

Identifying and resolving least privilege violations in software architectures

*Koen Buyens
Bart De Win
Wouter Joosen*

Report CW 532, December 2008



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Identifying and resolving least privilege violations in software architectures

Koen Buyens

Bart De Win

Wouter Joosen

Report CW 532, December 2008

Department of Computer Science, K.U.Leuven

Abstract

Security principles, like least privilege, are among the resources in the security body of knowledge that survived the test of time. The implementation of these principles in a software architecture is difficult, as there are no systematic rules on how to apply them in practice. As a result, they are often neglected, which lowers the overall security level of the software system and increases the cost necessary to fix this later in the development life-cycle.

This report improves the support for least privilege in software architectures by (i) defining the foundations to identify potential violations of the principle herein and (ii) eliciting architectural transformations that positively impact the security properties of the architecture, while preserving the semantics thereof. These results have been implemented and validated in a number of case studies.

Identifying and resolving least privilege violations in software architectures

Koen Buyens and Bart De Win and Wouter Joosen

December 19, 2008

Abstract

Security principles, like least privilege, are among the resources in the security body of knowledge that survived the test of time. The implementation of these principles in a software architecture is difficult, as there are no systematic rules on how to apply them in practice. As a result, they are often neglected, which lowers the overall security level of the software system and increases the cost necessary to fix this later in the development lifecycle.

This report improves the support for least privilege in software architectures by (i) defining the foundations to identify potential violations of the principle herein and (ii) eliciting architectural transformations that positively impact the security properties of the architecture, while preserving the semantics thereof. These results have been implemented and validated in a number of case studies.

Contents

1	Introduction and motivation	4
2	Motivating Example	5
3	Theoretical underpinnings	7
4	Identifying LP Violations	9
5	Resolving LP Violations	11
5.1	Solution spectrum	11
5.2	Splitting components	16
5.3	Splitting tasks	17
5.4	Splitting permissions	20
5.5	Removing unused actions from a component	21
5.6	Removing tasks from a user	24
5.7	Moving actions from a component to another	25
5.8	Rewiring the architecture	25
5.9	Moving tasks from one user to another	27
5.10	Moving actions from one permission to another	28
5.11	Dealing with multiple solutions	28
6	Measuring LP Violations	28
6.1	False positives?	30
7	Applying the results on several case studies	31
7.1	Detailed validation results	31
7.2	Broader validation results	33
8	Discussion	34
9	Related Work	34
10	Conclusion	36
A	Case study: Archstudio Chat System	38
A.1	Introduction	38
A.2	Software Architecture	38
A.3	Experiments	38
A.3.1	A first version	38
A.3.2	A second version	40
A.4	Conclusion	43
B	Case Study: Conference Management	44
B.1	Introduction	44
B.2	An agent-based architecture	45
B.2.1	Software Architecture	45
B.2.2	Experiments	49
B.3	OpenConf	49
B.3.1	Reverse Engineering the Software Architecture	50
B.3.2	Experiments	59
B.4	Conclusion	59

C Case study: publishing system	62
C.1 Introduction	62
C.2 Software Architecture	62
C.3 Experiments	64
C.4 Conclusion	68
D List of Figures, and Tables	70

1 Introduction and motivation

Security principles, like least privilege, are among the resources within the body of knowledge for secure software engineering that have survived the test of time [54]. These principles are statements of general security wisdom derived from real-world experience building systems [40]. Practicing them should reduce the likelihood of threat realization or impact if a threat is realized [56]. The principle of least privilege (LP) prescribes that a user is not assigned permissions that he does not require and, consequently, he cannot execute tasks that he is not allowed to execute.

These principles must be considered throughout the entire secure software development lifecycle. In recent times, several of such secure software development lifecycle processes have emerged from both academia and industry. Examples are the Touchpoints by McGraw [40], the SDL by Microsoft [27] and the Comprehensive, Lightweight Application Security Process (CLASP) by OWASP [56]. Those processes define security specific activities that increment traditional software processes in order to address security concerns in a sound, controlled, and more repeatable way. These secure software processes refer to the security principles and acknowledge their importance as they have specific activities that use these principles. Quite surprisingly, none of the above security processes explicitly provide guidance on how to apply these principles in practice [10]. Making this implicit knowledge explicit, e.g. in a set of rules that enforce the principle when applied, could help in developing more secure software [40].

While several techniques exist to reason about, and improve, the adherence to LP in software, no adequate support in the form of a systematic method is available at the architectural level, where the consequences are significant. Indeed, there is a research gap with respect to the principle of least privilege at architectural level. Several techniques have been developed to verify or enforce least privilege in software such as model checking, sandboxing, program separation and so forth (see Section 9 for a more in-depth discussion). However, while in theory applicable to the architectural level, most of the enforcement techniques focus on requirements specification or detailed design phases. Apart from that, a variety of testimonies witness the usefulness and the applicability of the principle at the architectural level (e.g., gmail [7], postfix [59] ...) but these testimonies fail to provide systematic rules or guidelines on how to apply the principle on a software architecture. Moreover, the consequences of not introducing the quality of security properly in the architectural phase are significant for two reasons. Not only do these quality related decisions determine the quality of the resulting software product [6], but changing or introducing them later on in the development life cycle is more expensive than introducing them in the architectural design phase [8].

This report argues that, at the architectural level, LP minimizes the capabilities of a (set of) component(s) that is to be executed as a single controllable unit (typically a process). Two important parameters in this context are (i) the controllable units and (ii) the access policy that is to be enforced: LP will be best adhered to if both the architectural structure and the policy are adequate (See Figure 1). Indeed, as shown in the example in the next section, it can be very awkward or even impossible to enforce LP with an inappropriate architecture. Unfortunately, to the authors' knowledge, no systematic techniques exist today to deal with this problem. One of the factors that makes this problem particularly challenging is the abstraction level of a software architecture and, hence, the limited amount of information available for reasoning about LP.

The approach presented in this report uses a list of use cases (sequence diagrams) and an architectural description to identify and predict LP problems in the final software product. The approach approximates a worst case assignment of permissions given the set of use cases. As represented in Figure 1 we are particularly interested in the architectural structure (without taking into account the access policy), since this ensures LP adherence in the final software system. The computed assignment is used to determine (potential) violations of the principle¹, which are then solved by architectural transformations. The contributions of the report are fourfold: (i) a theoretical underpinning of the meaning of LP in software architectures, (ii) an algorithm to identify violations of the principle for an architecture, (iii) architectural transformations to address violations, and (iv) an implementation of the former in a tool.

The rest of this report is structured as follows. Section 2 further motivates the problem by means of an

¹LP violations at architectural level are *potential* in the sense that one can never be certain about violations until the system has been fully implemented. For some problems, however, it is almost certain that they will persist in the final system.

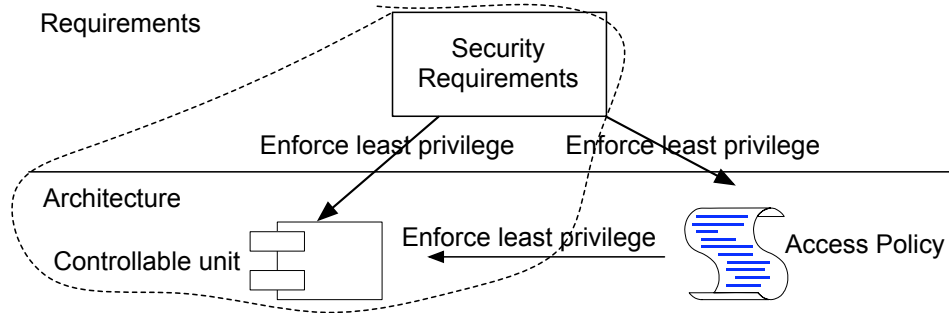


Figure 1: LP in software architecture: a combination of architectural structure and access policy.

example. Section 3 presents the formal foundations of the presented approach. Sections 4 and 6 use these foundations for identifying LP violations and measuring least privilege respectively. Section 5 presents transformations to solve a subset of these violations. In Section 7, the approach is applied on a number of case studies. Section 8 elaborates on the advantages, the drawbacks, and possible extensions of the approach. Finally, Section 9 discusses related work, and Section 10 concludes.

2 Motivating Example

Consider an integrated groupware system that consists of three main components (See Figure 2). The first, Calendar, is a component that enables a user to keep track of events. It also supports finding interesting public events to attend. The second, Repository, is a content management system that allows users to upload files and share these with other users. The third, Tasks, is a todo list and task management component that enables a user to create, update, and delete todo lists. Two components integrate the functionality of these main components and act as front-ends for end users: the Internal Groupware Client is used by employees, while the External Web Client is used by external users.

External Users use the External Web Client to (i) upload documents used and verified by these employees by means of a verification task, (ii) create public events and, (iii) confirm attendance of events organized by Internal Employees. *Internal Employees* use the Internal Groupware Client to (i) execute typical groupware related tasks such as add a task to a todo list or modify events, and to (ii) review input received from external users. Table 1 lists the tasks that have been assigned to the different users (i.e. security requirements).

Nb	User	Task (use case)
1	Employee	Upload, read, verify working document
2	Employee	Add, modify event
3	Employee	Add task
4	External User	Upload working document
5	External User	Confirm event, add public event

Table 1: Tasks assigned to the users of the calendar system

In order to enforce these requirements in our system, we need to specify an architectural-level access policy that expresses the rules in terms of component's interface methods. Several situations arise in which the current architectural structure jeopardizes or invalidates LP.

First, tasks can overlap such that permissions necessary for a set of tasks are sufficient to execute another task. For instance, if an external user has the right to *upload a working document* (perm1, perm2 for task T1), he will also be able to *add tasks* (perm2 for task T5), even though this might not be desirable. More complex overlapping scenarios that are more difficult to identify manually can arise.

Second, the granularity of rights specification in the access policy can be insufficiently fine-grained to grant the right to execute certain tasks, but not other tasks. This is particularly the case when the execution

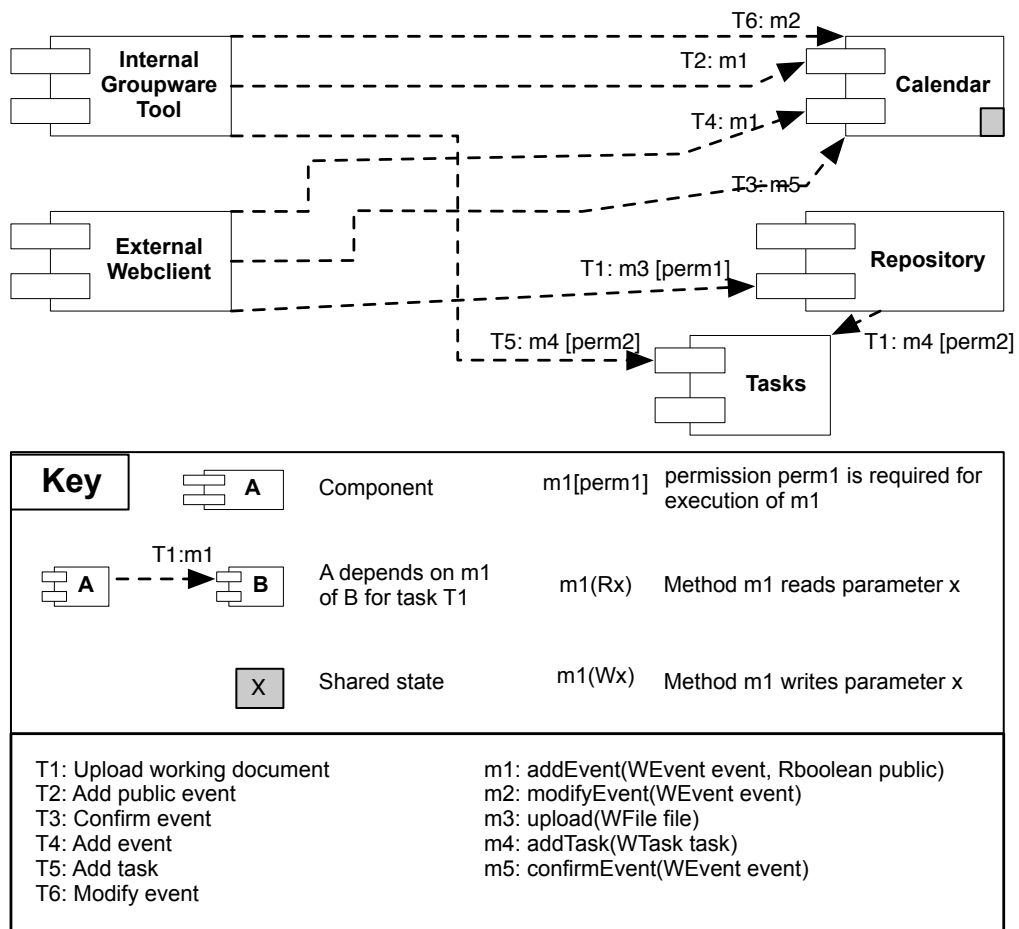


Figure 2: Excerpt from the component diagram of the an integrated calendar system

of a task is dependent on method parameters (rather than just methods), or when permissions represent a collection of methods rather than a single method. For instance, the system can not grant a user the right to *add public events* (T2), but refrain him from *add events* (T4), as the difference is represented by means of a boolean parameter of a single method. Note that this problem can be solved by increasing the granularity of the access policy, but this is rarely the case in practice.

Third, if two conflicting tasks are able to influence each other's operations, then the system might not be able to enforce LP correctly. Indeed, one can argue that influencing the operations of a task constitutes a weak form of having the permissions to execute it. In the groupware example, the *confirmEvent* method might for instance interfere with the *modifyEvent* method, since they probably make use of the same event store. Consequently, task T3 might interfere with task T6. This is not allowed, because external people might not be allowed to influence events based on the company policy.

Solving these issues properly can not be done solely by editing the access policy: a restructuring of the components is often required to address these problems.

3 Theoretical underpinnings

In this section, a formal model for software architecture is introduced to define the concept of LP (and violations thereof) at architectural level. This will allow us to identify LP problems at this level later on. The model, which was inspired by work of Jie Ren [52], is modelled in set theory. Obvious assumptions have been left out. Examples are *interfaces contain at least one action*, *there is at least one user*, and *all components are part of the software architecture*. Important assumptions are described at the end of this Section.

The model focusses on the software architecture's component and connector view [16]. Every *component* c can be described in terms of the *actions*² of its interfaces. These interfaces are used to interact with the component. Such an interaction is intentionally built into the system, often to realize a task (or use case).

$$\begin{aligned}
 SA(s) &= \langle \text{components}(s), \text{dependencies}(s), \text{parents}(s) \rangle \\
 \text{components}(s) &= \{ c_1, \dots, c_k \}; k \in \mathbb{N} \\
 c = \text{component} &= \langle \text{actions}(c), \text{interfaces}(c) \rangle \in \text{components}(s) \\
 \text{interfaces}(c) &= \{ I_1 \cup \dots \cup I_m \}; m \in \mathbb{N} \\
 \forall I \in \text{interfaces}(c): \text{actions}(I) &= \{ a_b, \dots, a_j \}; b, j \in \mathbb{N} \\
 \text{actions}(c) &= \{ a \mid a \in \text{actions}(I) \text{ and } I \in \text{interfaces}(c) \}; l \in \mathbb{N} \\
 \text{actions}(s) &= \{ a \mid c \in \text{components}(s) \text{ and } a \in \text{actions}(c) \} \\
 a &= \langle \text{name}(a), \text{params}(a) \rangle \in \text{actions}(c) \\
 \text{name}(a) &\in \{ a_1, \text{action}, a_2, \dots \} \\
 \text{params}(a) &= \{ \text{param}_1, \dots, \text{param}_n \}; n \in \mathbb{N} \\
 \text{param} &= \langle \text{name}(\text{param}), \text{type}(\text{param}) \rangle \\
 \text{type}(\text{param}) &\in \{ \text{String}, \text{int}, \dots \} \\
 \text{name}(\text{param}) &\in \{ p_1, p_2, \text{param}, \dots \} \\
 \text{pre}(a) &= \text{logical formula with params}(a) \\
 \text{post}(a) &= \text{logical formula with params}(a) \\
 \\
 \text{users}(s) = U &= \{ u_1, \dots, u_q \}; q \in \mathbb{N} \\
 \text{tasks}(s) = T &= \{ t_1, \dots, t_r \}; r \in \mathbb{N} \\
 \text{task} &= \{ (a_1, a_2), \dots, (a_p, a_{p+1}) \}^3 \\
 &\quad ; a_i \in \text{actions}(s); p, i \in \mathbb{N}; 0 \leq i \leq p \\
 UT &= \{ (u_i, t_j), \dots, (u_{i+f}, t_{j+g}) \} \subseteq U \times T; \\
 &\quad 0 \leq i, i+f \leq q, 0 \leq j+g \leq r, i, f, j, g \in \mathbb{N} \\
 \text{actions}(\text{task}) &= \{ a \mid a \in \text{actions}(s) \text{ and } x \in \text{actions}(s) \text{ and} \\
 &\quad ((x, a) \in \text{task} \text{ or } (a, x) \in \text{task}) \} \\
 \text{tasks}(c) &= \{ t \mid t \in T \text{ and } a \in \text{actions}(c) \text{ and } a \in \text{actions}(t) \} \\
 \text{users}(c) &= \{ u \mid (u, t) \in UT \text{ and } t \in \text{tasks}(c) \}
 \end{aligned}$$

We now focus on security related definitions. A component may personate multiple *subjects* (and *principals*) during its lifetime, because it may be used by different users to perform a range of (related) tasks. For now, we assume that a component represents exactly one subject.

²The term action is used instead of operation, since this fits well with the security-related part of the model.

³This is an ordered set.

A *permission* is a representation of the right to perform a set of actions (possibly a single action), – we consider components to be the resources of the system. The permissions necessary to execute a task is the union of all permissions needed for the actions that constitute the task.

$$\begin{aligned} AP(s) &= perm_1, \dots, perm_w ; w \in \mathbb{N} \\ perm &= \{ a_1, \dots, a_v \} \subseteq actions(s) ; v \in \mathbb{N} \\ permissions(action) &= \{ p \mid p \in AP(s) \text{ and} \\ &\text{action} \in p \} \\ permissions(task) &= \{ p \mid p \in permissions(action) \text{ and} \\ &\text{action} \in actions(task) \} \end{aligned}$$

A distinction was made between three types of permissions that will be used for reasoning about LP: required permissions, internal permissions, and indirect permissions.

The *required permissions* of a component are the permissions that are needed by the component for successfully completing the tasks (or parts of tasks) it is responsible for. In other words, they are the permissions associated with the actions the component depends on for the tasks it provides. A component is assumed to require permissions for all the actions in the task that follow his own contribution to the task. For instance, the External Web Client requires permissions 1 and 2 for uploading a working document (See Figure 2). Note that this set is actually equivalent to an ideal access policy.

$$\begin{aligned} reqperms(c) &= \{ p \mid p \in permissions(action) \text{ and } action \in usedactions(c) \} \\ usedactions(c) &= \{ a \mid t \in tasks(c) \text{ and} \\ &a \in actions(t) \text{ and } a \notin actions(c) \} \end{aligned}$$

The *internal permissions* of a component are the permissions linked to its own interfaces. A component is implicitly granted the permissions to execute all the actions defined in its own interfaces.

$$\begin{aligned} internperms(c) &= \{ p \mid p \in permissions(action) \text{ and } action \in ownactions(c) \} \\ ownactions(c) &= \{ a \mid t \in tasks(c) \text{ and} \\ &a \in actions(t) \text{ and } a \in actions(c) \} \end{aligned}$$

The *indirect permissions* are additional permissions that might be obtained by interfering in a component's shared state. Shared state is the internal state of a component that is used by its actions. Since an action might influence another one by changing the shared state on which the other is dependent, a task can actually influence the results of another task. See the third example in Section 2 for a concrete example. In the case of shared state, a component is attributed permissions from another component's task that is reachable via the shared state of the latter component.

$$\begin{aligned} indirectperms(c) &= \{ perm \mid c_2 \in reachablecs(c) \\ &\text{and } t \in sharedTasks(c_2, c) \text{ and} \\ &t_2 \in tasks(c_2) \setminus sharedTasks(c_2, c) \text{ and} \\ &sharedState(t, t_2, c_2) \neq \emptyset \text{ and} \\ &(perm \in internperms(c_2) \cup reqperms(c_2)) \text{ and} \\ &perm \in permissions(t_2) \} \\ reachablecs(c) &= \{ c_2 \mid c_2 \in components(s) \text{ and} \\ &t \in sharedTasks(c, c_2) \text{ and } c_2 \neq c \text{ and} \\ &\text{before}(t, c, c_2) \} \\ sharedTasks(c_1, c_2) &= \{ t \mid \\ &t \in tasks(c_1) \text{ and } t \in tasks(c_2) \} \\ \text{before}(task, c_1, c_2) &\Leftrightarrow \exists a_1, a_2 \in actions(task) \\ &\text{and } a_1 \in actions(c_1) \text{ and } a_2 \in actions(c_2) \\ &\text{and } \text{before}(a_1, a_2, task) \} \\ \text{before}(a_1, a_2, task) &\Leftrightarrow \\ &\exists (a_1, a_2) \in task \text{ or} \\ &(\exists (a_1, x) \in task \text{ and} \\ &\text{before}(x, a_2, task)) \} \\ perms(c) &= indirectperms(c) \cup internperms(c) \\ &\cup reqperms(c) \end{aligned}$$

We now come to the definition of LP. A system adheres to LP if all its components adhere to LP. A component does not adhere to LP if it, based upon the permissions attributed as described before, is capable of executing tasks it is not responsible for.

$$\begin{aligned}
\text{adherestolp}(s) &\Leftrightarrow \forall c \in \\
&\quad \text{components}(s) \mid \text{adherestolp}(c) \\
\text{adherestolp}(c) &\Leftrightarrow \\
&\quad \forall t1, t2 \in \text{executabletasks}(c) \mid \\
&\quad t1, t2 \in \text{tasks}(c) \text{ and } \text{executabletogether}(t1, t2) \\
\text{executabletogether}(t1, t2) &\Leftrightarrow \\
&\quad \exists u1 \in U \mid (u1, t1), (u1, t2) \in UT \\
\text{executabletasks}(c) &= \{ t \mid t \in T \text{ and} \\
&\quad \forall \text{perm} \in \text{permissions}(t): \text{perm} \in \text{perms}(c) \}
\end{aligned}$$

This definition is based on the fact that we assume that all behavior is executed via tasks and that we thus can relate components to users in the following way. The permissions attributed to a component determines the tasks that can be executed by this component. These tasks are related to users via the user-task assignment set. As such, a component will act on behalf of a possible set of users.

Our model defines least privilege correctly for the majority of the software architectures, but might be incorrect in the following cornercases: an architecture with one component per action, an architecture supporting one task, an architecture supporting many tasks each of which consisting of one action, an architecture used by one user, an architecture supporting many tasks each of which being used by a different user, an architecture supporting many tasks each of which requiring the same permission, and an architecture supporting tasks each of which requires one permission per used action. To solve this, we include the following assumptions.

- Every task in an architecture is executed by a user.

$$\forall t \in T \mid \exists u \in U \text{ and } (u, t) \in UT$$

Indeed, our model will not detect violations related to (sub)tasks that are not executed by a user, because the model identifies violations based on user-tasks assignments. Hence, tasks that are not part of the user-task assignment relation will not be used for detecting violations. This assumption is reasonable because in reality all tasks are initiated by some actor. For instance, withdrawing money is initiated by a bank customer, a cron job that executes a script that backs up data is initiated by the person wanting the back-up, and so forth.

- A software architecture has at least two users.

$$\exists u1, u2 \in U \mid \text{not } (u1 = u2)$$

Indeed, our model will not detect violations if one user executes all the functionality, because our model states that a component violates least privilege if it is able to execute tasks executed by different users. This assumption is reasonable, because systems are typically used by at least two types of users: administrators and others. If this is not the case, one can introduce a user that executes functionality that nobody is allowed to execute (i.e. attacks).

- A software architecture supports at least two tasks.

$$\exists t1, t2 \in T \mid \text{not } (t1 = t2)$$

Indeed, our model will not detect violations if there is only one task, because our model states that a component violates least privilege if it is able to execute tasks executed by different users. This implies that there should be at least two tasks. This assumption is reasonable, because a system typically supports more than one use case.

