Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems

Max E. Kramer, Martin Hecker, Simon Greiner, and Kaibin Bao

Karlsruhe Institute of Technology
Competence Center for Applied Security Technology (KASTEL), Karlsruhe
firstname.lastname@kit.edu

Abstract. Many software systems have to be designed and developed in a way that guarantees that specific information remains confidential with respect to adversaries. Such guarantees depend on the internal information flow inside individual components *and* the system architecture, e.g. the deployment on hardware nodes and properties of their communication links. Therefore, we propose a novel architecture-driven approach for specifying and analyzing the confidentiality of information processed by component-based systems. It includes an architectural analysis that is able to infer leaks of confidential information from abstract architecture models, adversary models, and confidentiality specifications. Our approach supports re-usability of components and specification parts across systems as well as specifications with custom labels, e.g. to specify the accessibility of hardware and service interfaces. Additionally, our information flow specifications for components are compositional and supported by tools for non-interference verification on source code level. In two case studies, we show how our specification approach is applied and how the architectural analysis is able to detect information leaks of a system in early design phases.

1 Introduction and Motivation

In distributed software systems, keeping data confidential while it passes the boundaries of logical components, physical machines, and communication stacks is a major security challenge. To avoid unintended information leaks, the flow of confidential data has to be considered during system design because modifying an existing architecture to increase the security can be very costly. Therefore, the confidentiality of data has to be addressed in a way that allows early up-front analyses. If components that were already developed and verified can be reused because these analyses are compositional, this can further decrease costs.

Current approaches analyze *either* inter-component *or* intra-component data flows and focus on specific concerns, such as protocols [1], policies [2], or generated code [3]. Whether a component-based system fulfills a confidentiality specification can, however, not be evaluated solely with such focused analyses: Many confidentiality leaks are the result of the *combination* of inter- and intra-component data flows and depend on the capabilities of potential adversaries.

Therefore, an integration of sound analysis methods should examine components, their connection and communication as well as adversary capabilities.

In this paper, we propose an approach for specifying and analyzing the confidentiality of information processed by component-based systems based on architectural models and specifications. After modeling the software, hardware, and communication of a system, confidentiality requirements can be added e.g. sets of data that shall not interfere or the accessibility of hardware nodes and service interfaces. Potential adversaries and their capabilities to acquire information, e.g. via communication links, can be specified. Our analysis is able to identify vulnerable locations where confidential information can be leaked by executing Prolog-based inferences rules on these models and specifications. In case the analysis finds no leaks, hardware and communication requirements are to be verified in the implemented and configured system as they cannot be guaranteed automatically. But these requirements are made explicit in our model to facilitate later assessment. Software requirements can be transformed to non-interference requirements that can be checked by tools like KeY [4] and JOANA [5] during system verification. In case the analysis finds a leak, the proof-tree is generated to document where and why a system is vulnerable with respect to an adversary. The specification and analysis code and all case studies are open-source[1].

The approach has four key properties: First, the architecture analysis with generic non-interference rules can directly be applied to models with custom confidentiality specification labels. Thus, domain experts can use domain concepts to label system and adversary properties when specifying confidentiality requirements. Both the architecture analysis and the code verification do not need to be adapted to specific systems or adversaries. Second, by inspecting an abstract model, the analysis allows identification of architectural designs that cannot meet the confidentiality requirements before the complete system is designed in detail. Third, the analyses are compositional so that results for components or subsystems that were already analyzed can be reused in new systems. Last, the approach scales to complex systems because specifications are only needed for interfaces of system-level components. Specifications for the internal structure of composite components can be added and refined as necessary.

The paper is structured as follows: In Section 2, we introduce the application scenario of a running example. In Section 3 we explain the concepts of the component-based modeling language that we use. In Section 4 and 5, we present our extension of this language for the specification of properties used for confidentiality analysis and the new concepts for adversary modeling. In Section 6, we explain the architecture and code analyses based on the modeling language. In Section 7, we present our evaluation case studies. In Section 8, we discuss related work and in Section 9 we draw some final conclusions.

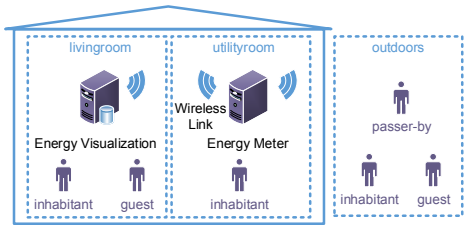

Fig. 1. The example scenario on energy visualization in the household

2 Exemplary Application Scenario

We briefly introduce a scenario to exemplify the proposed confidentiality specification and analysis. The scenario consists of a household where the energy consumption of the inhabitants is monitored (Figure 1). Consumption data is acquired by an energy meter and transmitted wirelessly to an energy visualization system. It stores the consumption data in the internal database and generates consumption graphs for the inhabitants of the household. The energy meter has a wireless consumer interface which is used by the inhabitant to read the meter values. It also provides additional information for billing purposes, for example the consumer identification. The energy meter is located inside a utility room and the visualization system is in a living room. The locations in Figure 1 are depicted as boxes with dashed lines. Whenever a user is able to gain access to a location, a user symbol is placed inside the box. There are three groups of users who are interacting with the system: inhabitants, guests and passer-bys. Inhabitants have access to all areas but guests have no access to the utility room. Passer-bys are not allowed to enter the house.

