is a mapping function from a node to its attribute or atomic attack, $E \subset ((T\times A_r) \cup (A_r \times T) \cup (A_0 \times T)$ is a set of edges between nodes (attribute nodes or atomic attack nodes).

Property 1. For every atomic attack node τ, let Pre(τ) be the set of τ 's pre-conditions and Post(τ) be the set of τ 's post-conditions, then $(\wedge L(a_i)) \rightarrow (\wedge L(a_k))$, where $a_i \in \text{Pre}(\tau)$, and $a_k \in \text{Post}(\tau)$, that shows when all the pre-conditions of atomic attack τ are true, every post-condition of atomic attack τ is true.

Definition 2. Let $\tau_1\tau_2...\tau_l$ is a finite sequence of atomic attacks in attack graph, where $\tau_i \in T$ for all $0 \le i \le l$, if $\forall a \in \text{Pre}(\tau_i)$, $a \in \cup_{k=1}^{i-1} \text{Post}(\tau_k) \cup A_0$, which shows the front atomic attacks prepares conditions for the latter atomic attack, then we define an *attack path* as the finite sequence of atomic attacks such that $\exists a' \in \text{Post}(\tau_l)$, $a' \in A_r$, which shows one of the last atomic attack's post-conditions is one of the reachable attributes.

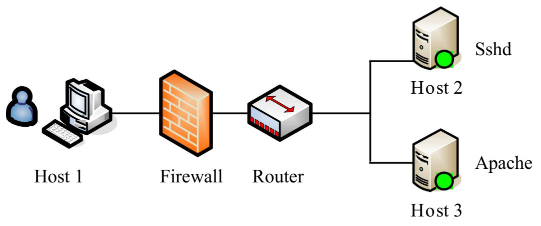


Fig. 2. SAMPLE network diagram

Table 1. Predicates in the attack graph

Predicate	**Description**
access(*s*,*d*)	The source host *s* accesses the destination host *d*
runProcess(*d*,*p*)	The destination host *d* is running the process *p*
hasVuls(*s*,*p*,*id*)	The process *p* of the host s has the vulnerability with identification *id*
user(*s*)	The attacker has the privilege user of the host s

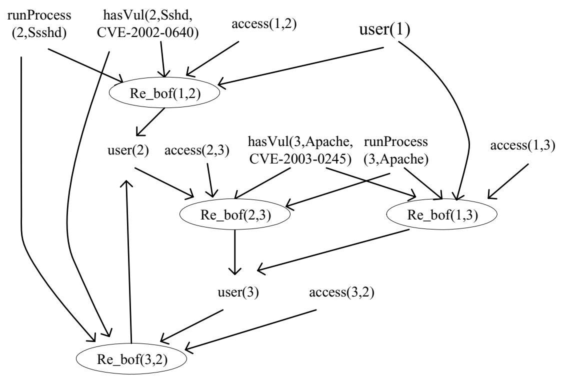


Fig. 3. Full attack graphs for the SAMPLE network

Figure 2 shows an SAMPLE network where the attacker's machine is denoted machine 1, and the two victim machines are denoted 2 and 3, respectively. The vulnerable process Sshd with CVE-2002-0640 is running in the host 2, and Apache with CVE-2003-0245 is running in the host 3. Both of the two vulnerable processes may be exploited by the remote buffer overflow attack pattern Re_bof that may be instantiated to yield atomic attacks. Re_bof(*a*,*b*) denotes the execution of the remote buffer overflow attack from machine *a* to machine *b* to have the privilege user of machine *b*. Figure 3 shows the attack graph for the network. In the figure, atomic attacks appear as ovals, and attributes appear as plain text. Table 1 shows the description of predicates in the attack graph.

4 VAML Modeling Language

In order to formally describe the model, we present a novel description language VAML(Vulnerabilities Analysis Modeling Description Language). We define the initial scenario as the initial attributes of the network and the attacker. This language has two parts where one is for attack patterns description and the other for the initial scenario description. In the remaining of this section, we discuss it in more detail.

4.1 Attack Patterns Description

The syntax of attack patterns description is a tuple <*Predicate*,*Type*,*AttackPattern*>, and the detail is discussed as the following.

- *Type* is a set of the types for each variable appearing the attack patterns. "Host", "Process" and "VulID" are the pre-defined types of the language.
- *Predicate* is a set of the predicates used to describe the pre and post-conditions of attack pattern. For the reasons mentioned in 3.2, we also consider it as the model of the network and attacker. "hasVul(*s*:Host, *p*:Process,*id*:VulID)" is the pre-defined predicate of the language and its implication is that the process *p* of the host *s* has the vulnerability with identification *id* from CERT, BugTraq CVE etc.
- *AttackPattern* is a set of attack patterns.
- Each attack pattern is a tuple <*Name*,*Vuls*,*Var*,*Pre*, *Eff*>, and the detail is presented as the following.
- *Name* is the name of the attack pattern.
- *Vuls* is a set of identifications of vulnerabilities referring to various sources (such as CERT, BugTraq CVE etc.) and these vulnerabilities could be exploited by this attack pattern.
- *Var* is a set of local variables and their scope is limited to the attack pattern description where they appear. Each variable is specified the corresponding type.
- *Pre* and *Eff* are a set of predicates describing the pre or post conditions of the attack pattern respectively and each predicate of them is in the set *Predicate*. The relation among predicates in *Pre* or *Eff* is AND. The pattern with the relation OR among pre or post-conditions may be divided to several different patterns with the single relation AND.

Re_bof
***Vuls**:*
CVE-2003-0245,CVE-2003-0466,CVE-2002-0640
***Var**:*
(*s*:Host),(*d*:Host),(*p*:Process),(*id*:VulID)
***Pre**:*
runProcess(*d*,*p*),hasVul(*d*,*p*,*id*),access(*s*,*d*,*p*), user(*s*),
***Eff**:*
user(*d*)

Fig. 4. The example of remote buffer overflow description

Figure 4 shows an example of remote buffer overflow description.

"Re_bof" is the name of the attack pattern, and the pattern has the capability of exploiting CVE-2003-0245, CVE-2003-0466 etc.(many other vulnerabilities are omitted). It has four local variables, where *s* is the source host that launches the attack, *d* is the destination host that is compromised by the attack, *p* is the vulnerable process running in the destination host. *id* specifies the identification of vulnerability and its domain is the set *Vuls*. The pre-conditions of the pattern are vulnerable process *p* running in the destination host *d* with the privilege of user, and the process *p* having the vulnerability *id*, the source host *s* having access to the destination host *d*, and the attacker having the privilege user of the source host *s*. The post-condition is the attacker having the privilege user of the destination host *d*.

The crucial and reasonable *monotonicity* assumption i.e., an attacker never relinquishes any obtained capa-bilities is introduced in [5] to effectively simplify the model. The VAML modeling language also supports the assumption owing to no negation quantifier appearing the pre and post-conditions of each attack pattern. In the CAVS, the language has the capability of describing all the attack patterns we have been discovered, and we have not found the practical attack patterns that can not be described.

4.2 Initial Scenario Description

The syntax of initial scenario descriptions is a tuple <*Object*,*InitialAttributes*>, and the detail of the each component is the following.

- *Object* is the set of constants describing the elements of the target network such as hosts and processes etc. They are used to instantiate the attack patterns and produce the corresponding atomic attacks.
- *InitalAttributes* is the set of facts describing the initial attributes of the target network and the attacker. The predicate of each attribute is in set of *Predicate***.**

For the target network and the given attacker, we can compactly describe their initial scenario. Figure 5 shows the initial scenario description for the SAMPLE network. *Object* specifies this network owning three hosts denoted 1 and 2 and 3 respectively, and two processes with the name Apache and Sshd respectively. The *InitialAttributes* shows the information about the initial attributes of network such as the running processes of each host and the vulnerability of each process and the connectivity among them. user(1) indicates that the attacker has the privilege user of the host 1 initially.

Object:
1,2,3:Host
Apache,Sshd:Process
InitialAttributes:
runProcess(2,Sshd),runProcess(3, Apache),
hasVul(2,Sshd,CVE-2002-0640),
hasVul(3, Apache, CVE-2003-0245),
access(1, 2,Sshd), access(2, 3, Apache), user(1),
access(1, 3,Apache),access(3, 2,Sshd)

Fig. 5. The initial scenario description for the SAMPLE Network

5 Full Attack Graphs Builder

The full attack graph builder shown in figure 6 is the core function component in the CAVS. Its input information includes the attack patterns knowledge database and the initial scenario corresponding to the initial attributes of the network and the attacker. Since the set of predicates leveraged to describe the pre and post-conditions of attack patterns specifies the attributes of the network and attacker necessarily known for generating attack graphs, we have developed or made use of some automatic tools to gain these initial attributes corresponding to the predicates. For instance, the predicate "hasVul" indicates the information about the vulnerabilities in the target network, thus we use the vulnerability scanner tools such as Nessus to acquire it. Once all the information about initial scenario has been known, the builder may produce full attack graphs.

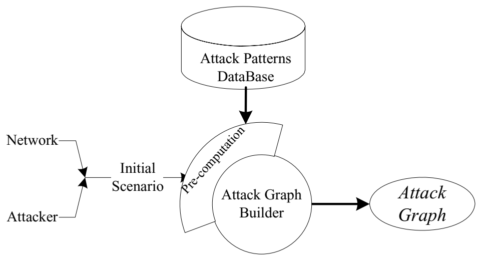


Fig. 6. The Full Attack Graphs Builder

In the set of *InitialAttributes*, the attributes corresponding to the pre-defined predicate hasVul indicate all the vulnerabilities in the target network. We use them to determine all the *possible* attack patterns. For example, in the SAMPLE network, we know "Re_bof" is one of *possible* attack patterns because the constant "CVE-2002-0640" in hasVul(2,Sshd,CVE-2002-0640) is contained in its set *Vuls*. We denote *P-AttackPatterns* as the set of *possible* attack patterns and it may be acquired in the pre-computation. Because there is no negation quantifier appearing the pre and post-conditions of each attack pattern, we need not pay attention to the mutual exclusion of pre-conditions and what we want to do is exploration of predicate space, i.e. we try to enumerate all instantiations of predicates that are reachable by beginning with the set of initial attributes.

5.1 Algorithms to Generating Full Attack Graphs

To compute the set of atomic attacks *AtomicAttacks* in attack graphs, we present the scalable algorithm α-Exploring shown in Figure 7 in detail. We make uses of a queue *AttibuteQueue* in which all attributes that are scheduled to be inserted into the set *ReAttributes*are stored. Initially, this queue consists of the attributes in *InitialAttributes*, while *ReAttributes*is empty. We then repeatedly remove the first attribute α from the queue, add it to the *ReAttributes* and instantiate all *possible* attack patterns whose preconditions are a subset of *ReAttributes* and include α . Add effects of the atomic attacks that are not yet stored in either the queue or *ReAttributes* to the back of the queue *AttibuteQueue*. This process is iterated until the queue is empty.

```
Procedure  α-Exploring
Input: P-AttackPatterns, InitialAttributes
Output: ReAttributes , AtomicAttacks
(1)  AttributeQueue is initialized with the attributes
     in InitAttributes
(2)  ReAttributes is an empty set initially.
(3)  While (AttributeQueue is not empty){
(4)    dequeue the attribute a from AttributeQueue;
(5)    insert a into ReAttributes ;
(6)    instantiate all the attack patterns in P-AttackPatterns
       whose pre-conditions include a;
(7)    choose atomic attacks whose pre-conditions are a
       subset of ReAttributes ;
(8)    add every effects of atomic attacks not appearing in
       ReAttributes and AttributeQueue to AttributeQueue
(9)    add the atomic attacks to AtomicAttacks
(10) }
```

Fig. 7. The Scalable Algorithm α- Exploring

The key advantage of the algorithm is that when instantiating attack patterns, only those instantiations need to be checked for which α is one of the preconditions, which means that we can bind all variables appearing in that preconditions to the corresponding value of α , thus dynamically reducing the number of degrees of freedom of the local variables. However, the variables having no the corresponding values of α need be instantiated each value in their domains. Since the number of these values might be great, which might take far long time to check each of them for the large target network size, we propose two optimization techniques to overcome this problem (they are discussed in section 5.2). Once *AtomicAttacks* are obtained, we could use some mature tools to easily draw the full attack graphs. In CAVS, we adopt the graphviz tool [16].

