Architecture-Driven Reduction of Specification Overhead for Verifying Confidentiality in Component-Based Software Systems

Kateryna Yurchenko, Moritz Behr, Heiko Klare, Max Kramer and Ralf Reussner
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: firstname.lastname@kit.edu

***Abstract*—Code verification techniques can be used to guarantee that some of the information processed in software systems remains confidential. For this, allowed information flows have to be specified for the system under analysis. Reducing the specification overhead could render code verification feasible where verification was considered too complex or costly so far. In this paper, we introduce a model-driven approach to reduce the overhead for creating and maintaining such specifications. Independent of the verification input format, developers can specify confidentiality for component-based architecture models, which are kept consistent with object-oriented code. They are supported in adapting the specifications to evolving systems in order to detect information leaks with less effort and in earlier development stages.**

I. Introduction and Motivation

For many software systems, the confidentiality of processed data is important. To guarantee this non-functional property, it is possible to verify the flow of information in the code. For this, confidentiality requirements have to be specified for every system under analysis. The effort required to specify and verify confidentiality represents a design and development overhead. A part of this effort is inevitable, and another part is due to accidental complexity. As a result, it is either more costly than necessary to develop secure software, or the software is less secure than required.

In this paper, we present an approach for reducing the overhead for creating and maintaining confidentiality specifications using the component-based development paradigm and model-driven techniques. With this approach 1.) confidentiality can be specified for services that are defined for component interfaces, 2.) architectural specifications for the component architecture are automatically translated into proof obligations for code verification, and 3.) evolving architectures, confidentiality specifications as well as code are kept semi-automatically consistent with each other. As a result, developers do not need to consider the internal realization of components or details of verification techniques. Furthermore, they can read and modify confidentiality specifications on their level of abstraction because specifications in the architecture and in the code are kept consistent with each other. Finally, by specifying confidentiality for architectural designs, developers can account for these requirements when developing or selecting components for reuse in a particular system. This can prevent late errors that are costly to fix.

II. Background and Foundations

Before we present our approach to architecture-driven confidentiality specifications, we briefly discuss the fundamental techniques that we use.

A. Component-Based Software Design

Our approach supports confidentiality specifications for component-based software. Components are reusable software units that provide and require services, for which signatures are defined in interfaces. We use the Palladio Component Model (PCM) [1] as architectural description language. With the PCM, service signatures are modeled with a return type, a name, and parameters, which have a name and a type. Confidentiality is specified based on these service signatures in line with the central idea of component-based software development: the internal realization of components is hidden behind interface contracts, which specify properties of the input and output for provided and required services.

B. Code Generation and Consistency

Techniques to automatically derive code from models are called code generation or model-to-text transformation techniques. Software is, however, often not developed in a strict forward engineering process where models are not changed after they have been used to produce code. Therefore, techniques for preserving consistency between code and models were developed. We use the Vitruvius framework [2] and its reactions language to keep Java code consistent with PCM instances based on monitored changes.

C. Specification and Verification of Confidentiality using Non-Interference

Confidentiality is specified and verified with our approach based on the notion of *non-interference*. The intuition for preventing information leaks is that confidential information may not have any direct or indirect influence on non-confidential information. For this, it has to be specified which input data of a component's service shall not interfere. The observable behavior of the component and output of the service has to be equivalent for all calls that are equivalent with respect to this specification [3]. If information is classified as "high" (confidential) and "low" (non-confidential), then high inputs may not have any observable influence on low outputs. Low inputs, however, may influence high outputs. In general, information can be classified based on arbitrary lattices, i.e. partially ordered sets with unique supremum and infimum for any two elements.

From architectural specifications we derive proof obligations that can be verified using the theorem prover KeY [4]. KeY can be used for automated and interactive proofs of requirements that are expressed using dynamic logic and specified with the Java Modeling Language (JML) [5]. Non-interference is verified with KeY using two symbolic executions that only differ in terms of a confidential input [6].

III. Architecture-Driven Confidentiality

We will explain how confidentiality can be specified, verified and co-evolved in an architecture-driven way based on a simple groupware example.

A. Groupware Calendar Example

In Figure 1, we show three snippets for a groupware calendar system. The first snippet shows an architectural model for a groupware component with a confidentiality specification. The component provides a service yielding all periods that are blocked by calendar entries and that are scheduled between two given timestamps. The second service yields all details for a specific calendar entry, such as the location and participants.

B. Confidentiality Specification

Confidentiality is not directly specified on the code level but on the level of component-based architectures. This way, developers do not need to know the verification input format and information flow only needs to be specified for inter- but not for intra-component communication.

First, so-called *data sets* have to be defined. They group data that may interfere with each other but not with other data sets. If information may flow from and to these data sets, it is specified for service signatures of architectural interfaces. Other flows are not permitted.

In our example in Figure 1, confidentiality specifications are shown as notes with the keyword *includes*. For the service `getBlockedPeriods`, a star yields two allowed information flows: 1.) non-confidential information (low) may influence all inputs, 2.) returned values may influence low information. For the service `getFullCalendarEntry`, it is specified that 1.) low information may influence the provided ID, 2.) returned values may only influence confidential information (high).

