Universität Augsburg

Report on the RS^3 Topic Workshop "Security Properties in Software Engineering"

Martin Ochoa, Sebastian Pape, Thomas Ruhroth, Barbara Sprick, Kurt Stenzel, and Henning Sudbrock

Report 2012-02 February 2012

Institute for Software & Systems Engineering

Institut für Informatik
D-86135 Augsburg

Report on the RS^3 Topic Workshop "Security Properties in Software Engineering"[*]

Augsburg, May 12–13, 2011

and its Follow-up Meeting

Dortmund, August 15-16, 2011

Martin Ochoa[2], Sebastian Pape[2], Thomas Ruhroth[2], Barbara Sprick[1], Kurt Stenzel[3], and Henning Sudbrock[1]

[1]FM-SecEng (TU Darmstadt, Modeling and Analysis of Information Systems)
[2]MoDelSec (Dortmund University, Software Engineering)
[3]IFlow (Augsburg University, Software and Systems Engineering)

Technical report 2012-02[†]

Augsburg University

[*]This work was sponsored by the priority programme 1496 "Reliably Secure Software Systems" of the Deutsche Forschungsgemeinschaft DFG.

[†]Available at `http://opus.bibliothek.uni-augsburg.de/volltexte/2012/1865/`

Contents

1 Introduction

The DFG priority programme "Reliably Secure Software Systems"[1] (RS^3) aims at a property centric approach to security. Within the priority programme, the cluster "Security Engineering" addresses questions of how such a property centric approach can already be considered at design time of a system.

Users and providers of software have certain security requirements on the software they use. In order to ensure that software meets these requirements, developers of software must consider the requirement already at design time, even during early phases. However, it has turned out to be unclear how concrete security requirements can be expressed in terms of security properties in the formalisms used during different phases in software development. How can security requirements that are expressed in natural languages be specified in the various formal and semi formal specification languages like e.g. UML, process algebras, trace systems, etc? The focus shall be on end-to-end security requirements modeled with information flow properties.

On the other hand, there are different security properties like, e.g. noninterference properties contained in the literature, that can be specified in various of these specification and modeling formalisms. However, it is not always clear, what these properties mean in terms of concrete security requirements.

The RS^3 cluster conducted a two day topic workshop at the university of Augsburg[2] and a 1.5 day follow-up workshop at TU Dortmund[3], where these questions were addressed and the relation between 'real' security requirements and formal security properties expressed in various modeling and specification languages was investigated.

Goals of the workshops were to exchange and gain insights on modeling and specifying security properties in security engineering and to acquire a common understanding on the following questions:

- What are relevant security requirements in practice?
- What are appropriate formalisms for specifying such requirements as formal security properties? What role plays the software development process?
- How can security properties be modeled/specified in these formalisms?
- What do security properties specified in the various formalisms mean intuitively? How can formal security properties be (re-)translated into user friendly languages? There questions are examined for concrete examples.

This report is a compilation of results from the two workshops as well as discussions from the RS^3 annual meeting held at Karlsruhe Institute of Technology in

[1]`http://www.reliably-secure-software-systems.de/`

[2]May 12–13, 2011, projects FM-SecEng (TU Darmstadt:Barbara Sprick, Henning Sudbrock), IFlow (University of Augsburg: Peter Fischer, Kuzman Katkalov, Gerhard Schellhorn, Kurz Stenzel), MoDelSec (TU Dortmund: Martin Ochoa, Sebastian Pape), MoVeSPAcI (TU Kaiserslautern: Christoph Feller)

[3]August 15–16, 2011, projects FM-SecEng (TU Darmstadt:Barbara Sprick, Henning Sudbrock), IFlow (University of Augsburg: Peter Fischer, Kuzman Katkalov, Kurt Stenzel), MoDelSec (TU Dortmund: Jan Jürjens, Martin Ochoa, Sebastian Pape, Thomas Ruhroth), MoVeSPAcI (TU Kaiserslautern: Christoph Feller), WS4Dsec (University of Rostock: Andreas Lehmann, Stefan Pfeiffer)

September 2011. It includes approaches to formally model security requirements of a reviewing system (Sect. 2); approaches to formally model confidentiality and integrity requirements for mobile applications (Sect. 3); lessons learned (Sect. 4), and a categorized collection of informal security requirements (Appendix A).

2 Formalizing Security Properties for a Reviewing System

For the concrete formalization of security properties, we selected the scenario of a reviewing system like EasyChair, that for example has the following security requirements:

1. Only PC members can see all submissions.
2. Only after submitting a review can a reviewer see other reviews.
3. A reviewer does not know if other reviewers have already submitted reviews.
4. Reviewers are not influenced by other reviewers.
5. Persons with multiple roles have restricted privileges (e.g. a PC member that also submits a paper can see all reviews except for her own paper during the review phase).
6. Reviewers are anonymous (to authors and possibly each other).
7. In a double blind review process the reviewers do not know the identities of the authors.
8. Only accepted papers are made public, and only after the end of the review phase, but statistics about submissions are published.
9. Reviews are never deleted, all remain accessible. A user cannot cause a review to become invisible.
10. A reviewer can only submit reviews to papers she was assigned for.
11. A reviewer may not write (or view) reviews for papers with a conflict of interest.
12. Authors are anonymous until after bidding.
13. An author (paper submitter) cannot see other submissions.

An important feature of a reviewing system is that it can be divided into several phases with different security requirements. After the end of one phase information is disseminated that was secret before. For example, an author does not see other submissions, but in the end gains knowledge about accepted papers, or bidding on papers for reviewing may be blind.

In two groups, we explored how to formalize some of the security properties in this scenario.

2.1 Security properties in a reviewing system (Group 1)

The participants of group 1[4] formalized security requirements of this scenario with UMLsec [Jür05] as well as in the Modular Assembly Kit for Security [Man00, Man03]. One focus of the work was the question of how to specify the different phases of the reviewing system without specifying a detailed system model. Another focus was to formalize and analyze the formalization of a seemingly simple security requirement.

2.1.1 System Specification

We consider a system that manages paper submission, paper reviewing, and author notification (like, for instance, Easychair). We model the following phases of the reviewing system:

Phase 1	Determining the program committee
Phase 2	Registration of authors and submission of articles
Phase 3	Assignment of papers to reviewers
Phase 4	Reviewing the papers
Phase 5	Acceptance respectively rejection of articles
Phase 6	Notification of Authors
Phase 7	Publication of accepted articles

To specify the system formally, we use the concept of an event system. An event system consists of a set of events E and a set of possible traces $Tr \subseteq E^*$, which is prefix-closed. An event $e \in E$ models an event occurring during a system run, and a trace $\tau \in Tr$ models a possible system run by the sequence of events that occur in the system run.

The set of events

For defining the set of events, we consider the following sets that we do not specify further: A set of persons P, a set of articles A, and a set of reviews R. Persons can be both reviewers and authors. We also considered modeling reviewers and authors with different (but not necessarily disjoint) sets, but here we just use one set.

We define the set of events as the union of all sets specified in the following list. The list is structured by the phase of the reviewing system in which an event typically occurs. For simplicity, we assume that each article is written by a single person (and not by a list of persons).