5.2 Two Optimization Techniques

The first is the static constraining domains technique. We define *one-way predicates* as the predicates appearing in the preconditions of one possible attack pattern, but never appearing in the post-conditions of all the possible attack patterns. We may pre-compute the initial attributes corresponding to *one-way predicates* to constrain the domain of the local variables of each attack pattern. For example, since the predicate "hasVul" is *one-way predicate*, we can constrain domains of the destination host variable in "Re_bof" to the determined value. In practice, we find the most predicates of the specific model are *one-way predicates*, For example, in the SAMPLE network, "hasVul", "runProcess", and "access" are the *one-way predicates*. Thus the optimization technique improves performance observably.

The second important optimization is the looking backward technique. We instantiates the precondition one at a time and immediately checking if it is an unsatisfied precondition, in which case we do not try to instantiate the next precondition.

6 Performance Analysis and Experiment Result

The complexity of algorithm α-Exploring lies in the number of possible attack patterns and the size of their local variables' domains. To analyze its scalability about the network size, we just focus on the variable of the type "Host" while the other variables are not affected by the size of the network.

The network attack happens between two hosts and the attributes include host connectivity information, thus the number of attributes stored in *AttributeQueue* is quadratic in the number of hosts. For each attribute, the number of different instantiations is lower than the number of hosts multiplied by the number of possible attack patterns, because the variable of the destination host could be specified by the attribute value corresponding to the predicate "hasVul" in the pre-computation. For our implementation, we used "HashMap" in vc++8.0's standard library, which has a constant-time lookup table to check if one precondition are in *ReAttribures.* Therefore, the upper bound for the algorithm is $O(MN^3)$, where N is the number of hosts, M is the number of possible attack patterns.

Our Full Attack Graphs Builder was tested in the following environment. The CPU was Intel Core Duo T7500 2.2GHz with 2GB of RAM, the operating system was Microsoft Windows XP Professional Version 2002 Service Pack 2.

We simulated the network with two topologies for a variety of network sizes, topologies and vulnerability densities. The "star" topology was simulated to consist of one centralized host (not the target host) that has two-way accessibility to every other machine. The non-centralized hosts, among which is the attacking host, have no direct network access to any other host. The "ring" topology was simulated with one host of the ring connected to the Internet, and all the other hosts (one is the attacking host) on the ring connected only to its two immediate neighbors with two-way access. All the vulnerabilities in the simulated network may be exploited by remote buffer overflow.

Figure 8 shows the graph generation CPU time for each of the simulated analysis problems of various sizes and topologies on the assumption that each host runs one vulnerable process. It illustrates the two topologies has the approximate CPU time and the CPU time does not exceed 90sec for the network size of 1000 hosts.

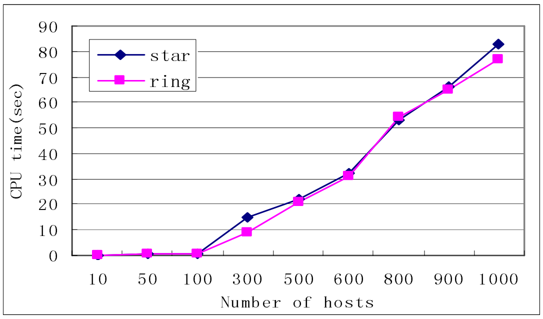


Fig. 8. Graph generation CPU usage as a function of network size for two network topologies

Figure 9 shows the two optimization techniques' impact on the graph generation CPU time of the simulated network with the "star" topology. It illustrates the two optimization techniques improves the performance observably.

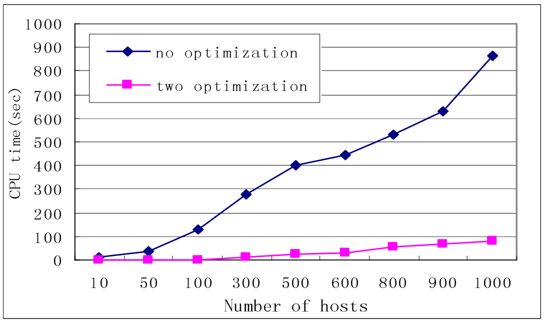


Fig. 9. The performance of improvement of two optimization techniques. The network with the "star" topology and one vulnerability per host.

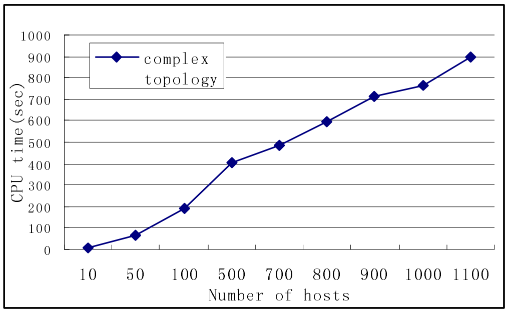


Fig. 10. Graph generation CPU usage as a function of network size for complex network topologies

To test the method's scalability to large enterprise network with complex topology, we design two lists. One is software list that consists of eleven kinds of software with vulnerabilities(four of them may be exploited by remote buffer overflow attack pattern, seven of them may be exploited by local buffer overflow attack pattern), the other is firewall rules list composed of twelve rules. We randomly choose three kinds of vulnerable software and two firewall rules from the two lists, and then assign them to each host in the simulated network. Figure 10 shows the graph generation CPU time for the simulated analysis problems of various size networks with complex topology. It illustrates the approach could potentially be applied to the operational large, enterprise-size networks with thousands of hosts.

7 Conclusion

In this paper, we present a modeling language VAML for attack graphs generation, which allows the formal definition of the different components of an attacker and a network description in a declarative manner. At the same time, we put forward a scalable approach to build *full attack graphs* which is capable of describing all the potential interrelations among all the known vulnerabilities in the targeted network. The analysis of the method performance illustrates it could be applied to the enterprise network with thousands of hosts. The prototype CAVS has integrated some mature vulnerability scanner tools (such as Nessus etc.) and network topology scanner tools (such as X-scan etc.) and the various vulnerabilities sources (such as CERT, BugTraq CVE etc.). In the future work, we will apply CAVS to the large operational enterprise network.

Acknowledgments. It is supported by the National Natural Science Foundation of china under Grant of No. 90604006.

References

1. Phillips, C., Swiler, L.: A graph-based system for network vulnerability analysis. In: ACM New Security Paradigms Workshop, pp. 71–79 (1998)
2. Ritchey, R., Ammann, P.: Using model checking to analyze network vulnerabilities. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, pp. 156–165 (2000)
3. Sheyner, O., Jha, S., Wing, J.M., Lippmann, R.P., Haines, J.: Automated Generation and Analysis of Attack Graphs. In: 2002 IEEE Symposium on Security and Privacy, Oakland, California (2002)
4. Lippmann, R.P., Ingols, K.W.: An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory, Lexington, MA, ESC-TR-2005-054 (2005)
5. Ammann, P., Wijesekera, D., Kaushik, S.: Scalable, graph-based network vulnerability analysis. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 217–224. ACM Press, New York (2002)
6. Jajodia, S., Noel, S., O'Berry, B.: Topological Analysis of Network Attack Vulnerability, vol. 5. Kluwer Academic Publishers, Dordrecht (2003)
7. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: A logic-based network security analyzer. In: 14th USENIX Security Symposium, Baltimore, MD,USA (August. 2005)
8. Ou, X., Boyer, W.F., McQueen, M.A.: A Scalable Approach to Attack Graph Generation. In: Proceedings of the 13th ACM conference on Computer and communications security, pp. 336–345 (2006)
9. Templeton, S., Levit, K.: A Requires/Provides Model for Computer Attacks. In: Proc. of New Security Paradigms Workshop, pp. 31–38 (2000-2009)

10. Cuppens, F., Ortalo, R.: LAMBDA: A language to model a database for detection of attacks. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 197–216. Springer, Heidelberg (2000)
11. Jha, S., Sheyner, O., Wing, J.: Two Formal Analyses of Attack Graphs. In: Proceedings: 15th IEEE Computer Security Foundations Workshop (CSFW 15), pp. 49–63. IEEE Computer Society Press, Los Alamitos (2002)
12. Wang, L., Noel, S., Jajodia, S.: Minimum-cost network hardening using attack graphs. Computer Communications 29(18), 3812–3824 (2006)
13. Wang, L., Liu, A., Jajodia, S.: An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 247–266. Springer, Heidelberg (2005)
14. Noel, S., Jajodia, S.: Correlating intrusion events and building attack scenarios through attack graph distance. In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004) (2004)
15. Wang, L., Singhal, A., Jajodia, S.: Toward measuring network security using attack graphs. In: Conference on Computer and Communications Security Proceedings of the 2007 ACM workshop on Quality of protection, pp. 49–54 (2007)
16. Graphviz - Graph Visualization Software, `http://www.graphviz.org/`
17. Common Vulnerabilities and Exposure, `http://cve.mitre.org/`
18. Common Attack Pattern Enumeration and Classification, `http://capec.mitre.org/`

MEDS: The Memory Error Detection System

Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, Virginia U.S.A.
{hiser,clc5q,mc2zk,jwd}@cs.virginia.edu

Abstract. Memory errors continue to be a major source of software failure. To address this issue, we present MEDS (Memory Error Detection System), a system for detecting memory errors within binary executables. The system can detect buffer overflow, uninitialized data reads, double-free, and deallocated memory access errors and vulnerabilities. It works by using static analysis to prove memory accesses safe. If a memory access cannot be proven safe, MEDS falls back to run-time analysis. The system exceeds previous work with dramatic reductions in false positives, as well as covering all memory segments (stack, static, heap).

1 Introduction

Modern computer software is absolutely essential to today's information and communications infrastructure. Software provides high performance, upgradeable, patchable, and diverse functionality in nearly every computer operated device from desktop and laptop computers, to cell phones and PDAs, and even automotive, marine, and aeronautical environments.

Considering the importance of computer software, it is no surprise that we demand software be reliable. Software must be reliable in two ways: it must be free from bugs which crash the program, and it must be free from vulnerabilities which might let a malicious user attack the system. Current software development practices yield a variety of memory errors: buffer overflow, array out-of-bound, double-free, and uninitialized pointer dereference errors. These errors may be relatively benign and only cause a program to crash. However, in a critical application, there may be significant dangers, such as important data being lost, aeroplanes being off course, or breach of security if the bug is exploited by a malicious attacker.

While there are a variety of memory-safe languages, such as Java and C#, that are available to address this problem, developers are reluctant or unable to use such languages for a variety of reasons. First, there is significant cost associated with maintaining memory safety, and software developers may decide that the system performance goals cannot be met with a memory safe language. Second, many applications existed before the wide availability of such languages, and the cost of moving from a non-safe language to a safe language is prohibitive. Finally, language support might be lacking on the platform in which the system is to be deployed. For example, Java run-time libraries might not exist for an embedded GPS device. Consequently, memory safe languages are not always a viable option for software.

An even worse situation arises when the software's source code is not available. Such a situation could occur because some or all of the software comes from an untrusted third party. For example, a third party library might be used for font rendering, or the entire project might be developed by an untrusted source and used in critical infrastructure. Another example is black-box testing, where the tester must find memory errors with no access to the source code. Thus, a tool to detect memory errors requiring only the binary executable file would be of great use for testing and security purposes.

To meet this need, we have developed MEDS: The Memory-Error Detection System. MEDS operates by using a combination of static and dynamic analysis to achieve a variety of advances to the state-of-the-art in binary-only memory error detection. MEDS provides:

1. Comprehensive protection of all memory segments, including global-static data, heap-allocated data structures, and stack allocations without requiring debug information. Previous techniques required debug information and failed to protect the stack due to excessive false positive rates.
2. Significant reduction of false positive rates. Previous techniques had several categories of false positives which would render those techniques useless for on-line protection, and much less desirable for offline use.
3. Aggressive static and dynamic analysis provide higher levels of protection with run-time performance comparable to previous, less effective techniques.
4. No reliance on source code, object code, or debugging information. Many memory error detectors require source code or source code changes, or object code so that custom allocation and tracking libraries can be linked.
5. An optional profiling step to help separate false positives from actual errors and provide further performance benefits. Previous techniques have no mechanisms for helping a user differentiate real errors from falsely reported errors.

Together, these advances demonstrate that MEDS provides significant improvement to state-of-the-art memory overwriting defences.