4 Identifying LP Violations

For the identification of LP violations, an algorithm was constructed based on the formal model. The input provided to the algorithm is threefold: a component and connector diagram, a list enumerating user-task assignments, and sequence diagrams mapping these tasks onto the component and connector diagram. The algorithm (See Figures 3, 4, 5, and 6) iterates over the different components, attributes permissions to each

```

SA(s) = software architecture
UT = user task assignments
EC = tasks

For each component in components(s)
  assign int. perms based on component actions
  For each task in tasks(c)
    propagate req. perms upward in task
    calculate ind. perms for downward components in task

For each component in components(s)
  determine LP violations based on assignment

```

Figure 3: Identification algorithm

```

for each component in components(s)
  for each action in actions(component)
    internperms.add(permissions(action))

```

Figure 4: Assign internal permissions

```

for each component in components(s)
  for each task in tasks(component)
    for each action in actions(task)
      for each action2 in actions(component)
        if (before(action2,action,task))
          requiredperms.add(permissions(action))

```

Figure 5: Propagate required permissions

```

//order of components(s) determines perms
for each component in components(s)
  for each component2 in reachablecs(component)
    for each task in sharedTasks(component2,component)
      for each task2 in (tasks(component2) \ sharedTasks(component2,component))
        if (sharedState(task,task2,component2).size > 0)
          for each perm in
            (intersection(union(internperms(component2),reqperms(component2)),
              permissions(task2)))
            indirectperms.add(perm)

```

Figure 6: Calculate indirect permissions

component, and verifies whether each component violates LP. The output of the algorithm lists the LP violations, specifying the violating component, the permissions causing the violation, and the conflicting tasks.

The algorithm can be parameterized with several options, two of which will be discussed here. A first option is the order in which the components will be iterated over. This order controls the assignment of indirect permissions, because these are propagated via the shared state of other components. Our algorithm currently starts with calculating permissions for "leaf node" components. Next, it follows these tasks in reverse order to determine the one-but-last components to calculate the permissions of, and so forth. Other propagation strategies such as root-node propagation could be considered as well.

Another option is the shared state approximation strategy. This determines the shared state between tasks in a component and thus controls the indirect permissions that are assigned to components. At this moment, the shared state is determined based on equivalence of the name and the type of the parameters of an action: if two methods share an equivalent parameter, they are considered capable of influencing each other. This is clearly a rough approximation of the shared state.⁴ This is illustrated in Figure 2, where the shared state of methods m1, m2, and m5 in component Calendar is approximated by the event parameter they all have. Other strategies could consist in working with semantic method annotations (such as pre- and postconditions) or on explicit annotations regarding shared state.

5 Resolving LP Violations

Different strategies exist to accommodate least privilege in a software architectural structure, among which: (i) splitting components into several isolated units and lowering permissions assigned to these units, (ii) rewiring the architecture (i.e. rerouting tasks to other components in order to split up conflicting privileges), (iii) splitting tasks, or (iv) applying well-known solutions and patterns (such as sandboxing) to introduce LP in selected parts of the architecture (see Section 9).

The remainder of this section elaborates on these strategies. Each strategy documents technical challenges, an architectural transformation, limitations of the transformation, and a proof that the transformation lowers permissions and thus privileges⁵.

5.1 Solution spectrum

Strategies to accommodate the property of least privilege in a software architectural structure are the ones that transform the fundamental elements used to express that property. Indeed, adherence to a property can only be introduced by modifying elements (or properties) affecting that property. For instance, a system won't support authorization (property) without introducing an authorization engine (element affecting authorization).

In general, modifications of (composed or sets of) elements are (See Figure 7): adding elements to a composed element, removing elements from a composed element, splitting a composed element into x elements, merging x composed elements into a new composed element, and moving basic elements from one composed element to another one.

In our model, fundamental elements affecting least privilege are: permissions, tasks, users, components, and actions (See Figure 8). Permissions and components are composed of actions, while tasks consists of action-tuples. Possible transformations on these elements are obtained by applying the general modifications on each of the fundamental elements. This results in the following transformations (See Table 2 for an overview):

Split Component splits a component into several components by splitting its actions into multiple sets.

Each new set is a new component. This transformation ensures that each newly created component requires less permissions than the original one, because the new ones offer less functionality (actions). The transformation can solve least privilege violations, because a component responsible for violating tasks can be split in a way that the resulting components do not violate LP anymore.

⁴Actually, in case of parameters of primitive types, it probably does not make much sense to use this type of approximation.

⁵We do not guarantee that applying these transformations result in the least amount of assigned permissions. However, applying these reduces the number of assigned permissions.

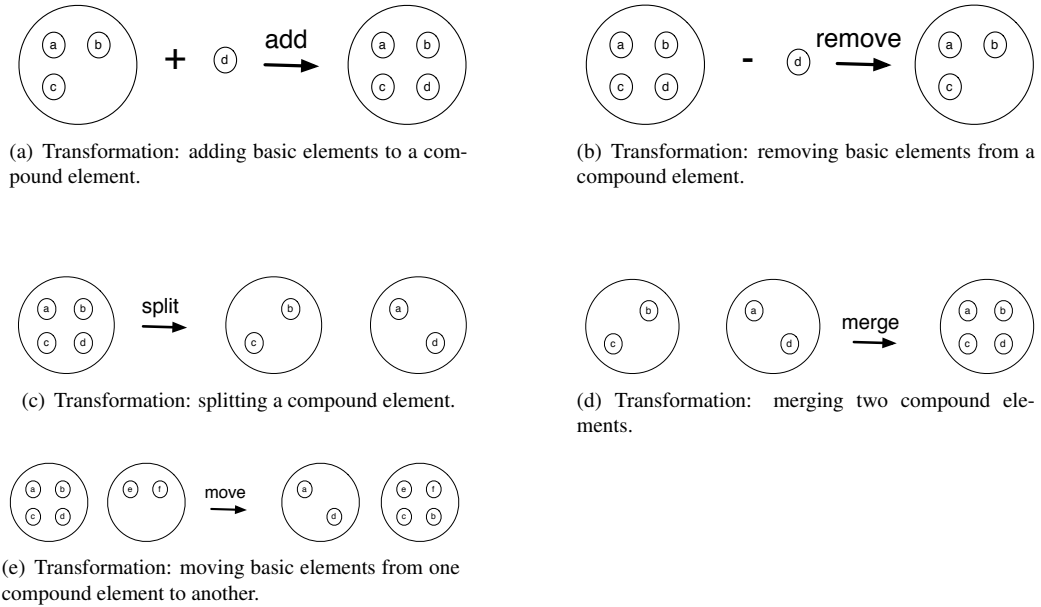


Figure 7: An overview of transformations of compound elements. These transformations are used to identify transformations on the elements of the formal model that define the property of least privilege.

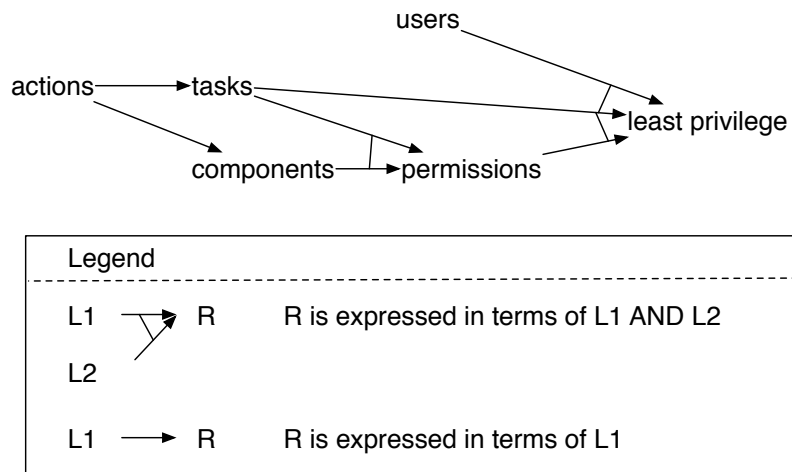


Figure 8: The property of least privilege is defined in terms of actions, permissions, tasks, users, and components.

element (basic element)	Component (action)	Task (action-tuple)	User (task from UT)	Permission (action)	Action (parameter)
Split	DS	DS	DN	IS	-S
Merge	IS	IS	IS	DN	IN
Remove basic element	DS	DS	DS	-S	-S
Add basic element	IN	IN	IS	-S	-S
Move basic element	DS	DS	DS	DS	DS

Table 2: Possible basic transformations that have an effect on least privilege. D means that this transformation can decrease the amount of permissions, I means that it can increase the amount of permissions, S means that it might solve least privilege violations, N means that it does not solve least privilege violations, S means that it solves LP problems but is not a good solution, and -S means that the number of permissions is not affected, but it can solve LP violations because the transformation enables better permission assignment. Transformations with a grey background have been detailed in the remainder of this section.

Split Task splits a task into several subtasks by splitting its sequence of action-tuples in multiple sequences of action-tuples. Each sequence is a new task. Splitting a task to the parts that are effectively executed reduces the required permissions of the components executing that task. The transformation can solve least privilege violations, because a task causing a violation can be split in a way that the actions requiring the permissions causing the violation are not part of that task anymore.

Split User splits a user into several new users by splitting his set of tasks in multiple sets of tasks. Each set is assigned a new user⁶. In other words, the tuples in the user-task assignment set having the user to be split as first element are modified in a way that a subset of those tuples have a new user as first element. Splitting a user reduces the permissions each new user requires, because he executes less tasks. However, this transformation won't solve least privilege violations due to our definition of least privilege. On the contrary, it might introduce new violations.

Split Permission splits a permission into several subpermissions by splitting its set of actions in multiple sets of actions. Each new set is a new permission. Splitting a permission increases the number of permissions assigned to a component, because the newly created permissions are also assigned to that component. However, splitting a permission might solve least privilege violations, because more finegrained permissions allow us to assign only the permissions required for a particular task.

Split action splits an action's set of parameters in multiple set of parameters. Each new set is a new action. Splitting an action does not necessarily lowers the number of assigned permissions, because the set of permissions required for the new actions is the same as the set of permissions required for the original action. However, splitting an action allows us to introduce more fine-grained permissions, which in turn might solve violations. Note that this transformation is not implementable, because the conditions for splitting an action are not known.

Merge Component merges several components into a single component by merging their sets of actions. Merging components ensures that the newly created component requires more permissions than each of the original ones, because its required permissions are the union of the required permissions of the original components. Hence, we assume that it won't solve least privilege problems.

Merge Task merges tasks into one new task by merging their sets of action-tuples. This increases the number of permissions assigned to a component (or user) if that component (or user) becomes responsible for a subtask it was not responsible for. This transformation solves violations due to our definition of least privilege: assigning a merged task that is created from two violating tasks to one user will remove the violation. Clearly, this transformation is not a good solution, because one user having the permissions to execute all the tasks has been causing problems for several years in the

⁶e.g. the creation of a new user role

real world. In order to solve this problem, we either have to (i) assume that the user task assignment is optimal, or (ii) redesign our definition of least privilege and thus our formal model. For now, we assume that the user-task assignment is optimal.

Merge User merges several users into a single user by merging their sets of tasks. This increases the number of permissions that a user requires, but solves violations due to our definition of least privilege. Again, this transformation is not a good solution for the same reason as the previous transformation.

Merge Permission merges several permissions into a single permission by merging their sets of actions. This lowers the number of permissions that are assigned to a component. However, it won't solve least privilege violations, because components will be able to execute more tasks with coarse grained permissions.

Merge Action merges several actions into a single action by merging their sets of parameters. This increases the number of permissions assigned to a component if at least one of the actions to be merged is an action of another component and the execution of that action requires a permission the component has not already. By consequence, we assume that this transformation will not solve LP violations.

Remove an action from a component removes an action from a component's set of actions. This lowers the number of internal permissions associated with that component if it does not have other actions with the permission(s) of the removed action. This transformation can solve least privilege violations, if the component's actions requiring internal permissions that cause a violation (i.e. give the ability to execute other tasks) are removed.

Remove an action-tuple from a task removes an action-tuple from a task's set of action-tuples. This lowers the number of required permissions of the components executing that task if these don't execute tasks that require the same permission. This transformation can solve least privilege violations, if an action from the action-tuple that is removed requires permissions that cause violations. However, note that this transformation is not implementable, because it is hard to identify the conditions under which tuples can be removed.

Remove a task from a user or prohibiting a user to execute a certain task is a transformation that removes (user,task) from the set of user-task assignments. This transformation decreases the number of permissions a user requires. Hence, it might solve least privilege problems. Note that this transformation is implementable, but that management should decide whether a user should be able to execute a certain task or not.

Remove an action from a permission removes an action from a permission's set of actions. This does not affect the property of least privilege, because the number of permissions stays the same. However, this transformation influences the granularity of permissions, and thus can optimize permission assignment.

Remove a parameter from an action removes a parameter from an action's set of parameters. This does not affect the property of least privilege, because the number of permissions stays the same. However, this transformation influences the granularity of actions, and thus can optimize permission assignment. Note that this transformation is not implementable, because it is hard to determine the conditions under which parameters can be removed.

Add an action to a component adds actions to a component's set of actions. This increases the number of internal permissions of the component if it does not already require the permissions associated with these new actions. Hence, we assume that this transformation does not solve LP violations.

Add an action-tuple to task adds action-couples to a task's set of action-couples. This increases the number of required permissions of the components executing this task, if they do not already require these permissions. Hence, we assume that this transformation does not solve LP violations.

Add a task to a user adds (user, task) tuples to the user-task assignment relation. This increases the number of permissions that a user requires, but solves violations due to our definition of least privilege. Clearly, this transformation is not a good solution.

Add an action to permission adds actions to a permission's set of actions. This does not affect the property of least privilege, because the number of permissions stays the same. However, this transformation can optimize permission assignment, which in turn can solve LP violations. Note that this transformation is not implementable, because it is hard to determine the conditions under which an action can be added to a permission.

Add a parameter to an action adds parameters to an action's set of parameters. This does not affect the property of least privilege, because the number of permissions stays the same. However, this transformation can optimize permission assignment, which in turn can solve LP violations. Note that this transformation is not implementable, because it is hard to determine the conditions under which a parameter can be added to an action.

Move an action from one component to another moves an action from a component's set of actions to another component's set of actions. This transformation is a combination of the *remove an action from a component* and *add an action to a component* transformations. Hence, the number of permissions of the first component lowers, while the number of permissions of the second component increases. This transformation can solve LP violations, because its subtransformation *remove an action from a component* can solve LP violations.

Move an action-tuple from one task to another moves action-tuples from a task's set of action-tuples to another task's set of action-tuples. This transformation is a combination of the *remove action-tuple from a task* and *add action-tuple to a task* transformations. Hence, the number of permissions the former requires decreases, while the number of permissions the latter requires increases. This transformation can solve LP violations, because its subtransformation *remove an action-tuple from a task* can solve LP violations.

Move a task from one user to another moves a task from a user set's of tasks to another user's set of tasks. This transformation is a combination of the *remove a task from a user* and *add a task to a user* transformations. Hence, the number of permissions of the first user decreases, while the number of the permissions of the second user increases. This transformation can solve LP violations, because its subtransformation *add a task to a user* can solve LP violations. Note that this transformation is implementable, but that management decides whether a user is allowed to execute a task or not.

Move an action from one permission to another moves an action from a permission's set of actions to another permission's set of actions. This transformation is a combination of the *remove action from permission* and *add action to permission* transformations. This transformation can decrease the overall number of needed permissions, because the permission granularity is modified. Hence, we assume that it can solve LP violations.

Move a parameter from one action to another moves a parameter from an action's set of permissions to another action's set of permissions. This transformation is a combination of the *remove parameter from an action* and *add parameter to an action* transformations. This transformation can decrease the overall number of needed permissions, because the permission granularity is modified. Hence, we assume that it can solve LP violations. Note that this transformation is hard to implement because, (i) the conditions under which a parameter can be moved are hard to determine due to the lack of information at architectural level.

Table 2 summarizes the effect of above transformations on least privilege. For each transformation is told whether applying the transformation affects the number of permissions and solves least privilege. D means that this transformation can decrease the amount of permissions, I means that it can increase the amount of permissions, S means that it might solve least privilege violations, N means that it does not solve least privilege violations, \$ means that it solves LP problems but is not a good solution, and -S means

that the number of permissions is not affected, but it can solve LP violations because the transformation enables better permission assignment. Transformations with a grey background are *implementable*⁷ transformations that aid in least privilege support. Hence, they have been detailed in the remainder of this section.

5.2 Splitting components

Splitting a component is a transformation that splits the component into several isolated units with lowered permissions assigned to these units.

One of the challenges of this transformation is that it has to be split in a way that preserves the semantics of the component: semantically related actions must remain adjacent even after splitting. The knowledge available for splitting is typically limited to the interfaces of the component, the actions described in these interfaces, and the parameters of these actions. Our approach uses these parameters to approximate related actions by applying a variant of the shared state approximation strategy on all possible subsets of the component's actions: actions that use the same (sub)set of parameters are related. However, in order to split a component that contains related actions used by violating tasks, we require extra information to be present in the architectural description: read/write on the action's parameters (see transformation itself).

Transformation If two tasks are delegated to a component via two actions, and the internal or required permissions associated with these actions cause a LP violation, then, based upon the shared state between the tasks, the component can be split as follows (See Figure 9).

1. If the shared state is empty, split the component in two disjunct parts by moving the interfaces/actions that one task uses to (a) new interfaces in a new component. Update the tasks accordingly.
2. If the shared set is not empty, and if one task reads state that is written by the other task, then create a new component containing a copy of the actions/interfaces of the writing task. Add a new interface on the original component to update the shared state. Extend the reading task to include the update actions provided by the new component.

Furthermore, if at least one indirect permission is causing the violation, then split the component with the shared state that propagates this indirect permission as described above.

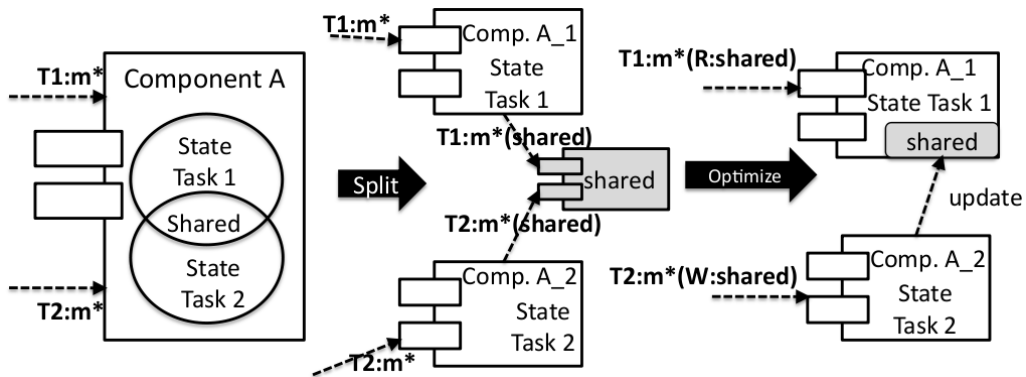


Figure 9: Transformation split component: a component with overlapping read methods and a write method can be split.

⁷An implementable transformation is a transformation that can be automated by a tool. Automatization can be achieved if sufficient architectural documentation is available to document the conditions that indicate when to apply the transformation and to identify the transformation itself.

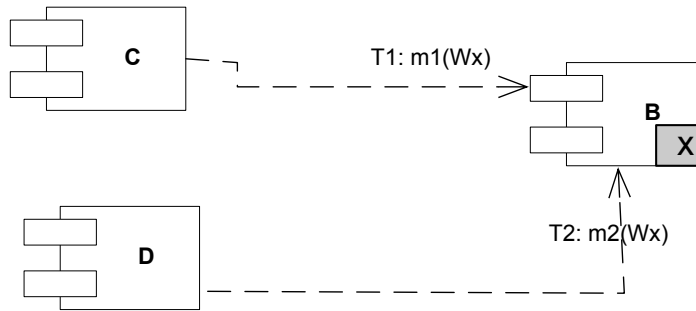


Figure 10: Transformation split component: a component with two write methods should not be split.

Known Problems The rule does not work if two violating tasks write on their shared state (See Figure 10). Indeed, splitting the violating component in a component per task, results in two components with shared state update methods for both tasks. Hence, these tasks can still influence each other via these components. Consequently, the indirect permissions of both components will not have been decreased.

Proofs Based on the model of section 3, it is easy to see that the number of permissions will reduce for one of the following reasons (See Figure 11).

First, partitioning a component will result in subcomponents each having less internal permissions by definition.

Second, partitioning a component in a way each partition is responsible for less tasks, will result in partitions requiring less required permissions by definition.

Third, if a component that grants indirect permissions to another component is split, then it is possible that these indirect permissions are not granted anymore, because the shared state propagating these indirect permissions does not exist anymore. Hence, the number of indirect permissions of that other component is lower or equal than the number of indirect permissions before splitting.

5.3 Splitting tasks

Splitting a task is the process of splitting a task in multiple tasks with each less required permissions.

One of the challenges in splitting a task is that it has to be split in a semantically correct way: each new task should be a logical unit of work that corresponds to a unit of work that a user wants to use the system for (typically a use case defined at requirements level). In other words, this unit of work should be *meaningful* to the end-user and thus have the *right* granularity. For instance, in a point of sale system a task make sale is meaningful, while a task save data in a database is not. Making this semantic distinction between meaningful tasks and meaningless tasks is hard to achieve in an automatic way, is another research domain, and therefore is not the focus of this report. By consequence, the software architect should determine if a certain subtask has a meaning at requirements level.

Transformation If a task (i) is executed by multiple users, and (ii) each user uses a different *branch* of that task, then split that task in the following way (See Figures 12, 13, and 14):

1. one task is the common part between both users and the branch that user 1 uses.
2. another task is the original task minus the branch that user 1 users.

Known problems Determining which user uses which branch of a task at architectural level is hard, because the architectural documentation is typically not detailed enough.

Proof. Assume that the following is given (from model):

$$\begin{aligned} \text{tasks}(c) &= \{t | t \in T \wedge a \in \text{actions}(c) \wedge a \in \text{actions}(t)\} & (1) \\ \text{ownactions}(c) &= \{a | t \in \text{tasks}(c) \wedge a \in \text{actions}(t) \wedge a \in \text{actions}(c)\} & (2) \\ \text{usedactions}(c) &= \{a | t \in \text{tasks}(c) \wedge a \in \text{actions}(t) \wedge a \notin \text{actions}(c) \wedge \dots\} & (3) \\ \text{perms}(c) &= \text{indirectperms}(c) \cup \text{internperms}(c) \cup \text{reqperms}(c) & (4) \\ \text{internperm}(c) &= \{o | p \in \text{permissions}(\text{action}) \wedge \text{action} \in \text{ownactions}(c)\} & (5) \\ \text{reqperm}(c) &= \{p \in \text{permissions}(\text{action}) \wedge \text{action} \in \text{usedactions}(c)\} & (6) \\ \text{sharedTasks}(c1, c2) &= \{t \in \text{tasks}(c1) \wedge t \in \text{tasks}(c2)\} & (7) \\ \text{reachablecs}(c) &= \{c2 | c2 \in \text{components}(s) \wedge t \in \text{sharedTasks}(c, c2) \wedge \dots\} & (8) \\ \text{indirectperms}(c) &= \{\text{perm} | c2 \in \text{reachablecs}(c) \wedge \dots\} & (9) \end{aligned}$$

The split component transformation splits the set $\text{actions}(\text{component_before})$ in 2 disjunct sets:

$$\begin{aligned} \text{actions}(\text{component}_{\text{after}1}) &\subset \text{actions}(\text{component}_{\text{before}}) & (10) \\ \text{actions}(\text{component}_{\text{after}2}) &\subset \text{actions}(\text{component}_{\text{before}}) & (11) \\ \text{actions}(\text{component}_{\text{after}1}) \cap \text{actions}(\text{component}_{\text{after}2}) &= \emptyset & (12) \end{aligned}$$

We have to proof that:

$$\begin{aligned} |\text{perms}(\text{component}_{\text{after}1})| &< |\text{perms}(\text{component}_{\text{before}})| & (13) \\ |\text{perms}(\text{component}_{\text{after}2})| &< |\text{perms}(\text{component}_{\text{before}})| & (14) \end{aligned}$$

Proof:

$$\begin{aligned} 4 \wedge 13 &\Leftrightarrow |\text{internperms}(\text{component}_{\text{after}1})| < |\text{internperms}(\text{component}_{\text{before}})| & (15) \\ &\vee |\text{reqperms}(\text{component}_{\text{after}1})| < |\text{reqperms}(\text{component}_{\text{before}})| & (16) \\ &\vee |\text{indirectperms}(\text{component}_{\text{after}1})| < |\text{indirectperms}(\text{component}_{\text{before}})| & (17) \end{aligned}$$

So it is sufficient if we proof 15, 16, or 17. We proof all three.