[4]Christoph Feller, Kuzman Katkalov, Martin Ochoa, Gerhard Schellhorn, Henning Sudbrock

Phase 1	
$\{addToPC(p) \mid p \in P\}$	person p added to the PC
(alternative: $\{addToPC(X) \mid X \subseteq P\}$	persons in X added to the PC)

Phase 2	
$\{register(p) \mid p \in P\}$	person p registers as an author
$\{p\text{-}submit(a, p) \mid a \in A, p \in P\}$	submission of article a by person p
$\{p\text{-}resubmit(a, a'p) \mid a \in A, p \in P\}$	resubmission – article a by person p is replaced with article a'

Phase 3	
$\{assign(a, p) \mid a \in A, p \in P\}$	article a is assigned to reviewer p

Phase 4	
$\{download(a, p) \mid a \in A, p \in P\}$	reviewer p downloads the article a
$\{r\text{-}submit(r, a, p) \mid r \in R, a \in A, p \in P\}$	reviewer p submits review r for article a
$\{r\text{-}view(r, a, p_1, p_2) \mid r \in R, a \in A, p_1, p_2 \in P\}$	reviewer p_2 reads review r of reviewer p_1 for article a

Phase 5	
$\{accept(a) \mid a \in A\}$	PC marks article a as accepted
$\{reject(a) \mid a \in A\}$	PC marks article a as rejected

Phase 6	
$\{notify(p, a, r, b) \mid p \in P, a \in A, r \in R^*, b \in \mathbb{B}\}$	author p is notified about acceptance/rejection (modeled by boolean value b) of article a, receiving the reviews in the list r

Phase 7	
$\{publish(a, p) \mid a \in A, p \in P\}$	Article a by author p is published
$\{statistics(n, m) \mid n, m \in \mathbb{N}\}$	statistics about acceptance rate are published (n articles accepted, m articles submitted)

The set of traces

We specify the set of traces via a state chart. Each state corresponds to a phase of the reviewing system, where only events corresponding to that phase are enabled (possibly with preconditions). Transitions from one phase to the following phase do not correspond to an event. The state chart is displayed Figure 1.

2.1.2 Security Requirements Specification

We considered the following two security requirements for the reviewing system:

1. The confidentiality requirement that reviewers must not learn about re-submissions of an article.

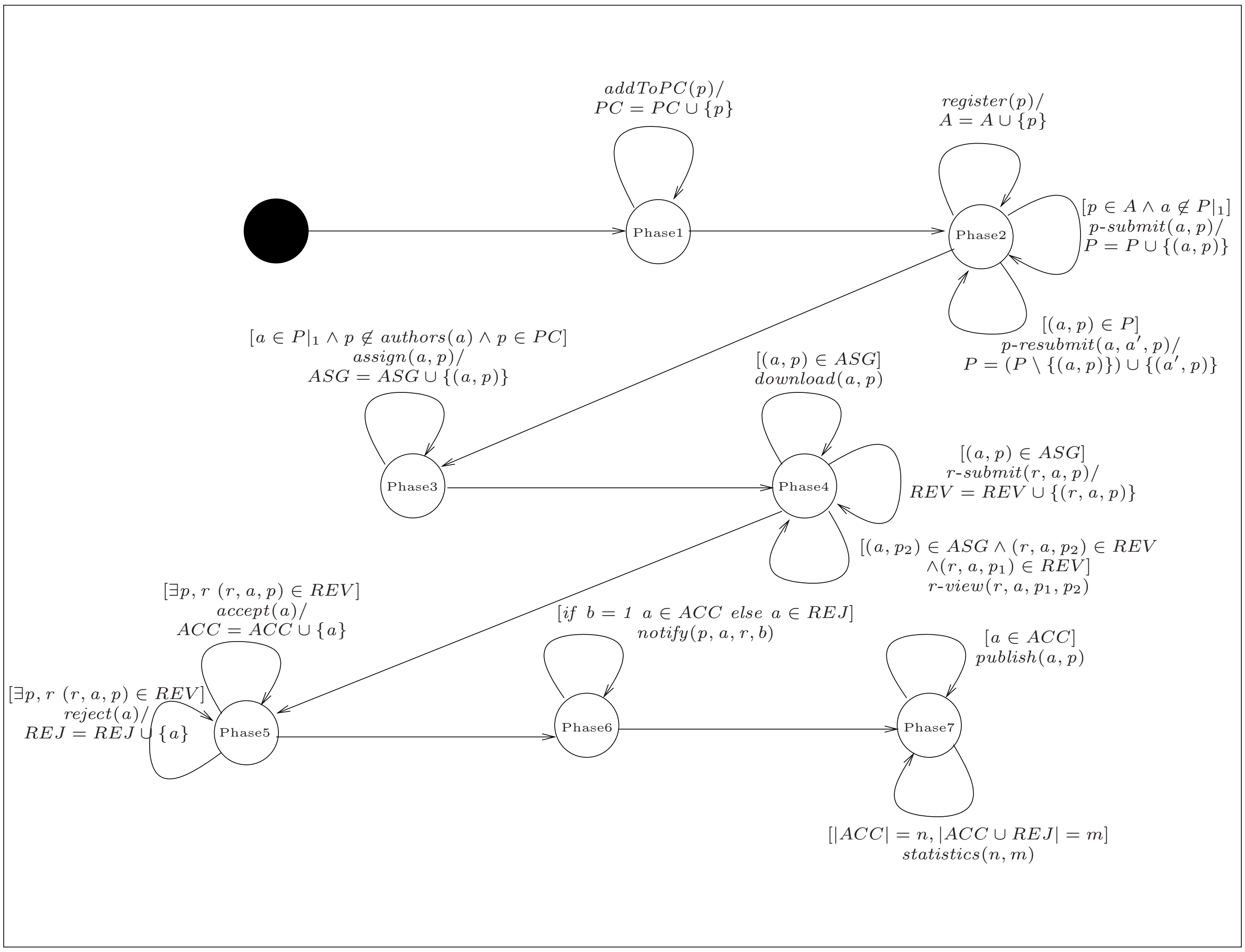

$addToPC(p)/$
$PC = PC \cup \{p\}$
$register(p)/$
$A = A \cup \{p\}$
Phase1
Phase2
$[p \in A \wedge a \notin P|_1]$
$p\text{-}submit(a, p)/$
$P = P \cup \{(a, p)\}$
$[(a, p) \in P]$
$p\text{-}resubmit(a, a', p)/$
$P = (P \setminus \{(a, p)\}) \cup \{(a', p)\}$
$[a \in P|_1 \wedge p \notin authors(a) \wedge p \in PC]$
$assign(a, p)/$
$ASG = ASG \cup \{(a, p)\}$
$[(a, p) \in ASG]$
$download(a, p)$
Phase3
Phase4
$[(a, p) \in ASG]$
$r\text{-}submit(r, a, p)/$
$REV = REV \cup \{(r, a, p)\}$
$[(a, p_2) \in ASG \wedge (r, a, p_2) \in REV$
$\wedge (r, a, p_1) \in REV]$
$r\text{-}view(r, a, p_1, p_2)$
$[\exists p, r\ (r, a, p) \in REV]$
$accept(a)/$
$ACC = ACC \cup \{a\}$
$[if\ \ b = 1\ \ a \in ACC\ \ else\ \ a \in REJ]$
$notify(p, a, r, b)$
$[a \in ACC]$
$publish(a, p)$
$[\exists p, r\ (r, a, p) \in REV]$
$reject(a)/$
$REJ = REJ \cup \{a\}$
Phase5
Phase6
Phase7
$[|ACC| = n, |ACC \cup REJ| = m]$
$statistics(n, m)$

Figure 1: State chart specifying the traces of the reviewing system

2. The integrity requirement that reviewers must not prohibit that other reviewers read a given review.

Our goal was to specify these security requirements with MAKS, the *Modular Assembly Kit for Security Properties* [Man00, Man03]. In the following, we provide the resulting formal definitions for the first of the two security requirements, as well as information on what we learned in the process of the specification.