The remainder of the paper is organized as follows: Section 2 describes the MEDS system in detail, while Section 3 gives experimental evidence of MEDS' effectiveness. Section 4 discusses related work, and finally Section 5 summarizes our findings.

2 MEDS

2.1 System Overview

MEDS, as shown in Fig. 1, takes a binary program as input. The binary is used for static analysis and to prepare the mmStrata run-time system. MEDS first prepares a run-time system using a binary instrumentation tool we call the Stratafier, so named because it inserts a software dynamic translation system called Strata into an executable image.

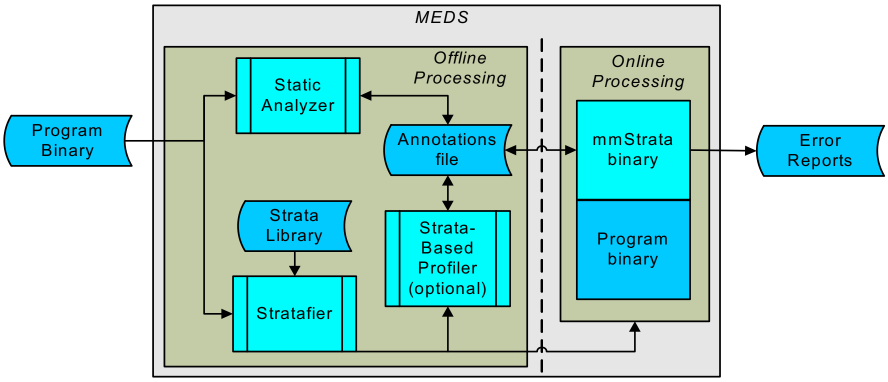


Fig. 1. High-level MEDS overview

After the binary has been "stratafied," MEDS runs a static analysis step to create an annotations file. The annotations file contains information obtained during all types of analysis to facilitate further analysis and performance improvements. Furthermore, the annotation file is the means for communication among all MEDS components.

Finally, the run-time system is ready to detect memory errors. When run, the revised program binary is dynamically instrumented by mmStrata. Memory writes that cannot be statically proven safe are checked for safety. Any violations detected result in further annotations (with diagnostic information) which are later reported to the user.

Sections 2.2-2.6 discuss these components in more detail.

2.2 MEDS Type System

To effectively detect memory errors, MEDS stores metadata for every program storage location: each hardware register and memory location is assigned its own metadata. The base system has two types of metadata:

- `n` – a numeric type (i.e. non-pointer) object is held in the storage location.
- $\texttt{p}_{\texttt{obj}}$ – a pointer is stored in the corresponding storage location with referent `obj`. Note that this metadata carries with it the bounds of `obj`, so that dereferencing of this object can be bounds checked. Furthermore, two objects of the same type, with different bounds, receive distinct metadata.

This data is initialized at program start-up, and updated for each program operation so that the metadata is consistent with the value held in each storage location. For example, consider a `mov eax,[0x8100800]` instruction. The metadata associated with storage location `0x8100800` is loaded, and stored into the metadata for register `eax`. For instructions that involve computation, a metadata computation is performed as well. Fig. 2 shows how metadata types are combined to compute a new metadata type. As the figure shows, most operations simply return that the result is numeric. `Add` and `subtract` operations are valid on pointers, and can result in a pointer type. A few operations, namely bitwise `and` and `or` operations, can result in either a pointer or a numeric type when a pointer type is used for input. For these operations, we use a simple heuristic that examines the result value. If the result stays within the referent object, then the result is a pointer, otherwise the result is a numeric.

+	p	n
p	n	p
n	p	n

-	p	n
p	n	p
n	n	n

&,|	p	n
p	p/n	p/n
n	p/n	n

*, /, %, ^, ~, <<, >>	p	n
p	n	n
n	n	n

Fig. 2. Rules for combining metadata types

For efficiency purposes, memory is divided into pointer-sized (4-byte) blocks and one metadata entry is kept for each block. In our implementation, the metadata is a pointer to a bounds-information object. The object contains the metadata type, as well as information about the referent if the type is a pointer. Furthermore, a reference count is maintained so we know when it is safe to deallocate the bounds-information in a garbage collected manner. In the dynamic systems, metadata information is kept in a small array for registers, and a splay tree (for fast common case look-ups) for memory. Bounds-information objects are allocated whenever the system detects the allocation of an object by the program, and are lazily deallocated after the corresponding program object is deallocated and the reference count falls to zero.

2.2.1 Challenges

Although the MEDS type system sounds easy to implement, there are a variety of challenges. These challenges are maintaining bounds information, type identification, code identification, and handling non-standard code. This section describes each challenge and gives a high-level view of how it was overcome. Complete details about the solutions are in following sections, as noted.

The first challenge the MEDS system faces is that every object created in the system needs to have a bounds-information object associated with it. For heap objects, this is as simple as having the dynamic system watch calls to allocation and deallocation routines. For static-global and stack objects, allocating the bounds-information object properly is not as easy. The information for static-global variables comes from the static analyzer (see Section 2.4). Stack variables are by definition at variable addresses, and bounds-information objects cannot be allocated at program start-up. Again, the static analyzer helps by analyzing stack frames (again, details in Section 2.4) which the dynamic system uses to create bounds-information objects dynamically. For functions that are unanalyzeable (because they contain dynamic stack allocation via `alloca`, or some other non-standard stack manipulation), the dynamic run-time system creates and updates bounds-information objects for the stack frames (see Section 2.6).

Besides tracking object creation, MEDS also needs to know whether a created object is a numeric type or a pointer to another object to set the metadata for the newly created object. For many objects this is easy, e.g. objects returned from `malloc` never contain a pointer initially. But consider the instruction `mov eax,$0x8108004`. Should `eax` be considered a pointer after this instruction? If so, to what object does it point? In the instruction `lea eax, [esp + 36]`, to which stack frame should `eax` point? If statically allocated memory were to contain the value 0x8108004, is this value a pointer? Compiler optimizations can produce code in which a pointer is initialized to point outside its referent data object, because accesses through the pointer will always contain an offset to bring the address within the referent object. These are all cases of a general *pointer identification problem*. The static analyzer and the profiler combine forces to resolve questions about any value which might be a pointer (Sections 2.4-2.5).

Another major challenge faced by MEDS is how to locate the executable code within a program. The static analyzer solves this problem (see Section 2.4).

Compiler optimizations and non-standard code can cause the type system to erroneously consider some objects to be numeric. Consider the code in Fig. 3 as an example of code commonly created by a combination of strength reduction and induction variable elimination [1]. In the loop preheader, register ecx gets assigned the offset from object a to object b, eliminating complex address arithmetic and multiple induction variable updates. However, the basic type system assigns ecx as type numeric. Thus, both the load and store instructions in the loop are seen as references to variable a, when clearly one is a reference to variable b. To solve this problem, we extend the basic type system with an offset type, $o_{ptr1,ptr2.}$ The offset type is created when the difference between two pointers is taken. In most operations, the offset type behaves as numeric, except in the case of adding a pointer and an offset; when $p_{ptr1}+o_{ptr1,ptr2}$ is calculated, the result is $p_{ptr2.}$ Care must be taken when deallocating objects to ensure that all offset types that reference the object are marked as invalid.

```
for(i=0;i<N;i++)          eax=&a             // eax is ptr to a
   a[i]=b[i];             ecx=&b             // ecx is ptr to b
                          ecx=ecx-eax        // what type is ecx?
                       L1:mov ebx,[eax+ecx]  // ptr to a?
                          mov [eax],ebx
                          add eax,4
                          ...
```

Fig. 3. Strength-reduction example

A problematic non-standard coding style is to use *block* operations to work on an object as an aggregate, e.g. using memcpy to copy a list node from one location in memory to a second location. Since pointers are almost always on a word boundary, and most block operations occur on word or double-word boundaries, the common case is handled without problem. However, if memcpy or a similar user-written routine uses byte-by-byte operations, then the byte load and byte store operations seem inherently to have a numeric type. Consequently, the copy can cause the type system to lose information about pointers in the destination of the byte-by-byte copy. MEDS solves this problem by considering the most significant byte of a pointer to be a pointer regardless of how or where it is stored. Thus, sign extension and truncation of the "pointer," as one byte of the pointer is copied, results in no issues. Likewise, we only set the metadata for a 4-byte memory storage location if the most significant byte is written.

Before moving to an in-depth description of each tool, we first briefly examine Software Dynamic Translation, a mechanism used to dynamically instrument binaries.

2.3 Software Dynamic Translation

Strata is a software dynamic translation (SDT) system designed for high retargetability and low overhead translation. Strata has been used for a variety of applications including system call monitoring, dynamic download of code from a server, and enforcing security policies [2, 3]. This section describes some of the basic features of Strata which are important to understanding the experiments presented later. For an in-depth discussion of Strata, please refer to previous publications [4, 5, 6].

2.3.1 Strata Overview

Strata operates as a co-routine with the program binary it is translating, as shown in Fig. 4. As the figure shows, each time Strata encounters a new instruction address (i.e., PC), it first checks to see if the address has been translated into the *fragment cache*. The fragment cache is a software instruction cache that stores portions of code that have been translated from the native binary. The fragment cache is made up of *fragments*, which are the basic unit of translation. If Strata finds that a requested PC has not been previously translated, Strata allocates a fragment and begins translation. Once a termination condition is met, Strata emits any *trampolines* that are necessary. Trampolines are pieces of code emitted into the fragment cache to transfer control back to Strata. Most control transfer instructions (CTIs) are initially linked to trampolines (unless the transfer target already exists in the fragment cache). Once a CTI's target instruction becomes available in the fragment cache, the CTI is linked directly to the destination, avoiding future uses of the trampoline. This mechanism is called *fragment linking* and avoids significant overhead associated with returning to Strata after every fragment [4].

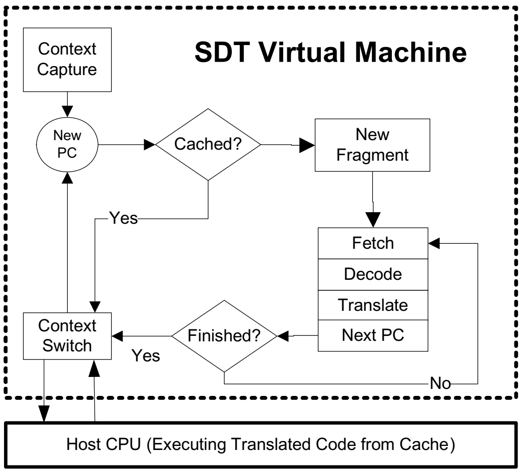


Fig. 4. High-level overview of Strata operation

Strata's translation process can be overridden to implement a new SDT use. In this paper, we modify Strata's default translation process to insert instrumentation to enforce the MEDS type system in both the profiler driven analysis (Section 2.5) and the online detector (Section 2.6).

2.4 Static Analysis

The MEDS Static Analyzer is implemented as a plug-in to the popular IDA Pro disassembler [7]. After IDA Pro completes its disassembly of the program binary, the static analyzer plug-in analyzes the program and produces *informative annotations*.

A preliminary step is to assist IDA Pro in the disassembly. Disassembly of a program binary involves solving the problem of precisely identifying code and data within the binary. This problem is not perfectly solvable at present. The two basic design approaches for disassemblers, *recursive descent* and *linear scan*, have different strengths and weaknesses when analyzing different program binaries. The static analyzer improves upon the recursive descent approach of IDA Pro by using a linear scan disassembler (the GNU Linux tool `objdump`) to get a second opinion on which addresses are code and which are data. If code identified by `objdump` but not identified by IDA Pro can now be successfully analyzed by IDA Pro, as requested by the static analyzer, the code is incorporated into the IDA Pro code database. This augmentation of IDA Pro's abilities improves the analysis coverage of MEDS and prevents false positives that would arise if code sections escaped static analysis and received no annotations.

Informative annotations are provided to identify all functions and static-global data objects by three attributes: name, starting address, and total size. If the executable has been stripped, the names will be dummy names generated by IDA Pro. The static-global data annotations will be used at run time to create bounds-information objects for static-global memory objects, as described in Section 2.2.1. The function annotations are used to identify the location of key library functions that allocate memory (such as `malloc`), so that heap memory referents can be tracked at run time.