3 Architecture Model

In this section, we introduce the architectural model on which our confidentiality specification and analyses are built. We use a subset of the Palladio Component Model (PCM) [6], which is a modeling language for component-based systems that also provides a graphical notation. It is based on the concept of components as composable software units that provide and require services [7].

A PCM model for the scenario of Section 2 is shown in Figure 2. For our specification and analysis of confidentiality we use the following model elements provided by the PCM: *Service*, *Parameter*, *Interface*, *Component*, *Assembly Context*, *Resource Container* and *Linking Resource*. These elements and their relationships are depicted in Figure 5 in the Appendix and described in the following paragraphs.

A *Service* realizes parts of the functionality provided by a system and may have input parameters and a return parameter. In the example, drawEnergyConsumptionGraph() (Figure 2, ①) is a *Service* with no input parameters (as it uses data provided by the DBMS), but a return parameter of type image.

[1] http://sdqweb.ipd.kit.edu/wiki/Model-Driven_Confidentiality_in_CBS

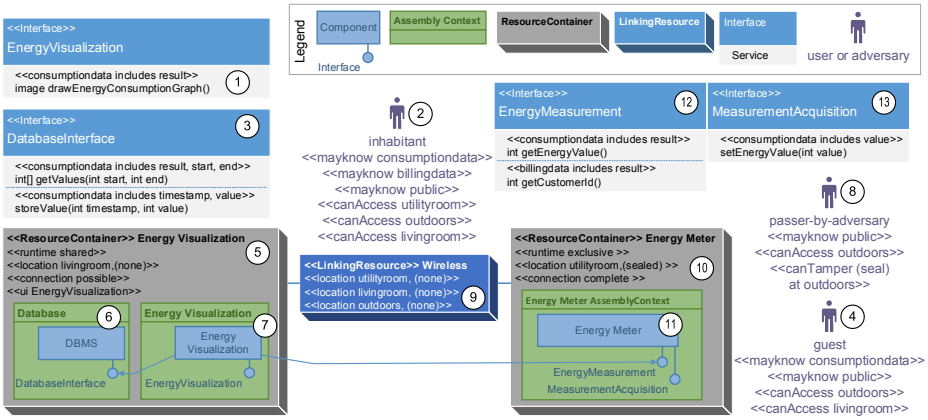

Fig. 2. Specification of the example scenario

An *Interface* groups multiple *Services*. E.g. DatabaseInterface (Figure 2, ③).

A *Component* is a reusable building block of software that is defined independent of its realization. Component-based software engineering is a contract-oriented development approach: every *Component guarantees* to provide *Services* via *Interfaces* if the environment provides the *Services* that are explicitly required by the *Component*. The *Component* Energy Meter in Figure 2, ⑪ provides the *Interface* EnergyMeasurement, which is required by the *Component* Energy Visualization (⑦). From the provides and requires relations between components and interfaces we can infer relations between components and their provided and required services.

To use a *Component* in a concrete system, the *Component* has to be instantiated in an *Assembly Context*. Every assembly context instantiates exactly one component, but a component can be instantiated in several assembly contexts. In the compact representation in Figure 2 the *Assembly Contexts* are represented as boxes surrounding *Components*, e.g. Energy Meter at ⑪.

A *Resource Container* represents hardware nodes, e.g. servers or embedded devices, on which *Assembly Contexts* are deployed. Every assembly context can be deployed in at most one resource container of a resource environment, but a resource container may host several assembly contexts. Our example has two *Resource Containers* in which two (⑤) and one (⑩) assembly contexts are deployed.

For communication, links between *Resource Containers* can be specified using *Linking Resources*. In the scenario, there is only one *Linking Resource* (Figure 2, ⑨) connecting the two resource containers (⑤ and ⑩).

4 Confidentiality Specification

We present a model-based security specification language with a formal basis. To express security-relevant properties for model elements of a component-based

system, we extended the concepts of Palladio Component Model with specification primitives, based on UML-like Stereotypes. The specification approach achieves a separation of concerns by specifying confidentiality in two steps:

First, domain-specific labels can be added to resource containers and linking resources to define how systems and components can be accessed by a user of the system, and possible adversaries. Second, input- and output-information of components can be divided into groups to specify information flow properties implied by these groups.

We give an informal presentation of the specification language, covering information flow due to parameters. Formal definitions, and treatment of information flow due to *presence* of service calls, are provided in a technical report [8].

Domain-Specific Labels The security requirements for a system usually depend on the domain they are used in. We consider three dimensions of security to be domain-specific: *Locations* on which hardware and wiring is physically installed, *Tamper Protection* mechanisms which are used for securing hardware, and types of information processed by the system: *Datasets*.