Reviewers must not learn about resubmissions – first approach

In a first step, we characterized the requirement more precisely:

> Property A: "Reviewer *Rev* does not learn that article *Art* has been resubmitted (i.e., that a new version of the article has been submitted), unless *Rev* submitted *Art* himself as an author of *Art*. However, *Rev* may learn that *Art* has definitely not been resubmitted."

The first step was defining an appropriate view:

- The set V of visible events contains all the events in which *Rev* is directly involved, and which *Rev* can hence observe:

$$V = \{\mathit{download}(\mathit{Art}, \mathit{Rev}), \mathit{r\text{-}submit}(r, a, \mathit{Rev}), \mathit{r\text{-}view}(r, a, p, \mathit{Rev}), \\ \mathit{p\text{-}submit}(a, \mathit{Rev}), \mathit{p\text{-}resubmit}(a, \mathit{Rev}) \mid r \in R, a \in A, p \in P\}$$

- The set C of confidential events contains the events where a new version of the article *Art* is submitted (but not by *Rev*):

$$C = \{\mathit{p\text{-}resubmit}(\mathit{Art}, p) \mid p \in P \wedge \ p \neq \mathit{Rev}\}$$

- The set N contains all remaining events:

$$N = E \setminus (V \cup C)$$

The second step was to select appropriate basic security predicates (BSPs). We quickly decided that we need a BSP that requires the deletion of confidential events, as reviewer R must not learn that a resubmission event occurred (but may learn that no resubmission event occured). After some more discussion, we convinced ourselves that the BSP R ("Removal") is sufficient, with the following reasoning:

- For each trace containing resubmission-events, R requires that there is another trace without any resubmission-events, such that the projections of both traces on events in V are equal (i.e., reviewer R cannot distinguish the two traces from his observations).
- In consequence, reviewer R can never tell from his observations that paper *Art* was resubmitted (as by R, at least one system behavior generating the same observations contains no resubmission-event).

In our discussion we also considered choosing the BSP D ("Deletion", which is stronger than R). Hence, we asked ourselves which intuitive security requirement we would have captured with the BSP D. We described it as follows:

> Property B: "Reviewer *Rev* does not learn whether a new version of article *Art* was submitted, even more, *Rev* does not learn how often a new version of article *Art* was submitted."

We became the feeling that it was difficult to choose the "right" BSP that adequately characterizes the intuitive security requirement. In particular, we were not completely convinced whether we had selected the right BSP. Moreover, we wanted to apply an approach for determining adequate BSPs without having to try different BSPs (respectively combinations) until finding one that seems to fit. Such an approach is described in the following section.

Reviewers must not learn about resubmissions – second approach

We followed the following methodology:

1. Step 1: Determine one or more closure properties that we believe to capture the informal security requirement.
2. Step 2: Determine for each of the closure properties which informal intuition it captures. In particular, discriminate the different closure properties with the informal intuitions.
3. Step 3: Select the closure property for which the informal intuition reflects the informal security requirement best. If necessary, make the informal security requirement more precise for this.
4. Step 4: Determine which combination of BSPs best captures the closure property, or, if no such combination captures it, which combination could be used as an approximation to the closure property.

We applied this approach for our property:

Step 1. A discussion resulted in the following three different closure properties:

$$\forall n \in \mathbb{N} : \forall \tau \in Tr : \mathit{length}(\tau|_C) = n \implies P,$$

where P is one of the following three formulas:

1. $\exists \tau' \in Tr : (\mathit{length}(\tau'|_C) < n) \wedge (\tau|_V = \tau'|_V)$
2. $\exists \tau' \in Tr : (\mathit{length}(\tau'|_C) \neq n) \wedge (\tau|_V = \tau'|_V)$
3. $\forall n' \in \mathbb{N} : \exists \tau' \in Tr : (\mathit{length}(\tau'|_C) = n') \wedge (\tau|_V = \tau'|_V)$

Step 2. We determined the following informal intuitions captured by the above closure properties:

1. Reviewer *Rev* does not know whether a new version of article *Art* was submitted, but *Rev* could deduce the maximal number of resubmissions for *Art*.
2. Reviewer *Rev* does not know whether a new version of article *Art* was submitted, but *Rev* could narrow down the number of resubmissions to two possible values.
3. Reviewer *Rev* has no information about how often a new version of article *Art* was submitted.

Step 3. We agreed that the third closure property best captures the security requirement we want to specify.

Step 4. For capturing the selected closure property, we decided that we need one BSP requiring the deletion of confidential events, and one requiring the insertion of confidential events. For example, we could choose the BSPs IA and R, or the BSPs IA and D, where the admissibility condition for IA expresses that resubmit-events need only to be inserted in the submission phase of the system. Our reasoning was as follows: Given a trace with n resubmission events, R respectively D guarantee that there is a trace with 0 resubmission events, and IA then guarantees that there are traces with n' resubmission events for arbitrary n'.

The following open questions remained and were not discussed in detail due to time constraints:

- Are, for our example, IA and R equivalent to IA and D?
- Should we choose backwards-strict BSPs (BSIA and BSD) over the non-strict variants (IA and D)?

Reviewers must not learn about resubmissions – using UMLsec

As discussed previously, through the Reviewing System exercise we focused on using MAKS for the specification of the security properties. We wanted however to compare this methodology to the specification of the same security properties using the UMLsec formalism [Jür05]. Although UMLsec was not designed to deal with information flow properties, it is however interesting to see how it compares to MAKS and how one could benefit from both approaches towards reaching an industrially applicable formally based security analysis.

UMLsec background In the UMLsec profile, there are two stereotypes that allow to define information flow security properties on UML state charts, namely «no-down-flow» and «no-up-flow». They have an associated tag { high } that allows to define which methods/attributes of the class (which behavior is represented by the state chart) are considered 'high'. All methods/attributes of the class which do not have the associated tag { high } are considered as associated with the tag { low }.

Intuitively, a UMLsec state-chart is a message-transforming function that maps sequences of input messages into sequences of output messages. Input messages are just calls to the interface methods with concrete parameters (if any) and output messages are the actions triggered by the input messages including possibly the return values of methods.

The security requirement of «no-down-flow» is that there exists no pair of input sequences that is equal after deleting the 'high' messages and such that the two outputs defer (also after purging the high messages). This semantics are inspired in the classical non-interference definition (Goguen/Meseguer, [GM82]) between two groups of users. Indeed, the classical definition requires that a group of users (for example the low users) sees the same output if another group (high) executes commands or not. Although very similar in nature and perhaps even equivalent, one natural question that arose in our discussion of the UMLsec definition was:

Are the UMLsec non-interference stereotypes formally comparable to the Goguen/Meseguer definition?

This is an interesting point to follow up, since to the best of our knowledge there is no such formal comparison in the UMLsec related bibliography.

UMLsec vs. MAKS The requirement 'Reviewers must not learn about resubmissions' involves only two user groups, that is the reviewers and a submitter. Therefore it is quite natural to try to map this to the *high* and *low* user levels allowed by the UMLsec definition. Given this and an explicit state chart representing the behavior of the reviewing system it is in principle possible to decide whether it respects the non-interference stereotype [5] .