The most important informative annotations describe the run time stack. Functions that allocate a local stack frame, or activation record, with space for local variables must have their stack memory objects tracked at run time. The instruction that allocates space for the stack frame (usually by subtracting the stack frame size from the stack pointer) is identified by a series of annotations that enable the run time dynamic analysis to divide the stack region into local variables, saved registers, and a saved return address. The run time dynamic analysis will then be able to check the bounds of stack references so that a memory access cannot overflow from one stack object to another (e.g. from a local variable to the return address). Precise identification of all sub-regions of the run-time stack permits monitoring of stack accesses without incurring false positives or false negatives. This synergy of static and dynamic analysis is a significant advance over comparable prior work in this area, which relied entirely on dynamic analysis and suffered from numerous stack-related false positives, as described in Section 4.

2.5 MEDS Profiler

The optional MEDS profiler is based on Strata, and uses information from the static analysis phase to mimic the MEDS type system. One of the main goals is to reduce false positives. The MEDS profiler does this by answering the question “is this object a pointer, and if so, to what does it point?”

To answer this question, any time an object might be a pointer (i.e. is pointer-sized and has a value that might legitimately make it a pointer), it is assigned a metadata type of `q`. The `q`-type tells the profiler that the decision on whether the object is a pointer has not yet been made, and carries along information about the creation point of the object. When the result of the operation depends on whether the `q`-type should have been a pointer, the profiler lazily evaluates it to either a `p`-type or an `n`-type by determining if there’s a corresponding referent for the object’s current value. Ultimately, if a `q`-type is dereferenced in the program, the profiler records which object was dereferenced. If, at the end of execution, the `q`-type always referenced the same object, then the profiler records that the `q`-type should be created as a pointer type during final execution. Otherwise, the profiler fails to prove that the object was a pointer, and safely reverts to the assumption that the created object is a numeric type. In either case, an informative annotation is written into the annotations file to inform the other parts of the system.

If no profiler information is available (for example, if the profiler was not run), MEDS falls back to a simple heuristic that works quite well. The heuristic assumes that a static constant is a pointer if and only if it is within the bounds of some data object. Section 3 gives evidence that this simple heuristic works well for many benchmarks, but that some benchmarks will require stronger static analysis or profiler information.

2.6 mmStrata

The MEDS dynamic monitoring system is called mmStrata (memory-monitor Strata). It makes the final determination of whether a memory access causes a memory error or not. To make this determination, it strictly enforces the MEDS type system. At program start-up, mmStrata uses informative annotations created during static and profile analyses to make decisions about which static entities are pointers and sets metadata appropriately. A bounds-information object is created for each program object at start-up (statically allocated data, as well as bounds-information objects for the program’s incoming arguments that exist on the stack), and the program begins to execute.

During execution, mmStrata watches for newly created objects. For example, calls to dynamic memory allocation routines are considered to create new objects. Likewise, when mmStrata reaches a program point that static analysis has determined creates a new stack frame, mmStrata creates new bounds-information objects for the stack frame. However, some functions fail to have a stack frame clearly identified, and the dynamic system must still create bounds-information objects to protect the non-analyzed stack frame. To create these objects, the dynamic system watches call instructions (or other control flow instructions that cross function boundaries). If the call instruction targets a function that failed the static analysis of the stack frame, then the dynamic system creates a bounds-information object to represent a new, empty stack frame. While within this function, changes in the stack pointer cause a change in bounds to the bounds-information object. For example, if a dynamic stack allocation (perhaps from `alloca`) extends the stack by 700 bytes, the bounds contained within the bounds-information object are extended. Another example is if outgoing arguments for a function call are pushed, the bounds within the bounds-information object are extended, then when the call returns, those arguments are removed from the stack, and the bounds shrink. In one function we analyzed, the stack frame was created by moving 0 into the `ecx` register, then pushing `ecx` 128 times. Consequently, we believe this mechanism for monitoring non-standard stack frames is a key part of providing complete protection.

MEDS deals with stack deallocations the same way. If the stack frame is being watched dynamically, the bounds within the bounds-information object shrink. If the bounds shrink to the point at which the object has negative size, we assume that the stack frame is no longer needed and mark the stack frame as invalid, and mark the bounds-information object ready for deallocation when the reference count falls to zero.

With the information provided by the static analysis phase, the profiler and the MEDS type system, mmStrata can detect a variety of memory errors. For example, mmStrata detects double-free errors by monitoring calls to allocation and deallocation routines. Out of bounds pointer writes are detected by examining the metadata associated with the pointer to determine if the write is in-bounds. Writes to stale objects are detected by examining the valid bit of the bounds-information object. MEDS clearly provides a general defence that can detect most memory errors within a binary executable. The next section discusses limitations where memory errors might be missed.

2.7 Limitations

MEDS currently has several limitations. These limitations are active areas of research.

The first limitation is how signals are handled. The base Strata system watches asynchronous signals correctly, efficiently, and transparently. The mmStrata extensions, however, need to be enhanced. In particular, new bounds-information objects need to be created for the stack area when a signal is caught. We do not expect this to be a challenging extension, but our system currently does not handle signals.

Like signal handling, system calls to `mmap` are a challenge. The `mmap` system call offers a wide variety of ways in which memory can be allocated, including mapping multiple address ranges to the same physical memory. MEDS only handles basic `mmap` calls that simply allocate memory. Again, we do not see this as a challenging extension.

Our system currently only supports statically linked code. Dynamic linking makes communication through the annotation file more difficult, as absolute addresses cannot be used. Instead, MEDS would have to use a path to a dynamically linked library and an offset within the library. The run-time system would have to watch dynamically loaded libraries and adjust its data structures appropriately. For libraries that are chosen based on dynamic input from the user, the system needs the ability to run the static analysis phase online. These same solutions would be necessary for a system that uses self-modifying code or generates code on the fly (such as a JIT). Our system currently implements none of these solutions.

Our system also does not currently handle kernel-level threading. To handle threading efficiently would be significant work, but not impossible. One solution would involving using a locking mechanism to protect all metadata updates. This could be prohibitively expensive, however, and a scheme that moves exclusive metadata access between threads would likely provide better performance. The best mechanism for threading support is an ongoing area of research.

A subtle, yet important limitation is that our system only protects variables at the allocation level. For example, consider a call to `malloc` that allocates the structure shown in Fig. 5. MEDS will provide protection so that no pointer to the object can be dereferenced outside the object.

```
struct foo {
    int rootID;
    char user_input[100];
    int (*function_pointer)();
}
```

Fig. 5. Allocation-level granularity example

However, the `user_input` buffer may still over- or underflow and cause the program to misbehave. We believe that modern methods for detecting variable types will allow us to solve this problem, but our current implementation of this mechanism is incomplete and not reported here. A related problem is how to deal with third party allocators (such as an arena allocator) built on top of system allocation routines (such as `malloc`, `sbrk`, or `mmap`). If the application uses a third party allocator, MEDS may only provide a very coarse-grained protection for the super-objects allocated with the underlying allocation routines. How to solve these problems is an area of ongoing research.

2.8 MEDS Summary

MEDS, the Memory Error Detection System, is a system to detect memory errors in binary executables. No source or object code is required to use the system. The system operates by first running a static analysis pass, then optionally a profiling pass. Finally, a software dynamic translation-based system called mmStrata is used to detect memory errors. Each of these systems work by applying the MEDS type system, composed in its most basic form of numeric and pointer types. Simple extensions allow for the offset type to reduce false positives, and the `q`-type in the profiler for further reduce false positives. Section 3 gives compelling evidence to the efficacy of this approach.

3 Experimental Results

We separate our evaluation of MEDS into two broad categories. First is a security evaluation in which we use a selection of real benchmarks to test for true and false overwriting detections, as well as any errors that the MEDS system may have missed (Section 3.2). The second category evaluates the performance of the MEDS system on a variety of benchmarks (Section 3.3). Before that, however, we briefly describe the experimental setup (Section 3.1).

3.1 Experimental Setup

Both exploit testing and benchmark timings were performed. All test programs were evaluated on an Opteron 148 CPU, running Linux Fedora Core 6, using the gcc 3.2.2 compiler w/static linking, and -O3 -fomit-frame-pointer optimization flags. While MEDS is designed to enable detection of read or write memory errors, only memory overwrites were monitored in the configuration tested.

The benchmark programs evaluated are shown in Fig. 6. The applications in the evaluation included standard benchmark suites with no expected vulnerabilities such as the SPEC CPU2000 benchmark suite [8]. Applications with known or seeded vulnerabilities, including the Apache web server, many of the relevant cases in the SAMATE static analysis test suite, the Wilander buffer overflow suite, and the BASS vulnerability suite, were also included in the evaluation [9, 10, 11]. In addition, commonly used applications and test benchmarks were included in the evaluation such as the binutils-2.18 utility suite and the vpo [12] regression test suite, which includes benchmarks such as fm-part (VLSI placement program), matrix multiply, 8-queens solver, sieve of Eratosthenes, wc, Whetstone, Dhrystone, a travelling salesperson problem solver, etc.

Benchmark Suite	**Description**
SPEC CPU 2000	ammp, art, bzip2, crafty, equake, gap, gcc, gzip, mcf, mesa, parser, perlbmk, twolf, vortex, vpr
Wilander buffer overflow suite	buffer overflows on the stack, heap, and BSS
Benchmarks for Architectural Security Systems (BASS)	buffer overflows: 01_overflow_fp, 02_overflow_variable, 04_overflow_shellcode_injection
SAMATE Reference Dataset	test cases related to memory overwriting
Apache	web server with manually seeded vulnerability
binutils	nm, objdump, readelf, size, strings
VPO compiler test suite	ackerman, arraymerge, banner, bubblesort, cal, cb, dhrystone, fm-part, grep, hello, iir, matmult, od, puzzle, queens, quicksort, shellsort, sieve, strip, subpuzzle, wc, whetstone
nasm	netwide x86 assembler

Fig. 6. Benchmarks evaluated

In addition, several of the vpo test suite benchmarks were seeded and tested with four categories of vulnerabilities: buffer overflows, array out of bounds accesses, double free/dangling pointer references, and uninitialized pointer dereferences.

3.2 Error Detection Evaluation

The benchmarks listed above were run to evaluate detection of memory overwriting.

3.2.1 False Positives

A warning generated by the MEDS system is considered a false positive if it is determined that a memory overwrite or underwrite has been detected by the system, but no overwrite or underwrite actually occurred.

As seen in Fig. 7, for SPEC CPU2000, some false positives occurred when the profiling pass was not performed. For the set of applications with known or seeded vulnerabilities, mmStrata produced warnings only when memory overwriting was attempted, i.e. it generated no false positive reports. For the other applications tested, only apache and queens produced false positives, which were eliminated by profiling. No other tests yielded false positives, even without the profiling pass, as shown in Fig. 8.

Benchmark	Pre-Profile False Positives?	Post-Profile False Positives?	Required Offset Type Extension?	MEDS Slowdown (ref input)
ammp	No	No	No	24.4
art	No	No	No	9.5
bzip2	Yes	No	No	44.8
crafty	Yes	No	No	35.4
equake	Yes	No	No	20.4
gap	Yes	No	Yes	64.9
gcc	No	No	No	61.9
gzip	Yes	No	No	34.9
mcf	No	No	No	15.0
mesa	No	No	No	34.4
parser	Yes	No	Yes	39.7
perlbmk	Yes	No	No	51.0
twolf	Yes	No	No	34.8
vortex	No	No	Yes	52.0
vpr	No	No	Yes	35.9
Geo. mean				32.6

Fig. 7. False positive and performance results for SPEC CPU 2000

For all benchmarks which produced false positive reports prior to profiling, the false positive was eliminated by the profiling pass, due to profiler assistance in solving the problems described in Section 2.2.1. The profiling pass did not generate any new false positive reports, because the profiling algorithm is conservative.

Several benchmarks from SPEC CPU 2000 (`gap`, `parser`, `vortex`, `vpr`) contained code which was generated as the result of the combination of strength reduction and induction variable elimination. After the introduction of the offset type to the shadow type system, false positives due to encountering this type of code were eliminated.