Now we proof 15.

$$\begin{aligned} 10 \wedge 1 &\Rightarrow \text{tasks}(\text{component}_{\text{after}1}) \subset \text{tasks}(\text{component}_{\text{before}}) & (18) \\ 2 \wedge 18 &\Rightarrow \text{ownactions}(\text{component}_{\text{after}1}) \subset \text{ownactions}(\text{component}_{\text{before}}) & (19) \\ 19 \wedge 5 &\Rightarrow \text{internperm}(\text{component}_{\text{after}1}) \subset \text{internperm}(\text{component}_{\text{before}}) & (20) \\ 20 &\Rightarrow 15 & (21) \end{aligned}$$

Now we proof 16.

$$\begin{aligned} 3 \wedge 18 &\Rightarrow \text{usedactions}(\text{component}_{\text{after}1}) \subset \text{usedactions}(\text{component}_{\text{before}}) & (22) \\ 6 \wedge 22 &\Rightarrow \text{reqperms}(\text{component}_{\text{after}1}) \subset \text{reqperms}(\text{component}_{\text{before}}) & (23) \\ 23 &\Rightarrow 16 & (24) \end{aligned}$$

□

Proofs Based on the model of section 3, it is easy to see that the number of required permissions of the users reduce for the following reason (See Figure 15). Removing a part of a task from a component (possibly) lowers the internal and required permissions of that component (in case they are not required for another task). The proof is similar to the one of rewiring tasks.

Proof. Now we proof 17.

$$18 \wedge 7 \Rightarrow \text{sharedTasks}(\text{component}_{\text{after1}}) \subset \text{sharedTasks}(\text{component}_{\text{before}}) \quad (25)$$

$$25 \wedge 8 \Rightarrow \text{reachablecs}(\text{component}_{\text{after1}}) \subset \text{reachablecs}(\text{component}_{\text{before}}) \quad (26)$$

$$9 \wedge 26 \Rightarrow \text{indirectperms}(\text{component}_{\text{after1}}) \subset \text{indirectperms}(\text{component}_{\text{before}}) \quad (27)$$

$$27 \Rightarrow 17 \quad (28)$$

□

Figure 11: This proof proves that splitting a component lowers the number of permissions.

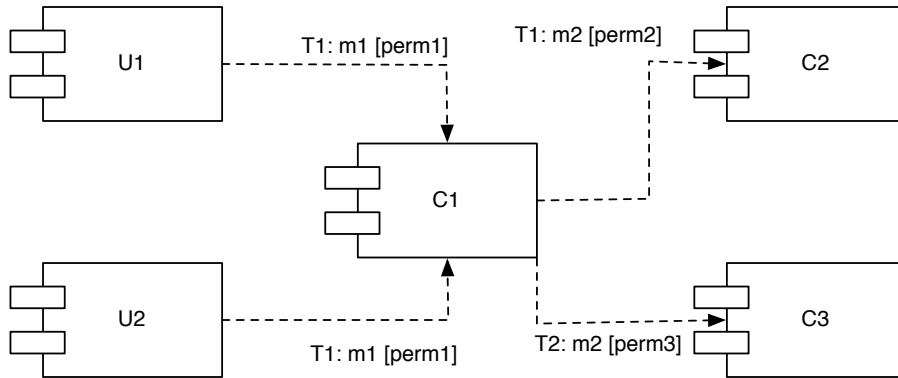


Figure 12: Transformation split tasks: structural view before splitting.

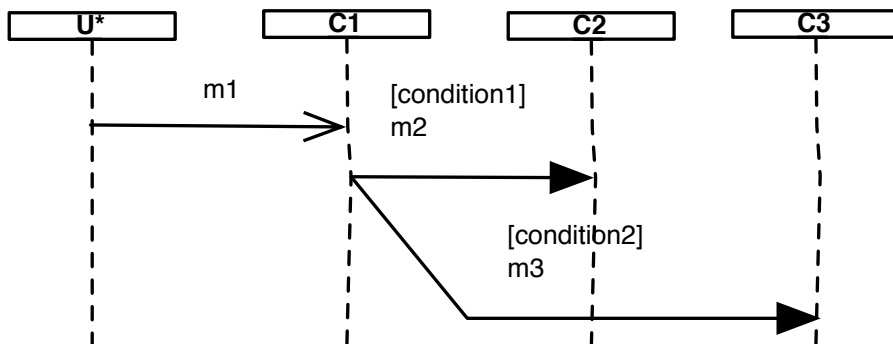


Figure 13: Transformation split tasks: behavioral view before splitting.

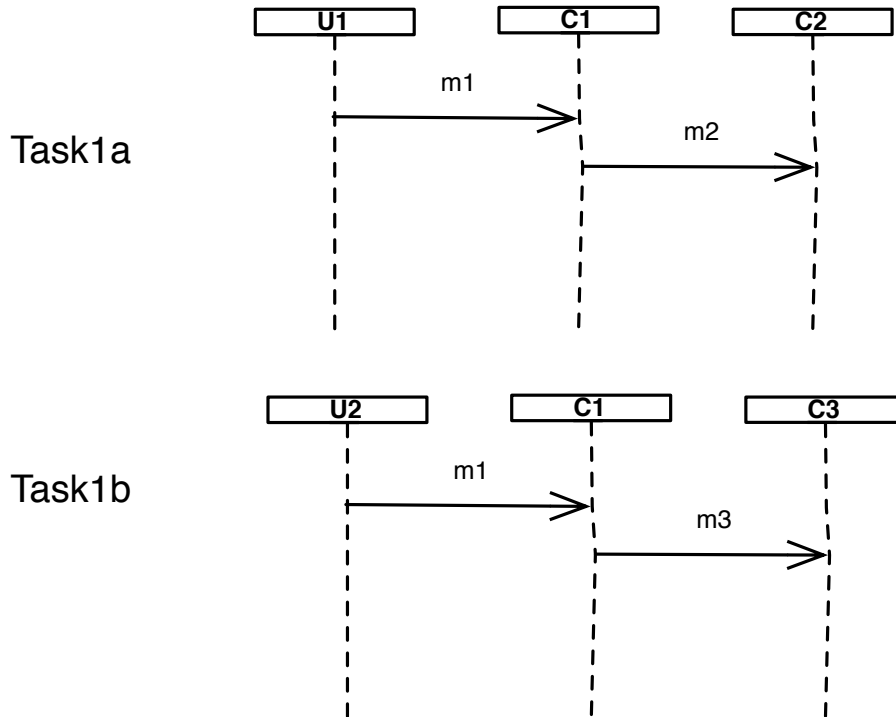


Figure 14: Transformation split tasks: behavioral view after splitting.

5.4 Splitting permissions

Splitting a permission is a transformation that tries to optimize permission-actions assignments by moving a group of actions of the permission to newly created permissions.

Transformation If task 1 requires a permission p , task 2 requires the same permission p , this permission is associated with a set of actions, task 1 uses a subset of these actions (subset1), task 2 uses a different subset (subset2), and both tasks are executed by different users, then split permission p in the following way:

1. $p1 = \{ \text{subset1} \}$
2. $p2 = \text{subset2} \setminus \text{subset1}$
3. $p3 = \text{subset1} \cap \text{subset2}$
4. $p4 = p \setminus (\text{subset1} \cup \text{subset2})$

This is illustrated in Figure 16.

Known Problems This transformation does not work in following situations:

- tasks depending on the same actions. Indeed, in this situation the transformation reduces the number of permissions, but can not solve the violation because both tasks use the same actions, and thus require the same permissions. Hence, they still can influence each other.

Proofs Splitting a permission increases the number of permissions assigned to a component, because the newly created permissions are also assigned to that component. However, splitting a permission might solve least privilege violations, because more finegrained permissions allow us to assign only the permissions required for one task. A proof is left as exercise to the reader.

Proof. The splitting task transformation splits a task in two parts. For simplicity, we assume that both parts are disjunct.

$$actions(task_{after1}) \subset actions(task_{before}) \quad (29)$$

$$actions(task_{after2}) \subset actions(task_{before}) \quad (30)$$

$$actions(task_{after1}) \cap actions(task_{after2}) = \emptyset \quad (31)$$

We have to proof for every component relying on the task being split that its number of permissions lower. c_{before} means the component before the task was split, while c_{after} is the same component, but after splitting the task.

$$\forall c \in components(c) \wedge task \in tasks(c) \mid |perms(c_{before})| < |perms(c_{after})| \quad (32)$$

The proof is as follows:

$$4 \wedge 13 \wedge 31 \Leftrightarrow |internperms(component_{after})| < |internperms(component_{before})| \quad (33)$$

$$\vee |reqperms(component_{after})| < |reqperms(component_{before})| \quad (34)$$

$$\vee |indirectperms(component_{after})| < |indirectperms(component_{before})| \quad (35)$$

So it is sufficient if we proof 33, 34, or 35.

We proof 33 first.

$$2 \wedge 29 \Rightarrow ownactions(c_{before}) \subset ownactions(c_{after}) \quad (36)$$

$$36 \wedge 15 \Rightarrow internperms(c_{before}) \subset internperms(c_{after}) \quad (37)$$

$$37 \Rightarrow 33 \quad (38)$$

Next, we proof 34.

$$3 \wedge 29 \Rightarrow usedactions(component_{after}) \subset usedactions(component_{before}) \quad (39)$$

$$6 \wedge 39 \Rightarrow reqperms(component_{after1}) \subset reqperms(component_{before}) \quad (40)$$

$$40 \Rightarrow 16 \quad (41)$$

□

Figure 15: Proof that the splitting tasks transformation lowers the number of permissions.

5.5 Removing unused actions from a component

Removing unused actions from a component is a transformation that removes an action from a components set of actions. This lowers the number of internal permissions associated with that component if it does not have other actions with the permission(s) of the removed action. This transformation can solve least privilege violations, if the components actions requiring internal permissions that cause a violation (i.e. give the ability to execute other tasks) are removed.

Transformation If an action of a component is not used by a task, then remove that action from that component.

This is illustrated in Figure 17.

Known Problems This transformation has one main drawback: it might still be possible that removed actions would have been needed in the future.

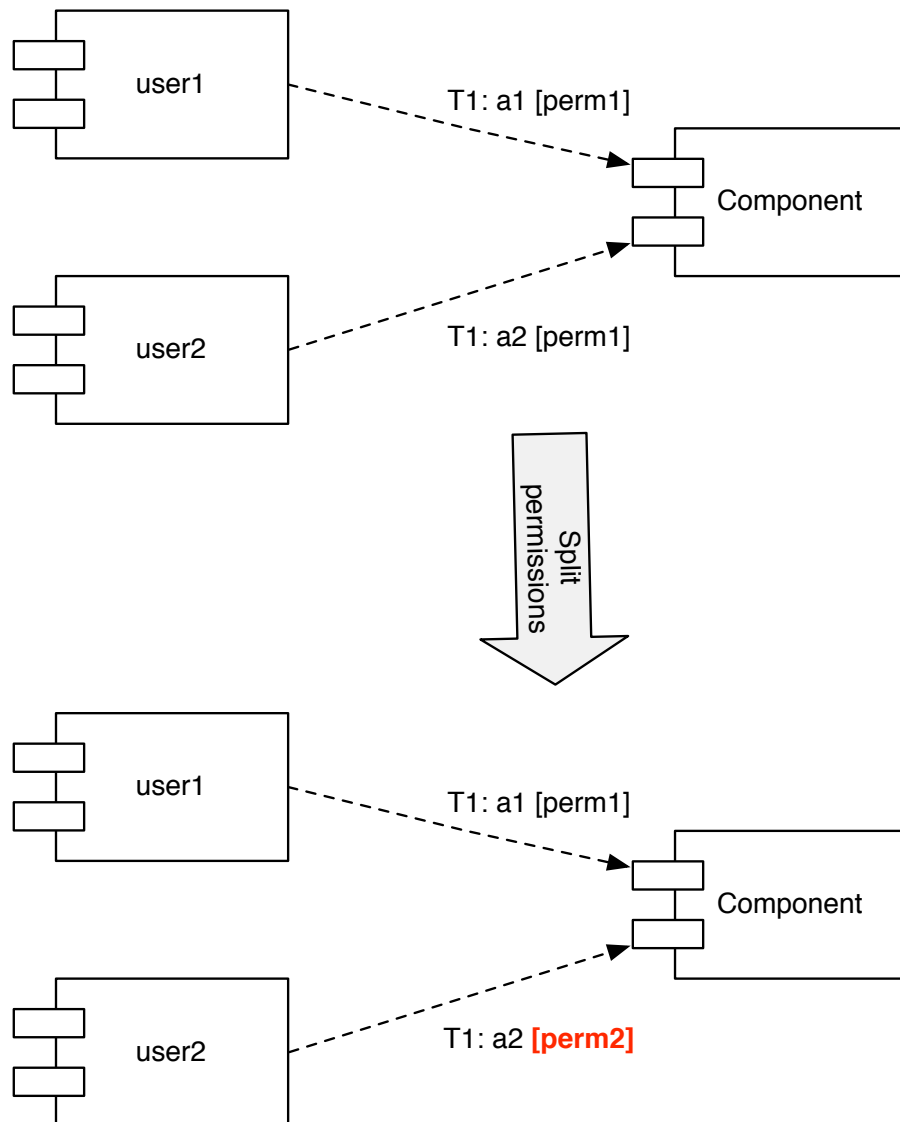


Figure 16: Transformation: splitting permissions.

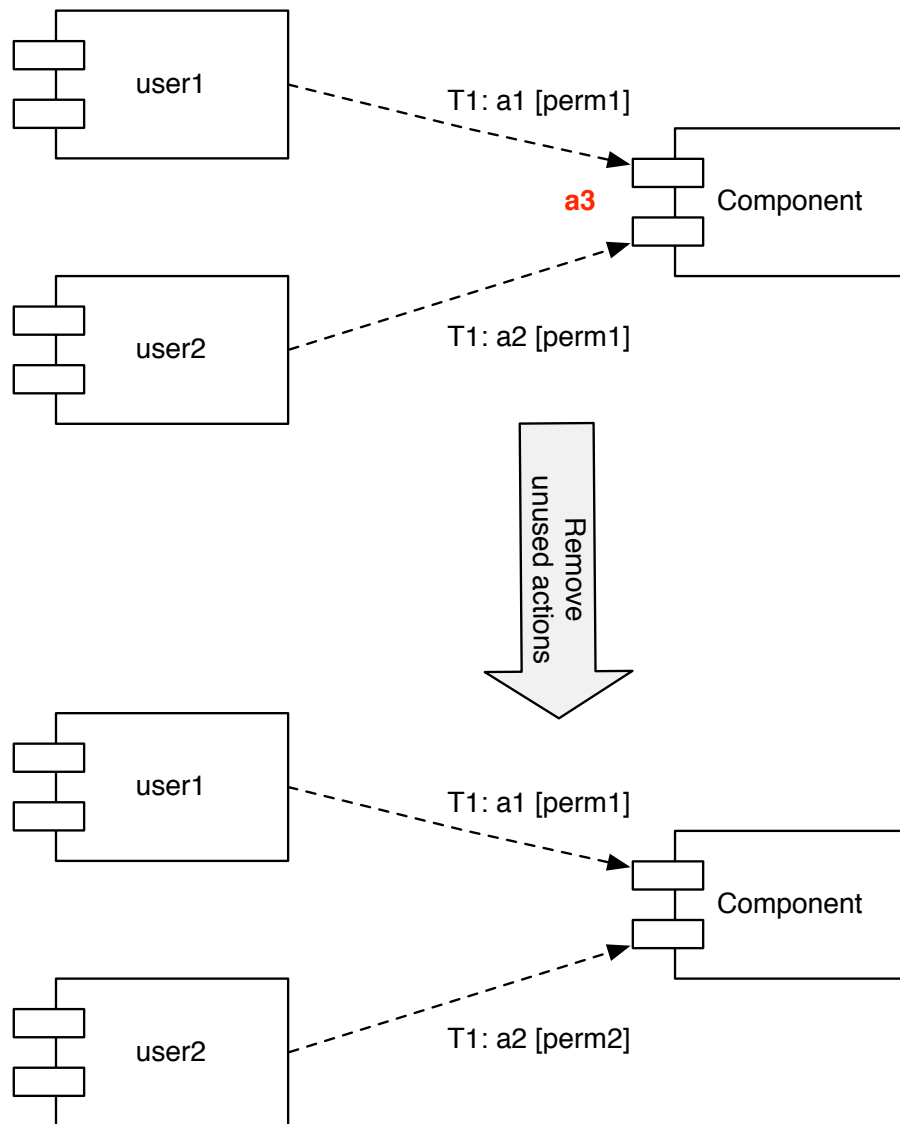


Figure 17: Transformation remove unused actions from a component.

Proofs This transformation lowers the number of internal permissions associated with that component (See Figure 18). It also solves least privilege violations, because unneeded internal permissions that give the ability to execute other tasks are taken away from the component.

Proof. Assume that the following is given: The transformation remove actions from a component:

$$actions(c_{after}) \subset actions(c_{before}) \quad (42)$$

We have to proof that the number of permissions of that component lowers. As seen in the other proofs, it is sufficient to proof that the number of internal permissions lower.

$$|internperms(c_{after})| < |internperms(c_{before})| \quad (43)$$

The proof is as follows:

$$2 \wedge 18 \Rightarrow ownactions(c_{after}) \subset ownactions(c_{before}) \quad (44)$$

$$5 \wedge 44 \Rightarrow internperms(c_{after}) \subset internperms(c_{before}) \quad (45)$$

$$45 \Rightarrow 43 \quad (46)$$

□

Figure 18: Proof that the removing unused actions from component transformation lowers permissions of that component.

5.6 Removing tasks from a user

Removing a task from a user or prohibiting a user to execute a certain task is a transformation that removes (user,task) from the set of user-task assignments. This transformation decreases the number of permissions a user requires.

One of the challenges of this transformation is that it is hard to prohibit a user from executing tasks he needs to execute. Indeed, our approach can prohibit a user from executing a task that causes many violations, but in the end management decides whether a user is prohibited to execute that task.

Transformation If user u executes a task t that causes a significant percentage of the violations (see Section 6), then

1. Remove (u,t) from the set of user-task assignments.
2. If the number of tasks that this user is able to execute is 0, then remove the user from the set of users.
3. If the task is not executed anymore by a user, then remove the task from the set of tasks.

Proofs Assume that the following is given: The transformation remove task from a user:

$$usertasks(c_{after}) \subset usertasks(c_{before}) \quad (47)$$

We have to proof that this transformation results in less violations.

An outline of the proof is as follow:

- The number of user task tuples reduces.
- The number of conflicts in the function executabletogether reduces (power set).
- The number of possible violations of the principle reduces as well.

5.7 Moving actions from a component to another

Moving actions from a component to another moves an action from a components set of actions to another components set of actions. This transformation is a combination of the remove an action from a component and add an action to a component transformations. Hence, the number of permissions of the first component lowers, while the number of permissions of the second component increases. This transformation can solve LP violations for the same reasons as the split component transformation.

Transformation The transformation is similar to splitting a component.

Proofs This proof is similar to the proof of splitting a component.

5.8 Rewiring the architecture

Rewiring the architecture (= moving action-tuples in tasks) is the process of changing the components being responsible for certain tasks. It ensures that less privileges have to be attributed to the different components if the process properly delegates (parts of) existing tasks to other components.

One of the challenges in rewiring a task is that it has to be rewired in a semantically correct way: the component a task is rewired to should offer the same (or similar) actions than the component the task is rewired from. The knowledge available for rewiring is typically limited to the actions described in a component's interfaces, the parameters of these actions, and the pre- and post conditions associated with these actions. Our approach uses these to identify similar actions. An action a_1 is similar to another action a_2 iff a_1 and a_2 have the same signature and $a_1.pre \subseteq a_2.pre$ and $a_1.post \rightarrow a_2.post$. Note that a_1 similar to a_2 does not imply a_2 similar to a_1 .

$$\begin{aligned} \text{similar}(a_1, a_2) &\leftrightarrow \text{samesignature}(a_1, a_2) \text{ and } a_1.pre \subseteq a_2.pre \text{ and } a_1.post \rightarrow a_2.post \\ \text{samesignature}(a_1, a_2) &\leftrightarrow \text{params}(a_1) \subseteq \text{params}(a_2) \text{ and } \text{params}(a_2) \subseteq \text{params}(a_1) \end{aligned}$$

Transformation If two tasks are delegated to a component via two actions, the internal or required permissions associated with these actions cause a LP violation, and one of these actions (or a similar one) is offered by another component, rewire that task to another component in the following way (See Figure 19):

1. Modify the task to use the action of that other component instead.
2. Change the dependencies accordingly.

Let's illustrate this by a small example (See Figure 19). Suppose that three components are responsible for two tasks: t_1 and t_2 . The first task accesses method m_3 of the third component via method m_1 from the first component and m_2 from the second component. The second task does the same, but via method m_4 , m_5 , and m_6 . Because the pre and post conditions of m_2 are the same, one can change the task t_1 to make use of component 3 directly instead of passing through component 2. This results in component 1 and 2 having less permissions.

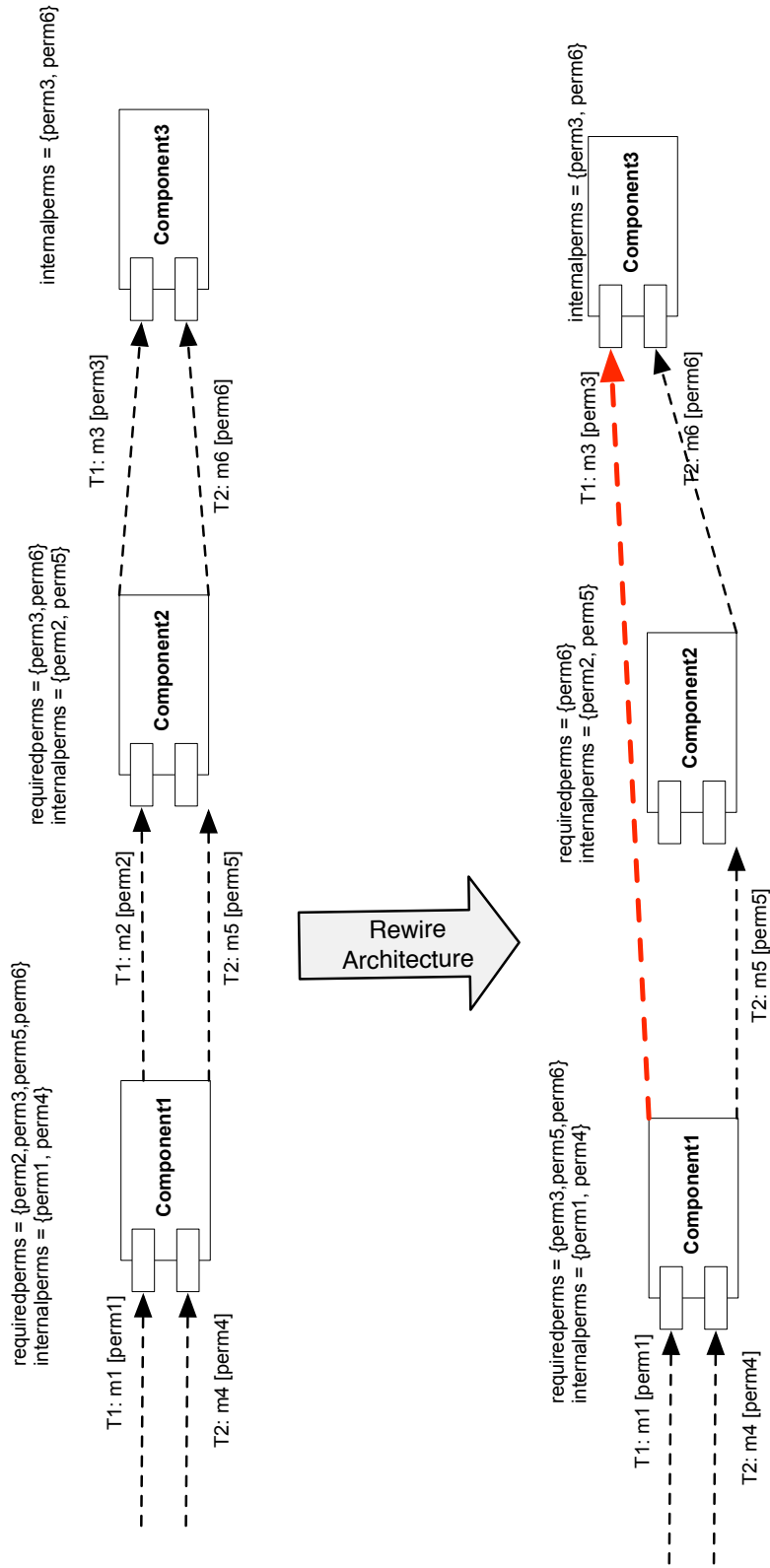


Figure 19: Transformation rewiring: a small case that illustrates that rewiring lowers the overall required privileges.