On the one hand, one obvious question is: Is «no-down-flow» comparable to any of the BSPs? If so, to which one? Given its 'pruning nature' it is very tempting to say that UMLsec is very similar to the 'R' BSP of MAKS. It will be indeed interesting work to try to establish this formally.

Now, if we can formally map the semantics of UMLsec to some BSP of MAKS, can we extend UMLsec so that we can use all of the BSPs ? In principle this sounds feasible but it would need a deeper comparison of the underlying trace models. For example it was not very clear to the group if the use of concrete Integer values in the method parameters as in UMLsec is somehow incompatible with the MAKS definitions.

Towards verification Although we wanted to focus on the specification of security properties and to compare different formally based techniques, another important distinction point between UMLsec and MAKS must be made. Indeed, MAKS allows a rich variety of subtle security definitions, whereas as discussed before UMLsec is not so advanced with respect to information flow properties. The strength of UMLsec resides however in its philosophy to combine the easiness of system specification with automatic verification of security properties. Traditionally, the kind of properties verifiable in UMLsec (and partially also tool supported) are not information flow properties. It is an interesting research path however, to investigate how could one automatically verify some information flow properties given an explicit system construction (i.e a state chart). Here one could benefit from the existing verification results (for example the very general unwinding techniques) and compositionality results of MAKS.

To experiment with the verification techniques, we could benefit from a sketch of the system behavior we have depicted for the reviewing system by means of a UML state chart (compare Figure 1).

2.2 Security properties in a reviewing system (Group 2)

Group 2[6] focused on an approach how to obtain a formal specification from an informal requirement. The formalism taken into account by this group were UMLSec [Jür05] and the Modular Assembly Kit for Security [Man00, Man03]. One focus of the work was to try a systematic approach towards formalization of the properties.

2.2.1 Informal Security Requirement

We focus on two of the properties listed at the beginning of Sect. 2:

- Reviews are anonymous.
- Reviewers are not influenced by other reviewers.

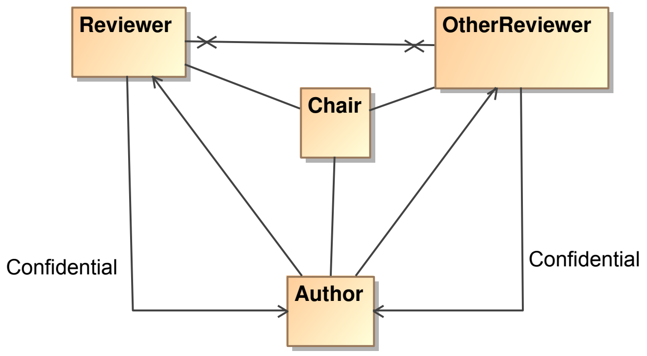
The idea is that in this version of the reviewing system the reviewing phase is distinct from a discussion phase. Only in the discussion phase a reviewer may see other reviews.

[5] Although there is not an explicit verification strategy in UMLsec, just the definition of the verification goal.

[6] Peter Fischer, Sebastian Pape, Barbara Sprick, Kurt Stenzel

2.2.2 1. Step: Security Policy

In a first step the security policy below is defined. Technically, this is a UML class diagram. Unfortunately, UML does not provide the types of arrows used for MAKS security policies. Therefore, the following convention is used: Dashed arrows denote N flow (not visible, not confidential), lines without any arrows denote visible flow in both directions, lines with two crosses at each end denote confidentiality in both directions, simple arrows named "confidential" denote confidentiality in this direction, and a simple arrow without name denotes visible flow in the indicated direction. (The advantage of using UML – instead of xfig or inkscape – for the policy is that the policy can be used by a tool for something useful, e.g. input for a prover.)



To simplify the model a distinction between *reviewer* and *OtherReviewer* is made. The idea is to express properties like *Reviewer does not know if Other-Reviewer has already submitted a review or not.*.

2.2.3 Step 2: Identifying Events

We identify the following events and assign them to security domains:

Event	**Assignment**
assign(Author, Reviewer)	Reviewer
assignOther(Author, OtherReviewer)	OtherReviewer
review(Author, Reviewer)	Reviewer
reviewOther(Author, OtherReviewer)	OtherReviewer
submit(Author)	Author
endSubmission	Chair
endReview	Chair

We consider only two phases of the reviewing process, the submission phase where authors can *submit* papers, and the reviewing phase. The submission phase ends with the *endSubmission* event and the reviewing phase ends with the *endReview* event. Both can be initiated by the program chair. In the reviewing phase a reviewer must first be *assigned* to a paper written by an author before she can *review* the author's paper. The assignment of events to security domains leads to the following views:

Event	Reviewer	OtherReviewer	Author	Chair
assign	V	C	C	V
assignOther	C	V	C	V
review	V	C	C	V
reviewOther	C	V	C	V
submit	V	V	V	V
endSubmission	V	V	V	V
endReview	V	V	V	V

2.2.4 Step 3: Select BSPs

For the property

Reviewer does not know if OtherReviewer has already submitted a review or not.

the BSPs IA + D seem appropriate at first for the *Reviewer* view:

$D = \forall \alpha, \beta \in E^* \forall c \in C : \beta.\langle c\rangle.\alpha \in Tr \wedge \alpha|_c = \langle\rangle \Rightarrow \beta.\alpha \in Tr$

$IA = \forall \alpha, \beta \in E^* \forall c \in C : \beta.\alpha \in Tr \wedge \alpha|_c = \langle\rangle \wedge Adm^\rho(Tr, \beta, c) \Rightarrow \beta.\langle c\rangle.\alpha \in Tr$

with $\rho = \{submit, endSubmission, endReview\}$ and

$Adm^\rho(Tr, \beta, e) = \exists \gamma \in E^* : \gamma.\langle e\rangle \in Tr \wedge \gamma|_\rho = \beta|_\rho$

Admissibility is chosen because we have a system in mind where first only submissions by authors occur until *endSubmission*. Then reviewers are assigned and do reviews until *endReview*. This means a reviewer knows that no other reviews have been done before *endSubmission* occurs. So we have to modify our informal property to

In the review phase Reviewer does not know if OtherReviewer has already submitted a review or not.

Unfortunately, the formal specification IA + D includes *all* confidential events, i.e. also *assignOther*. This means the specification translated back to informal language means

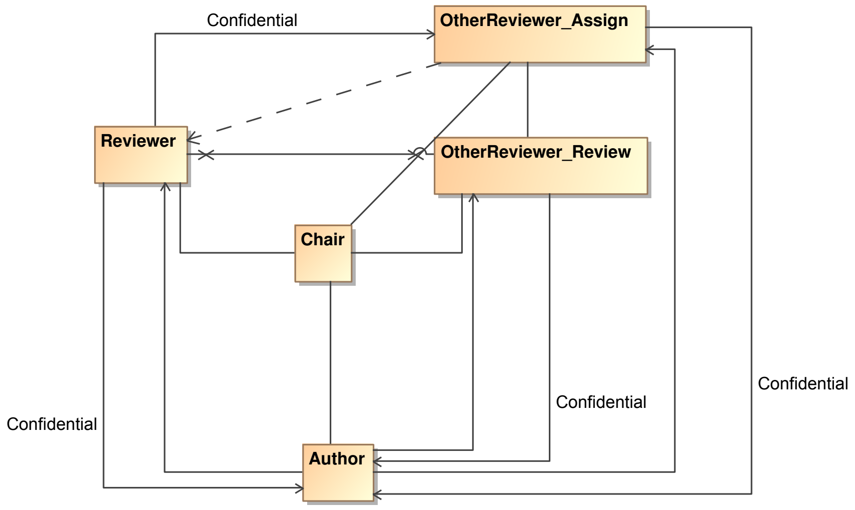
In the review phase Reviewer does not know if OtherReviewer has already submitted a review or not or was assigned to a paper or not.