Benchmark Suite	**Pre-profiler False Positives?**	**Post-profiler False Positives?**	**Required Offset Type Extension?**
Wilander	No	No	No
BASS	No	No	No
SAMATE	No	No	No
Apache	Yes	No	No
binutils	No	No	No
VPO compiler test suite	queens: Yes Others: No	No	No
nasm	No	No	No

Fig. 8. False positive results for assorted benchmarks

3.2.2 False Negatives

False negatives are recorded when the MEDS system does not generate a warning report when a memory overwrite should be detected. To evaluate false negatives, we verified that all the memory overwriting occurrences in our benchmarks were detected by our system. We also tested several vpo regression tests seeded with the following four categories of vulnerabilities: buffer overflows, array out of bounds accesses, double free/dangling pointer references, and uninitialized pointer dereferences. Our tool detected every instance of the seeded vulnerabilities. No false negatives for coarse-grained memory overwrites were generated for our test applications. Some fine-grained false negatives occurred, but these are outside the scope of this paper.

3.3 Performance

Fig. 7 shows the performance of the MEDS system normalized to native execution for a variety of SPEC CPU2000 benchmarks. The best performing benchmark is `179.art` at only 9.5 times slower than native speed. The worst performing is `254.gap` at 65 times slower than native speed, while the geometric mean of the benchmarks is about 33 times slower than native execution. We realize that this level of run-time overhead is too high for many application domains. However we believe that it is very suitable for off-line testing and debugging. Furthermore, it may be useful in secure environments for programs that do not have high throughput requirements, such as I/O bound applications, interactive applications or lightly loaded server programs.

As our implementation has only had modest tuning effort, we are encouraged that MEDS performs as well as past techniques, even though it is a more comprehensive system with a more in-depth type system. The closest related work, *Annelid* based on Valgrind, reported a geometric mean slowdown of 36.7 times (for the SPEC benchmarks they report), without protecting the stack, but with the additional overhead of protecting memory reads [13]. For the same benchmarks, MEDS shows 32.6 times slowdown. Continued overhead reduction is an area of ongoing research.

4 Related Work

Over time, a wide variety of memory overwriting exploits have been invented, and a corresponding variety of software defenses have been developed. Some defenses are specific to particular subsets of all memory overwriting exploits, such as stack smashing, format string, code injection, or buffer overflow exploits [14, 15, 16, 17, 18, 19, 20]. Many memory overwriting defenses require source code or pre-linkage object code, unlike MEDS, making their use infeasible in many computing environments [20, 19, 17, 21, 22]. Rewriting software in a memory-safe language (e.g., Java, C#) would prevent memory overwriting exploits, but would require source code and great time expenditure. Some defenses are probabilistic, using randomization, and therefore subject to being defeated by brute force attacks [23]. Many defenses are designed only to protect control data, i.e. code addresses used in control flow, such as return addresses and function pointers [20, 24]. However, security-critical data can include non-control data [25]. MEDS protects against all memory overwrites, whether the target of the overwrite is control data or not, and regardless of whether the attack vector is a buffer overflow, format string exploit, integer overflow of a pointer, double-free, etc.

The most comparable prior work is the Annelid tool, which was based upon the Valgrind SDT [13]. Annelid detects out of bounds reads and writes to global-static and heap memory objects. Lacking a profiler and static analyzer, it incurred too many false positives for stack objects, and the stack portion of Annelid was disabled before completion. Annelid also encountered the problems with false positives discussed in Section 2.2.1. The pointer identification problem was left unsolved, causing some false positives. The difference between pointers problem was also left unsolved, although the authors proposed that a pointer offset type (the MEDS solution) could be implemented in the future. Annelid segments (equivalent to MEDS bounds-information objects) have an unsafe cleanup mechanism. The only sound solution proposed by the authors was a slow run-time garbage collection mechanism that would have increased overhead. Finally, Annelid makes use of some (not usually available) debug information in the executable, unlike MEDS. It appears that Annelid is not being maintained or used.

5 Summary

This paper has described MEDS, the Memory Error Detection System. MEDS detects common memory errors, such as buffer overflows, within binary executable programs: no source or object code is required. MEDS starts with a static analysis phase which analyzes functions and objects in the program binary. The static analyzer writes informative annotations into a file, which are read by an optional profiling step. The profiling step helps avoid several classes of false positives unsolved by previous work. Finally, MEDS instruments the program via software dynamic translation to detect memory errors. We show how extending the type system to include the offset-type eliminates a variety of common false positives. Performance of the system is 33 times slower than native execution for our SPEC CPU2000 benchmarks. This performance is suitable for off-line uses and I/O intensive or low throughput applications.

Acknowledgements

This work was supported in part by the National Science Foundation under awards CNS-0551560 and CNS-0716446. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-07-2-0029. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Kumar, N., Misurda, J., Childers, B.R., Soffa, M.L.: Instrumentation in software dynamic translators for self-managed systems. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems, pp. 90–94. ACM Press, New York (2004)
3. Zhou, S., Childers, B.R., Soffa, M.L.: Planning for code buffer management in distributed virtual execution environments. In: VEE 2005: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, pp. 100–109. ACM Press, New York (2005)
4. Scott, K., Davidson, J.: Strata: A software dynamic translation infrastructure. In: IEEE Workshop on Binary Translation. IEEE, Los Alamitos (2001)
5. Scott, K., Kumar, N., Childers, B., Davidson, J.W., Soffa, M.L.: Overhead reduction techniques for software dynamic translation. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, p. 200. IEEE, Los Alamitos (2004)
6. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: CGO 2003: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 36–47. IEEE Computer Society Press, Los Alamitos (2003)
7. Eagle, C.: The IDA Pro Book. No Starch Press, San Francisco (2008)
8. Hening, J.L.: SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computer 7, 28–35 (2000)
9. Black, P.E.: Software assurance metrics and tool evaluation. In: Proceedings of the 2005 International Conference on Software Engineering Research and Practice (2005
10. Poe, J., Li, T.: Bass: A benchmark suite for evaluating architectural security systems. In: SIGARCH Computer Architecture News, pp. 26–33. ACM Press, New York (2006)
11. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the Network and Distributed System Security Symposium, pp. 149–162. Internet Society (2003)
12. Benitez, M.E., Davidson, J.W.: The advantages of machine-dependent global optimization. In: Proceedings of the 1994 Conference on Programming Languages and Systems Architectures, pp. 105–124. ACM, New York (1994)
13. Nethercote, N., Fitzhardinge, J.: Bounds checking entire programs without recompiling. In: Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004) (2004)

14. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanovic, D.: Randomized instruction set emulation. ACM Transactions on Information Systems Security 8, 3–40 (2005)
15. Baratloo, A., Singh, N., Tsai, T.: Transparent run-time defense against stack smashing attacks. In: Proceedings of the USENIX Annual Technical Conference, pp. 251–262. USENIX (2000)
16. Liang, Z., Sekar, R., DuVarney, D.C.: Automatic synthesis of filters to discard buffer overflow attacks: A step towards self-healing systems. In: Usenix 2005 Annual Technical Conference, pp. 375–378 (2005)
17. Ruwase, O., Lam, M.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security (NDSS) Symposium, pp. 159–169 (2004)
18. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: CCS 2003: Proceedings of the 10th ACM conference on Computer and communications security, pp. 272–280. ACM Press, New York (2003)
19. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: FormatGuard: Automatic protection from printf format string vulnerabilities. In: Proceedings of 10th USENIX Security Symposium, pp. 191–200 (2001)
20. Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium. pp. 26–29. USENIX (1998)
21. Necula, G.C., McPeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy code. In: POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 128–139. ACM Press, New York (2002)
22. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with wit. In: IEEE Symposium on Security and Privacy, pp. 263–277. IEEE, Los Alamitos (2008)
23. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of 12th USENIX Security Symposium, pp. 105–120. USENIX (2003)
24. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206. USENIX (2002)
25. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th Usenix Security Symposium, pp. 177–192. USENIX (2005)

Idea: Automatic Security Testing for 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. With the increasingly important role of web applications in online services and business systems, vulnerabilities such as SQL Injection have become serious security threats. Finding these vulnerabilities by manual testing is a time-consuming and error-prone practice that may result in some potential vulnerabilities being missed due to some execution branches being missed. In this paper, we describe an automatic security testing method to find vulnerabilities in web applications; this method utilizes test data generation techniques for improving the code coverage. Our security testing involves automatic attack request generation and automatic security checking using dynamic tainting technique that detects dangerous contents originating from untrustworthy sources in commands and outputs. Automatic constraint-based test data generation helps to create test data for executing program branches that may have remained unexecuted in previous tests. The experimental results indicate that our method is effective to find new vulnerabilities, and test data generation may help to improve the effectiveness of detection.

Keywords: Web application vulnerabilities, security testing, dynamic tainting, test data generation.

1 Introduction

Recent years have witnessed a significant increase in the number of vulnerabilities in web applications. Vulnerabilities can be exploited by attackers to obtain unauthorized access to data stored in back-end database systems or to illegally execute malicious commands on host computers. Therefore there is an obvious requirement for effective methods to find vulnerabilities in web applications. Dynamic testing and static analysis are the two main approaches that are currently used for finding vulnerabilities in web applications. Both approaches have certain advantages and disadvantages, and generally they should be used in a complementary manner to improve the detection result.

In this study, we propose a new automatic security testing method for web applications. Our method is inspired by advancement in the dynamic tainting technique used for finding vulnerabilities and the dynamic test data generation technique used for leveraging the code coverage of the test.Our proposed test is completely automated except for one step - the testers have to specify one or several entry points to the web applications in advance. Other entry points are automatically discovered during the test. Attack requests are automatically generated using a small set of attack codes that are injected into request parameters corresponding to each entry point. By using the dynamic tainting technique, our method automatically tracks the taintedness of data that originates from untrustworthy sources such as inputs sent by requests at the character level. This information is used to detect potentially dangerous tainted contents in commands and outputs generated dynamically at runtime. One of the main contributions of this work is the consideration of test data generation for improving the code coverage of the test. Automatic constraint-based test data generation helps to create test data for executing program branches that may have remained unexecuted in previous tests. We have developed a tool called Volcano to implement our proposed method for the security testing of web applications written in PHP to detect SQL Injection vulnerabilities. We focused on PHP because of its widespread use; furthermore, a large number of vulnerabilities have been reported in web applications written in PHP [1]. In our experimental evaluation, we found a total of 40 vulnerabilities in 8 sample real-life web applications using Volcano, of which 25 were new. The test data generation has helped to find 2 vulnerabilities.

2 Automatic Security Testing

Automatic Vulnerability Detection. Dynamic tainting technique has been used in many studies [2,3] for effectively detecting if dangerous data goes into sensitive sinks. Dangerous data may originate from untrustworthy sources such as user inputs; for example, in SQL Injection attacks, such inputs could change the structure of SQL queries in an application program to ones not of intended by the developer, and these may result in corruption of the data in database or leaking of data to end users (or attackers). Because input data of web application programs are all strings, our method tracks the taintedness of the values of all string variables at the character level. Input data from untrustworthy sources such as GET, POST or Cookies parameter values, or data from database, are marked as tainted. Taintedness is propagated across assignments and function calls at the granularity of a character. As compared to tracking the taintedness at the string level, tracking at the character level allows more flexibility to apply alternative security checking for different vulnerabilities.

Automatic Security Checking. We create security policies that help to separate dangerous and benign commands and outputs generated by web application programs. For the case of SQL Injection attacks, the security policy is that tainted data must not be used as SQL keywords or meta characters in SQL queries, otherwise, they could possibly change the intended structure of the queries. Then, we can find SQL Injection vulnerabilities by simply looking for tainted SQL keywords and meta characters in generated SQL query strings.

Automatic Creation of Attack Requests. Our method requires testers to first specify one or several entries for the target web application, such as the URL of the top page. If there is a login page, the user name and password are also required. After sending requests to that entries, other entry points are automatically discovered by extracting static links and form submissions from the response pages. Attack requests are created by injecting malicious strings with the intent of triggering specific vulnerabilities existing in web applications. For example, quotes, double quotes, and backslashes are used to perform SQL injection attacks. If these tainted input characters are added to SQL queries as meta characters, the test successfully detects an SQL Injection attack.