Proofs Based on the model of section 3, it is easy to see that the number of permissions can reduce for the following reason. Removing a task from a component (possibly) lowers the internal and required permissions of that component (in case they are not required for another task). Adding a task to another component only increases the permissions if that component does not already have these permissions. Hence, rewiring might reduce the number of overall permissions. An outline of the proof is given in Figure 20. The details of the proof are left as an exercise to the reader.

Proof. Assume that the following is given:

$$task = \{(a_1, a_2), \dots, (a_p, a_{p+1})\} \quad (48)$$

$$actions(task) = \{a | a \in actions(s) \wedge x \in actions(s) \wedge ((x, a) \in task \vee (x, a) \in task)\} \quad (49)$$

The rewiring transformation replaces a part from a task with other actions. We assume that the rewiring consists of 2 steps. A first step removes a part of the task, while a second step adds a part to the task.

$$actions(task_{after1}) \subset actions(task_{before}) \quad (50)$$

$$actions(task_{after1}) \subset actions(task_{after2}) \quad (51)$$

We have to proof that there exists components relying on the task being rewired whose number of permissions lower. c_{before} means the component before the task was rewired, while c_{after} is the same component, but after the first step of rewiring.

$$\forall c \in components(c) \wedge task \in tasks(c) | perms(c_{before}) < |perms(c_{after})| \quad (52)$$

The proof is as follows:

$$4 \wedge 50 \Rightarrow |internperms(component_{after})| < |internperms(component_{before})| \quad (53)$$

$$\vee |reqperms(component_{after})| < |reqperms(component_{before})| \quad (54)$$

$$\vee |indirectperms(component_{after})| < |indirectperms(component_{before})| \quad (55)$$

So it is sufficient if we proof 53, 54, or 55.

We proof 53 first.

$$2 \wedge 50 \Rightarrow ownactions(c_{before}) \subset ownactions(c_{after}) \quad (56)$$

$$56 \wedge 15 \Rightarrow internperms(c_{before}) \subset internperms(c_{after}) \quad (57)$$

$$57 \Rightarrow 53 \quad (58)$$

Next, we proof 54.

$$3 \wedge 50 \Rightarrow usedactions(component_{after}) \subset usedactions(component_{before}) \quad (59)$$

$$6 \wedge 59 \Rightarrow reqperms(component_{after1}) \subset reqperms(component_{before}) \quad (60)$$

$$60 \Rightarrow 54 \quad (61)$$

Indirect permissions also lower, because they are based on required and internal permissions.

We also have to proof that the permissions do stay the same if actions whose permissions are already required are added to the task. This is left as an exercise to the reader. \square

Figure 20: Proof that rewiring lowers the number of permissions.

5.9 Moving tasks from one user to another

Moving tasks from one user to another moves a task from a user sets of tasks to another users set of tasks. This transformation is a combination of the remove a task from a user and add a task to a user transforma-

tions. Hence, the number of permissions of the first user decreases, while the number of the permissions of the second user increases. This transformation can solve LP violations, because its subtransformation add a task to a user can solve LP violations.

Transformation If a user is allowed to execute a significant part of the tasks causing a violation, then:

1. Identify a user who can execute the task without causing a conflict. If such user does not exist, create a new one.
2. Move the task to the this user.

Proofs The proof is similar to the proof of remove a task from a user.

5.10 Moving actions from one permission to another

Moving actions from one permission to another moves an action from a permissions set of actions to another permissions set of actions. This transformation is a combination of the remove action from permission and add action to permission transformations. This transformation can decrease the overall number of needed permissions, because the permission granularity is modified. Hence, we assume that it can solve LP violations.

Transformation If two actions used by different tasks used by different users have the same permission, then:

1. Identify a permission of an action used by a first task. If such a permission does not exist, create a new one.
2. Move the action used by the first task to this permission.

5.11 Dealing with multiple solutions

The attentive reader might have noticed that the above strategies can lead to multiple solutions, mainly because a set of actions can be partitioned in multiple ways. Therefore, a strategy is needed that searches the best possible solution.

A naive strategy first creates a tree of all possible solutions by applying all possible transformations on all possible violations in every possible order (See Figure 21). Next, this strategy calculates the best solution based on several architectural metrics such as size, complexity, and number of violating components. Obviously, this strategy finds the correct result, but requires a lot of memory and CPU-time.

Another strategy selects the best solution on the fly by ranking the identified violations based on several metrics. For instance, solve the violation whose solution does not significantly change other architectural qualities such as size or complexity first (See also Section 7). Obviously, this strategy requires less memory and is faster than the first one, but we are not sure that the result is the best one. Indeed, it can be a local optimum in the complete search tree.

For now, we opted for the second strategy. In the future, we should create an algorithm that allows the architect to trade-off between *best* and space or time.

6 Measuring LP Violations

This section sketches metrics that facilitate insight into least privilege at architectural level. These metrics can be used to answer questions such as:

- How many tasks cause least privilege violations?
- How many components violate least privilege?

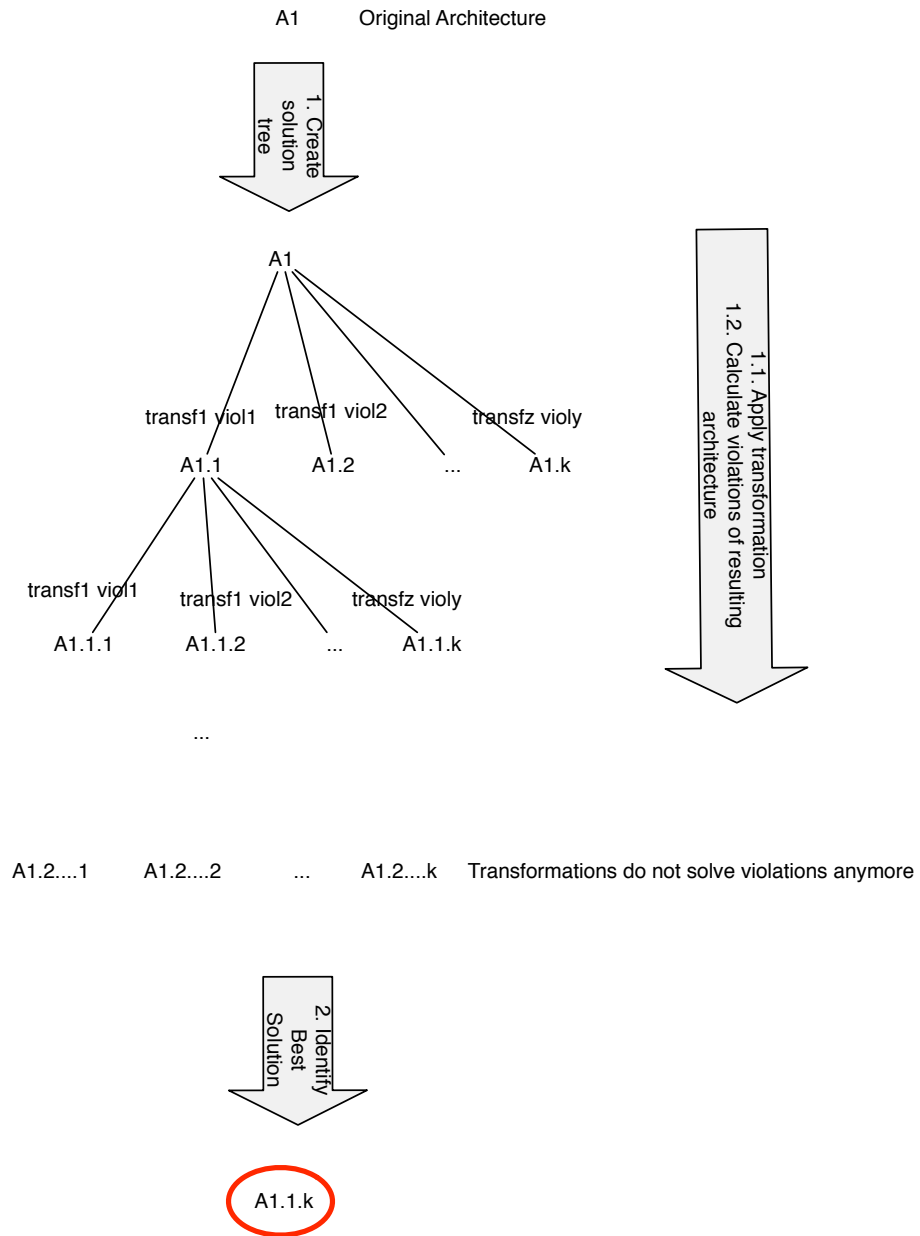


Figure 21: Multiple solutions: a naive strategy creates a tree of all possible solutions. Nodes are architectures. Edges are transformations.

- Is one task responsible for the majority of the least privilege violations or is the problem scattered throughout?
- Is one component responsible for the violation or is it scattered throughout the whole software architecture?
- Which version of a system's software architecture supports least privilege the most?

For each metric, we describe its (i) goal, (ii) in what situations it can be used, and (iii) the metric itself.

Number of violating components The goal of the metric *Number of violating components* is to have an idea how many components of a system are violating least privilege. This metric can be used for comparing two iterations of the same system, but can not be used for two totally different systems. Indeed, it is hard to tell whether a 2-component system with 1 violating component supports least privilege better than a 20-component system with 2 violating components.

```
numberofviolatingcomponents(system) = |violatingcomponents(system)|
violatingcomponents(system) = { c | not(adherestolp(c)) and c ∈ components(system) }
```

Number of violating components per all components The metric *Number of violating components per all components* eliminates the shortcoming of the *number of violating components* metric by normalization. Hence, it can be used to compare systems with a different amount of components.

```
numberofviolatingcomponentsperall(system) =
  numberofviolatingcomponents(system) / |components(system)|
```

Number of violating tasks The goal of the metric *Number of violating tasks* is to have an idea how many tasks are causing least privilege violations. This metric can be used for comparing two iterations of the same system, but can not be used for two totally different systems. Indeed, it is hard to tell whether a system supporting 2 tasks, 1 of which causing a violation, supports least privilege better than a system supporting 20 tasks, 2 of which causing a violation.

```
numberofviolatingtasks(system) = |violatingtasks(system)|
violatingtasks(system) = { t | t ∈ violatingtasks(component) and
  component in components(system) }
violatingtasks(component) = { t | t, x ∈ tasks(component) and
  t, x ∈ executabletasks(component)
  and (not(executabletogether(t, x)) or not(executabletogether(x, t))) }
```

Assumption: a tasks only causes a violation if it is not executable together with another task.

Number of violating tasks per all tasks The metric *Number of violating tasks per all tasks* eliminates the shortcoming of the *number of violating tasks* metric by normalization. Hence, it can be used to compare systems supporting a different amount of tasks.

```
numberofviolatingtasksper100(system) = numberofviolatingtasks(system) / |tasks(system)|
```

6.1 False positives?

The reader might have noticed that these metrics do not take into account our certainty of a potential violation being a real one (See footnote in Section 1). Hence, these metrics also count potential violations that turn out not to be real ones. This might lead to a significant number of false positives.

We assume that required permissions and internal permissions won't lead to a significant number of false positives, while indirect permissions do. Indeed, implemented components will most likely have the required and internal permissions for executing the tasks they are responsible for. However, a component's indirect permissions are a worst-case assignment of what might go wrong with its estimated shared state. So, we could say that the certainty of a potential indirect permission based violation being a real one is lower than the certainty of a potential required or internal permissions based violation being a real one.

In order to reduce this number of false positives, one can split the different types of permissions and work with weighted violation detection.

7 Applying the results on several case studies

This section presents the validation results of the presented approach. In particular, the goal was to assess whether the LP properties of the software architecture improved, while other software architecture qualities such as size, complexity, or security properties did not deteriorate. The evaluation was based on an implementation of the identifying and solving techniques described earlier.

In the rest of the section, one particular case study is first elaborated upon in order to appreciate the type of problems and solutions that can be addressed in practice. Afterwards, a summary of the results of applying the approach to several case studies will be discussed from a broader perspective.

7.1 Detailed validation results

The case study used in this section involves a subset of the requirements and architecture of a digital publishing system. The complete description is available in [33]. The system automates the cross-media publishing workflow of a corporate publishing company. Its main features are input management, user management, and content management and distribution.

A wide range of different actors make use of this system, among which the advertiser, the journalist, and the manager. The *advertiser* is the main income source of the company because he buys commercial space or time. The *journalist* forms the bridge between consumers and producers by using the publishing system to distribute finished content. The *manager* manages the publishing company by creating a publishing strategy and assigning tasks to journalists.

Obviously, the execution of each of these tasks is restricted to certain actors. For instance, a journalist is not allowed to create a corporate strategy, because that requires too much responsibility.

In the architecture, components responsible for these features are the following. An Media Advertising System (MAS) is used by Advertisers to submit produced commercials to be stored in the Content Management System. The Content Management System (CMS) is responsible for storing and retrieving content items, and a Journalist News Desk is responsible for making content ready to be published. The Planning System (PS) is used by both Journalists and Managers to manage their planning and assign tasks.

We discuss some of the problems that have been identified in the Planning System. A first problem is that the Planning System is responsible for the *plan corporate strategy* and *plan edition strategy* tasks. The former is executed by a manager, while the latter is executed by a journalist. Having permissions for both tasks was explicitly forbidden by the company policy (as illustrated in Figure 59).

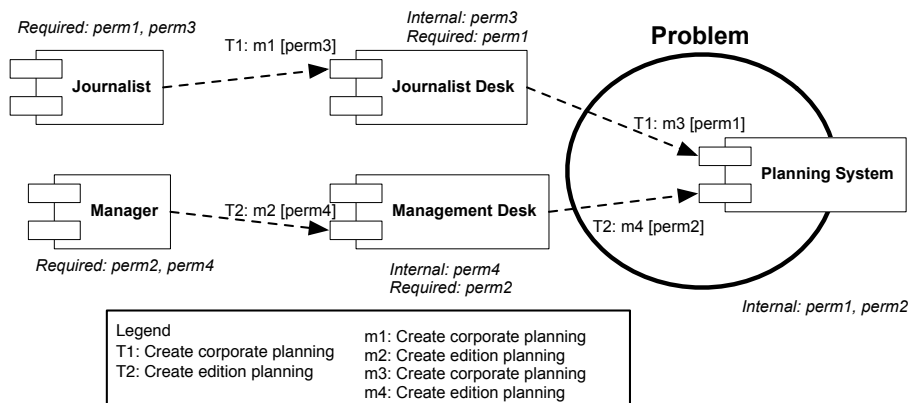


Figure 22: A violation caused by too many internal permissions.

A second problem is that the advertiser is able to obtain enough permissions to modify the advertisement workflow (part of planning tasks) (See Figure 23). The design problem is that planning system is responsible for both *notification* and *edition planning* tasks.

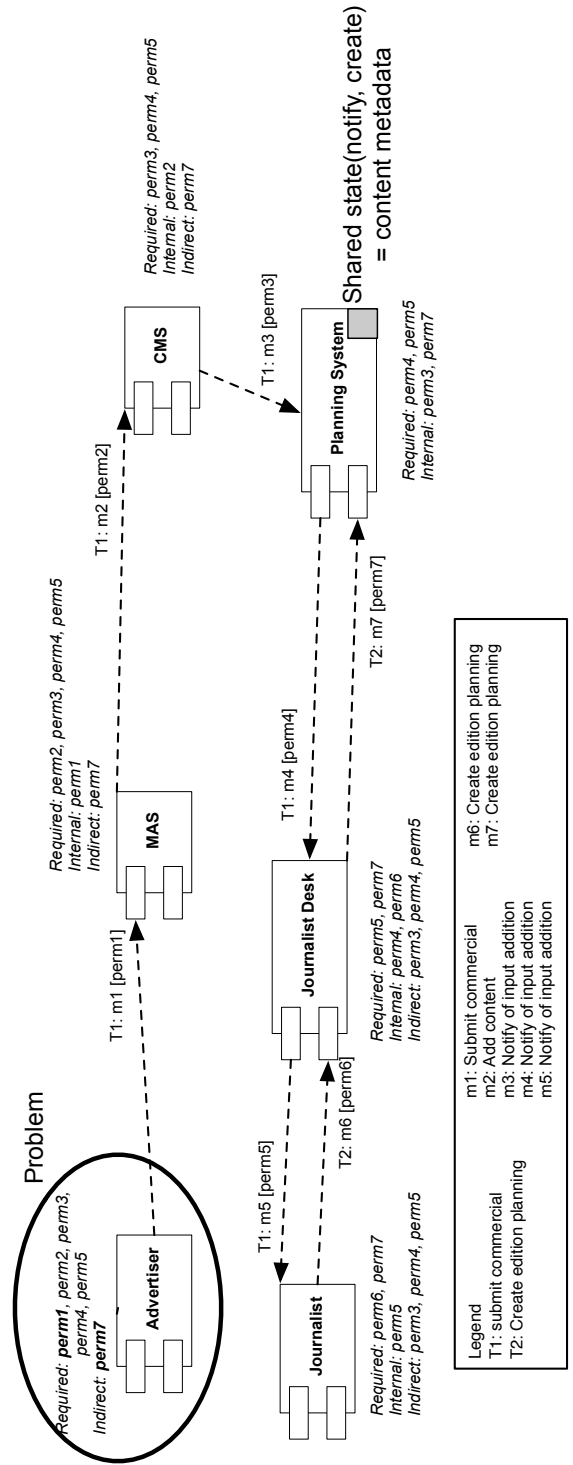


Figure 23: A violation caused by too many indirect permissions.

A solution to these problems can be found by applying the transformation described in Section 5. The first problem can be solved by splitting the component in two parts. The first part contains actions related to corporate strategy, while the second part contains actions related to edition planning. The second problem can be solved by decoupling notification from creation. As such, the risk of LP violations in the final software architecture will be greatly reduced.

7.2 Broader validation results

In this section, the quality of the presented approach is assessed quantitatively. For this purpose, the impact of our algorithms on size, complexity, and security was measured. Size and complexity were selected because our solution strategy impacts these explicitly: it creates new components, introduces additional dependencies, and so forth. Size was measured by the number of components, the number of interfaces per component, and the number of actions per interface, while complexity was measured by CBMC [34], connector complexity [62], and McCabe [38] for architecture. *Security* was selected because our strategy should improve the LP security properties of a software architecture, but not invalidate or deteriorate other security properties such as the size of the attack surface. LP was measured by the number of violating components, and the number of tasks causing the violation, while the attack surface was measured by a (simplified version of a) metric defined by Manadhata [36].⁸ Table 3 presents the results of the application of the approach on the three architectures from three different domains: a modified version of the small chat application example delivered with ArchStudio[18] (case1), a conference management system [45] (case2), and (a subset of) the publishing system [33] (ps sub and ps full). Details are available in Appendices A, B, and C.

Metric Case	Size			Complexity		Security		
	#comp	#inf /#comp	#acts /#inf	#tasks	Mccb.	#viol comps (indirect)	# viol tasks (indirect)	attack surface
case 1 before	3	1.67	1.2	2	3	1 (0)	2 (0)	
case 1 after	4	1.5	1	2	4	0 (0)	0 (0)	same
case 2 before	7	1.0	2.9	11	4	6 (5)	11 (8)	
case 2 after	9	1.1	2.1	11	6	8 (6)	11 (8)	larger
ps sub before	13	2.38	2.65	8	6	4 (0)	6 (0)	
ps sub after	20	2.6	1.58	8	7	0 (0)	0 (0)	same
ps full before	13	2.38	2.64	22	8	13 (12)	22 (16)	
ps full after	13	2.38	2.64	22	8	13 (12)	22 (16)	same

Table 3: Measurements of the cases in the three domains and three variants of the publishing case.

The first analysis examined whether system size (#comp) worsened after the application of our approach. In general, this was indeed the case. While the smaller case studies grew in size with 1 or 2 components, the publishing case almost doubled in size. Indeed, large systems might require more conflicting permissions to solve, because (i) their components have more actions, and (ii) they support more tasks. The increase for small systems is still acceptable, while the increase for the large system is not. Note, however, that this number can be reduced by lowering the number of false positives in the violation set or by improving the architectural transformations.

The second analysis examined whether component size (#inf/#comp and #acts/#inf) worsened after the application of our approach, which was not the case. Indeed, if a component is split, the number of actions per interface decreases as a subset of these are moved to another component.

The third analysis examined whether complexity (Mccb.) increased. In general, this was the case. A possible explanation for this is that dependencies, one of the main parameters of complexity, between the old set of components and the newly created components are introduced. This increase in complexity is considered acceptable.

⁸The original metric requires pre- and postconditions associated with the actions, which our case studies do not provide. Furthermore, two of the metric's parameters, trusted data items and direct and indirect entry points, are estimated by making use of the tasks (data flow).

The fourth analysis examined whether LP improved (in terms of the number of violating components (# viol comps), and tasks causing a violation (#viol tasks) after the application of our approach. In general, LP did improve. However, in two cases (case2 and ps full) an improvement was not noticed at all. This might be caused by the inability of the transformation to identify a set of subcomponents that each contain a different violating action. These numbers are remarkably high, which might indicate, among others, that the approximation of indirect permissions produces a significant number of false positives.

The fifth analysis examined how the attack surface was impacted. In general, this was not the case, which is plausible since two of the main parameters of attack surface, indirect entry points and untrusted data items, are not influenced by the splitting transformation. However, in case 2 the attack surface increased, because it increases the number of indirect entry points by creating shared state update methods based on existing indirect entry points, which count as indirect entry points as well.

In conclusion, we could say that our detection algorithm detects least privilege violations, but has a high rate of false positives. Our splitting transformation works if the components that have to be split, are divisible in subcomponents, which mainly depends on the size of the shared state.

8 Discussion

A number of observations driven by the results of our experiments are worth further discussion. The identified approach and transformations have at least the following limitations.

The identified transformations do not always work well: sometimes it is not possible to split the component in subcomponents, because two violating tasks both use an action or a group of actions update shared state. Hence these actions can not be part of two components. Possible solutions to this problem are (i) using smarter ways for creating subcomponents (e.g., another shared state identification algorithm), and (ii) creating new transformations that are not based on splitting architectural elements to solve LP violations, like splitting tasks.

On the other hand, one should also carefully ponder the options that influence the result of the identification and solving algorithm, since naively applying it might lead to extreme architectures or might cause unwanted side effects. An extreme architecture is for instance an architecture in which every component has been split (possibly several times). It is clear that such architectures are not useful in practice. Note also that sometimes a component should not be split when many tasks pass through these components on purpose. Some security-specific components (like the Audit Interceptor [3]) exhibit this type of behavior.

Some current assumptions considerably limited the practical applicability of the approach. First, a process typically consists of multiple components, while we assume that it only consists of one. This assumption can be dropped if other architectural views such as the process or runtime view is incorporated in our model. Second, not all security relevant use cases are typically available, while we assume they are. Furthermore, there are also two current limitations in the current implementation tool: (i) no support for UML (useful for reading architectural diagrams), and (ii) no meaningful names for newly created components and interfaces.

An interesting added value of the approach is the ability to reason about separation of duty policies at architectural level. Indeed, separation of duty (SoD) deals with splitting and distributing tasks that, when combined, can cause a lot of damage. SoD can be seamlessly integrated in the approach as specific constraints that are imposed on particular tasks. As such, one can easily reason about SoD conflicts at architectural level.

9 Related Work

This work is strongly related to three research domains: security engineering, software refactoring, and model checking. The remainder of this section surveys and relates work in each of these domains to our work.

Security engineering is about building systems to remain dependable in the face of malice, error, or mischance. As a discipline, it focuses on the tools, processes, and methods needed to design, implement, and test complete systems, and to adapt existing systems as their environment evolves [4]. In other words,

this domain tries to build dependable systems by using a plethora of techniques. Hence, there is a lot of related work in this area. Therefore, we limit our discussion to the context of secure engineering processes, and further focus on methods related to least privilege: (i) program separation, and (ii) execution monitoring.