Choosing IA + D with the same ρ for the Author view translates back to

Between endSubmission and endReview an author neither knows if (and what) reviewers were assigned, nor if (and what) reviews have been submitted.

2.2.5 Step 4: Modify Security Policy

The problem is that the security policy was defined with all security properties in mind (all security properties under consideration). The result was a formal property that was stronger than the informal one; actually it was a conjunction of two informal ones. To formalize each property precisely different security policies are needed. The policy for *In the review phase Reviewer does not know if OtherReviewer has already submitted a review or not.* is shown below. Here the security domain for *OtherReviewer* is divided into two, one for the confidential *reviewOther* event, and one for the not visible, not confidential *assignOther* event.



2.2.6 Ideas for a Systematic Approach

We start with a set of informal security requirements that we want to formalize. One step will be to decide what formalism is appropriate for each requirement. For information flow this means to identify those requirements that will be formalized with sets of traces. Others can be formalized for example with one trace, or for one state.

This may require a reformulation of requirements, or a division of one requirement into several. The relation between original and modified requirements should be documented.

We see three approaches to a formalization of the requirements (in our case the information flow requirements to be formalized with MAKS): individual, accumulated, or incremental. The three approaches are described in turn.

Individual. The idea is to treat each requirement as if it was the only property to formalize. Often, a security property describes interests or restrictions of one stakeholder. In this case a security domain for the stakeholder must be defined, and also for others. After identifying the relevant events, a view (possibly several) can be defined, and BSPs selected. This is done for every requirement.

The result will be a set of specifications with different events, views, and security domains. An actual system that must fulfill all requirements will have only one set of events. This means that in a second phase the different specifications must be aligned in some sense. One idea could be to take the union of all events, and modify all specifications:

- New Events are the union of all events.
- The views must be extended for the new events. Probably they will be either visible or not visible, but not confidential.

- The BSPs remain unchanged.

It must be checked if the new specification still is an appropriate formalization of the informal requirement.

The resulting specifications will have the same events, but probably conflicting views for one security domain. (An event can be visible for one property, but must be confidential for another.) However, this seems to be no problem.

Accumulated. Here, all requirements are considered at once. A security policy is defined that may contain several security domains for each stakeholder. All events are defined and mapped to domains. Then BSPs are selected for the views, and the result is the formalization of all requirements at once.

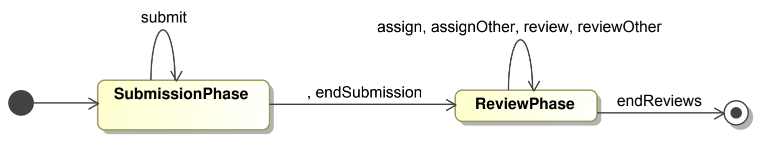
As the example in the previous section shows it is no longer possible to specify each property individually. Rather a set of BSPs for one view may formalize a conjunction of several original properties, or a stronger property than an original one. The formalization should be translated back into informal language, and the relation to the original properties should again be documented (and checked to be appropriate).

In contrast to the individual approach there is only one view for each security domain. An actual system again must fulfill all BSPs.

Incremental. Perhaps it is possible to start with one property, then consider a second property, and modify the events, security domains, etc. to have a specification for both properties. In the end the result is similar to the accumulated approach.

Perhaps it is possible to further divide the properties into groups, and treat each groups individually with an accumulated approach.

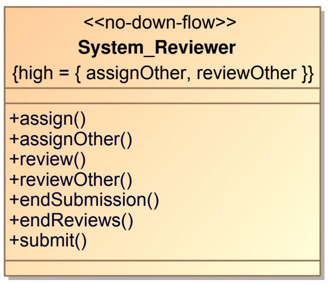
Sanity check. It is easy to make the restrictions on information flow too strong. Usually, one has some kind of system model in mind that could serve as an abstract specification of the system. In our reviewing system we have in mind a submission phase (where only submissions occur), then a review phase (with review assignments and actual reviews).

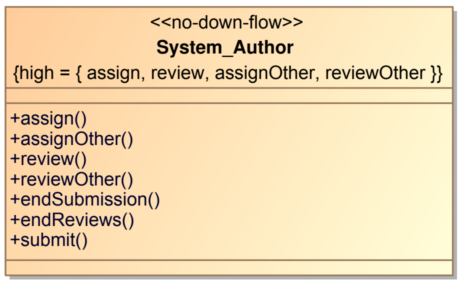


It may be a good idea to check the formalized requirements against this model. This may help to detect impossible requirements. (Example: Before the end of the submission phase every reviewer knows that no reviews have been submitted yet. The requirement concerning no knowledge about other reviews makes sense only in the review phase. This requires a BSP with admissible predicate.)

2.2.7 Formalization in UMLsec

Based on the no-down-flow example for accounts, a formalization of a security property could be class with "high" annotations but no associated state machine. (The state machine already describes an actual system.) Each security properties can be described by one class. Two examples are shown below.





Naming conventions are used to show that these are two requirements for the same system. Both classes have the same events (defined as UML operations). An alternative is to make the "high" annotation more complex by including different views, e.g.

$$\begin{aligned}high = \{&View(Reviewer) = \{assignOther, reviewOther\},\\ &View(Author) = \{...\}\}.\end{aligned}$$

Later on each class can be associated with the same state machine (at least conceptually, it may not be possible in UML), and UMLsec can be used to check each system.

2.2.8 Further Ideas

An integrity property may be:

> The chair is not influenced by the (number of) submissions in his decision to end the submission phase.

To formalize this property means that the *submission* event is confidential for the *chair*. Indeed, if the chair has no knowledge about submissions he cannot be influenced in his decision by them.

3 Formalizing Security Properties for Mobile Applications

During the follow-up meeting, the considered scenario was that of applications running on mobile devices. On the first day, there was a common session with all participants addressing the general methodology of the ongoing considerations regarding the informal definition of information flow and the scenarios resulting out of it. On the second day, there were two groups. One group used requirements from this scenario to take a closer look at the particularities in the treatment of events that are neither confidential nor observable. The other group considered the specification of integrity properties at the example of a mobile devices scenario.

Since both groups took a different focus, the concrete scenario and requirements considered will be introduced in the sections of the respective groups.

3.1 Common Session on the General Methodology

We still considered the scenario of applications (apps) running on mobile devices. We first discussed which kind of communication between mobile devices we need to address resulting in informal definitions of information flow scenarios.

3.1.1 Step 1: Finding Informal Definitions of Information Flow Scenarios

Possible differences in scenarios regarding different demands on information flow properties are the way the apps communicate with each other. There are at least three different ways to distinguish the communication channels to regards:

- Apps do not only communicate via open channels. Rather, they may also use covered channels. The resulting requirement would be that apps do not influence each other in any way. This would in particular demand the underlying hardware and operating system to meet certain requirements. Especially, if the resources available to the apps – such as cpu power or used memory – are also considered and not allowed to be a source of influence.
- Apps do only communicate via open channels. In this case the requirements need to be instantiated for particular applications.
- Apps do neither communicate via open channels nor do they communicate via covered channels. If the considered apps do not communicate, the requirement is trivially fulfilled.