Automatic Test Data Generation. In this research, we focus only on character string predicates for data generation because such predicates are very common in web applications. Other types will be considered in future studies. For data generation, we used the algorithm described in [7]. For example, in order to generate data for an input data contained in a variable *str*, such as predicate *str* == "mod" becomes TRUE, the algorithm first maps each string to a unique non-negative integer by the definition $\xi(str) = \sum_{i=0}^{L-1} str[i] \times w^{L-i-1}$, where L is the length of the string *str* and w is set to 128. Then, the distance function is defined for computing the distance between two strings: $dis(str_1, str_2) = |\sum_{i=0}^{L_1-1} str_1[i] \times w^{L-i-1} - \sum_{i=0}^{L_2-1} str_2[i] \times w^{L-i-1}|$. The algorithm searches for an appropriate input by gradually changing each character of input data so that *str* finally has the value "mod". Detail of the algorithm can be found in [7].

3 Implementation

We developed a tool called Volcano to implement our proposed method for the security testing of web applications written in PHP to find SQL Injection vulnerabilities.

Taintedness Tracking. The PHP interpreter is written in the C language, and each variable is expressed by a structure called *zval*. We add a new structure member *char∗ taint* to the structure *zval* in PHP 5.2.2 to track taintedness of each character in a string variable. All functions related to string manipulations and assignment operations have been modified for correct taintedness propagation. These modifications do not change the behavior of the original operations, and thus, it is not necessary to rewrite the web applications. Because we target SQL Injection vulnerabilities, the function *mysql_query*, which sends SQL queries to database server, is considered as a potential vulnerable point. Every string that is input to this function as a parameter is then checked using the security checking function.

Security Checking. We implement an SQL query parser to parse dynamically generated SQL queries. If any tainted meta character or keyword is found in the SQL queries, the request is considered to be an SQL Injection attack.

Test Data Generation. We have modified the PHP interpreter to add a small code in order to capture the execution information (data- and control-flow information). This information is expressed in the XML format, and it includes the current execution point (file name and line number), runtime values of variables, results of predicates, and so on. This information helps to generate input data for new requests to increase the code coverage and provides the tester with information for fixing vulnerabilities after the test.

Compound Predicates. The structures of compound predicates are rebuilt by using the obtained data- and control-flow information. The desired value of each elemental predicate is computed based on the desired value of the compound predicate that is required to cause the program to execute a conditional branch that may have remained unexecuted in previous tests.

4 Experimental Results

We selected 8 sample applications from the well-known bug tracking site BugTraq [6] as listed in Table 1. These have been written in PHP, and they have the latest reported SQL Injection vulnerabilities. The result is compared with the test result obtained using an existing tool called Paros [4], a well-known vulnerability scanner. Paros detects vulnerabilities by simply searching for some error messages, e.g. "not a valid MySQL", in the response pages. In Table 1, *Vuln.Points* indicates the number of vulnerable *mysql_query* functions present in the program. *Vuln.* indicates the number of vulnerabilities found by using corresponding tool.

Volcano is quite effective in that it found 40 vulnerabilities in the 8 sample applications as compared to only 4 found by Paros; furthermore, only 15 of the 40 vulnerabilities had been reported previously in BugTraq. Among the 25 new vulnerabilities, the effects of some could only be identified within the web server; these effects cannot be detected using Paros. Some of them have not been reported in BugTraq because the tester may have manually reviewed the source code or only attempted to find interesting vulnerabilities as opposed to the total number of vulnerabilities.

Table 1. Result of security testing for finding SQL Injection vulnerabilities

Applications	Lines of code	Vuln. Points	Volcano			Paros			BugTraq
			Total warns	Vuln.	False positives	Total warns	Vuln.	False positives	Reported Vuln.
en.faname	1817	10	7	7	0	0	0	0	2
shnews	3689	13	9	9	0	0	0	0	2
ajchat	3383	24	1	1	0	0	0	0	1
Neuronnews	1995	60	7	7	0	0	0	0	3
PHPEchoCMS	13557	1	1	1	0	1	1	0	1
taskfreak	22872	3	3	3	0	2	1	1	1
GestDown	1486	38	8	8	0	2	2	0	3
tutorialCMS	5475	87	4	4	0	2	0	2	2

Effect of test data generation. When only character string predicates were considered, test data generation helped find 2 in 9 vulnerabilities found in the `shnews` application.

5 Related Works

Many methods such as Pixy [2] rely on static analysis for statically finding vulnerabilities. Other methods such as Amnesia [5] combine static analysis and dynamic model checking for detecting vulnerabilities at runtime. Although static analysis provides complete code coverage, it often leads to imprecise results because of its conservativeness. Penetration testing used in Paros [4] is a common testing method that provides precise results; however, the code coverage is low.

6 Conclusion and Future Work

We proposed a new automatic security testing method for web applications. Except for a small manual configuration in the initial stage, the test is completely automated. The main idea is to automatically detect vulnerabilities within the web server by using the dynamic tainting technique and to improve the code coverage of the test by automatic test data generation. We developed a tool called Volcano to implement our proposed method for the security testing of web applications written in PHP to find SQL Injection vulnerabilities. The experimental results indicate that our method is effective to find known and even some unknown vulnerabilities. In future work, we will consider alternative techniques for improving the effectiveness of test data generation. Static analysis may possibly be used to solve the problem of dependencies in compound predicates.

References

1. Lemos, R.: PHP security under scrutiny (2006), `http://www.securityfocus.com/news/11430.SecurityFocus`
2. 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, pp. 258–263. IEEE Computer Society, Washington (2006)
3. 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)
4. Chinotec Technologies Company. Paros, `http://www.parosproxy.org`
5. Halfond, W., Orso, A.: AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 174–183 (2005)
6. SecurityFocus. BugTraq, `http://www.securityfocus.com`
7. 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)

Report: Functional Security Testing Closing the Software – Security Testing Gap: A Case from a Telecom Provider

Albin Zuccato[1,2] and Clemens Kögler[1]

[1] TeliaSonera, Product Security, Sweden
[2] Stockholm University, Department of Computer and System Science, Sweden

Abstract. To offer successful products and services in the telecom business requires to show that the specified security is implemented as expected. Functional security testing is suitable for this purpose as it bridges the gap between software and security testing. In this paper we describe the aspects of such a functional security testing approach. Further we provide evidence for a practical application of our approach and show the benefits we found.

1 Introduction

Our interconnected communication world implies an entity which facilitates this communication by providing the necessary infrastructure and services. As a telecom provider we see it as our role to be this entity. In this role as facilitator we see a need to deal with a number of requirements that are put upon us. Among those we can find information security (further on called security) on a prominent position. The challenge for us as a Telecom provider is to offer the "right" kind of security and more important show that our systems really provide it.

To codify the right kind of security we rely on security requirements [1] [2] as they address the actual needs instead of the coarse results provided by security risk analysis. Security requirements have another clear advantage when it comes to showing security as they provide a criteria that can be verified. Under the assumption that it is both suitable and possible to craft security requirements that have a certain degree of uniformity [2] we can also create security test cases that provide assurance for those requirements. Such security test cases show requirement realization and allow to conclude security assurance.

Security testing is therefore important for us. To rely so strongly on testing to gain assurance origins from the fact that as a telecom provider we act in many cases as a system integrator without access to the source code. Therefore other assurance techniques like module tests, static analysis, design/code reviews or formal verification are not suitable for us.

As a commercial organization we have naturally strong demands for a security testing approach that is both efficient (e.g. short lead times, easy to integrate into business practices . . .) and effective (e.g. good security, performable in our projects by normal staff - i.e non-security-experts. . .). After investigating many existing approaches, which we discuss in the next chapter, we found the need for what we call functional security testing to bridge the gap between software and security testing. In this paper we describe the static (artifacts) and dynamic (processes) aspects of our functional security testing approach. Subsequently we present practical experiences we made. Last we offer our conclusions and suggest further research.

2 Security Testing Today

To gain assurance that an information system is secure in its operative environment it is necessary to test its behavior. As most telecommunication systems consist to a great deal of software it seems apparent to use software testing techniques. However, this is only partially correct as security testing is different in certain aspects. The most significant difference between software and security testing [3], [4], [5], [6], [7] is that security testing has to assume that the opponent searches after vulnerabilities, in many cases actively trying to create them, whereas normal testing works under the assumption of purpose oriented usage by the user.

This implies that security testing has not only to verify the requirements work but also to assure that not more than the requirements demand can be done. This can be tricky as many security requirements are ambiguous and it is not possible to enumerate all permutations of the requirement [6]. Consequently it is not possible to give a suitable set of security test cases. We also find that side effect behavior that is not part of the specification and under-fulfilment of the specification [8] can imply security vulnerabilities. Therefore side effects and under-fulfilled areas of the specification have to be identified and tested to assure that they do not compromise security.

These differences imply that specific testing approaches, which complement software testing, are necessary. [7] discusses penetration testing and risk based testing as suitable approaches for this. Both testing methods dig deep into a certain aspects of a requirement. While penetration testing is basically a malicious approach to exploit mistakes, side-effects and environmental conditions, risk-based testing verifies analyzed risk and corresponding vulnerabilities by testing critical parts where the vulnerability could occur. [3] suggested to use threat based testing, going even earlier in the security risk identification chain, where not the risks but the potential threats the system faces are used to craft the tests. These approaches are complemented by attack patterns [9], [10]. This testing method is based on "canned" (i.e. prefabricated) penetration tests with a malicious intent. In comparison to penetration tests, attack patterns are not that sophisticated and individually tailored to a system but much faster to perform. In contrast to risk-based testing it is more focused on breaking the system under test than on evaluating a critical part of a requirement.

However, the above approaches leave a gap to software testing. They seem to cover many of the security problems coming from malice. Unfortunately, quite a number of security problems originate from implementation/design mistakes both in the normal and the security functionality and from vulnerability exploitation based on curiosity (i.e. usually not considered the highest risk) and are therefore not fully covered. [7] gives the feeling that some of these issues are tackled by software testing. [3], [8] seems to make a good case that software testing alone is not capable to address these problems because they are not entirely in its scope [11]. It seems therefore that we need an approach that helps verifying the full scope, the correct functionality and potential side effects of the security requirements to complement and fill the gap.

3 Bridging the Gap with Functional Security Testing

We have seen above that there seems to be a gap between normal functional testing and requirements testing when it comes to security. We have to analyze this gap further to motivate why we need a new testing approach and what requirements we have towards it.

Risk based testing requires a "security" risk analysis (including the identification of assets, threats, vulnerabilities and risks). However with a change towards security requirements [12], [13], which have a broader scope, to rely solely on the vulnerabilities and risks to identify test cases implies that not the full range is covered. However, these broader scope will not be covered by normal tests either as we said before when discussing the need for special security testing. To put it as a requirement we find that as much as possible of the specified security (relevant) functionality has to be covered. Or as [3] puts it "security testing is about proving that the defensive mechanisms work correctly".

A second concern emerges form the fact that the more holistic security requirements suffer from the same ambiguity and side effect problem [8] mentions. That means that the security requirements will have functionality (i.e. side effects and environmental dependencies) that are not covered in the specification and which have to be tested. We find therefore that we need to investigate also around the borders of the specification to make sure that the system does not do more than it should.

A final criteria we consider important to delimit functional security testing is to state that it should not be malicious in intent. Malicious intent in our view almost naturally causes a stronger focus towards the environment. This implies a danger that the domain specific aspects codified in the requirements are possibly ignored. We therefore think that the malicious intent is far better covered with attack patterns and penetration tests and would like to emphasize the non-malicious character of functional security testing.

We therefore define "*functional security testing aims to verify the security requirements at significant functionality and around the boarders of the system under test without a malicious intent during creation and execution of the tests*".

A dedicated approach to perform such functional security testing is described in the next section. It is noteworthy that functional security testing is complementary to existing security testing approaches and does not aim to replace them.

4 Functional Security Testing Approach

Testing does not start after the implementation ends but already when the requirements are defined. As [3] notes "security cannot be tested into a product" but is a part of the overall security process. Instead security testing has to be understood as the "health check" that says that the expectations in respect to security are satisfied. A tight integration into the Product Security Life Cycle, starting at specification and ending at product retirement, is therefore an important goal.

From a practical perspective this integration requires that certain properties have to be satisfied by the approach. First of all it must be performable by project staff which usually are non-experts in respect to security. This implies clear guidelines and tools that allow a pragmatic application. A second concession to practice we deem necessary is that the impact on time effort, especially on lead time, has to be small. This means that an easy solution which fits good enough (i.e. low flexibility and low degree of freedom in respect to security) is preferred over another solution that is more accurate but requires too many security related decisions which require expertise.