Secure development processes such as SDL [26], Touchpoints [39], and CLASP [48] combine and glue various current practices such as threat modeling, risk management, or secure coding into an integrated and more comprehensive construction method. The relation with our approach is threefold. First, the meaning of one of the design principles mentioned in those processes, namely least privilege, is made explicit [10]. Hence, (in)experienced designers using our interpretation of the principle may apply (part of) these processes more correctly. Second, the meaning is extended to the other lifecycle phases of these processes, such as architectural design. Third, the use of our tool reduces the cost of applying these secure development processes, because less time has to be spent on (design) activities.

Program separation, a technique to separate a program in multiple processes with clean interfaces, has been successfully applied in several end-user programs such as Vsftp [12], qmail [7], and postfix [59] to support LP. The approach presented in this report actually provides a systematic and automated means for program separation at architectural level. Another more general approach is privilege separation [9][32], a technique that partitions the implementation of an existing program into two processes: a privileged program called the monitor and an unprivileged program called the slave. Our approach can be considered an extension of privilege separation, because it optimizes the number of privileged processes for LP.

Execution monitoring is another technique that limits the privileges a program is allowed to have. These techniques block system calls and/or access file and network resources based on policies. Examples are Systrace [51], Mapbox [1], BlueBox [11], Consh [2], Janus [60], Jain and Sekar's system call interposition [30], Peterson's general purpose API [50] and Walker's domain type enforcement [61]. The main drawback of these mechanisms that it is hard to specify policies in terms of application-specific resources and functions, because these resources and actions don't always map on files and system calls (application-level versus system level-view). Schneider confirms this by arguing that these systems can only enforce LP in a meaningful way if they use application-dependent security policies that depend on the application's state along with the semantics of that applications abstractions [55]. Another drawback is that one component, the sandbox, is assigned a lot of privileges.

Software refactoring is the process of improving the internal structure of a software system without disrupting its external behavior [20]. This improvement of the internal structure can be based on a specific quality goal, such as modifiability or security. Security can be for instance the support for least privilege. A lot of work has been published in this area. Therefore, we limit this discussion to refactoring of and transformations on architectural artifacts.

Several papers have surveyed software refactoring [44, 21, 41, 42, 17]. Mens' proposed approach for refactoring consists of the following steps: (i) identify where the software should be refactored, (ii) determine which refactorings should be applied, for instance refactoring bad smells such as code duplication to improve maintainability [20], (iii) guarantee that the applied refactorings preserve its behavior, for instance by using pre and post conditions to ensure that the input and output values are the same after the refactoring [47], (iv) apply the refactorings, (v) assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort) using metrics, statistics, and controlled experiments, and (vi) maintain the consistency between the refactored software elements and other software artifacts. Czarnecki's taxonomy [17] organizes transformations according to eight areas of variation. It differs from Mens by more detailing *apply refactorings* and adding tracing, and rule composition. Apply refactorings consists of a rule (transformation) application strategy that determines the order in which the rule gets applied: there may be more than one match for a rule within a given source scope; and a rule scheduling strategy that determines the order in which individual rules are applied. These transformations are expressed in several formalisms, among which: graph transformations [23], XSLT [14], transformation framework that is defined in the Common Warehouse Model [46], text-based tools like awk and perl, Gen-Gen, Mercury [21], and F-logic [21].

Our proposed approach refactor a software architecture for least privilege. Hence, we can relate to the general refactoring steps as follows. The identification of where software should be refactored corresponds to identifying the least privilege problems. Determining the rules to apply corresponds to our rule selection algorithm. Guaranteeing behavior preservation is implicitly part of our rules. Applying refactorings is

equal to applying the rules. Assessing the effect of the refactorings on quality characteristics is something we plan to do in the future. Maintaining the consistency with other artifacts is out of scope for this report. In the future, we should use an existing formalism for expressing our transformations.

Model checking is a collection of techniques for the automatic analysis of systems interacting with their environment [15]. Such techniques are typically automatic, model-based, and property-verification approaches. Our approach is a model checking technique as it uses a predicate logic model for automatically verifying the property of least privilege. A lot of work has been published in this area as well. Therefore, we focus on (i) a general overview of modelling formalisms, and (ii) formalisms used for verifying least privilege.

Modeling formalisms are typically based on a formal logic language [43]. Predicate logic can be used for micro-models that do not use the concept time [28], while linear and branching-time temporal logic can be used for more complex models taking time into account.

Examples of predicate logic based languages include (but are not limited to) Alloy, Z, and UML with OCL constraints. Alloy [29] is a formal specification language based on first-order predicate logic. It finds models which form counter examples to assertions made by the user. This language is supported by the tool Alloy Analyzer that allows the user to generate instances of invariants, animate the execution of operations, and check user-specified properties. Z [57] is a formal specification language used for describing and modeling computing systems. It is based on first-order predicate logic and lambda calculus. The language is supported by the tools ZLive and ProZ. An UML model that is annotated with properties expressed in OCL constraints can be used to verify these properties [53]. Aydal and his colleagues [5] compare tools for the above formalisms by modeling and validating the same toy example in all languages (and tools).

Examples of tools supporting linear-time or branching-time temporal logic based languages include Spin and NuSMV. Spin [25] is a generic verification system that supports the design and verification of asynchronous process systems. Its models are focussed on process interactions. NuSMV [13] supports the SMV language, finds counterexamples, supports fair model checking, and computes quantitative characteristics of the model.

Formalisms that can be used to verify least privilege include UMLSec. In his PhD thesis [31], Jürjens explains how one can use his UMLSec approach to enforce LP by formulating LP requirements and verifying UMLsec specifications (including policy specifications) with respect to these requirements. The major difference with this approach is that is functions independently from the access policy. Thuong Doan's UML model checking approach is similar [19]. Rubacon [24] is a tool that checks UML models and their configuration data for adherence to security policies. The tool allows to express security rules that the permissions need to satisfy in a formal way. An example of such a security rule can be separation of duty. In essence, the tool investigates chains of information flow, such as determining user permissions from user names or roles. Rubacon and our work share a similar idea: identify possible (sub)tasks (transactions) that can be executed by granted permissions. However, the authors focus more on the end user having too much permissions, while we focus on architectural components having too much permissions. In addition, we propose rules for restructuring the architecture in order to be able to enforce least privilege (and separation of duty), while their tool only detects violations. On the other hand, their tool allows to verify more rules than our least privilege rule, by taking into account configuration data.

10 Conclusion

This report proposes a technique that improves the identification and resolution of LP violations in software architectures. To this aim, the concept of architecture-level LP has been modelled formally. This model was used to create an algorithm that indicates when an architecture violates LP. Subsequently, architectural transformations that solve a subset of these violations was proposed. The approach has been validated by means of several case studies which indicate that, overall, properties such as the number of violations and the average component size improve, while other software properties such as system size, system complexity, and attack surface are negatively affected.

While this work is a first milestone in this context, many opportunities and issues remain. The focus of (near term) future work will be threefold. First, the false positive rate should be further reduced such that the outcome of the violation identification algorithm is more usable. A first idea in that direction is to

split the different types of permissions and work with weighted violation detection. Second, other architectural views (e.g., process view) can be included in order to make the technique more applicable. Finally, architectural refactorings that preserve the architectural semantics, possibly aided by special-purpose architectural annotations, can be significantly improved. In that context, the alternatives described in the report will be further studied, implemented and tested thoroughly.

Subject	Task
home user	send message to home user
home user	receive message from home user
business user	send message to business user
business user	receive message from business user

Table 4: A chatting policy stating that home users are not allowed to view business users' messages.

A Case study: Archstudio Chat System

This Section applies the presented identification and solving algorithms on a modified version of the small chat application example delivered with ArchStudio[18]. This application illustrates a limitation of our model and proposes ideas for illuminating it. The restriction is that the identification algorithm does not work if it can not distinguish security relevant tasks.

This case study is structured as follows. First, Section A.1 introduces the application domain. Next, Section A.2 gives an overview of the software architecture, and the most important sequence diagrams. Finally, Section A.3 briefly presents and interprets the results of our experiments and Section A.4 concludes.

A.1 Introduction

A synchronous conferencing system is a chat system that automates real-time textual one-to-one and group-based communication. Such a communication tool is useful for a wide range of different users. The *End User* utilizes the system to informally chat with other end users, while the *Business User* uses the system to discuss a wide range of possibly sensitive topics in group. Due to this sensitive nature, home users should not be able to read these discussions (See policy in Table 4).

A.2 Software Architecture

ArchStudio's chat application consists of three components: a chat server and two chat clients. Each chat client is an instance of the same chat client component. The chat client is used by a user to communicate with other clients via the chat-server. The client contacts the chat-interface of the server to send messages to other clients. This server forwards this message to the addressees by contacting their chatevents-interface. This is illustrated in Figures 24, and 25.

We modified the example to make it more interesting in the following way (See Figure 26). Suppose we have 2 groups of 2 clients each. The first group is a group of home users, while the second group is a group of business users. Each group uses the chatserver to communicate with other members of its group. The constraint we imposed do not allow home users to read business users' communication. In other words, the Chat Server component has too many permissions if it is responsible for both business and home users.

A.3 Experiments

This Section briefly presents the application of our identification and solving algorithms on two versions of the given software architecture. In the first version, we assigned internal permissions in a naive way, while in the second version we assigned these in a more clever way.

A.3.1 A first version

Formally modeling the component and connector diagram (See Figure 26) is rather straightforward. Therefore, we focus on the non-trivial elements of the model, namely internal permissions and the least privilege policy. Each action defined in Figure 25 is associated with precisely one internal permission. For instance, *send message* is associated with permission *perm1*. The least privilege policy contains the send message task that each user is allowed to execute (see policy in Table 5).

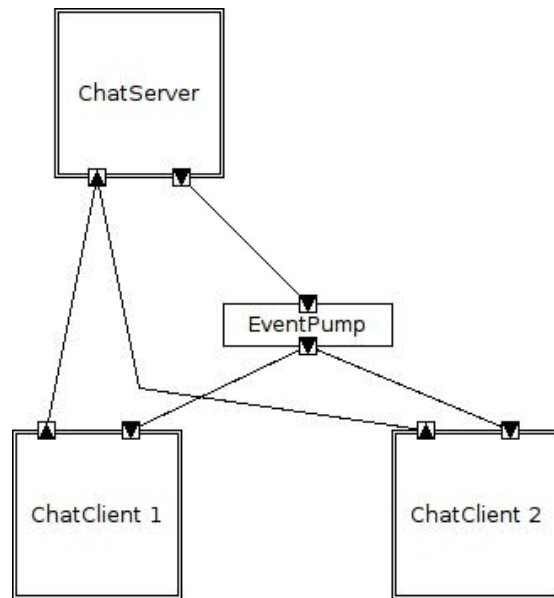


Figure 24: The component and connector diagram of the example chat application.

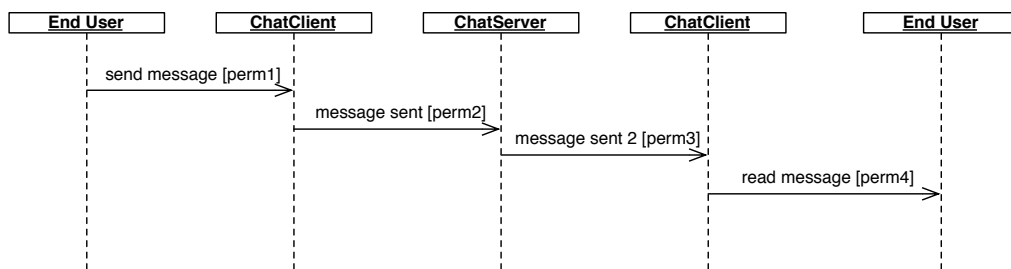


Figure 25: The sequence diagram of the send message use case.

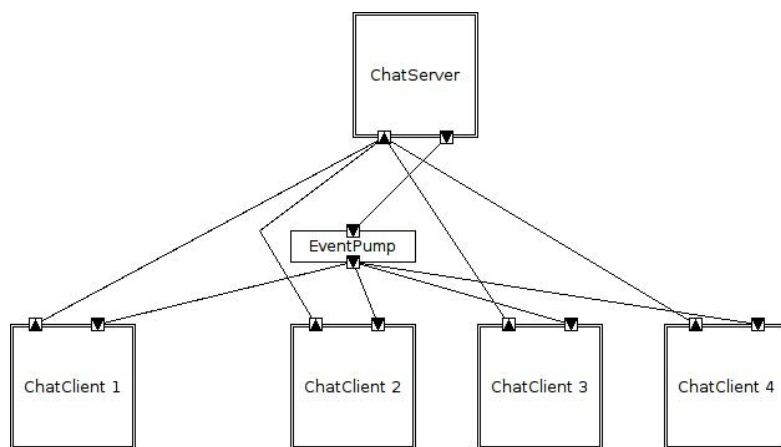


Figure 26: The component and connector diagram of our extension of the example chat application.

Subject	Task
Home User	Send Message
Business User	Send Message

Table 5: Policy stating what tasks each user is allowed to execute.

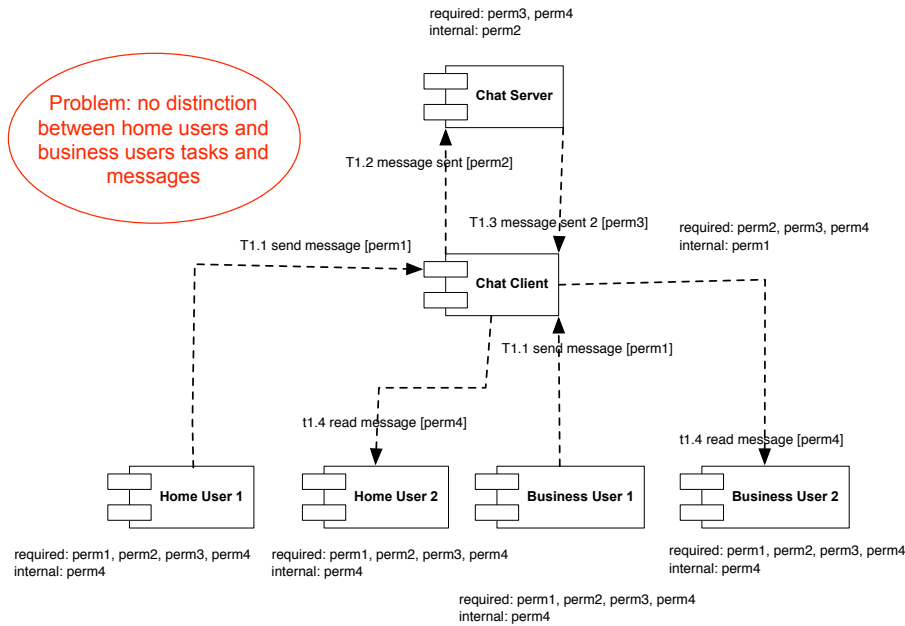


Figure 27: The identification algorithm fails to identify a violation because it can not distinguish between the business and home user tasks.

The identification algorithm determines the required and indirect permissions based on the internal permissions. The required permissions for the *Send Message* task are the following: perm1, perm2, perm3, and perm4. There are no indirect permissions, because there is no shared state.

The identification algorithm should indicate that the Chat Server-component has too many permissions, because it is responsible for home user as well as business user tasks. However, the algorithm fails to do so, because the model is not accurate enough (See Table 6 and Figure 27). Indeed, two tasks causing the violation are represented as one task. The following section will try to solve this shortcoming.

A.3.2 A second version

This version suggests three modeling choices for solving the violation identification failure, namely (i) renaming the client components to business client components and home client components, (ii) making use of the deployment and/or process view, and (iii) analyzing data by forward reasoning.

Renaming the client components to business client components and home client components allows enforcement of the policy because a distinction between the *send message*-task that transfers business messages and the one that transfers home messages can be made.

Using the deployment view and/or the process view allows us to enforce the policy because we can make a distinction between business processes and home user processes, and thus also between a send message task that sends business user's messages and one that sends home user's messages.

Manually annotating data that is considered to be privileged, also allows us to enforce the policy.

For now, we focus on the first option, because the second option requires extending our model with a process view, while the third option is considered a quick hack. Application of the first option is illustrated

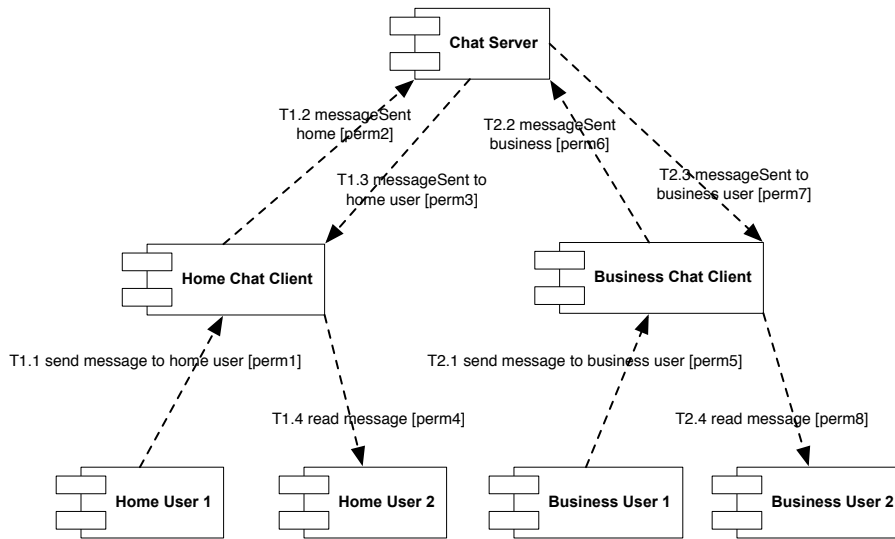


Figure 28: This Figure illustrates how the architecture would look like if we modify the formal representation of it.

in Figure 28.

The least privilege policy contains the send message task that each user is allowed to execute and is as follows.

Subject	Task
Home User	Send message to home user
Business User	Send message to business user

The identification algorithm identified the least privilege problems by using the formal model, and the list of user-task assignments (See Table 6). In short, there was 1 violating component, namely the Server Component.

This identification was done in two steps. First, required and indirect permissions based on the internal permissions were determined. The required permissions for the *Send message to home user* task are the following: perm1, perm2, perm3, and perm4. The required permissions for the *Send message to business user* task are the following: perm5, perm6, perm7, and perm8. There are no indirect permissions, because there is no shared state. Second, these permissions were used to determine that the server component is responsible for business users and home users. This is illustrated in Figure 29.

The solving algorithm identifies a solution by applying the rules on the least privilege violations. In this case, there is only one solution. The Chat Server is split in a business chat server and a home chat server, which is illustrated in Figure 30.

Metric Case	Size			Complexity		Security		
	#comp	#inf /#comp	#acts /#inf	#tasks	Mccb.	#viol comps (indirect)	# viol tasks (indirect)	attack surface
version 1 before	2	1.5	1.0	1	2	0 (0)	0 (0)	same
version 1 after	2	1.5	1.0	1	2	0 (0)	0 (0)	same
version 2 before	3	1.67	1.2	2	3	1 (0)	2 (0)	same
version 2 after	4	1.5	1.0	2	4	0 (0)	0 (0)	same

Table 6: Measurements of the two versions of the chat system case. User components are not included.

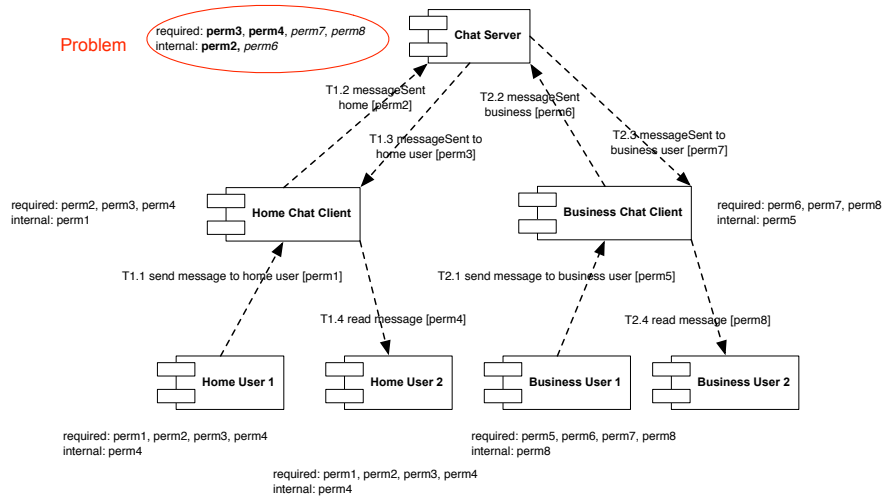


Figure 29: The Chat Server has too many permissions and thus privileges.

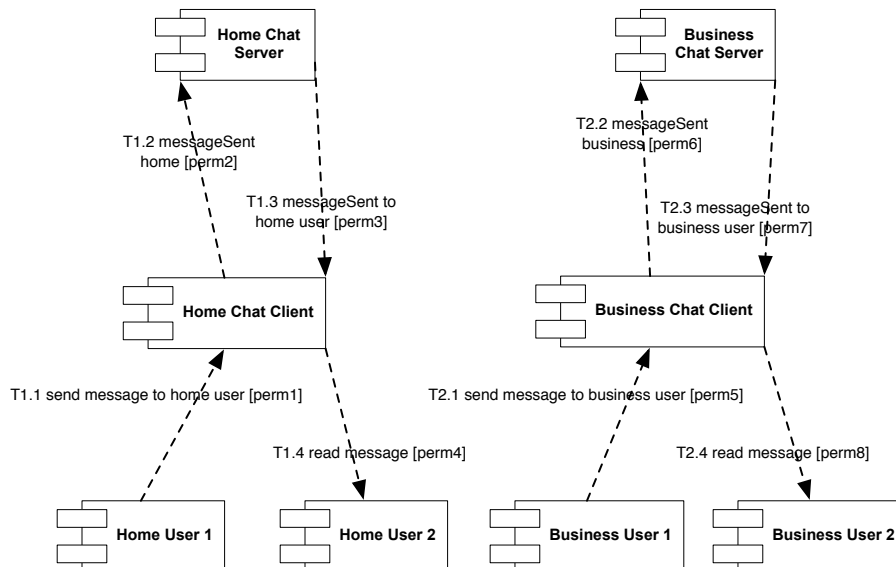


Figure 30: A possible solution: the Chat Server has been split.

A.4 Conclusion

A modified version of the small chat application example delivered with ArchStudio[18] illustrates a limitation of our model and proposes ideas for overcoming these limitations. The restriction is that identification algorithm does not work if it can not distinguish security relevant tasks. This was solved by renaming those tasks and corresponding permissions.

Review Process	Setup	Submission	Selection	Announcement
Setup CFP	■			
Invitation	■			
Abstract Submission		■		
Bidding		■		
Full paper submission		■		
Assignment			■	
Reviewing			■	
Revision			■	
Notification				■
Camera Ready				■

Table 7: The review process as presented in the confious conference management system[49]. Phases are ordered according to the time when they are executed. A light gray activity is an optional one.

B Case study: conference management (agent-based and openconf)

This Section applies the presented identification and solving algorithms on two software architecture versions for conference management: a conceptual agent based one [45], and a reversed engineered version of a real-world implementation (openconf) [22].

This case study is structured as follows. Section B.1 introduces the application domain. Next, Sections B.2 and B.3 elaborate on the application of the identification and solution algorithms on the agent based system and openconf respectively. For both cases, an overview of the software architecture, the most important sequence diagrams (Sections B.2.1 and B.3.1 respectively), and an presentation and interpretation of the results of our experiments is given (Sections B.2.2 and B.3.2 respectively). Finally, Section B.4 concludes.

B.1 Introduction

A conference management system is a system that automates (parts of) a scientific peer review process. Such a process is performed by a number of people in order to submit, select, and prepare papers to be published.

There are at least three user roles that execute the process. The Program Committee Chair (or Chair) is responsible for the coordination of the peer review process tasks. The Program Committee Member (or Reviewer) evaluates the quality of a submitted paper that is in his area of expertise. The Contact Person (or Author) submits papers of research.

The process (see Table 7) consists of several activities organized in four phases[49, 37]. The initiation phase involves sending out the call for papers, and optionally asking reviewers whether they are willing to review (invitation). In the submission phase the authors first submit the abstract of their paper. Next the reviewers optionally bid on the papers they would like to review. Finally, the authors submit the full paper. In the selection phase, reviewers are assigned papers to review, and submit reviews about these papers to the system. Based on these reviews, the program committee (reviewers and chair) selects the best papers (revision). In the announcement phase, the authors are notified of acceptance. Next, the selected papers' authors are requested to submit an improved camera ready version of their papers.