3.1.2 Step 2: Refining the Requirements to Comply with the Chosen Setting

The three described ways are already ordered in decreasing complexity. Since the case that apps do not communicate with each other already fulfills the requirement, the most natural way is to consider the easiest of the remaining cases: Apps do only communicate via open channels.

Some instantiations of real live examples for this case are:

1. Messages sent to Google Maps App may not influence to whom the Google Maps App is sending GPS coordinates.
2. Google Maps App is not allowed to transmit any GPS data outside the mobile device.
3. An app is allowed to update online. Other apps may neither initiate nor prohibit the update.
4. As item 3, but other apps may additionally not detect if an update was performed.
5. An app has special permissions to delete files on the mobile device's SD card. Other apps may not influence this app to actually delete files.
6. As item 5, but other apps may cause the app to delete files – after the user's explicit confirmation.

3.1.3 Step 3: Finding Adequate Meanings of MAKS-BSPs

For this step we need to use particular examples of the instantiations. We chose the examples 3 and 4 from Sect. 3.1.2 since they are on the one hand simple and on the other hand descriptive. Therefore, they are appropriate to serve as demonstration.

We distinguish between App1, the app (possibly) performing the update and App2, the adversary trying to observe, initiate or prohibit App1's update. For each BSP there are the following meanings.

R:
- Independently of App2's observations, App2 may not conclude that App1 did not perform an update.
- App2 may not conclude from its observations that App1 performed one or more updates.

SD:
- Independently of App2's observations, App2 may not conclude, how many updates App1 performed[7].
- App2's observations do not help App2 learning how many updates App1 performed. But App2 may get knowledge on the number of updates from the system's specification.
- App2 may conclude that App1 performed at most n updates, but App2 may not conclude how many of the first n updates were really performed by App1.

SI:
- Independently of App2's observations, App2 may not conclude that App1 has not performed an update.
- Independently of App2's observations, App2 may not exclude that App1 performed an update.
- Independently of App2's observations, App2 may not exclude that App1 performed n updates (for arbitrary n).
- Independently of App2's observations, App2 may not exclude that after the updates App1 actually performed, App1 performed further updates.
- App2 may possibly conclude, that App1 performed n updates, but App2 may not conclude if App1 performed further update afterwards.

D / BSD:
- Analogous to item SD: App2 may possibly conclude, that App1 performed at most n updates, but App2 may not conclude how many of the first n updates were really performed by App1.

However, these intuitive statements are not the full truth, several observations, comments and questions arise:

Observation on SD and SI The referred BSPs describe more than we specified as intuitive statement. That means, if the BSPs SD and SI are fulfilled, they make a statement that App2 is not able to learn something on the number of updates in a certain sense. But this statement could also be given with statements which do not delete or insert events at the end of the trace but on arbitrary positions in the trace.

[7]Neglecting the fact that the maximum number of updates may be learned from knowing the set of possible traces.

Comment It may be useful to also introduce some N-events for the comparison of the BSPs.

Upcoming questions

- What happens if N-events are only sometimes seen? For example due to different routes of packages through the network.
- Given the events "request update" and "send new data", do they both need to be confidential? Or only one of them? Do we need to find a more precise informal requirement for this? Variants:
  - Request for update is confidential
  - Receipt of the update is confidential
  - Request for and receipt of the update are confidential
- Is it possible at all to keep the receipt of the update confidential or is it only possible to keep the installation of the update confidential? Do we need a further event "install update"?
- Is non-determinism dissolved at implementation time? Is non-determinism dissolved at running-time?
- Some non-determinism may support security, some may not. How can we distinguish them?
- General question regarding SD: Why is the *last* C-Event deleted and not an arbitrary one?

3.2 Corrections in Basic Security Properties (Group 1)

In this group[8] on the intuition of MAKS-BSPs D, SD, BSD and I, SI, BSI. In particular, we wanted to explore when each of the three variants (Non-strict, strict, backwards-strict) is sensible.

Intuitively, using non-strict or backwards-strict BSPs permits to explore flexibility provided by the events in the set N, i.e., events that are neither visible to the attacker nor confidential. Such BSPs do not require that deleting / inserting confidential events into a trace results in another trace, but that it should results in another trace that agrees on confidential and visible events.

An example in [Man03] illustrates that exploiting this flexibility might not be adequate in the presence of Trojan horses (Example 3.4.22).

Inspired by this example, we investigated the adequacy of D, SD, and BSD for a scenario with the following attacker model:

- The attacker has successfully placed a Trojan horse with access to secret information. The Trojan horse provides "high" inputs to a system in order to transmit this information to the attacker.
- The attacker observes the system over which the Trojan horse attempts to transmit information. The attacker observes "low" outputs of the system. From these observations, he tries to deduce which inputs were provided by the Trojan horse.

[8]Christoph Feller, Peter Fischer, Andreas Lehmann, Martin Ochoa, Sebastian Pape, Henning Sudbrock

We observed that one can consider the Trojan horse as a refinement of the system, and that this may lead to problems using non-strict BSPs:

If a Trojan horse is influencing the system, some previously possible traces may become unfeasible. That means, if the behavior of the system is described by a set of possible traces, this set may contain traces which do not occur when the Trojan horse is influencing the system. This behavior may regarded as conducting a refinement by including the Trojan horse into the system's specification. Regarding the system's security the resulting problem is that BSPs are not preserved under refinement. The intuition of the Trojan horse's consequence is as follows: A BSP assures that for each observed trace there exist at least one more trace with different confidential events which also explains the adversary's observations. If the Trojan horse prevents the other possible traces from appearing, it may draw conclusions of its observations and thus the security guarantee is lost.

This leads to the following ideas for future work:

Which conditions need to be demanded for refinements of the set of possible traces to preserve the validity of the particular BSPs? Which meanings do these requirements have regarding the power of the Trojan horse?

Or, to turn the question around: For which refinements/Trojan horses are the BSPs D/SD/BSD suitable? Are there any requirements regarding the set of (possible) traces itself?

So when is D, BSD, respectively BSD adequate? The following describes what we believe could be the intuition:

- D is suitable, if Trojan horses are not regarded by the security requirements.
- BSD is suitable, if only Trojan horses are regarded, which act at a certain point of time only depending on already occurred events.
- SD is suitable, if only Trojan horses are regarded, which act at a certain point of time only depending on already occurred or future events.

Claim 1: Since there are other Trojan horses (for an example see below), for some of them neither SD nor BSD nor D are suitable. Claim 2: The considerations above should work analogous for the BSPs I, BSI and SI.

Claim 1 and Claim 2 need to be verified.