For each dimension, the domain expert defines labels. Alternatively, it is possible to define the labels according to levels of protection, like *high-security area* and similar. The relevant anti-tampering mechanism in the example in Figure 2 are represented by the labels sealed, and the default none.

To describe different types of information, *Datasets* are used. In the example, they are defined according to the purpose of information by using consumptiondata, billingdata, and public as labels. In other cases, it may be reasonable to define datasets according to stakeholders, who may have an interest in some kind of information, or according to security classifications of information.

Hardware Protection and Accessibility Hardware can be protected from physical manipulation by placing it on a secure location or by preventing persons from physically tampering with the hardware. Applying this stereotype specifies that a *Resource Container* or *Linking Resource* is placed on the specified location and secured against tampering by the specified mechanisms. A cable, for example, may bridge several locations with different security levels and may be secured differently at each location. In Figure 2, the Energy Meter ⑩ is specified to be installed in the utility room and it is protected against tampering using a seal. In order to validate the configuration and deployment of a system, a checklist for each resource container and linking resource can easily be generated from the model.

Connectivity Linking Resources represent connections between resource containers in the model. Apart from the connections explicitly modeled, a resource container may provide further possibilities for physical connections. If we assume that there can be no connection that is not modeled, we would have to model all connections for every container in every system, which is not feasible. Instead, we provide three options: The stereotype ≪*connection* complete≫ applied to a resource container expresses that only explicitly modeled connections can exist. ≪*connection* possible≫ expresses that the resource container does provide further ports that are currently not connected, for example, USB or Ethernet

ports. To express that some of the existing connections are not modeled, the stereotype ≪*connection* existing≫ is used. In our scenario, the Energy Meter ⑩ is a closed system that only has a wireless connection. The Energy Visualization ⑤, however, does provide further possibilities for connections.

Hardware Usage Whether additional software is deployed on a resource container has implications on the confidentiality of information. Similar to the problem of whether or not to completely specify all possible connections of a resource container, it is important whether all deployed software is modeled. The stereotype ≪*runtime* shared≫ expresses that further software might be deployed on a system, as shown in the scenario for the Energy Visualization ⑤. If the system is closed and all software is represented in the model the stereotype ≪*runtime* exclusive≫ is used, as for the Energy Meter in our example ⑩.

Confidentiality of Information The goal of our specification and analysis approach is to ensure the confidentiality of information provided to the modeled system as input and provided by the system as output. The stereotype ≪*d includes P*≫, with dataset d and parameters P, specifies for a service that all information in the parameters in P is contained in the dataset d.

In our scenario, the interface specification for Measurement Acquisition ⑬ expresses that all information in the input to the service setEnergyValue() is in the dataset consumptiondata, representing information about the consumption of the household. The specification of the Energy Visualization interface ① states that all information in the output of the service is in the dataset consumptiondata.

These abstract information flow specifications using datasets allow a black box view on components. Thus, domain experts only require the specification of a component to judge which information may be contained in an output.

Encryption of Communicated Information Encryption can be used to ensure the confidentiality of information communicated over insecure channels. But even for encrypted data some information, like the size of the data or protocol control information (IP addresses, MAC addresses, ...) cannot be kept confidential. Therefore, the stereotype ≪*encrypts except* d≫ can be applied to a *Linking Resource*, to express that all information communicated over it is encrypted except for information in dataset d. If a *Linking Resources* does not have any ≪*encrypts except* d≫ annotation, no encryption is specified to be used at all.

5 Adversary Specification

Our contractual approach towards adversary specification is driven by physical accessibility of adversaries to locations, and abilities an adversary is willing to use. The goal is to support abstract adversary modeling in cooperation with domain experts who can assess which persons have physical access to which parts of the system, allowing decisions on which types of adversaries and what abilities are considered. This can be done in an iterative process based on previous analysis results or, for example, based on a risk assessment step. The domain expert can exclude particular types of adversaries due to other security measures, e.g. access control to the premises, guards on the premise or camera surveillance.

We model adversaries as entities that are potentially interacting with the system under design. They can represent persons with malicious intentions, but also legitimate users. An adversary is graphically represented as shown in Figure 2 (②, ④, ⑧) and three properties can be specified: First, which domain-specific locations an adversary may gain access to. Second, which actions an adversary is willing to perform in terms of physically tampering with hardware of a system. Finally, which information an adversary is intended to gain access to.

An adversary has physical access to a *Location l* if the stereotype ≪*canAccess* l≫ is applied to it. In Figure 2, the adversary inhabitant ② has access to all locations in the domain, while passer-by ⑧ is only able to access the locations outside of the house. It is up to the domain expert to know why an adversary does not have physical access to other locations and by which means this is ensured in order to model the right properties.