Obviously, each (group) of these activities have constraints imposed on them. For instance, an author is not allowed to review papers he has written, a chair is not allowed to submit papers to a conference he organizes, a paper has at least two associated reviews before best paper selection starts, and the author only receives reviews about his own paper. These constraints are expressed in the policy in Table 8.

This policy is used to apply the identification and solving algorithms on two applications: an agent-based architecture, and openconf. In order to be able to use this policy as input, we have to map it to the tasks both systems support.

Subject	Task
Chair	Send call for papers
Chair	Send PC Invitation
Chair	Assign submission responsibilities
Chair	Read submission responsibilities (assignment)
Chair	Rank and select paper (revision)
Chair	Send notification
Author	Submit abstract
Author	Submit paper
Author	Submit camera ready
Author	Read review
Reviewer	Bid paper
Reviewer	Read paper
Reviewer	Read abstract
Reviewer	Submit review

Table 8: A simplified policy for the review process: an author is not allowed to be chair or reviewer.

B.2 An agent-based architecture

This section applies our identification and solving algorithms on the conference management system architecture presented in [45]. The paper’s architecture supports the concepts and tasks defined in Section B.1 to a great extent. However, the paper also introduces additional tasks and one additional actor: the publisher who publishes proceedings. The additional tasks are *coordinate conference*, *support paper submission*, *deal with reviews*, and *get proceedings*. We refined and extended the architecture ourselves in case the documentation lacked detail for successfully completing our experiments. For instance, tasks that were not explicitly supported and straightforward to support, were added by us.

B.2.1 Software Architecture

This system consists of four actors interacting with four components (See Figure 31).

The **Conference Manager** is used by the chair for conference coordination. This component has one interface: Conference Manager Interface. This interface contains several methods, among which send call for papers, send program committee invitation, send notification, and assign submission responsibilities.

The **Paper Manager** aids several actors in the paper review process. The author uses this component to submit papers, while the other actors use it to obtain a version of the submitted paper. The Paper Manager has the following interface: Paper Manager Interface. This interface contains several methods, such as support paper submission, manage submissions, collect finals, and get papers to review.

The **Proceedings Manager** is mainly used by the publisher to create conference proceedings. It has one interface: Proceedings Manager Interface. This interface contains methods like deal with proceedings, and get proceedings.

The **Review Manager** is mainly used by the reviewer to submit reviews. It has only one interface: Review Manager Interface. This interface offers several methods among which deal with reviews, and collect reviews.

We show how the system supports the tasks presented in Section B.1 by mapping the tasks onto the component diagram. A selection of tasks is presented as sequence diagrams.

Chair The chair delegates several tasks to the system.

- Send call for papers. (See Figure 32)
- Send PC invitation.

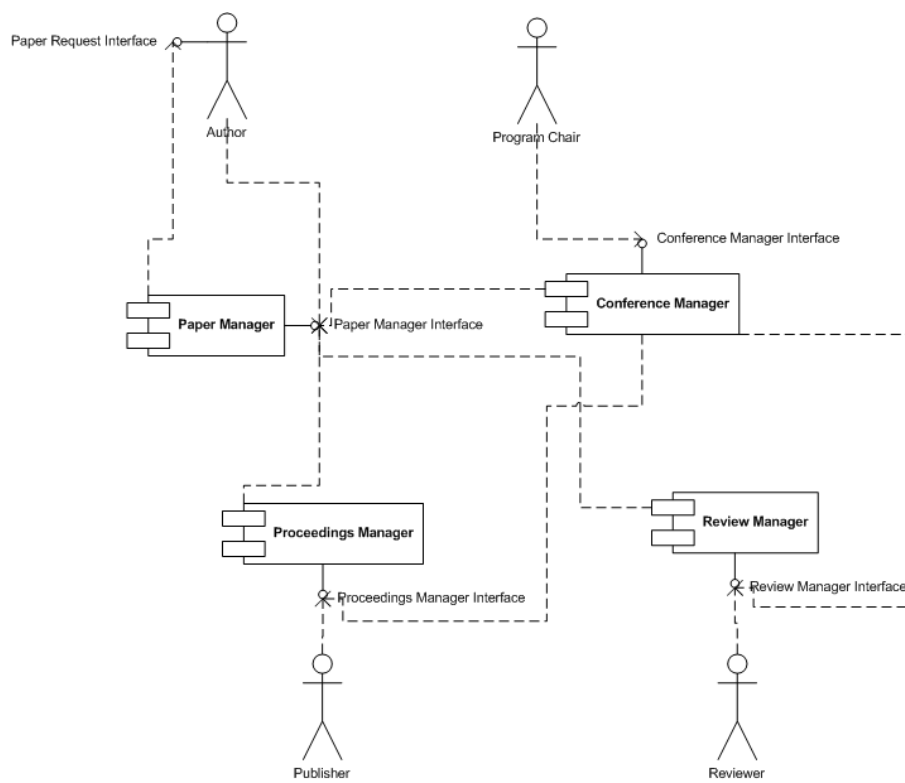


Figure 31: Agent-based CMS: component diagram of the software architecture presented in [45] extended with interfaces.

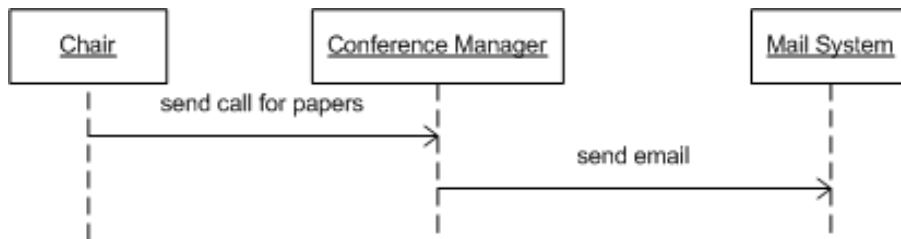


Figure 32: Agent-based CMS: sequence diagram of send call for papers.

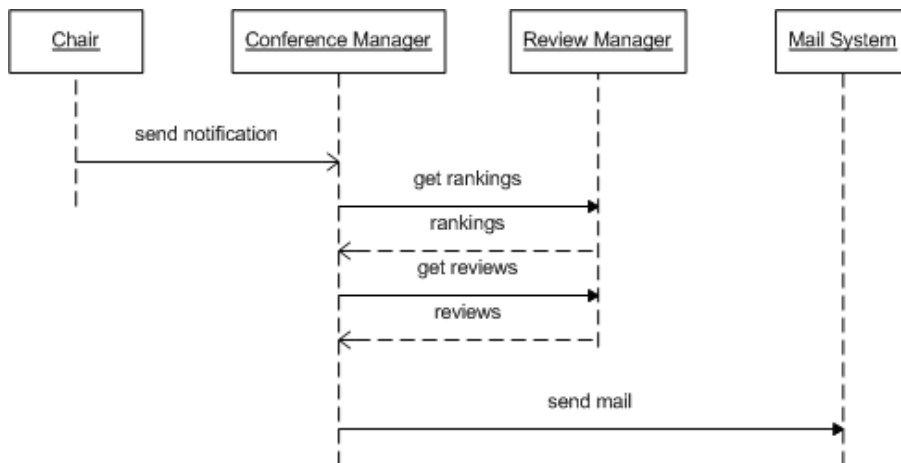


Figure 33: Agent-based CMS: sequence diagram of send notification.

- Send notification. (See Figure 33)
- Assign submission responsibilities. We assume that this task is automated by the *Paper Manager* as it has a *get papers to review* method.
- Rank and select papers. We assume that this task is automated by the *Review Manager* as it collects all the reviews.
- Coordinate conference. This task is composed out of the previous tasks.

Author The author delegates several tasks to the system.

- Submit paper. (See Figure 34)
- Submit abstract.
- Submit camera ready.
- Read review. This is not part of the system, as the author receives the reviews by the notification mail.

Reviewer The reviewer delegates several tasks to the system.

- Read paper. (See Figure 35)
- Read abstract.

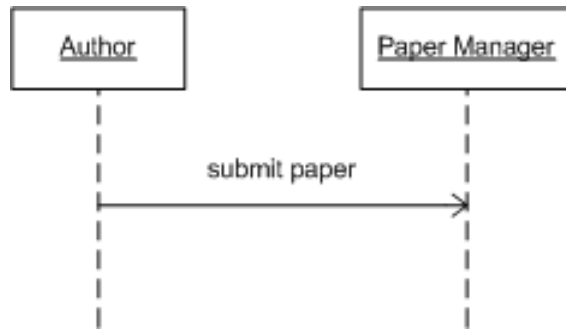


Figure 34: Agent-based CMS: sequence diagram of submit paper.

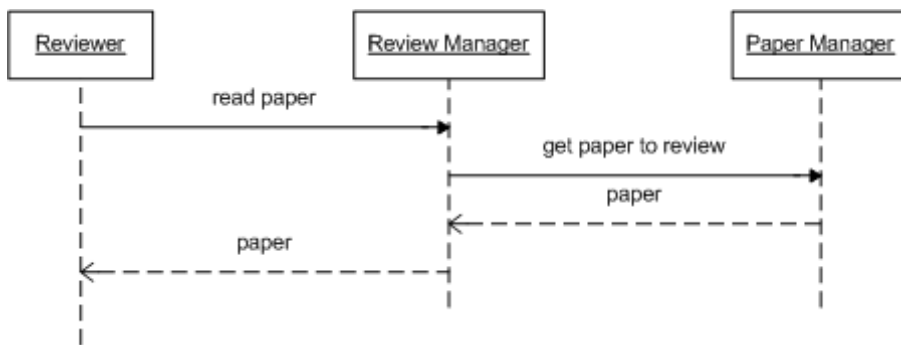


Figure 35: Agent-based CMS: sequence diagram of read paper.

- Bid paper, and assign submission responsibilities are not explicitly supported by the system. However, the *get papers to review* method provided by the *Paper Manager*-component indicates that this component knows what papers the reviewer has to review. Therefore, we assume that this component is responsible for this task.
- Submit review.
- Deal with reviews is composed out of the above tasks.

Publisher The publisher uses the system for only one task, namely get proceedings. This task is illustrated in Figure 36.

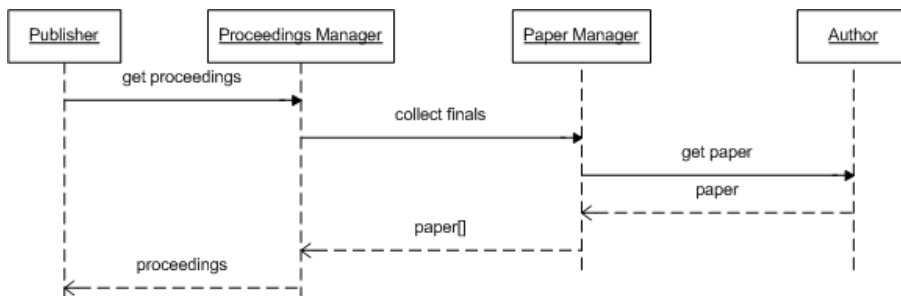


Figure 36: Agent-based CMS: sequence diagram of get proceedings.

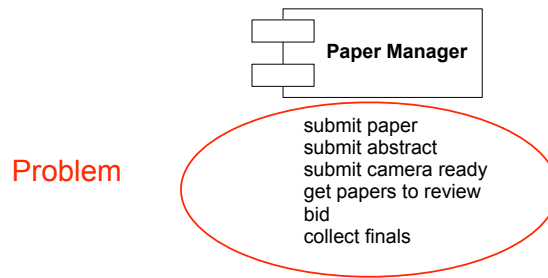


Figure 37: Agent-based CMS problem: Paper Manager has too many permissions.

B.2.2 Experiments

Presenting the component and connector diagrams in a formal way (See Figure 31) is rather straightforward. Therefore, we focus on two non-trivial elements of the model: composed tasks and components initiating a task.

A first non-trivial element to model is a component and not an end-user being responsible for a task. For instance, the Paper Manager automatically assigns reviewers to papers. By consequence, it seems impossible to determine on what users's behalf that component executes during its life-cycle. However, in the requirements phase is clearly defined that the chair is responsible for this task. Consequently, the Paper Manager executes on behalf of the Chair.

A second non-trivial element to model are composed tasks. For instance, the *coordinate conference* task consists of five subtasks, namely *send call for papers*, *send program committee invitation*, *assign submission responsibilities*, *select papers*, and *send notification*. By consequence, each composed task consists of an ordered subset of tasks. Subtasks have the same properties as the composed task: it is executed by the same user, requires the same permissions, etc.

We identified the least privilege problems by using the formal model, and the list of user-task assignments (See Table 9). In short, 6 out of 7 components have too many permissions. Only a subset of the problems is described for reasons of brevity and clarity. The identification of other problems is similar.

A first problem is *Paper Manager* having too many required permissions (See Figures 37 and 38): namely, the ones for the bid paper and submit paper tasks. These are not allowed to be executed together. Hence, at least one task has to be removed from that component.

Another problem is *Conference Manager* having too many permissions (See Figure 38): namely the ones for the send notification and submit paper tasks are obtained via the shared state of the Review Manager.

The solution algorithm iterates over all violations and tries to solve them by applying a set of rules. The algorithm failed to find a suitable solution, because the number of violating components raised from 6 to 8. This might be caused by the fact that a lot of tasks use a central repository and thus share a lot of state. Hence, that component is indivisible.

Metric Case	Size			Complexity		Security		
	#comp	#inf /#comp	#acts /#inf	#tasks	Mccb.	#viol comps (indirect)	# viol tasks (indirect)	attack surface
agents before	7	1.00	2.86	11	4.00	6 (5)	11 (8)	
agents after	9	1.11	2.10	11	6.00	8 (6)	11 (8)	larger

Table 9: Measurements of the two versions of the conference management case study.

B.3 OpenConf

This Section applies the least privilege violation identification and solution algorithm on a reverse engineered version of OpenConf. OpenConf is a [...] conference management system that facilitates the

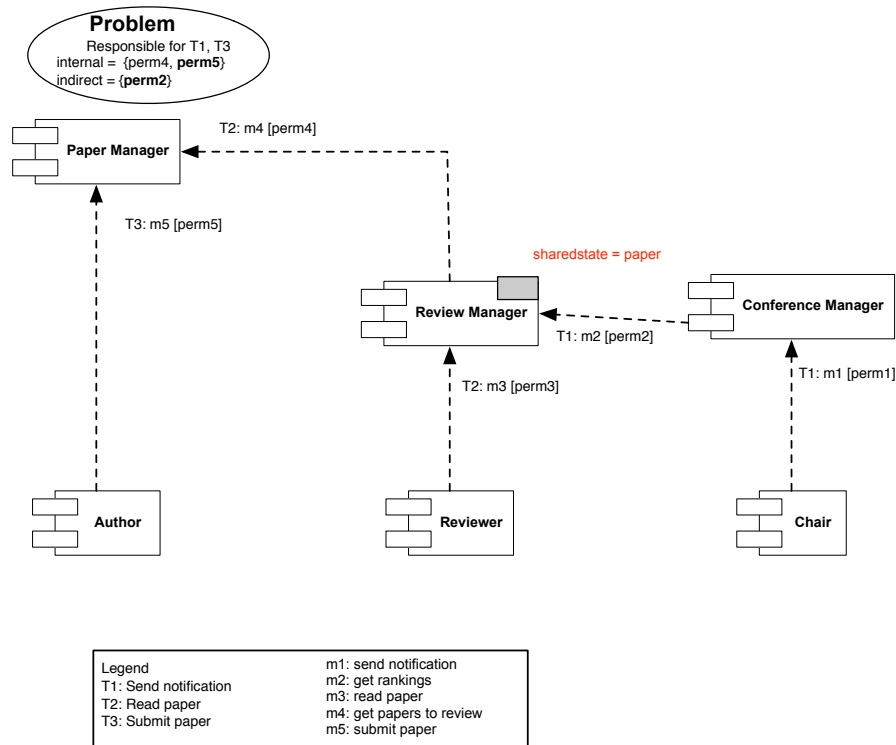


Figure 38: Agent-based CMS problem2: Paper Manager has too many permissions.

electronic submission, review, advocacy, and acceptance of papers, along with management of the whole process by a Program Chair or Editor[22]. Openconf supports the tasks defined in Section B.1 to a great extent. Tasks that openconf offers additionally are: configuration tasks such as install or backup the database, information statistics tasks such as give summary or list authors by country, and advocacy tasks such as defending a certain paper. However, openconf does not support the bidding task.

B.3.1 Reverse Engineering the Software Architecture

We reverse engineered the software architecture of openconf, because no architectural documentation was available at the time of writing.

The system is a typical three-tiered web application as illustrated in Figure 39. The Client Tier is the webbrowser that all actors use to interact with the system. The Presentation and Business Tier consists of the php-application code that is executed by the webserver. The Backoffice Tier consists of the mysql server that is used to store data. Our reengineering focussed on the application and business tier, because OpenConf source code focusses on them.

We created the component and connector diagram by applying the following rules on the application's source code. For each rule, we provide a brief reasoning.

1. Each folder of the source tree represents a different component with as name the name of the folder. In our opinion, this rule partially adheres to the definition of a component[58], because a folder in the source tree represents a coherent unit out of which the application is composed.
2. Create a dependency from component A to component B if one of A's source files uses one of B's source files explicitly as indicated by a *require*, *require_once*, *include*, or a *fopen* statement.
3. Create a dependency from component A to component B if one of A's source files uses one of B's source files's methods.

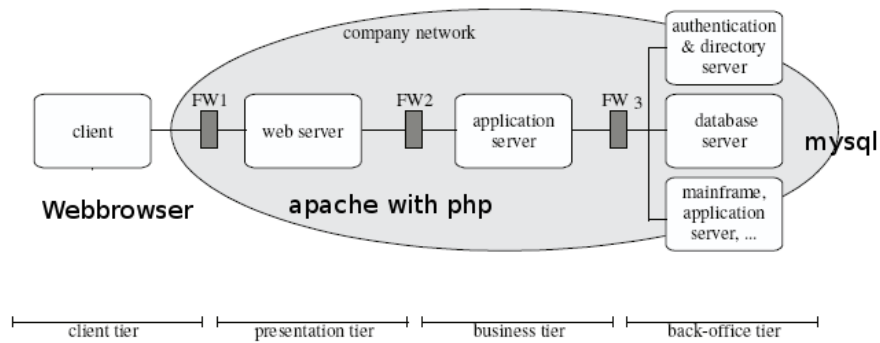


Figure 39: Openconf architecture: conference management system mapped onto webapplication tiers.

4. Create a dependency from component A to component B if one of A's source files outputs a hyperlink to one of B's source files. This hyperlink is denoted with $\langle a \rangle \text{link} \langle a \rangle$ or with *header(link)*.
5. Create a dependency from component A to a database component if the A's source code contains database-specific methods such as *mysql.select*.
6. Create a dependency from component A to a file system component if A's source code contains file system specific operations such as *is.file*.
7. Create an action in interface I of component A if component B outputs an hyperlink to one of A's source files. If the hyperlink is referring to a specific method m in the source code, then $a = m$, else $a = \text{name}(\text{source file})$.
8. Create an action a in an interface I of component A if component B uses an action defined in one of A's source files.
9. The source-files that contains code interfacing with users and other components correspond to the provided interfaces of the component. Again, this matches the intuitive notion of an interface[58] - page 42, because it are the component's access points.
10. The action in an interface's source code that are used by external entities are actions. For end users, they correspond with the code that is executed when they interact with the application via the user interface. This is illustrated in Figure 40.
11. If the actions are grouped in the user interface, they belong to a separate interface. This is illustrated in Figure 40. Each group of actions in the user interface typically groups actions logically belonging together. As such they can be seen as a different service catering for different client needs. Thus it can be considered as a different access point and thus interface.
12. Remove an action from an interface if it can only accessed via another action in the same interface, because it is considered to be a part of that action.

A first version of the software architecture is obtained by applying the first six rules and compounds the following components (See Figure 41). An *Author Component* provides functionality for submitting papers and an abstract. A *Review Component* provides functionality for submission and management of reviews. A *Chair Component* provides functionality for managing the conference. An *Utilities Component* provides the remaining functionality and corresponds to the main directory of the source tree.

A simplification of this first architecture is obtained by applying rules 7-9. Subsequently, applying rules 10-12 results in the final version of the software architecture (See Figure 42).

The tasks are mapped on the software architecture in the following sequence diagrams:

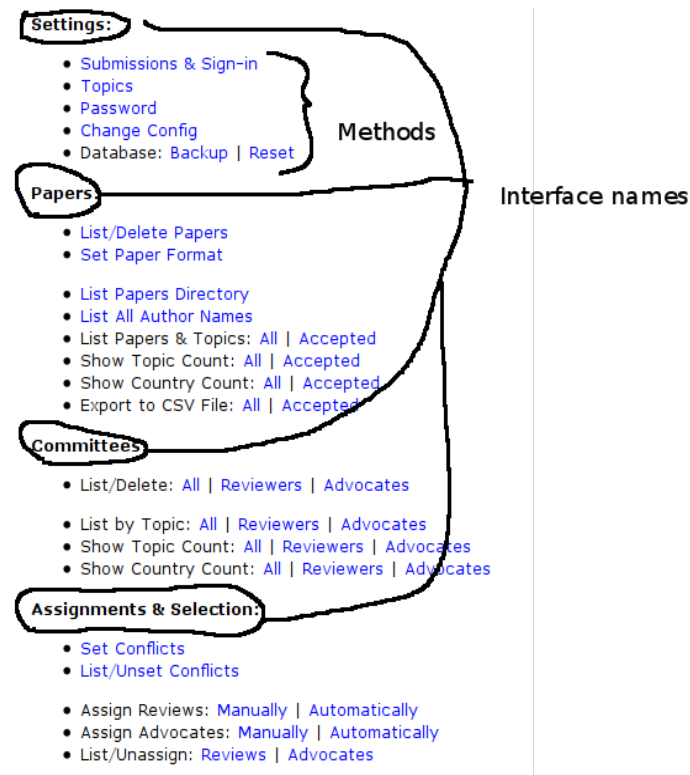


Figure 40: Openconf architecture: separate elements in the user interface are seen as separate interfaces.

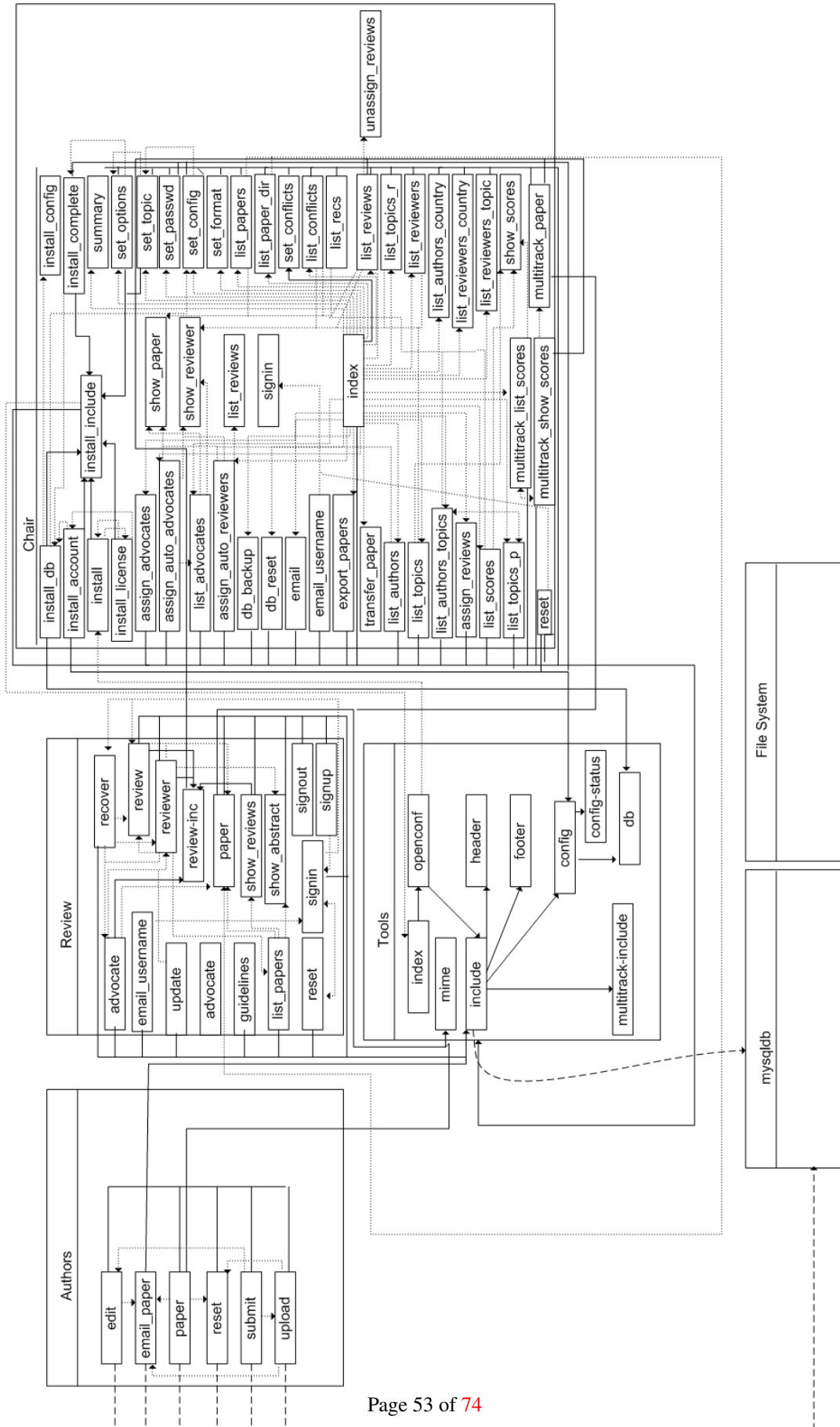


Figure 41: Openconf architecture: initial version.