Example for which BSD is not adequate:

- Set of Traces: ((N(x),V(x)) — (C(x),V(x)))*, x is a number
- N(x) happens if for a certain time no C(x)-Event has happened (timeout)
- The Trojan horse assures the timeout does not occur by initiating C(x)-Events
- BSP BSD is fulfilled: If a C(x)-Event is deleted, a N(x)-Event may be inserted instead

Example for which BSD is not adequate:

- Set of Traces: (V(x) — (C(x),V(x)))*, x is a number
- V(x) happens if for a certain time no C(x)-Event has happened (timeout)
- The Trojan horse assures the timeout does not occur by initiating C(x)-Events
- BSP SD is fulfilled: If a C(x)-Event is deleted, the trace is still valid

Our conclusion: Three possible aspects one could use when deciding which BSP(s) to use:

1. As in [Man03]: Does one want to keep the existence or non-existence of confidential events secret? $\Rightarrow$ Choose either Insertion or Removal/Deletion-BSP
2. Does one want to keep the existence or the number of occurred confidential events secret? $\Rightarrow$
   - Choose BSP R to keep the existence of a C-Event secret.
   - Choose BSP D, SD or BSD to allow the adversary learning that at most n C-Events occurred, but to keep the exact number which of those n events at least occurred secret.
     Caution: BSP D, SD and BSD do not assure that the order of occurred/occurring C-Events is kept secret.
   - Choose BSP I, SI, BSI, IA, SIA or BSIA to allow the adversary learning that at least n C-Events occurred, but to keep the information secret if after these n C-Events any other C-Event(s) happened.
     Note: For I-Variants this require that an infinite number of C-Events may occur at arbitrary positions.
   - Choose combinations of the BSPs I and D to keep the number of occurred C-Events secret.
3. Does one want to regard systems where it can be neglected that Trojan horses want to communicate via the system, or do Trojan horses need to be regarded?[9]
   - Choose D/R/I/IA-BSP if no Trojan horse needs to be considered.
   - Choose BSD/BSI/BSIA-BSP [10], if only Trojan horses should be regarded, who act only depending on the previous system's behavior.
   - If arbitrary Trojan horses should be regarded, there seems to be no suitable BSP(s).

3.3 Formalizing Integrity with MAKS (Group 2)

In this group[11] we considered the following scenario: Two applications App1 and App2 are running in parallel on the same device. Both applications can trigger a communication with the other app. Both applications have some internal behavior that is independent of the other app. Furthermore, app 1 can request and receive updates from internet and then install them.

The integrity property we looked at is as follows:

App2 can neither trigger nor prohibit an update/install of App1.

Integrity is considered dual (complementary, inverse) to confidentiality. Hence, we assume that a formalization can invert the confidentiality formalization.

This means we define a view for App1 (instead of App2) with the following classification of events:

[9]Note: This intuitions are more vague than those before.

[10]alternatively one of the SD/SI/SIA-BSPs may be chosen. That would be more restrictive. Maybe the BSP SR is also interesting. Regarding the BSP BSR it is not clear what is meant by "backwards strict"

[11]Jan Jürjens, Kuzman Katkalov, Stefan Pfeiffer, Thomas Ruhroth, Barbara Sprick, Kurt Stenzel

“corrupting”	C =	{ communicateAB, communicateBA, computeB }
	N =	{ request, send, computeA }
“veritable”	V =	{ install }

Since we are talking about integrity, C should be seen as meaning *corrupting* instead of *confidential* and V as *veritable* instead of *visible*. Intuitively, the occurrence or non-occurrence of C events cannot influence V events which is the integrity property.

Does the classification above make sense? Yes. *install* is the event to protect. *communicateBA* is an event under the control of App2, and a message from App2 to App1 should not influence *install*, i.e. it is in C. The same holds for *computeB*. Depending on the assumptions about the execution environment this may be trivially true, but putting it in C says that it should not influence *install*, independent of the execution environment. *communicateAB* in C may seem strange at first since the event is controlled by App1. However, it also could be a way to influence *install*, e.g. by an answer from C, or in case of a synchronous communication, by blocking. In case of an asynchronous communication *communicateAB* could be in N. The N events *request* and *send* obviously are necessary for *install*, i.e. influence *install*, and *computeA* probably will also influence *install*. To summarize, the view captures the intuition of the integrity property.

What BSPs are adequate?

- SR is too weak. Example1:

  { [], [communicateBA], [install] }

  fulfills SR, but if App2 starts with *communicateBA*, no *install* happens, i.e. App2 prohibits the update (actually blocks App1 completely).

- IA is too weak. Example2:

  { [], [communicateBA], [communicateBA.install] }

  IA is fulfilled with $\rho(View) = E$, but here App2 triggers *install* with *communicateBA*.

- Perhaps D + IA (or BSD + BSIA) are adequate.

We also considered separability, i.e. arbitrary interleaving of V-traces and C-traces. This property seems to be too strong:

- Example3:

  { [], [install], [install.communicateAB] }

  This system is secure since corrupting events happen only *after* the *install*. However, it does not fulfill separability. On the other hand, IA is fulfilled.

The question remains if a combination like BSD + BSIA is also too strong, i.e. excludes systems that are (intuitively) secure concerning the integrity property.

Nevertheless, integrity properties can be formalized in a systematic manner similar to confidentiality properties.

4 Lessons learned and Conclusion

The workshop was very successful. Participants from different projects with different fields of expertise worked together productively on several problems and aspects of formalizing security properties. Besides learning something the joint work has led to a common understanding and ideas for future collaboration. Some of the lessons learned are:

- Informal security requirements usually cannot be formalized directly (or exactly) because they are too imprecise or because they encompass several properties at once.
- Formalizing a requirement with an information flow property requires the definition (and assignment) of security domains. Here is a choice: If the requirement should be captured exactly then different (assignments of) security domains may be necessary for different requirements. If the aim is to have only one definition (and assignment) of security domains then the formalization will capture several individual requirements.
- Formalizing a requirement with an information flow property requires the assignment of security domains to all events/operations. Often the requirement is only concerned with one event. Then it is not clear how to assign the other events.
- It is very important to translate a formal property back into informal language to validate whether the original requirement was captured adequately.
  This can lead to an iterative process where the informal requirement and the formalization are refined so that they 'fit' together.
- It is easier to formalize a security requirement for an existing system than to formalize it before the system exists. On the one hand some part of the system must be mentioned in the specification (e.g. events), on the other hand the formalization should not be too restrictive for real systems. For example, in a real system it may be acceptable that a property holds only during a certain application life cycle phase.
- Integrity can be formalized with information flow properties, but requires a different thinking than confidentiality.

Acknowledgments. The topic workshop was sponsored by the priority programme 1496 "Reliably Secure Software Systems" (RS^3) of the Deutsche Forschungsgemeinschaft DFG, and the authors of this technical report by the DFG projects FM-SecEng in the Computer Science Action program (MA 3326/1-2), IFlow in the priority programme RS^3 (RE 828/15-1), and MoDelSec in the priority programme RS^3 (JU 2734/1-1).

A A Collection of Security Properties

This section presents a first version of a collection of security properties for different application areas.

It is not intended to be comprehensive or structured (the properties sometimes contradict each other), but may serve as a starting point from a software engineering point of view.

The areas are

1. Smart phone apps
2. Separation kernels, esp. in
    (a) Automotive (different control circuits on one bus)
    (b) Avionics (different safety-critical applications on one chip)
3. Multi-applicative smart cards
4. Conference reviewing system
5. E-Voting
6. Web server applications and browsers
7. Intrusion detection and system administration
8. Enterprise systems
9. Cloud computing
10. Travel planner (as a concrete smart phone app)

A.1 Smart Phone Apps

1. I (the user/owner of the smart phone) know to what destination an App transmit data.
2. I can control how data is transmitted and/or stored.
3. My data is transmitted only to the intended recipient.
4. My private data is never transmitted.
5. An App never accesses private data.
6. Apps have no covert channels (their behavior does not depend on private data).
7. Apps never influence each other or only in controlled ways.
8. No data about App usage is transmitted, not even statistical or anonymized data.
9. Location data is never stored (or only in a secure manner)
10. It is not possible to track the movements of a smart phone owner.