The functional security testing process, shown in Fig. 1, is an instantiation of a normal testing process [11]. However, minor modifications are made to address the non-expert support issue for security testing. Most important to mention is that security tests are selected from the a-priori available security test cases stored in the test case database. This implies changes in the test design and the test data generation activities. Secondly, the evaluation is enhanced to enable system wide security assurance assessment.

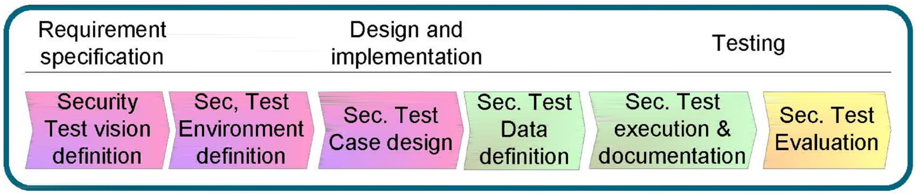

Requirement specification
Design and implementation
Testing
Security Test vision definition
Sec, Test Environment definition
Sec. Test Case design
Sec. Test Data definition
Sec. Test execution & documentation
Sec. Test Evaluation

Fig. 1. Functional Security Testing Approach

From a bird's eye view we find the distinct steps of test planning, execution and evaluation. The integration of the planning has to start with the requirement phase and stretches into the design phase of the life cycle. During implementation, when enough details become available, the activities start focus more on execution. This continues into the test phase which naturally has a clear focus on execution. During deployment a gradual shift from execution towards evaluation is suggested. It is important to mention that as today's life cycles are basically iterative the test activities are recurring and the here outlined structure addresses one requirement at the time. After this coarse outline we describe each activity in more detail.

Test Planning

The test planning activity starts testing early in the product life cycle. It starts by defining a vision which describes the sophistication of testing we investigate the criticality of security for the business. Important to note is that this covers not only resources protection but also business enabler considerations – as they also determine the effect a failure in a security mechanism would have. Secondly, we describe the expected skills of an attacker we want to protect against as this directly influences the test ambition level. To support our non-expert goal we provide three sophistication levels covering low (no great impacts on the business and low attacker skills), medium (normal importance and interested attackers) to high (critical if security fails and skilled attacker).

Subsequently we describe the preconditions toward our test environment. We mainly consider necessary system accounts, network openings, tool landscape (e.g. testing tools, observation tools, log viewers ...) and environmental data which have to exist (i.e. functional data that are necessary for system operation).

The last step in planning is to select the test cases based on the requirements. Test cases are developed a-priori by security experts for each requirement and further used by non-security-expert staff in each project utilizing the requirement. When developing tests it turns out that two different groups of requirements, *Generic requirements* and *Specific Requirements*, exist. Generic requirements represent security mechanisms which are common and widely used in various systems. The requirement parameters are similar for all systems and only the implementation differs. Due to that test cases for such requirements have to be generally applicable regardless of the actual implementation. Therefore to define test cases a-priori they must aim at top level functionality rather than specific technical details. Specific requirements on the other hand are specified by strict technical standards which provide clarification about implementation details and avoid ambiguity in actual implementations. Hence it is possible for a-priori defined test cases to take such details into account.

To find the relevant test cases the database is searched with the requirement as starting point. As the linking between requirements and test cases is already codified in the database this process is semi-automatic and requires almost no security knowledge and only little time.

Test Execution

In many cases the generation of test data is seen as a part of test design. However, due to our goal of defining test cases a-prior this is not entirely suitable. The problem areas are the test cases which are based on *Generic requirements*. The reason is that the actual test data for these tests have to be adapted to the given environment. Hence a pure a-priori definition could result in weak or inadequate test data (caused by slightly different implementations from different vendors). Therefore the interpretation of the environmental dependencies must be left to the executing person which violates our simplicity requirement a bit. We try to tackle this by defining test data types and indicate suitable ranges which seems pragmatic enough to not hamper overly. Such test data types can reach from regular expressions, in a format which must be easy to understand for non-experts, to unstructured textual descriptions, which still have to create a clear and unambiguous delineation of the expected data to avoid weak test data caused by miss-interpretation. The situation is different for the test cases for *Specific Requirements*. Standardization regulates so strongly here that it is possible to specify also generally applicable test data in advance. Some minor adaptation is still required but this seems to be without problem.

We defined from the beginning that the execution of the tests must be performed by humans. An automated execution of tests is out of scope. The reasons are the broad range of diverse requirements the approach has to cover and the variety of system we find in the telecommunication ecosystem. Therefore, the designed test cases contain instructions how to perform them. As testers are used to this procedures this is not perceived as a problem. During the execution of the tests the results have to be stored in the database. We found the use of the database extremely useful as completeness and consistency checks can be performed automatically which ensures sufficient data quality.

Test Evaluation
To gain the assurance we are seeking we have to perform an evaluation of the test results for each requirement. The achieved test results documented in the database are semi-automatically compared to expected results. Semi-automatic means that although the system correlates the test plan with the test result a human has to perform the decision if the expectation is meet by the test result. As each test case , which includes the expected results, is linked to a requirement it is then possible to conclude towards the fulfillment of the requirement.

5 Examples for Test Cases

The test cases constitute a cornerstone of our approach as they encapsulate all security knowledge. They have to be available a-priori so that the tester can select the appropriate ones based on the requirements she wants to test. Currently our approach includes over 50 requirements which are linked to around 80 test case specifications[1] and over 150 atomic test cases[2]. Due to that we think that we have achieved a holistic coverage of many security requirements normally emerging in a telecommunication environment. Test case specifications are used as aggregation layer for reusability reasons in the test case database between the performable atomic test cases and requirements.

[1] A test case specification represents an abstract test concept that is constructed from atomic test cases.

[2] An atomic test cases represent non-overlapping tests which can be performed. An atomic test case can be associated with multiple test case specifications.

We find that test cases are constructed bottom-up[3] by analyzing the critical functionality and eventual boundary issues of the requirement. This bottom up treatment requires a solid understanding of security. However, the security expertise is only needed during construction. During application the test cases are already available and can be applied by non-experts as described in the last chapter. Therefore usage is performed top-down which is more suitable for non-experts that start out from requirements.

Due to space limitations we only show a generic and a specific test case. A detailed treatment of these issues can be found in [14].

5.1 One Factor Authentication

As above mentioned the *One Factor Authentication* is a generic requirement. This implies that the top level functionality is taken into account when preparing test cases. The following listing gives an overview of the relevant functionality: (a) Authentication functionality, (b) Secret change, (c) Input boundary, and (d) Authentication robustness [9], [15], [3].

These parts represent critical areas of the requirement which have to be tested. We therefore create an atomic test case for each of them. *Authentication functionality* is concerned about the basic functionality of providing only correct authenticated users access to the system. *Secret change* assures the correct functionality of the secret change mechanisms. Thereby the test includes common problems for such mechanisms. The *Input boundary* test targets on common boundary problems in one factor authentication mechanisms. *Authentication robustness* assures that the requirement behaves as intended even when pushed to areas slightly beyond it's specification.

These atomic test cases are aggregated into test case specifications which are linked to the requirement indicating the relevant security- and regression- levels for these entities. Security levels define the depth a requirement is tested with. Regression levels classify a test case specification according to its usage in regression tests [16]. The way of aggregating the atomic test cases to test case specifications depends on the need of how to test (defined by an organization's policies and guidelines concerning security). For this generic requirement we find that test have to be available for one security level and two regression levels. Therefore two test case specifications are created. One covers basic functionality and the most critical tests. The second assures the correctness of the implementation for more advanced areas and therefore contains the remaining tests. This provides the opportunity to chose the relevant tests according to the available resources and necessary degree of regression.

[3] Naturally the bottom-up constructions requires completeness verification as it does not necessarily come automatically. We therefore perform for each test cases a verification against the testing techniques useable for security described in [6] and document this for later reference in the database.

5.2 SSL/TLS

SSL/TLS is in contrast to the above explained a specific requirement. It is defined by a technical specification which every implementation has to follow. Due to this standardization it is possible to create tests which have a stronger internal focus. As basis for the test cases the specification from [17] and common problems presented in [18] have been considered. Due to that the following relevant functionality can be identified: (a) Transfer, (b) Algorithm, (c) Client Handshake, (d) Server Handshake, and (e) Package Boundaries.

Transfer is about assuring the principal functionality of the requirement. Hence it includes confidentiality and integrity aspects of the established communication channel. The *Algorithm* test focus on the used ciphers by the implementation and their compliance with the organization's policy. The initial handshake between the two involved parties in a *SSL/TLS* connection can be seen as another critical part. Hence the correctness of this task is assured in the *Client Handshake* and *Server Handshake* tests. The last significant functionality of the requirement is *Package Boundaries*. As the length and content of some packages is strictly prescribed by the specification, this test assures that these limitations are handled correctly by the implementation.

The assumption for creating the test case specification is in this case determined by two diverse security levels and one regression level. Hence two test case specifications are created. The one which can be used to gain assurance for the lower security level includes not all of these tests, while the other one aggregates all test cases to provide assurance for all identified functionality of the requirement.

6 Practical Experiences

To test the practical applicability of the functional security testing approach a small project called "My Family 2.0 – Calendar" was chosen. This offering aims to bind entire families to the provider by offering interesting price models and additional functionality. In our case this is a calendar application where the entire family has shared calendars via mobile phone.

We started out by creating the test plan. The ambition level was easy set as the calendar data directly suggest an average to high ambition. To verify our approach we conducted a normal test planning and our approach with two different testers. For our approach we selected the relevant test cases from the database by iterating over the requirements. A critical issue we found was the missing graphical interface[4]. Due to that it was necessary to perform SQL statements to extract the needed data. When the tester extracted the relevant data from the database he identified eight TestCaseSpecifications in total. Six of these specifications matched directly to the given requirements. The remaining two had some shortcomings as some aspects of the requirements were not considered in detail.

[4] We have now a graphical interface available which eliminated this problem in follow up tests.

However, due to the use of the test case specifications the tester mentioned the positive effect of discovering missing or ambiguous parts in the system's specification. Altogether 35 atomic test cases where identified with our approach. The verification endeavor with traditional testing found 15 test cases for the same problem – all of which also our approach covered. We attribute this to the completeness checks we could perform due to the structured representation in the database. We also found significant differences in the time effort. With our approach we needed two hours for a first draft of the test plan and another hour for finalizing. In contrast the verification endeavor needed six hours for this task. We expect that time saving around this size are to expect also in future.

During test execution we could not find any systematic differences between the approaches as they basically relied on the same structures and people. Naturally more test cases implied an increase in test time but delivered far better assurance which we consider a worthwhile trade-off.

In respect to the security test evaluation we have no comparison as this was not performed earlier. From the result perspective we find a positive impact of the evaluation on assurance. A finding we expected due to the explicit mentioning of this activity in the common criteria.

To summarize we can say that we could test security more thoroughly with a decrease in planing time. We consider this very promising results.

7 Conclusion

In this article we set out to present functional security testing as a complimentary security testing technique that helps us as a telecommunication provider to show to our customers that our systems are in compliance with our stated security requirements. Earlier we had problems to assure the full scope of our security requirements as the existing techniques could not take non-risk originate requirements into account. With our new approach we are confident that we can do this as it bridges the gap between normal and security testing.

We presented our test process which aims to support non-security-expert tester. This is an important feature to make testing successful in an environment where constantly many new products and services are developed. If security testing would not be performable by the non-experts to a satisfactory degree it would be practically impossible to gain security assurance as the security staff would never manage the sheer amount of tests. A second advantage we see is that the security staff can now focus an crafting new test cases which are then reused by a broad public. Altogether we also find that the execution and lead time during test planning is reduced significantly and so more time on the actual testing can be spent. A fact we associate ultimately with improved security assurance.

Although we have already made some practical experiences the here presented approach is currently in the early phases of usage. We find it currently satisfactory for the non-malicious part of security requirement assurance. In the long run we hope to also include some more aggressive security testing by means of attack patterns. We also work constantly with adding more test cases and better guidelines to allow a more precise selection of test cases (i.e. test cases better matched to the requirement under test).