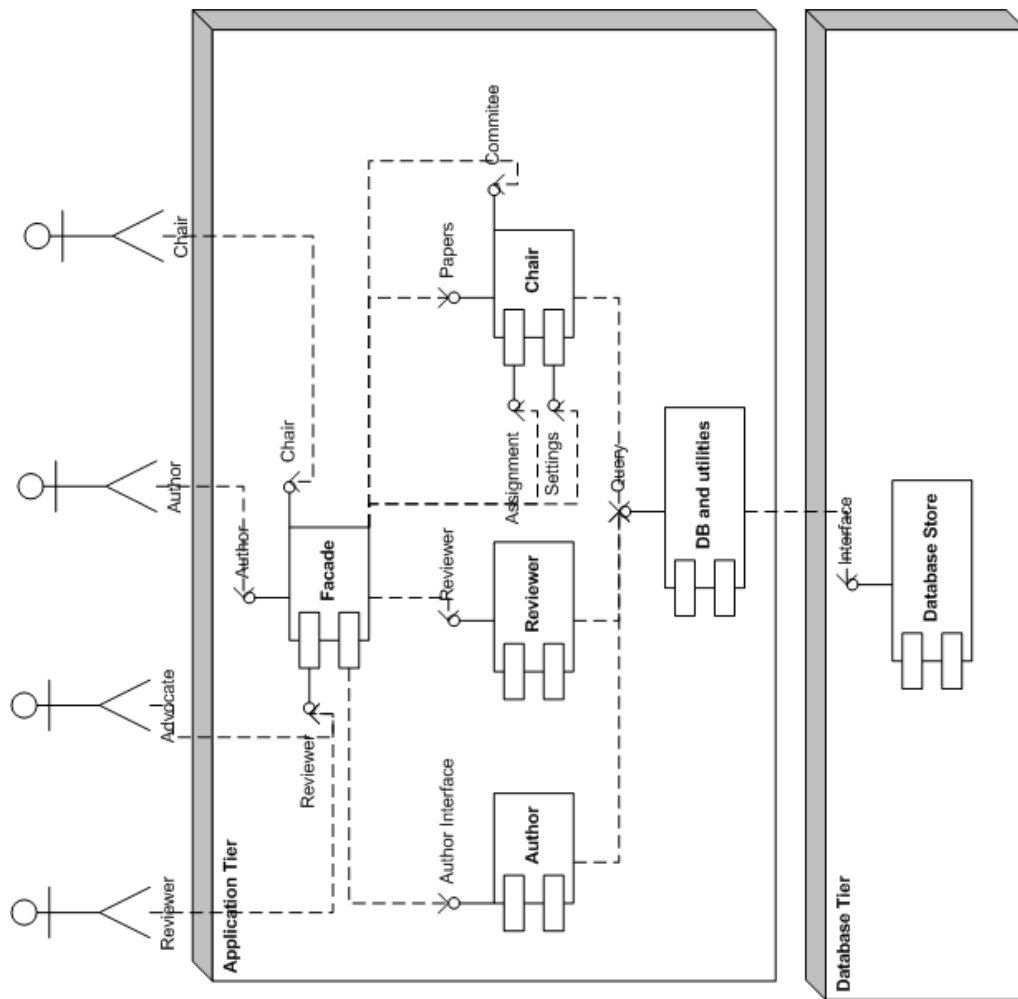


Figure 42: Openconf architecture: component connector diagram.



Figure 43: Openconf architecture: sequence diagram of submit paper.

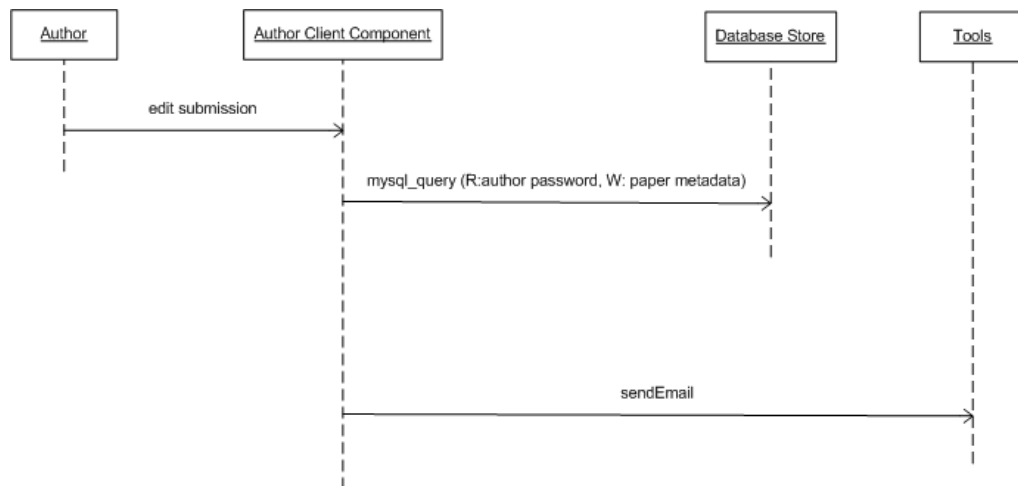


Figure 44: Openconf architecture: sequence diagram of edit submission.

Author The author can execute the following tasks:

- Submit paper (See Figure 43).
- Edit submission (See Figure 44).
- Reupload paper (See Figure fig:case3:openconf:sequencediagram:reuploadpaper).
- View paper (See Figure 46).
- The email papers task is not accessible via the userinterface. However, directly entering the URL allows the author to mail the papers.
- The reset password password is not accessible via the userinterface. However, directly entering the URL allows the author to reset his password.

Reviewer The reviewer is allowed to execute the following tasks.

- review paper (See Figure 47).

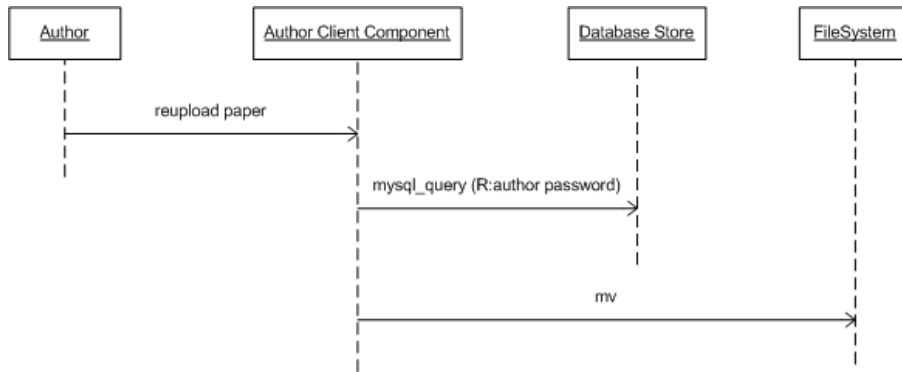


Figure 45: Openconf architecture: sequence diagram of reupload paper.

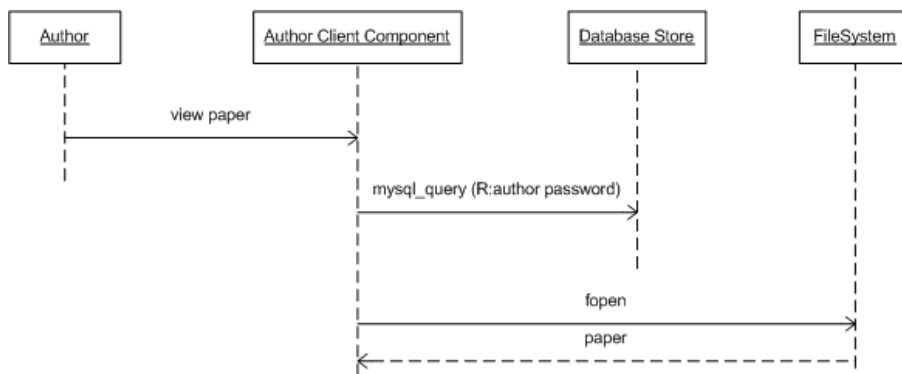


Figure 46: Openconf architecture: sequence diagram of view paper.

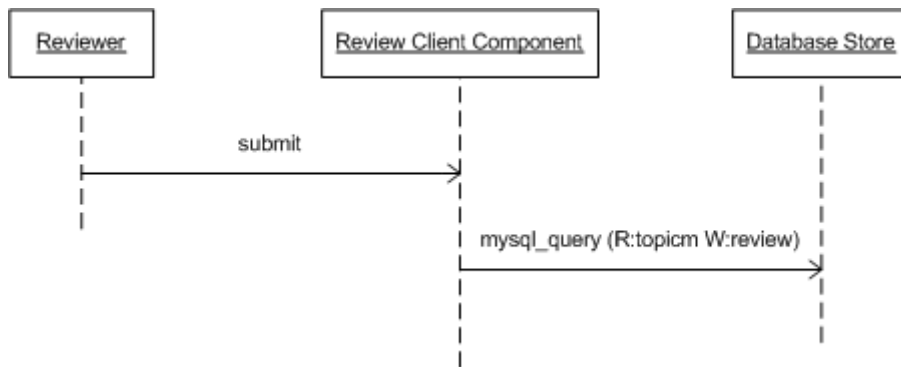


Figure 47: Openconf architecture: sequence diagram of review paper.

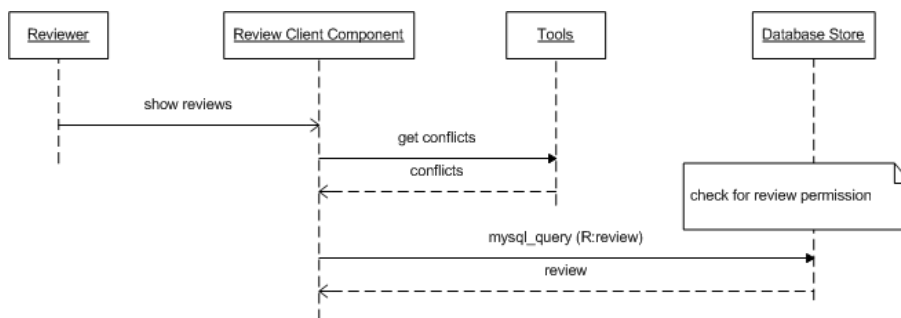


Figure 48: Openconf architecture: sequence diagram of show reviews.

- show reviews (See Figure 48).
- show paper (See Figure 49).
- show abstract (See Figure 50).
- update review: see review.

Chair The chair is allowed to execute the following tasks.

- set topics (See Figure 51).
- auto assign reviewers (See Figure 52)
- unassign reviews.
- set conflicts.
- list conflicts (See Figure 53).
- remove conflicts.
- assign reviewer.
- send email.

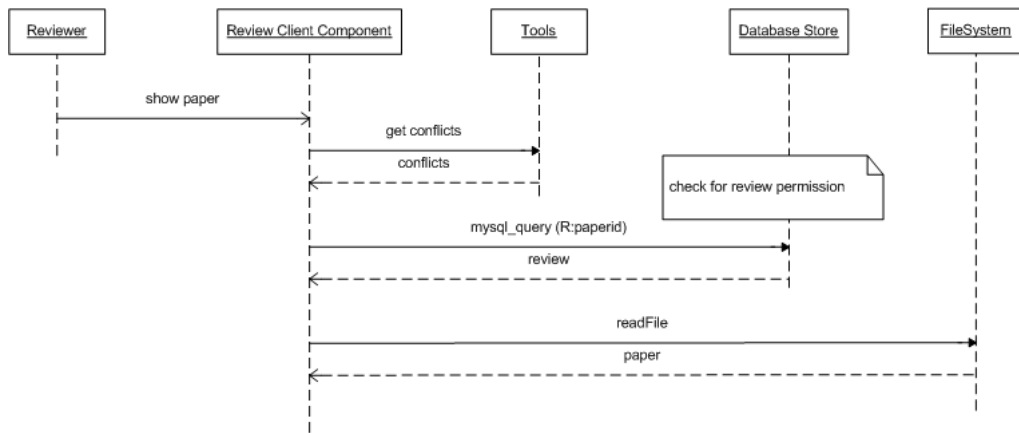


Figure 49: Openconf architecture: sequence diagram of show paper.

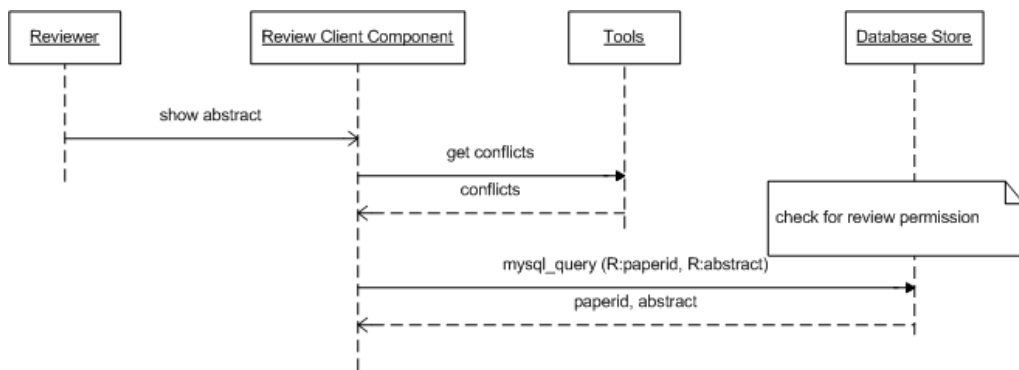


Figure 50: Openconf architecture: sequence diagram of show abstract.

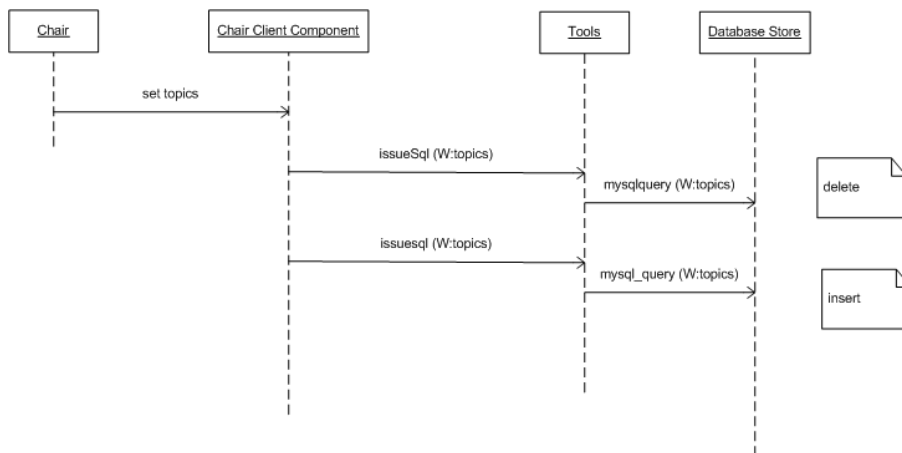


Figure 51: Openconf architecture: sequence diagram of set topics.

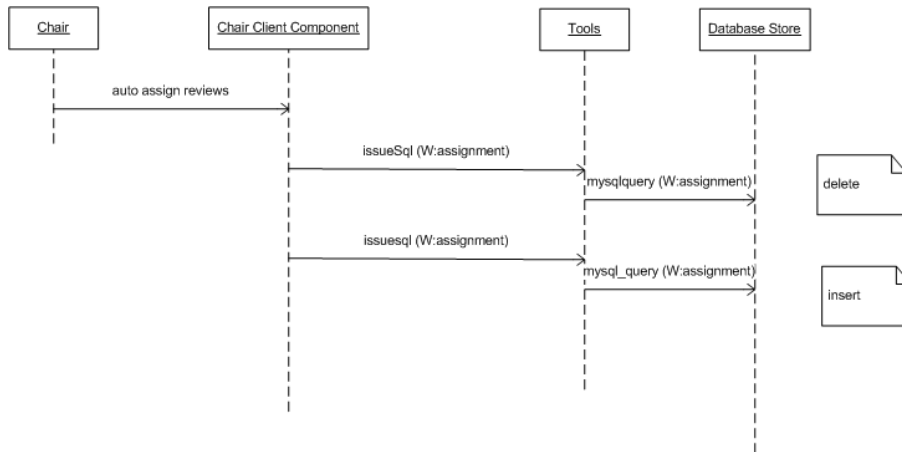


Figure 52: Openconf architecture: sequence diagram of auto assign reviewers.

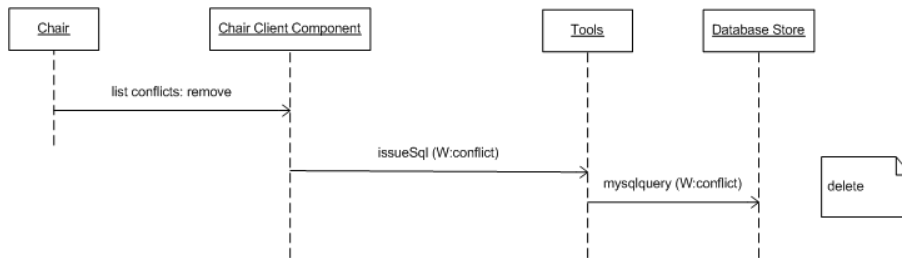


Figure 53: Openconf architecture: sequence diagram of list conflicts.

B.3.2 Experiments

Formally modeling the component and connector diagram 42 is rather straightforward. Therefore, we focus on one non-trivial element of the model: one action used by all tasks hinders the shared state detection algorithm.

Every task uses the *mysql_select* -method from the database component which has only one string parameter. By consequence, the identification of indirect problems via shared state might not work very well. This can be solved by annotating the method with the real data that passes when a certain task is executed. Although, this might be a cumbersome task.

The identification algorithm identified several violating components, among which *DB*, *FS*, and *Tools*. All three have too many indirect permissions, namely the ones that allow them to execute (parts of) the review paper and submit paper tasks. This is illustrated in Figure 54.

A possible solution splits these components into new ones.

B.4 Conclusion

Two conference management systems (agent-based conference management system and openconf) illustrate that our algorithms are able to detect and solve least privilege violations. However, our splitting rule does not work very well when the state shared between tasks is very large, e.g. when there is a central repository.

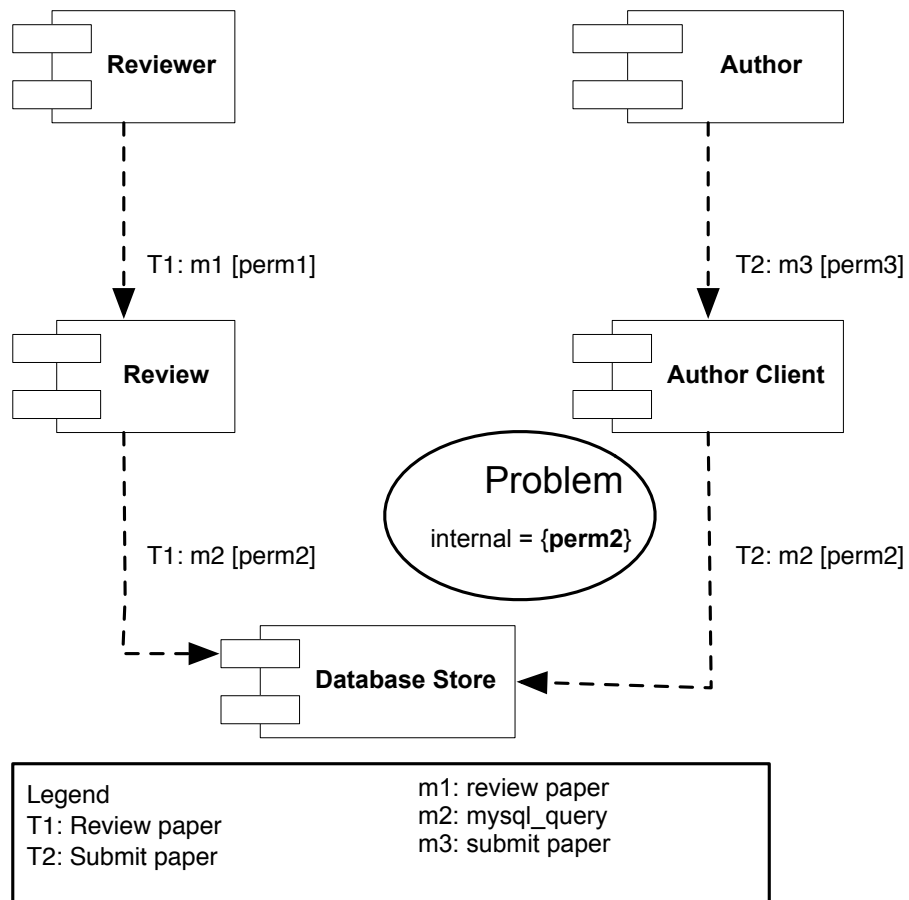


Figure 54: Openconf problem: DB has to many permissions.

Subject	OpenConf task	Requirements Task
Author	submit paper	submit abstract
Author	submit paper	submit paper
Author	reupload paper	submit revision
Author	reupload paper	submit camera ready
Author	edit submission	submit revision
Author	edit submission	submit camera ready
Author	email papers	-
Author	reset password	-
Reviewer	advocate paper	-
Reviewer	email username	-
Reviewer	read guidelines	-
Reviewer	update review	submit review
Reviewer	list papers	-
Reviewer	show paper	read paper
Reviewer	show reviews	read review
Reviewer	show abstract	read paper abstract
Reviewer	review paper	submit review
Reviewer	signup	-
Reviewer	signin	-
Reviewer	reset	-
Reviewer	-	bid paper
Chair	backup database	-
Chair	reset database	-
Chair	set password	-
Chair	set options	-
Chair	set config	-
Chair	set topics	send call for papers
Chair	install	-
Chair	Auto assign reviewers	assign submission responsibilities
Chair	Unassign reviews	assign submission responsibilities
Chair	set conflicts	assign submission responsibilities
Chair	list conflicts	read submission responsibilities
Chair	assign advocates	-
Chair	auto assign advocates	-
Chair	list advocates	-
Chair	list scores	-
Chair	assign reviewer	assign submission responsibilities
Chair	give summary	-
Chair	sign in	-
Chair	sign out	-
Chair	set format	-
Chair	export papers	-
Chair	list papers	-
Chair	list paper directory	-
Chair	list authors topics	-
Chair	list authors country	-
Chair	list authors	-
Chair	list topics	-
Chair	send email	Notification

Table 10: Mapping of the tasks openconf supports and the tasks defined in the introduction.

C Case study: publishing system

This Section applies the presented algorithms on the the publishing system presented in [33]. In order to limit the complexity of the material presented here, we apply the algorithms on the functional viewpoint.

This case study is structured as follows. First, Section C.1 introduces the application domain. Next, Section C.2 gives an overview of the software architecture, and the most important sequence diagrams. Finally, Section C.3 briefly presents and interprets the results of our experiments and Section C.4 concludes.

C.1 Introduction

A publishing system is a system that automates the cross-media publishing workflow of a corporate publishing company. The system is a next-generation end-to-end media platform using various wired and wireless communication channels for publishing, allowing personalized services based on user-profile and context.