11. Others (e.g. provider) cannot identify a smart phone owner.
12. A payment operation with the phone happens only after explicit confirmation.
13. I control what data is exchanged in 0-click apps.
14. If my contact data is transmitted to another phone it is not disseminated further.
15. The data received by an app (or network) provider is not disseminated further.
16. The smart phone cannot be used by other persons.

A.2 Separation Kernels

1. Car: The infotainment system does not interfere with brakes, air bag, etc.
2. Car: It is not possible to interfere with the internal circuits from outside the car (e.g. opening the doors by manipulating the circuitry of the external mirror).
3. Car: The odometer cannot be manipulated.
4. Car: logged data cannot be manipulated, and can be accessed only by authorized entities.
5. Car: The sequence (order) of events has no influence on functionality (example with a truck: entering, turning radio on/off, leaving by the other door blocks the starter)
6. Avionics: cabin events do not interfere with flight systems (e.g. switching the reading light on does not shut down the auto-pilot).
7. Avionics: Several systems on one chip behave as if each system had its own chip.
8. Avionics: Systems never influence each other except in well-defined situations (take-off or landing)
9. The order of events has no influence on system behavior.

A.3 Multi-applicative Smart Cards

1. Apps can exchange data on in a controlled manner.
2. Dynamic (i.e. after the smart card was issued) installation of new apps does not influence already installed apps.
3. It must be possible that a dynamically installed app communicates with an existing app (e.g. a payment app).
4. A smart card and its environment (e.g. phone and SIM) interact only in a controlled way.

A.4 Conference Reviewing Systems

1. Only PC members can see all submissions.
2. Only after submitting a review can a reviewer see other reviews.
3. A reviewer does not know if other have already submitted reviews.
4. Reviewers are not influenced by other reviewers.
5. Persons with multiple roles have restricted privileges (e.g. a PC member that also submits a paper can see all reviews except for her own paper during the review phase).
6. Reviewer are anonymous (to authors and possibly each other).
7. In a double blind review process the reviewers do not know the identities of the authors.
8. only accepted papers are made public, and only after the end of the review phase, but statistics about submissions are published.
9. Reviews are never deleted, all remain accessible. A user cannot cause a review to become invisible.
10. A reviewer can only submit reviews to papers she was assigned for.
11. A reviewer may not write (or view) reviews for papers with a conflict of interest.
12. Authors are anonymous until after bidding.
13. An author (paper submitter) cannot see other submissions.

A.5 E-Voting

1. It is (and remains) secret if and what I have voted.
2. Every vote is counted correctly.
3. I can verify if my vote was counted correctly.
4. It is not possible to prove to others what I have voted.
5. It is possible to vote, but/and only eligible persons can vote.
6. Everybody can vote at most once.
7. The final tally is secret until the end, and no intermediate results become known.
8. The final tally corresponds to the actual votes.
9. Different actors may have only certain (what?) knowledge.

A.6 Web Server Applications and Browsers

1. Entering data on the server does not influence the behavior of other clients (XSS, cross site scripting).
2. CSFR (cross site request forgery).
3. User input causes only the intended modifications and results to a data base (SQL injection).
4. Input channels can only be used for the intended information.
5. A click means what the user expects (click highjacking).
6. Same-origin policy.
7. Pages in different tabs are independent from each other.

A.7 Intrusion Detection and System Administration

1. A user has no knowledge about the IDS (whether it runs or not, how it is configured, etc.).
2. A user cannot find out if the account of another user is disabled.
3. An intruder does not notice that he was detected.
4. An intruder cannot influence the IDS in his favor (e.g. disable it).
5. A user cannot learn if another user is logged in or not, and learns nothing about her activities.

A.8 Enterprise Systems

1. Access to single or aggregated data is allowed, but not direct access to all data.
2. It is allowed to access data, but not to pass it on.
3. It is not possible for a user to give himself the permission for an action (if somebody else should do it).
4. Separation of duty, 4 eye principle.
5. Chinese wall: After access to certain data the access to certain other data is prohibited.
6. A backup cannot interfere with running processes (e.g. an emergency system in a hospital).
7. A service center working with data from different sources cannot link the data (e.g. hospital bill and insurance for one person).
8. Human resources department is separated from payment system is separated from illness days.
9. Elena (e.g. a future employer cannot access my current payment)
10. Separation of private and business data (e.g. Bahncard).

A.9 Cloud Computing

1. Two users cannot gain knowledge about each other and cannot interfere with each other.
2. The provider cannot access user data.
3. It is possible to really delete data.
4. Information that was deleted by the user do not influence future actions (this includes backups, version control systems etc.).
5. Access to data by employees is logged.
6. Data must remain within certain bounds (or limits, because of legal reasons).

A.10 Travel Planner: General Requirements

The travel planner is a mobile App that is used as a running example in the IFlow project. The user selects an appointment from her calendar App and wants to book a flight and a hotel. The travel planner App communicates with a travel agency to obtain flight and hotel offers. Booking is done directly with an airline or hotel using the credit card data stored in another App on the smart phone. The travel agency gets a commission for a booking from the airline or hotel.

1. A booking takes place only after explicit confirmation.
2. Credit card data is used only for the booking.
3. The travel agency does not know the booking data, has not knowledge about a booking (in a version without commission).
4. The travel agency does not know the identity of the user.
5. Only the minimal necessary data of the selected event is transmitted, and only to the travel agency.
6. Booking data is transmitted only to the selected airline/hotel.
7. No other data is transmitted.
8. Only the travel details are stored in the calendar, nothing else is modified.
9. The credit card data is never modified (a malicious agent cannot influence the occurrence of the event "modify credit card data").

A.11 Travel Planner: Specific Requirements

1. Private calendar entries do not leave the calendar app and can be seen (or modified) only by the user. (Other agents cannot be the cause that entries are seen/modified.)

2. Public calendar entries may flow to the travel planner app if the app was started by the user that also made the entry. The skeleton data of the entry influences future queries. Only the skeleton data may leave the phone.

3. The TravelOptions entered by the user are incorporated in the FlightRequest and the HotelRequest. The policy for this user data must be weakened to the policy of the requests.

4. The FlightRequest is visible to the airline. The travel agency transmits the request. (Can the travel agency read the request?)

5. The filtered FlightOffers are destined for the user/travel planner instance. The travel agency transmits the data to the recipient. The airline does not know the user.

6. The travel agency does not know which (or if a) FlightOffer was selected. The FlightOffer may flow only to the intended airline. However, the contained BookingInformation must also flow to the travel agency. (This requires different policies in one message.)

7. The travel agency cannot link a BookingInformation with a FlightRequest. (Assuming that linking by timing is not possible because many requests are submitted in a short period of time.) Or the travel agency cannot read a request at all.

8. Items 4–7 similarly hold for the hotel booking.

9. Credit card data may leave the credit card app only after confirmation by the user, and may flow only to the airline/hotel of the booked flight/hotel.

References

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 11–20, Oakland, CA, USA, 1982. IEEE Computer Society.

[Jür05] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[Man00] H. Mantel. Possibilistic Definitions of Security – An Assembly Kit. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, UK, 2000.

[Man03] H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Secure Information Flow.* PhD thesis, Saarland University, Saarbrücken, Germany, 2003.