Even if an adversary is, in principle, able to circumvent a given anti-tampering mechanism, he might not be willing to do so. For example, an adversary might be able to pick a lock, and yet refrain from doing if he has to be afraid of being caught while tampering with it due to video surveillance. Therefore, an adversary's willingness of tampering with a given protection mechanism depends on the location. Again, it is up to the domain expert to assess which assumptions according the willingness and ability of tampering should be made.
If an adversary is labeled with the stereotype stereotype ≪*canTamper* (T) at l≫, he is willing and able to overcome tamper protections T at location l. In order to actually tamper, the adversary additionally has to have access to the respective location.The adversary passer-by in the example specification (Figure 2, ⑧) has the ability to overcome seals, but not access to the utility room.

Finally, an adversary might be a legitimate user of a systemand therefore obtain knowledge about *some* information processed in the system. In many non-trivial systems everybody may get knowledge about some but not all information, for example: the current time of the system or information displayed on a welcome screen on an ATM. Therefore, we provide the possibility to specify that an adversary may know about information of a dataset d using the stereotype ≪*mayknow d*≫. In our example the adversary guest (Figure 2, ④) may gain knowledge about the consumption data in the system, but he may not get any information about the billing data.

6 Confidentiality Analyses

The security properties (Section 4, 5) for a architecture model (Section 3) express requirements to be fulfilled by the system. The intra-component information flow requirements as expressed by ≪*d includes P*≫ annotations can be checked using formal code-level analyses on the actual implementation. Other requirements, such as expressed by ≪*canAccess* l≫, can only be checked informally.

Fulfillment of every single requirement, however, is not sufficient to ensure confidentiality. It also has to be ensured that no violations of confidentiality are inherent to the architecture model. To this end, we propose an automatic access

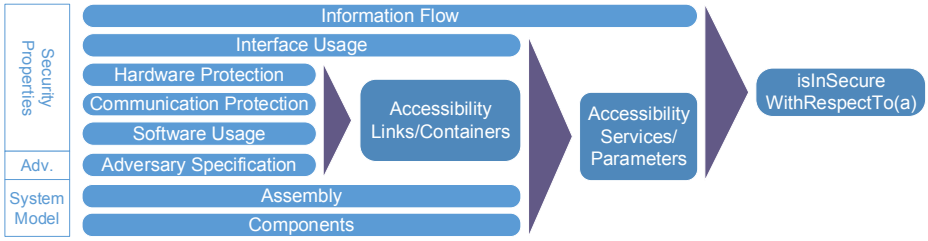

Fig. 3. Work flow of the three-stage architectural access analysis

analysis. This analysis reports every unavoidable leak and therefore identifies all faulty architecture models. Only if system external properties do not hold although they are guaranteed, for example because cryptographic keys are not used correctly, the analysis result could be considered a false positive.

6.1 Access Analysis

Under the assumption that

- each component fulfills its information flow specification,
- the protection mechanisms for resource containers and linking resources are properly employed, and location specifications are followed,

the access analysis determines whether the system is *vulnerable* against any of the specified adversaries, i.e. whether they can obtain unintended access to information. For every vulnerability the analysis generates an *explanation* of how a given adversary may exploit it. The expanation guides the architect or domain expert to those model and specification elements that cause the vulnerability.

The analysis process is depicted in Figure 3. From the architecture and adversary models and the confidentiality specification the analysis determines which links, containers, and interfaces are accessible to a given adversary. It then checks whether the adversary can access information that is not intended for him by these means, e.g. by circumventing a protection mechanism at a location.

As explained in the technical report [8], the analysis is designed for extensions to the specification primitives.

Our analysis is explained in terms of *inference rules* that formalize the intuition given for the different security properties.

Similarly to the relational notation of the architecture model (Section 3), the confidentiality and the adversary specifications are stated in terms of relations between model elements. For example, the annotation $\ll ds\ includes\ P \gg$ states that the relation *includes* holds for all $p \in P$, i.e.: $includes(ds, p)$. In addition to these relations that we directly obtain from the architecture model and the confidentiality- and adversary-specification, we define *inferred* relations (Figure 4), such as $containersFullyAccessibleBy(a, rc)$, which holds whenever an adversary a can – by whatever means – obtain access to all information available at a resource container rc. Ultimately, we are interested in the inferred relation $isInsecureWithRespectTo(a)$, where a is an adversary.

Links/Containers
$containerAccessibleBy(a, rc, l)$
$containersFullyAccessibleBy(a, rc)$
$linkAccessibleBy(a, link, l)$
$linksDataAccessibleBy(a, link, d)$

Services/Parameters
$accessibleParameters(a, p)$
$observableServices(a, s)$

Interface Usage
$providedInterfacesAccessibleTo(a, i)$
$requiredInterfacesAccessibleTo(a, i)$

$parameterAllowedToBeAccessedBy(a, p)$
$serviceAllowedToBeObservedBy(a, s)$
$isInsecureWithRespectTo(a)$

a : *Adversary*, rc : *Resource Container*, l : *Location*, $link$: *Linking Resource*, ds : *Dataset*, i : *Interface*, p : *Parameter*, s : *Service*

Fig. 4. Overview of inferred access relations

As an example, consider the inference rule for *sharing* and *location*:

$$\frac{containerAccessibleBy(a, rc) \qquad sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{possible})}{containersFullyAccessibleBy(a, rc)}$$

It defines that any information that reaches a resource context rc is fully accessible to an adversary a if a

- can access an resource container physically, i.e. by reaching its location,
- rc is openly shared, i.e. an adversary can coerce some (unmodeled) program running on rc to access data processed by a modeled component on rc,
- and further connections are possible (to communicate data to the adversary)

The access analysis depicted in Figure 3 is *staged*: relations of one stage are inferred only from relations of preceding stages. Accessibility relations for links and resource containers, for example, are inferred only from the hardware protection, communication protection, software usage, and tampering-properties. Figure 7 in the Appendix shows a selection of the corresponding inference rules.