A wide range of different actors make use of this system: the input source, the media consumer, the advertiser, the journalist, and the manager. The *input source* is the entity that produces content, like an author, or a musician. At the other end of the content consumption chain, the *media consumer* is the individual who wants to obtain and consume content. For example, a home user who wants to download and read the news of the day. The *advertiser* is the main income source of the company because he buys commercial space or time. The *journalist* forms the bridge between consumers and producers by using the publishing system to distribute finished content. The *manager* manages the publishing company by creating a publishing strategy and assigning tasks to journalists.

The publishing system's main features are input management, content management, content distribution, user management, and planning tasks and strategy. In this discussion, we limit ourself to a subset of tasks for reasons of practicality and in order to not loose focus in the example. Incorporating the other tasks is similar and a straightforward job. An overview of publishing use cases can be found in [35]. The tasks are listed in Table 11.

Obviously, each of these tasks have constraints imposed on them. For instance, an advertiser is not allowed to edit articles, because that allows him to advertise for free. A journalist is not allowed to create a corporate strategy or delegate tasks to him, because that allows him to look busy while being idle. These constraints are expressed in a policy in Table 11.

C.2 Software Architecture

The software architecture of the publishing system is described in four viewpoints. The functional viewpoint comprises the basic and core functionality of the business of the publisher, namely publishing. The Integration with additional services viewpoint presents integration of the core service with additional services. The Billing viewpoint shows how billing is dealt with within the publishing system. The Context awareness and tracking viewpoint adds the possibility of tailoring services to the preferences, needs, and more generally the context of the media consumer.

In the system's architecture, the components responsible for the publishing features are the following (See Figure 55). An *Input Management System* (IMS) is used by an input source to annotate and prepare produced content to be stored in the *Content Management System* (CMS). The *Journalist Desk* (JD) is used by the journalist to make content ready to be published. A *Newspaper Service* (NS) is utilized for distributing editions towards the media consumer. The *User Management System* (UMS) takes care of handling user information. The *Planning System* is used by the manager and the journalist to create tasks, corporate, and newspaper planning.

The authors made the following changes to the architectural description of the publishing system for the sake of clarity and simplicity:

- Corporate News Desk Worker became Manager
- Corporate News Desk Worker Desk became Management Desk
- Service News Desk Worker became Journalist

User	Task
Media Consumer	Subscribe to service
Media Consumer	Unsubscribe from service
Media Consumer	Pull an edition
Media Consumer	Send viewing information
Media Consumer	Send context information
Media Consumer	Send feedback information
Input Source	Retrieve the number of views of the stories built from their input
Input Source	Submit input to the publishing system
Advertiser	Submit input to the publishing system
Advertiser	Retrieve the number of views of a commercial
Manager	Plan or adapt the overall business strategy
Journalist	Plan and replan a service and their editions
Journalist	Verify input
Journalist	Verify commercials
Journalist	Replanning triggered by input
Journalist	Add meta-data to content or change it
Journalist	Build a story from usable content
Journalist	Build an edition from stories and commercials for a certain service
Journalist	Assist the publishing system in lay-outing and edition
Journalist	Push an edition
Journalist	Create/update journalist task assignment
Manager	Create/update task assignment

Table 11: A list of user-task assignments.

Task
plan corporate strategy
subscribe to a service
unsubscribe from a service
pull an edition
submit a commercial
retrieve number of views
plan edition strategy
plan service strategy

Table 12: Publishing system subset: supported tasks.

- Service News Desk Worker Desk became Journalist Desk
- Only 1 service is supported: Newspaper Service
- The name of actions that do not have a unique name have been replaced with the name concatenated with its component's name.

Some relevant tasks are illustrated in Figures 56, 57

C.3 Experiments

Formally modeling the component and connector diagrams (See Figure 55) is rather straightforward. Therefore, we focus on the non-trivial elements of the model.

A first non-trivial element to model are composed tasks. For instance, the *verify new content* task is executed as final part of tasks that add content to the system. Example of such tasks are *Submit input to the publishing system*, *Submit a commercial to the publishing system*, and *Send feedback information*. In other words, each composed task consists of an ordered subset of tasks. Determining the properties of these tasks such as the user it is executed by is done conservatively. Determining the properties of an unknown parent task is done by creating the union of the properties of its subtasks. Determining the properties of an unknown subtask is done by giving them the same properties as the parent task.

A second non-trivial element to model is a component and not an end-user being responsible for a task. By consequence, it seems impossible to determine the user under which this component executes. For instance, the *verify new content* task starts with the Content Management System sending a notification to the manager via the Planning System. However, this task is a subtask of various other tasks. Hence, these other tasks can be used to determine the user under which the component executes.

A third non-trivial element is determining the users associated with tasks that involve multiple users. For instance, the *verify new content* task consists of three parts. In the first part a “there is new content” notification is sent to the manager. In the second part, the verification task has been delegated by the manager to a journalist. The third part consists of the actual verification by the journalist. In other words, different parts of a task are executed by different users. Acting conservatively, we decided to assign all the users to the whole task.

We identified the least privilege problems by using the formal model, and the list of user-task assignments (See Table 13) on two versions of the software architecture. The first version is the full architecture, while the second version is the same architecture with tasks causing shared state removed (See Table 12). Both cases caused least privilege violations: in the full architecture, all components are violating least privilege, while in the subset only 4 components did. Only a subset of the problems is described for reasons of brevity and clarity. The identification of other problems is similar.

A first problem is that the Planning System is responsible for the *plan corporate strategy* and *plan edition strategy* tasks. The former is executed by a manager, while the latter is executed by a journalist. Having permissions for both tasks was explicitly forbidden by the company policy (as illustrated in Figure 59).

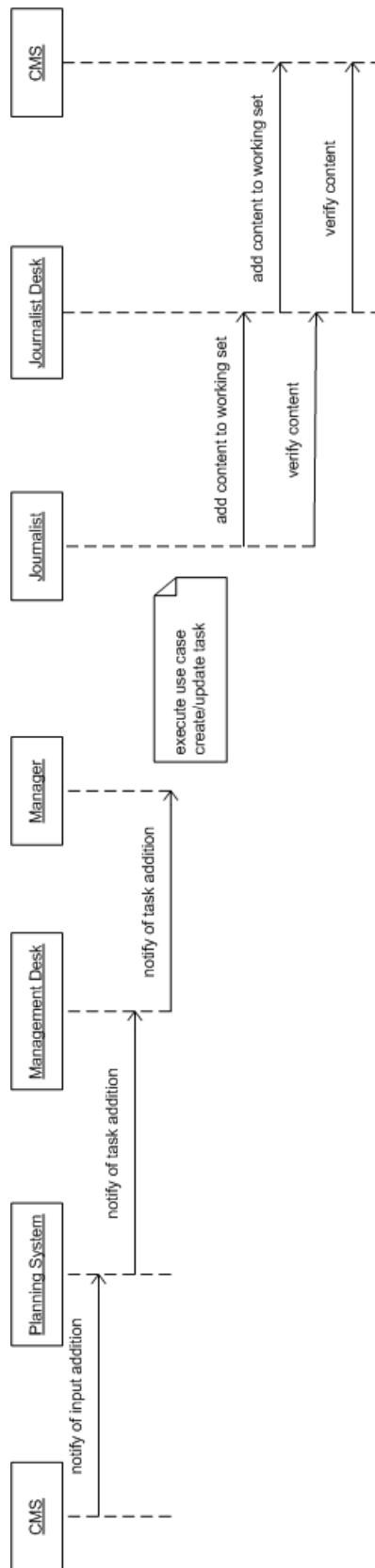


Figure 56: The sequence diagram of the verify new content task.

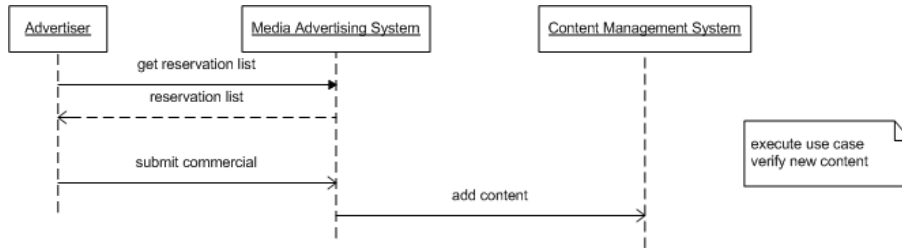


Figure 57: The sequencediagram of the submit commercial task.

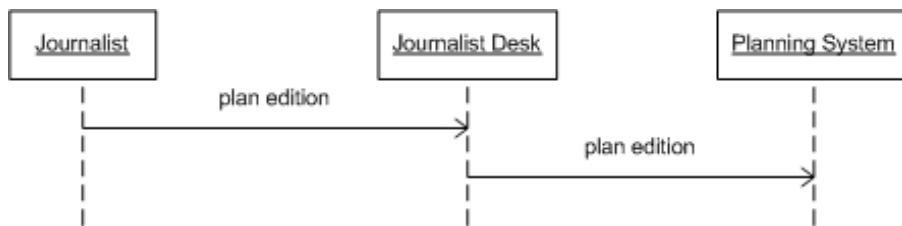


Figure 58: The sequencediagram of the plan edition task.

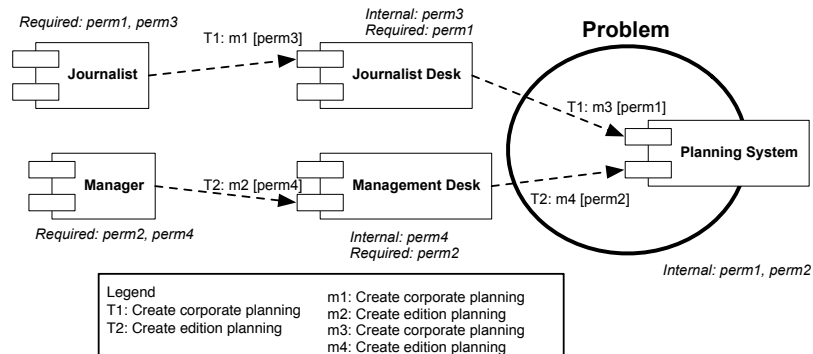


Figure 59: A violation caused by too many internal permissions.

A second problem is that the advertiser is able to obtain enough permissions to modify the advertisement workflow (part of planning tasks) (See Figure 60). The design problem is that planning system is responsible for both *notification* and *edition planning* tasks.

A solution to these problems can be found by applying the transformation described in Section 5. The first problem can be solved by splitting the component in two parts. The first part contains actions related to corporate strategy, while the second part contains actions related to edition planning. The second problem can be solved by decoupling notification from creation. As such, the risk of LP violations in the final software architecture will be greatly reduced.

	Size			Complexity		Security		
ps sub before	13	2.38	2.65	8	6	4 (0)	6 (0)	
ps sub after	20	2.6	1.58	8	7	0 (0)	0 (0)	same
ps full before	13	2.38	2.64	22	8	13 (12)	22 (16)	
ps full after	13	2.38	2.64	22	8	13 (12)	22 (16)	same

Table 13: Measurements of the cases in the three domains and three variants of the publishing case.

C.4 Conclusion

A publishing system illustrates that our algorithms are able to detect and solve least privilege violations. However, our splitting rule does not work very well when the state shared between tasks is very large, e.g. when there is a central repository.

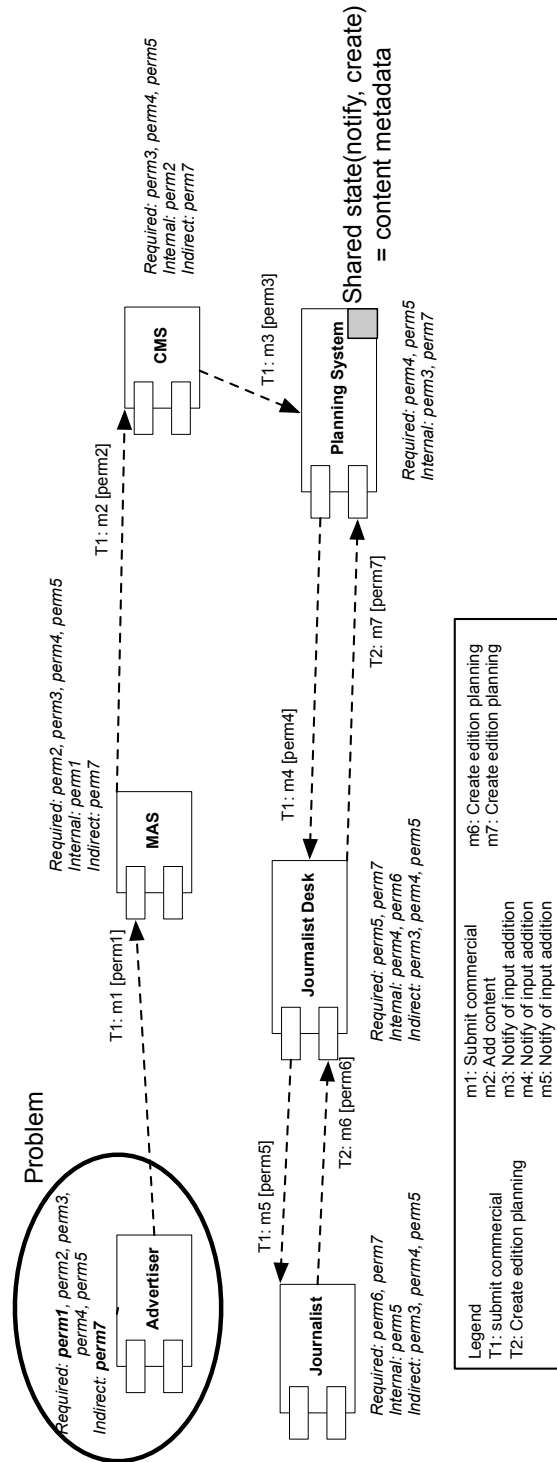


Figure 60: A violation caused by too many indirect permissions.

D List of Figures, and Tables

List of Figures

1	LP in software architecture: a combination of architectural structure and access policy. . .	5
2	Excerpt from the component diagram of the an integrated calendar system	6
3	Identification algorithm	10
4	Assign internal permissions	10
5	Propagate required permissions	10
6	Calculate indirect permissions	10
7	An overview of transformations of compound elements. These transformations are used to identify transformations on the elements of the formal model that define the property of least privilege.	12
8	The property of least privilege is defined in terms of actions, permissions, tasks, users, and components.	12
9	Transformation split component: a component with overlapping read methods and a write method can be split.	16
10	Transformation split component: a component with two write methods should not be split.	17
11	This proof proves that splitting a component lowers the number of permissions.	19
12	Transformation split tasks: structural view before splitting.	19
13	Transformation split tasks: behavioral view before splitting.	19
14	Transformation split tasks: behavioral view after splitting.	20
15	Proof that the splitting tasks transformation lowers the number of permissions.	21
16	Transformation: splitting permissions.	22
17	Transformation remove unused actions from a component.	23
18	Proof that the removing unused actions from component transformation lowers permissions of that component.	24
19	Transformation rewiring: a small case that illustrates that rewiring lowers the overall required privileges.	26
20	Proof that rewiring lowers the number of permissions.	27
21	Multiple solutions: a naive strategy creates a tree of all possible solutions. Nodes are architectures. Edges are transformations.	29
22	A violation caused by too many internal permissions.	31
23	A violation caused by too many indirect permissions.	32
24	The component and connector diagram of the example chat application.	39
25	The sequence diagram of the send message use case.	39
26	The component and connector diagram of our extension of the example chat application.	39
27	The identification algorithm fails to identify a violation because it can not distinguish between the business and home user tasks.	40
28	This Figure illustrates how the architecture would look like if we modify the formal representation of it.	41
29	The Chat Server has too many permissions and thus privileges.	42
30	A possible solution: the Chat Server has been split.	42
31	Agent-based CMS: component diagram of the software architecture presented in [45] extended with interfaces.	46
32	Agent-based CMS: sequence diagram of send call for papers.	47
33	Agent-based CMS: sequence diagram of send notification.	47
34	Agent-based CMS: sequence diagram of submit paper.	48
35	Agent-based CMS: sequence diagram of read paper.	48
36	Agent-based CMS: sequence diagram of get proceedings.	48
37	Agent-based CMS problem: Paper Manager has too many permissions.	49
38	Agent-based CMS problem2: Paper Manager has too many permissions.	50
39	Openconf architecture: conference management system mapped onto webapplication tiers.	51

40	Openconf architecture: separate elements in the user interface are seen as separate interfaces.	52
41	Openconf architecture: initial version.	53
42	Openconf architecture: component connector diagram.	54
43	Openconf architecture: sequence diagram of submit paper.	55
44	Openconf architecture: sequence diagram of edit submission.	55
45	Openconf architecture: sequence diagram of reupload paper.	56
46	Openconf architecture: sequence diagram of view paper.	56
47	Openconf architecture: sequence diagram of review paper.	57
48	Openconf architecture: sequence diagram of show reviews.	57
49	Openconf architecture: sequence diagram of show paper.	58
50	Openconf architecture: sequence diagram of show abstract.	58
51	Openconf architecture: sequence diagram of set topics.	58
52	Openconf architecture: sequence diagram of auto assign reviewers.	59
53	Openconf architecture: sequence diagram of list conflicts.	59
54	Openconf problem: DB has too many permissions.	60
55	The component diagram of the functional viewpoint of the publishing system.	65
56	The sequence diagram of the verify new content task.	66
57	The sequencediagram of the submit commercial task.	67
58	The sequencediagram of the plan edition task.	67
59	A violation caused by too many internal permissions.	67
60	A violation caused by too many indirect permissions.	69

List of Tables

1	Tasks assigned to the users of the calendar system	5
2	Possible basic transformations that have an effect on least privilege. D means that this transformation can decrease the amount of permissions, I means that it can increase the amount of permissions, S means that it might solve least privilege violations, N means that it does not solve least privilege violations, S means that it solves LP problems but is not a good solution, and -S means that the number of permissions is not affected, but it can solve LP violations because the transformation enables better permission assignment. Transformations with a grey background have been detailed in the remainder of this section.	13
3	Measurements of the cases in the three domains and three variants of the publishing case.	33
4	A chatting policy stating that home users are not allowed to view business users' messages.	38
5	Policy stating what tasks each user is allowed to execute.	40
6	Measurements of the two versions of the chat system case. User components are not included.	41
7	The review process as presented in the confious conference management system[49]. Phases are ordered according to the time when they are executed. A light gray activity is an optional one.	44
8	A simplified policy for the review process: an author is not allowed to be chair or reviewer.	45
9	Measurements of the two versions of the conference management case study.	49
10	Mapping of the tasks openconf supports and the tasks defined in the introduction.	61
11	A list of user-task assignments.	63
12	Publishing system subset: supported tasks.	64
13	Measurements of the cases in the three domains and three variants of the publishing case.	68

References

- [1] Anurag Acharya and Mandar Rajee. Mapbox: Using parameterized behavior classes to confine applications. Technical report, Santa Barbara, CA, USA, 1999.
- [2] Albert Alexandrov, Paul Kmiec, and Klaus Schauer. Consh: A confined execution environment for internet computations. *USENIX Annual Technical Conference*, 1999.

- [3] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc. Mountain View, CA, USA, 2003.
- [4] R.J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc. New York, NY, USA, 2001.
- [5] Emine G. Aydal, Mark Utting, and Jim Woodcock. A comparison of state-based modelling tools for model validation. pages 278–296. 2008.
- [6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [7] Daniel Julius Bernstein. Qmail home page.
- [8] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall Englewood Cliffs, NJ, 1981.
- [9] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.
- [10] Koen Buyens, Riccardo Scandariato, and Wouter Joosen. Process activities supporting security principles. In *COMPSAC (2)*, pages 281–292. IEEE Computer Society, 2007.
- [11] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.
- [12] Chris Evans. Comments on the Overall Architecture of Vsftpd, from a Security Standpoint. Internet, February 2001.
- [13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 495–499, 1999.
- [14] J. Clark et al. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, 16(11), 1999.
- [15] E.M. Clarke, O. Grumberg, D.A. Peled, and I. NetLibrary. *Model checking*. Springer, 1999.
- [16] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
- [17] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.
- [18] Eric Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society.
- [19] Thuong Doan, Steven Demurjian, T. C. Ting, and Andreas Ketterl. Mac and uml for secure software design. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 75–85, New York, NY, USA, 2004. ACM.
- [20] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.
- [21] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 90–105, 2002.
- [22] Zakon Group. Openconf, July 2008. <http://www.zakongroup.com/technology/openconf.shtml>.

- [23] L. Grunske, L. Geiger, A. Zundorf, N. Van Eetvelde, P. Van Gorp, and D. Varro. Using graph transformation for practical model driven software engineering. *Model-driven Software Development*, 2005.
- [24] S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In *Proceedings of the 13th international conference on Software engineering*, pages 875–878. ACM New York, NY, USA, 2008.
- [25] GJ Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.
- [26] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.
- [27] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.
- [28] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.
- [29] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [30] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In ISOC Network and Distributed System Security*, 2000.
- [31] Jan Jürjens. *Secure Systems Development With UML*. Springer, 2005.
- [32] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.
- [33] Dimitri Van Landuyt, Johan Gregoire, Sam Michiels, Eddy Truyen, and Wouter Joosen. Architectural design of a digital publishing system. Technical report, Katholieke Universiteit Leuven, October 2006.
- [34] Mikael Lindvall, Roseanne T. Tvedt, and Patricia Costa. An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8(1):83–108, March 2003.
- [35] Tom Mahieu, Wouter Joosen, Dimitri Van Landuyt, Johan Gregoire, Koen Buyens, and Eddy Truyen. System requirements on digital newspapers. Technical report, Katholieke Universiteit Leuven, March 2007.
- [36] Pratyusa K. Manadhata, Dilsun K. Kaynar, and Jeannette M. Wing. A formal model for a systems attack surface. Technical Report CMU-CS-07-144, Carnegie Mellon University, 2007.
- [37] G. Jason Mathews and Barry E. Jacobs. Electronic management of the peer review process. *Comput. Netw. ISDN Syst.*, 28(7-11):1523–1538, 1996.
- [38] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [39] Garry McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.
- [40] Gary McGraw. *Software Security: Building Security In*. Addison Wesley, 2006.
- [41] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [42] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions of Software Engineering*, 30(2):126–139, 2004.

- [43] S. Merz. Model Checking: A Tutorial Overview. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 3–38, 2001.
- [44] A. Metzger. A systematic look at model transformations. *Model-driven Software Development*, 2:978–3, 2005.
- [45] Mirko Morandini, Duy Cu Nguyen, Anna Perini, Alberto Siena, and Angelo Susi. Tool-supported development with tropos: The conference management system case study. In Michael Luck and Lin Padgham, editors, *Agent Oriented Software Engineering VIII*, volume 4951 of *LNCS*, pages 182–196. Springer, 2008. 8th International Workshop, AOSE 2007, Honolulu, HI, USA, May 14, 2007, Revised Selected Papers.
- [46] OGM. Common WareHouse Model.
- [47] William F. Opdyke, William F. Opdyke, Ph. D, and Ralph E. Johnson. Refactoring object-oriented frameworks. Technical report, University of Illinois at Urbana-Champaign, 1992.
- [48] OWASP. Comprehensive, lightweight application security process. <http://www.owasp.org>, 2006.
- [49] Manos Papagelis, Dimitris Plexousakis, and Panagiotis Nikolaou. Confious: Managing the electronic submission and reviewing process of scientific conferences. In Anne H. H. Ngu, Masaru Kitsuregawa, Erich J. Neuhold, Jen-Yao Chung, and Quan Z. Sheng, editors, *WISE*, volume 3806 of *Lecture Notes in Computer Science*, pages 711–720. Springer, 2005.
- [50] David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, Berkeley, CA, USA, 2002. USENIX Association.
- [51] Niels Provos. Sysrtrace - interactive policy generation for system calls.
- [52] Jie Ren. *A connector-centric approach to architectural access control*. PhD thesis, Long Beach, CA, USA, 2006. Adviser-Richard N. Taylor.
- [53] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 265–277, 2000.
- [54] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
- [55] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
- [56] SecureSoftware. CLASP security principles.
- [57] JM Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, 2008.
- [58] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [59] Wietse Zweitze Venema. Postfix home page.
- [60] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.
- [61] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (dte). In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, pages 3–3, Berkeley, CA, USA, 1996. USENIX Association.
- [62] Sherif M. Yacoub and Hany H. Ammar. A methodology for architecture-level reliability risk analysis. *IEEE Transactions on Software Engineering*, 28(6):529–547, June 2002.