References

1. International Organization for Standardization: ISO/IEC 17799:2000, Information technology – Code of practice for information security management (2000)
2. Zuccato, A., Endersz, V., Daniles, N.: Security requirement Engineering at a Telekom Provider. In: Jakoubi, S., Tjoa, S., Weippl, E. (eds.) ARES 2008 proceedings. IEEE Computer Society Press, Los Alamitos (2008)
3. Howard, M., LeBlanc, D.: Writing Secure Code, 2nd edn. Microsoft (2003)
4. Whittaker, J.A., Thompson, H.H.: How to Break Security. Addison-Wesley Longman, Amsterdam (2003)
5. Potter, B., McGraw, G.: Software security testing. IEEE Security & Privacy (2004)
6. Michael, C.C., Radosevich, W.: Risk-based and functional security testing. Technical report, U.S. Department of Homeland Security and Cigital Inc. (2005)
7. McGraw, G.: Software Security: Building Security. Addison-Wesley Software Security Series (2006)
8. Thompson, H.H.: Why security testing is hard. IEEE Security & Privacy (2003)
9. Hoglund, G., McGraw, G.: Exploiting Software: How to Break Code. Addison-Wesley Software Security Series (2004)
10. MITRE: Common attack pattern enumeration and classification. Technical report, MITRE Corporation and U.S. Department of Homeland Security (2008), `http://capec.mitre.org`
11. Craig, R., Jaskiel, S.: Systematic Software Testing. Artech House Publishers, Northwood (2006)
12. Gerber, M., von Solms, R.: From Risk Analysis to Security Requirments. Computer & Security 20(7), 577–584 (2000)
13. Zuccato, A.: Holistic security requirement engineering for electronic commerce. Computers & Security 23/1, 63–76 (2004)
14. Kögler, C.: Functional security testing – an approach for a global telecommunication company. Technical report, TeliaSonera (03, 2009)
15. Chess, B., West, J.: Secure Programming with Static Analysis. In: Getting Software Security Right with Static Analysis. Software Security Series. Addison-Wesley, Reading (2007)
16. Kaner, C., Falk, J., Nguyen, H.Q.: Testing Computer Software, 2nd edn. John Wiley & Sons Inc., Chichester (1999)
17. IETF: The TLS Protocol. Technical report, Internet Engineering Taskforce - Network Working Group (1999)
18. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 Protocol. In: USENIX Workshop on Electronic Commerce. USENIX Association (November 1996)

Idea: Measuring the Effect of Code Complexity on Static Analysis Results

James Walden, Adam Messer, and Alex Kuhl

Department of Computer Science
Northern Kentucky University
Highland Heights, KY, 41099

Abstract. To understand the effect of code complexity on static analysis, thirty-five format string vulnerabilities were studied. We analyzed two code samples for each vulnerability, one containing the vulnerability and one in which the vulnerability was fixed. We examined the effect of code complexity on the quality of static analysis results, including successful detection and false positive rates. Static analysis detected 63% of the format string vulnerabilities, with detection rates decreasing with increasing code complexity. When the tool failed to detect a bug, it was for one of two reasons: the absence of security rules specifying the vulnerable function or the presence of a bug in the static analysis tool. Complex code is more likely to contain complicated code constructs and obscure format string functions, resulting in lower detection rates.

Keywords: Static analysis, code complexity.

1 Introduction

As an increasing number of vital operations are carried out by software, the need to produce secure software grows. Because of the complexity of software and the rapidly changing nature of vulnerabilities, it is impractical to identify vulnerabilities through developer code reviews alone. To help address this problem, developers have been increasingly using static analysis tools to identify security vulnerabilities.

We conducted an experiment that measured detection and false positive rates of static analysis tools using a set of thirty-five format string vulnerabilities from the National Vulnerability Database[6]. Two code samples were analyzed for each vulnerability, one containing the vulnerability and one in which the vulnerability had been fixed. These vulnerabilities were analyzed to determine the impact of code complexity on the error rates of static analysis tools.

Several comparative evaluations of static analysis tools have been published [9,10,3]. These studies used two techniques: microbenchmarks, which are small programs containing a single security flaw[3], and samples extracted from known security flaws found in open source software[10]. While Kratkiewicz[3] categorized each test case according to local code complexity categories, such as aliasing depth or type of control flow, the samples are too small to demonstrate the complexity found in complete programs. Zitser[10] extracted small samples containing the vulnerabilities from open source software because the tools evaluated in the study could not successfully analyze the complete source code. This study differs from the ones mentioned above by analyzing complete open source applications, allowing us to study static analysis in the conditions under which it is normally used.

2 Test Procedures

Thirty-five format string vulnerabilities in open source Linux software written in C or C++ were selected randomly from the National Vulnerability Database. For a test case to be evaluated, source code for both the vulnerable and fixed versions of the software had to be available. The software must also be able to be compiled with the GNU C compiler on Red Hat Enterprise Linux 4. Test cases that did not meet these criteria were replaced with another test case selected randomly.

For each vulnerability, both the version of the software with the reported vulnerability and a later version of the software where the vulnerability was reported to be fixed were evaluated. Several software packages had multiple entries in the NVD for format string vulnerabilities. Only the first sample selected for such a package was used, so no software was evaluated twice.

Software was compiled using either gcc 3.4.6 or 3.2.3 on Red Hat Enterprise Linux 4. Some software had to be patched in order to compile with these versions of gcc. The patches fixed differences in include files or C language variants, such as older versions of gcc permitting the presence of an empty case at the end of a switch statement[1]. No patch altered lines of code where the selected vulnerabilities existed or where they were fixed.

We used Fortify Software's Source Code Analyzer 4.5.0. Older open source static analysis tools, such as flawfinder and ITS4, were not selected because they rely on simple lexical analysis techniques that produce many false positives[10]. While numerous modern tools exist, the freely available tools were not suitable for this study, as they were not able to analyze larger pieces of software, did not support common variants of C, or required extensive configuration specific to each piece of software[2].

If the flaw was identified in the vulnerable version of software, the result was recorded as a successful detection. If the flaw was not found, a false negative was recorded. If the flaw continued to be detected in the patched software, a false positive was recorded. If no vulnerability was found in the patched version, the analysis was marked as correct.

3 Results

Two complexity measures were computed for each application. The first metric was Source Lines of Code (SLOC), which is the number of lines of code excluding comments or blank lines. We measured SLOC using SLOCCount[8]. The second metric was McCabe's cyclomatic complexity[4]. The tool used to measure

Class	Lines of Code	Samples	Detections	Complexity	Samples	Detections
Very Small	< 5000	9	7	< 1000	10	7
Small	5000-25000	9	7	1000-5000	10	8
Medium	25,000-50,000	7	4	5000-10,000	5	3
Large	50,000-100,000	6	2	10,000-25,000	6	2
Very Large	$> 100,000$	4	0	$> 25,000$	4	0

Fig. 1. Size and Complexity Class

cyclomatic complexity was PMCCABE[7]. We divided the applications into five classes by size and complexity values as described in Fig. 1.

Of the 35 vulnerabilities examined, 22 (63%) were detected by the static analysis tool. Fig. 2 shows that detection rates of format string vulnerabilities decreased with increasing code complexity ($p = 0.02$). While there was no substantial difference in the quality of static analysis results between the very small and small categories, the quality of results declined noticeably as complexity increased beyond the small category, declining to zero for the very large category, which included software larger than 100,000 lines of code.

We found two causes of failed detections of format string vulnerabilities. Four of the thirteen (31%) failed detections resulted from the first cause, format string functions that were not in the rule set of SCA. One of the format string functions that was not detected was the `ap_vsnprintf()` function from the Apache Portable Runtime. It is impossible for a tool to track all potential format string functions. However, these mistakes could be fixed by the developer adding rules to detect the format string functions that are used in the developer's application.

The second cause was a bug in how Source Code Analyzer counts arguments that are passed into a function using the C language's varargs mechanism. In these cases, the application contains a variadic function that wraps the call to the dangerous format string function. Nine of the thirteen (69%) failed detections resulted from this cause. This bug has been reported to Fortify.

Neither of these causes has a necessary relationship to code size or complexity. However, larger projects are more likely to use their own specialized functions for input and output instead of directly using functions from the language's standard library. This means that larger projects are more likely to call format string functions through wrappers. Large software projects also tend to include a larger number of developers, which typically provides a wide range of knowledge. This knowledge can lead to use of a broader subset of a language's features than would be used in smaller projects, resulting in heavier usage of the varargs feature.

Divided into five classes, there was no significant difference ($p < .01$) between the results for lines of code and those for cyclomatic complexity.

Discrimination[5], a measure of how often an analyzer passes the fixed test case when it also passes the matching vulnerable test case, is shown in Fig. 3. This metric provides an important check on the results of a static analysis tool, as a tool can achieve a high detection rate through overeager reporting of bugs, which also produces many false positive results. Discrimination ensures that the tool accurately detected both the vulnerability and its fix. The discrimination

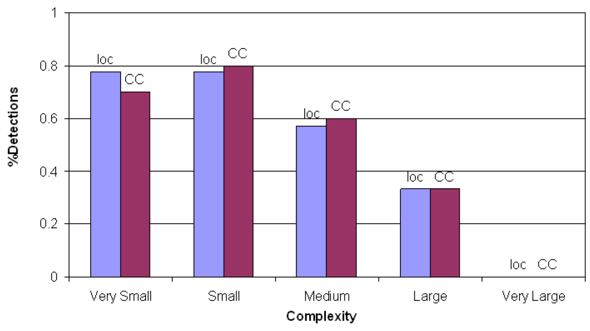


Fig. 2. Detections by Complexity Class

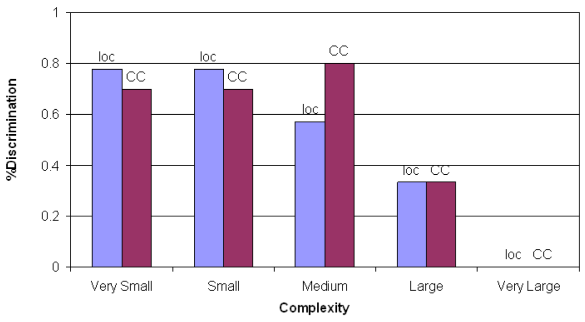


Fig. 3. Discrimination by Complexity Class

graph closely resembles the complexity graph above, as Source Code Analyzer returned only two false positives during the analysis of the 35 fixed samples. Like detection rate, discrimination rate decreases with complexity ($p = 0.02$).

In order to determine how far these results can be generalized, we need to measure the effect of code complexity using different types of vulnerabilities and software written in other languages. Analyzing software written in a language such as Java would also enable us to use a broader range of open source static analysis tools.

4 Conclusion

Thirty-five format string vulnerabilities were analyzed to determine the limitations of the usefulness of static analysis tools. We examined the effect of code complexity, measured using lines of code and cyclomatic complexity, on the quality of static analysis results. Our results show that detection rates of format bugs decreased with increasing code complexity. There were two reasons why the tool failed to detect vulnerabilities: use of format string functions absent from the tool's rule set and a bug in processing parameters submitted to variadic functions.

Acknowledgements

The authors thank Brian Chess, Maureen Doyle, and Pascal Meunier for their comments on the paper. We would also like to thank Fortify Software for permitting use of their Source Code Analysis tool for this project.

References

1. Gcc 3.4 changes, `http://gcc.gnu.org/gcc-3.4/changes.html`
2. Heffley, J., Meunier, P.: Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security? In: Proceedings of the 37th Hawaii International Conference on System Sciences. IEEE Press, New York (2004)
3. Kratkiewicz, K., Lippmann, R.: Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In: 2005 Workshop on the Evaluation of Software Defect Tools (2005)
4. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
5. Newsham, T., Chess, B.: ABM: A Prototype for Benchmarking Source Code Analyzers. In: Workshop on Software Security Assurance Tools, Techniques, and Metrics. U.S. National Institute of Standards and Technology (NIST) Special Publication (SP), pp. 500–265 (2006)
6. NVD, `http://nvd.nist.gov/`
7. PMMCABE, `http://www.parisc-linux.org/~bame/pmccabe/overview.html`
8. SLOCCount, `http://www.dwheeler.com/sloccount/`
9. Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools For Static Intrusion Prevention. In: Proceedings of the 7th Nordic Workshop on Secure IT Systems, Karlstad, Sweden, pp. 68–84 (2002)
10. Zitser, M., Lippmann, R., Leek, T.: Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. SIGSOFT Software Engineering Notes 29(6), 97–106 (2004)

Author Index