The analysis answers the question "Is the architecture model vulnerable with regard to the modeled adversaries?" For the example scenario in Figure 1, we could start by inspecting the adversary guest. One might be tempted to conclude that there are no vulnerabilities concerning guest, because billingdata is only available from the interface EnergyMeasurement, which is not provided to users by the system. A guest can also access consumptiondata, which is intended. The automated access analysis, however, exposes four vulnerabilities for guest.

The explanation for the first vulnerability is that it is possible for guest to obtain access to billingdata via return values of the service getCustomerId. This service is part of the interface that is used for the link Wireless to connect the resource containers Energy Visualization and Energy Meter. This link is located at livingroom, which is accessible to guest, but billingData is *not* encrypted on the link Wireless. Thus, guest has unintended access to billingData via the link Wireless. The access analysis reports a second, similar vulnerability at location outdoors and a third and fourth vulnerability because the resource container Energy Visualization has a shared runtime and possible connections but lacks tamper protection.

Given such an explanation, the architect may decide to either require stronger guarantees in the security specification, to alter the architecture model, or to leave models and specification unchanged, but to document and assess the possible risk associated with the given vulnerability and the corresponding adversary.

To eleminate the first two vulnerabilities in the sample scenario, an obvious modification of the security specification is to require encrypted communication by labeling Wireless with $\ll$*encrypts except* public$\gg$. This modification imposes new requirements on implementations of the linking resource Wireless.

Alternatively, one may decide to remove any services dealing with billingdata from the interface EnergyMeasurement and to put them into a new interface that is *not* required by the component EnergyVisualization. Implicitly, this forbids the component EnergyVisualization to call any of these services. Hence, no billingdata is transmitted along Wireless, and there is no need to encrypt communication via the Wireless link, at least to achieve confidentiality with respect to guest.

Implementation The inference rules form a stratified [9] (even: hierarchical [10]), allowed – and hence: flounder-free [11] – general logic program. Therefore, we were able to directly implement them in Prolog. The architecture model, adversary specification and security properties are expressed as Prolog *facts* and the model entities are *atoms*.

For a given adversary a, the analysis result is simply the answer to the query `isInsecureWithRespectTo(a)`. Whenever it succeeds, the system is insecure, and for each violation an explanation is given in form of a proof-tree, which can be generated by a suitable proof-collecting Prolog (meta-)interpreter [12]–[15]. The explanation given in the example above is just an informal rendering of the proof tree where the leafs are *facts*. Figure 6 in the Appendix shows the proof tree for the first vulnerability concerning the adversary guest.

6.2 Code Level Analysis

Annotations $\ll d$ *includes* $P\gg$ specify information flow requirements for parameters and return values of components. In order to check whether a component fulfills these requirement and does not – by accident or on purpose – leak information, we formally verify *non-interference*: Input and output is classified as *low* or *high* and it is required that low output is at most influenced by low input, i.e. low output cannot reveal any information about high input.

The datasets and the information flow specification for parameters and return values implies such a classification for our code analysis: Each in- and output included in a dataset d is considered *low* and every other in- and output is considered *high*. A components implementation fulfills its information flow specification if it is non-interfering with respect to these parameter classifications.

Non-Interference for components is compositional, because non-interferent components can be composed to non-interferent compositions and systems. A thorough account on non-interference in component-based systems is out of scope of this paper, but can be found in [16].

Several tools for language based security can verify non-interference program code. Two such tools for Java are KeY[4] and JOANA[5]. Both support the specification of non-interference requirements via source code annotations in Java programs, which can be derived from $\ll d$ *includes* $P \gg$ annotations. Similarly, tool independent information flow policies for Java programs in the RIFL specification language[17] can be derived.

Code and architecture analyses do not depend on each other, even if models and code evolve: the architectural analysis can be used if parts of the code specification are outdated and the code analysis can be used if the architectural model is outdated as long as the specification input was corrected according to code changes. We are currently improving the support for co-evolution of architectural models, specification and code. If the architectural model is outdated, the code can still be verified if the specification input was kept consistent with code changes. Likewise, the architectural analysis could still be used if generated code stubs are not only completed but also modified as long as these changes are not in conflict with the architectural model. Fully automated co-evolution of code, specification and implementation is an unsolved problem in the literature but we already keep parts of JML specifications consistent with Java code [18].

7 Case Studies

We applied our approach to two case studies to evaluate its applicability: a distributed travel planner system from literature [19] and a two tier cloud hosting system. They can be downloaded from the homepage referenced in Section 1.

The travel planer system of the first case study consists of a travel agency, an airline, a travel planer and a credit card center component. The system offers services for booking flights, confirmation of payment information, setting available flights and others to stakeholders like the airline, a travel agency and customers. An example for a requirement for the confidentiality of information in this system is "The user's credit card data does not flow to the travel agency."

We extended the original specification with explicit adversaries, links and physical access properties. In order to keep our specification as close to the original as possible, we used datasets for each invovled party. The original system defines a temporal declassification property of the form "The credit card data flows to the airline only after explicit confirmation and declassification". We modeled this property by adding a declassifying service to the model.

We applied our architectural analysis to the model and verified that there is no architectural leak. As our approach does not explicitly support temporal declassification, we are, however, not able to verify the information flow in the source code of the declassifying component with KeY or JOANA.

The second case study is based on a typical multi-tenant cloud service – a file synchronization service. Two tiers are involved in providing that service: The cloud service provider, who is responsible for the synchronization service, and the cloud platform provider, who provides the (virtual) machines and the network infrastructure. As service consumers, we consider two end users and a guest

user. The deployment of the architecture involves two application servers for the synchronization service behind a load balancer. We modeled an API similar to WebDAV for the access to the files. Additionally, we modeled a database and a file server which are used to store the end user data. Each server has an administration and a logging component.

The two case studies showed that we can model and analyse examples from literature and more complex systems to identify confidentiality vulnerabilities.

8 Related Work

Various approaches for the design or analysis of distributed systems that process confidential data have been presented in the literature. But to our knowledge, no method for verifying confidentiality requirements both at the design and implementation level according to architectural confidentiality requirements for abstract components was presented so far. A thorough review of model driven development approaches focusing on security properties can be found in [20]. We restrict our presentation here to the most prominent and closest approaches.

UMLsec [1], [21], for example, provides analyses for security properties that can be checked formally for the design of individual components. The approach does, however, not combine results of implementation and design analyses and an only provides an explicit link to source code for role-based access control (RBAC) [22]. UMLsec does not support modeling physical aspects of a system.

Another related approach is IFlow [19], which introduced the travel planner system that we used as a case study. The component model of IFlow is very similar to that of our approach. IFlow uses application-specific security domains and a specification of allowed flows between security domains to specify the confidentiality of messages (i.e. service calls). As we showed in the case study, our approach allows a very similar specification technique using datasets, but also provides additional specification possibilities. IFlow does not support explicit adversaries, specification of physical accessibility properties or links. But IFlow supports explicit temporal declassification of information. For our approach this has to be emulated on the model level because we do not support it on the source code level. Further, IFlow supports behavioral modeling of components and services using UML sequence diagrams. In contrast to our approach, which enables automated generation of implementation level specifications for KeY or JOANA, IFlow does not provide a bridge to the actual implementation.

Other approaches are restricted to different security concerns, e.g. RBAC for SecureUML [23], or to special domains, e.g. web-services for SECTET [2] or smartcards for SecureMDD [3]. They also lack information flow analyses for code.

The access analysis we propose in this paper is formulated in terms of logical facts and inference rules. It can be seen as an automatic generation of *attack trees* (e.g. [24]) from an architectural system description. Interpreted this way, the adversary capabilities modeled in our approach determine whether the attacks (the tree's leafs) are feasible. Attack tree generation from process algebraic specifications instead of architectural specifications is described in [25]. [26] gen-

erate attack graphs from network models, also modeled as logical inference rules. There, the possibility of multi-stage network penetration attacks not the confidentiality of information is analyzed.

Note that it is not necessary to express security mechanisms and patterns explicitly with our approach. If the effects of a mechanism or pattern are correctly specified, the analysis can consider the resulting information flow. Therefore, it indirectly supports patterns such as complete mediation or minimum exposure.

9 Conclusions and Future Work

We presented an integrated method for the specification and analysis of confidentiality in component-based systems. It requires abstract information set specifications for the in- and output of a system and its components as well as accessibility specifications for hardware and communication links. Based on this confidentiality specification and additional adversary models, we presented architecture and code analyses, which can be used to eliminate faulty designs, data leaks and specification violations.

Our approach provides a consistent way to specifiy confidentiality at different stages of development and takes confidentiality into account from the very beginning. Information about the deployment in terms of resource containers, locations, and linking resources as well as user roles and datasets can be specified with custom labels for confidentiality requirements. Based on the judgment of domain experts, it can be specified which users and adversaries are allowed to know information of which datasets and which locations they are allowed to access. Similar, for resource containers and linking resources it can be defined how they can be accessed at specific locations and how they are protected against tampering. By assigning parameters and return values to datasets, we are able to infer which information flow is permissible, thus complementing code-level non-interference analysis methods. At model level, possible violations of confidentiality can be identified in an automated architecture analysis by taking the abilities of adversaries into account. By focusing on the information flow at an abstract level, our analysis method reveals weaknesses at an early design stage but also offers a way to re-use components and code analysis results in a confidentiality-preserving manner.

An obvious line of future work is to implement support for security properties apart from confidentiality, e.g. integrity. One could also add support for frequently used protection mechanisms, like role-based access control. On the level of components that are deployed on a resource container, it could be useful to have a possibility of modeling sandboxing mechanisms. Furthermore, we plan to evaluate how much vulnerabilities can be identified with our architectural analysis based on histories of architecture models.

Instead of checking for vulnerabilities with respect to a given set of adversaries, we could take advantage of the fact that the access analysis is implemented by a logic program and *infer* those adversaries that are able to obtain access to a given dataset by *abductive logic programming* (e.g.: [27]). We plan to investigate

the practicability of this approach, e.g. by using a suitable notion of *weakest* adversaries.

References

[1] J. Jürjens, *Secure systems development with UML*. Springer-Verlag, Berlin, Germany, 2005, UMLSec.

[2] M. Alam *et al.*, "Model-driven security engineering for trust management in sectet," *Journal of Software*, vol. 2, no. 1, pp. 47–59, 2007, SECTET.

[3] N. Moebius *et al.*, "A modeling framework for the development of provably secure e-commerce applications," in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, SecureMDD, 2007, pp. 8–8.

[4] C. Scheben and P. H. Schmitt, "Verification of information flow properties of java programs without approximations," in *FoVeOOS*, 2011, pp. 232–249.

[5] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009.

[6] R. Reussner *et al.*, "The Palladio Component Model," KIT, Fakultät für Informatik, Tech. Rep., 2011.

[7] C. Szyperski *et al.*, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. ACM Press and Addison-Wesley, 2002.

[8] M. E. Kramer *et al.*, "Model-driven specification and analysis of confidentiality in component-based systems and its formal foundation," Department of Informatics, Karlsruhe Institute of Technology, Tech. Rep., 2016, to be published.

[9] K. R. Apt *et al.*, "Foundations of deductive databases and logic programming," in, Morgan Kaufmann Publishers Inc., 1988, ch. Towards a Theory of Declarative Knowledge, pp. 89–148.

[10] K. Clark, "Negation as Failure," in *Logic and Data Bases*, Springer US, 1978, pp. 293–322.

[11] J. W. Lloyd and R. W. Topor, "A basis for deductive database systems II," *The Journal of Logic Programming*, vol. 3, no. 1, pp. 55–67, 1986.

[12] G. Pemmasani *et al.*, "Online justification for tabled logic programs," in *Functional and Logic Programming*, ser. LNCS, vol. 2998, Springer Berlin Heidelberg, 2004, pp. 24–38.

[13] H.-F. Guo *et al.*, "Speculative beats conservative justification," in *Logic Programming*, ser. LNCS, vol. 2237, Springer Berlin Heidelberg, 2001, pp. 150–165.

[14] G. Specht, "Generating Explanation Trees even for Negations in Deductive DataBase Systems." *LPE*, vol. 1993, pp. 8–13, 1993.

[15] C. Viegas Damásio *et al.*, "Justifications for logic programming," in *Logic Programming and Nonmonotonic Reasoning*, ser. LNCS, vol. 8148, Springer Berlin Heidelberg, 2013, pp. 530–542.

[16] D. Grahl and S. Greiner, "Non-interference with what-declassification in component-based systems," Department of Informatics, Karlsruhe Institute of Technology, Tech. Rep. 2015,10, 2015.

[17] S. Ereth *et al.*, "Towards a common specification language for information-flow security in rs3 and beyond: rifl 1.0 - the language," TU Darmstadt, Tech. Rep. TUD-CS-2014-0115, 2014.

[18] M. E. Kramer *et al.*, "Change-driven consistency for component code, architectural models, and contracts," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, ser. CBSE '15, ACM, 2015, pp. 21–26.

[19] K. Stenzel *et al.*, "A model-driven approach to noninterference," *JoWUA*, vol. 5, no. 3, pp. 30–43, 2014.

[20] P. H. Nguyen *et al.*, "An extensive systematic review on the Model-Driven Development of secure systems," *Information and Software Technology*, vol. 68, pp. 62–81, 2015.

[21] J. Jürjens, "Umlsec: extending uml for secure systems development," in *«UML»2002 – The Unified Modeling Language*, ser. LNCS, vol. 2460, Springer Berlin Heidelberg, 2002, pp. 412–425.

[22] L. Montrieux *et al.*, "Tool support for code generation from a umlsec property," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, ACM, 2010, pp. 357–358.

[23] D. Basin *et al.*, "Model driven security: from uml models to access control infrastructures," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 39–91, 2006, SecureUML.

[24] C. Salter *et al.*, "Toward a Secure System Engineering Methodolgy," in *Proceedings of the 1998 Workshop on New Security Paradigms*, ser. NSPW '98, ACM, 1998, pp. 2–10.

[25] R. Vigo *et al.*, "Automated Generation of Attack Trees," in *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, IEEE, 2014, pp. 337–350.

[26] X. Ou *et al.*, "A Scalable Approach to Attack Graph Generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, ACM, 2006, pp. 336–345.

[27] M. Denecker and A. C. Kakas, "Abduction in logic programming," in *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, Springer-Verlag, 2002, pp. 402–436.

Appendix

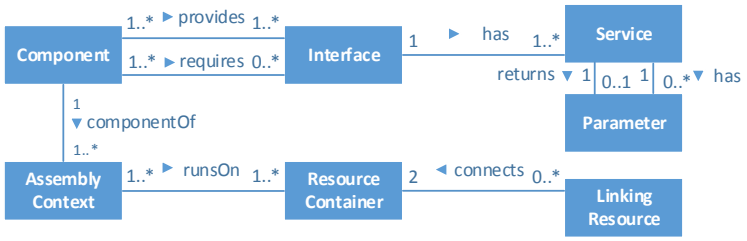

Fig. 5. Entities of architectural models and their relationships

```
isInSecureWithRespectTo(guest)
+- accessibleParameters(guest,return(getCustomerId))
|  +- linksDataAccessibleBy(guest,wireless,billingData)
|  |  +- linkAccessibleBy(guest,wireless,livingRoom)
|  |  |  +- linkLocation(wireless,livingRoom,none)
|  |  |  '- locationsAccessibleBy(guest,livingRoom)
|  |  +- linkLocation(wireless,livingRoom,none)
|  |  +- tamperingAbilities(guest,livingRoom,none)
|  |  |  '- locationsAccessibleBy(guest,livingRoom)
|  |  '- exposesPhysicallyAccessibleDataTo(wireless,guest,billingData)
|  |     '- encryptsExcept(wireless,billingData)
|  +- connects(wireless,energyVisualizationRC,energyMeterRC)
|  +- hasService(energyMeasurement,getCustomerId)
|  +- requires(energyVisualization,energyMeasurement)
|  +- parametersOf(getCustomerId,return(getCustomerId))
|  '- includes(return(getCustomerId),billingData)
'- not parameterAllowedToBeAccessedBy(guest,return(getCustomerId))
   +- includes(return(getCustomerId),billingData)
   '- not mayknow(guest,billingData)
```

Fig. 6. Proof tree of a vulnerability (excerpt)

$$\frac{location(rc, loc, t) \qquad locationAccessibleBy(a, loc)}{containerAccessibleBy(a, rc, loc)}$$

$$\frac{linkLocation(link, loc, t) \qquad locationAccessibleBy(a, loc)}{linkAccessibleBy(a, link, loc)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, loc) \\ tamperingAbilities(a, loc, t) \qquad location(rc, loc, t)\end{array}}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, loc) \\ sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{possible})\end{array}}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{existing})}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{\begin{array}{c}linkAccessibleBy(a, link, loc) \\ tamperingAbilities(a, loc, t) \qquad location(rc, loc, t) \qquad encryptsExcept(link, ds)\end{array}}{linksDataAccessibleBy(a, link, ds)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, l) \\ providedInterfacesOn(rc, i) \qquad uiInterfaceOn(rc, i)\end{array}}{providedInterfacesAccessibleTo(a, i)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, l) \\ requiredInterfacesOn(rc, i) \qquad uiInterfaceOn(rc, i)\end{array}}{requiredInterfacesAccessibleTo(a, i)}$$

$$\frac{\begin{array}{c}providedInterfacesAccessibleTo(a, i) \\ hasServices(i, s) \qquad returnParameter(s, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{requiredInterfacesAccessibleTo(a, i) \qquad hasServices(i, s) \qquad hasParameter(s, p)}{accessibleParameters(a, p)}$$

$$\frac{\begin{array}{c}containersFullyAccessibleBy(a, rc) \\ providedInterfacesOn(rc, i) \vee requiredInterfacesOn(rc, i) \\ hasServices(i, s) \qquad returnParameter(s, p) \vee hasParameter(s, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{\begin{array}{c}linksDataAccessibleBy(a, link, ds) \qquad connects(link, rcleft, rcright) \\ runsOn(assemblyleft, rcleft) \qquad runsOn(assemblyright, rcright) \\ componentOf(assemblyleft, componentleft) \qquad requires(i, componentleft) \\ systemAssembledTo(assemblyleft, i, assemblyright) \qquad hasServices(i, s) \\ returnParameter(s, p) \vee hasParameter(s, p) \qquad includes(ds, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{includes(p, ds) \qquad mayknow(a, ds)}{parameterAllowedToBeAccessedBy(a, p)}$$

$$\frac{accessibleParameters(a, p) \qquad \neg parameterAllowedToBeAccessedBy(a, p)}{isInsecureWithRespectTo(a)}$$

Fig. 7. Access Analysis: selected inference rules. Rules concerning the observation of service *calls* (i.e.: their presence, but not their content) are omitted.