C. Code Generation and Verification

To obtain a fully functional and verified system, we generate, complete, and verify code in four steps. First, Java code and confidentiality annotations are generated from PCM instances and the architectural specifications. Then, the empty method stubs for component services are manually completed. Next, the completed code with confidentiality annotations is copied and the annotations are translated into JML proof obligations. In this step, the generator produces all required JML proof obligations for cases that do not need to be distinguished by developers in architectural specifications, such as different obligations for inputs and outputs to service calls. Finally, the code copy is verified by proving the JML obligations using KeY (subsection II-C). Figure 1 illustrates these two code representations of confidentiality specifications as Java annotations (middle), and as JML proof obligations (right). The purpose of the annotations is to reduce the complexity for developers by providing them a code representation of the architectural specifications. These two representations only have syntactic differences: data set definitions are turned into Java enums and stereotype applications are turned into annotation usages. The level of abstraction with data sets and information flow permissions only for inputs and outputs of component services is the same.

D. Consistency across Models and Code

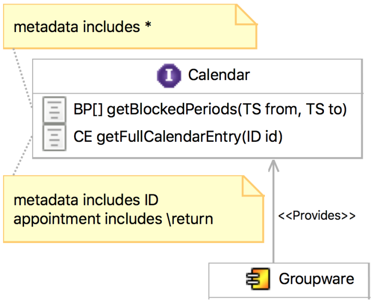
We introduce the intermediate annotation representation for confidentiality specifications not only to reduce specification complexity, but also to ease automated consistency preservation for it. The goal is to support development processes in which code, architectural models and confidentiality specifications may co-evolve. To this end, we reuse a mechanism that keeps component-based architectures (PCM instances) and Java code consistent with each other by reacting to monitored changes (subsection II-B). We are currently extending this mechanism to keep confidentiality specifications on the architectural level consistent with specifications in code. The JML input for KeY is, however, always regenerated and therefore no consistency preservation is needed for it.

IV. Realization and Evaluation

In our prototypical implementation[1], data set definitions and pairs that link data sets and parameters or return values are persisted in a confidentiality specification model that is kept separate from the architectural models. Stereotypes of a confidentiality profile for PCM can be applied to service signatures of architectural interfaces in order to refer to these pairs as tagged values. By using this non-invasive profile mechanism the architecture models remain compatible to all Palladio tooling.

To obtain Java code and confidentiality annotations from PCM instances and specifications, we have created a generator based on Xtend. An additional generator for JML

[1] github.com/KASTEL-SCBS/PCM2Java4Key


metadata includes *
Calendar
BP[] getBlockedPeriods(TS from, TS to)
CE getFullCalendarEntry(ID id)
metadata includes ID
appointment includes \return
<<Provides>>
Groupware

```
enum DataSets {APPOINTMENT(),METADATA();}
enum IFPairs {
 METADATA_STAR(DataSets.METADATA,"*"),
 METADATA_ID(DataSets.METADATA,"id"),
 APP_RETURN(DataSets.APPOINTMENT,"\return");
}
interface Calendar {
 @NISPEC({IFPairs.METADATA_STAR})
 BP[] getBlockedPeriods(TS from, TS to);
 @NISPEC({IFPairs.METADATA_ID, IFPairs.APP_RETURN})
 CE getFullCalendarEntry(ID id);
}
```

```
//@ model \seq metadata;
//@ \determines this.getBlockedPeriods(\result)
//@ \by this.getBlockedPeriods(from),
//@   this.getBlockedPeriods(to)
//@ \preserving metadata;
BP[] getBlockedPeriods(TS from, TS to) {...}
//@ model \seq metadata;
//@ \by this.getFullCalendarEntry(id)
//@ \preserving metadata;
//@ model \seq appointment;
//@ \determines this.getFullCalendarEntry(\result)
//@ \preserving appointment;
CE getFullCalendarEntry(ID id) {...}
```

Figure 1. Architectural model and specification (left), Java code with annotations (middle), and JML comments (right) for a groupware calendar example

proof obligations is under development. Both generators process the complete input in batch mode and overwrite previous output. We are currently developing consistency preservation reactions, which update annotations after changes in architectural specifications. Java code and architectural models are already kept consistent based on previous work [2].

To ensure the quality of both batch generators and of the incremental reactions, we create unit tests that compare the obtained output with expected outputs for all types of specification possibilities and changes. Furthermore, we will perform cross-validation by always comparing the result of incremental consistency preservation with that of a batch generation of code and annotations.

We will evaluate how confidentiality can be specified and maintained in an architecture-driven way using three systems: a simple web shop, a cloud-based file sharing platform, and a common case study realizing a trading system for retail shops.

V. Related Work

Jürjens presents a model-driven approach where security-relevant information is specified on the system design level using the UMLsec extension for the Unified Modeling Language (UML) [7]. This approach does, however, not support automatic proof obligations generation for code.

The IFlow approach [8] allows the model-driven development of distributed systems consisting of mobile apps and web services which are secure w.r.t. information flow. Using the system's UML model, IFlow automatically generates code and a formal model of the system based on abstract state machines, which is used to verify information-flow properties. IFlow does not support automatic consistency preservation on both design and code levels during the system evolution.

To the best of our knowledge, there is no other approach for verifying component-based software against architectural confidentiality requirements or for preserving consistency between confidentiality specifications in software architecture descriptions and code.

VI. Conclusions

In this paper, we have presented a model-driven approach that facilitates the creation and maintenance of confidentiality specifications for the verification of component-based software. We have presented architecture-driven confidentiality specifications for which JML proof obligations can be automatically derived so that developers do not need to know the verification input. Furthermore, we have proposed a mechanism for keeping evolving architectures, confidentiality specifications and code consistent with each other. With it, developers can design and realize software with confidential data in an incremental way and on their level of abstraction. The goal of the approach is to help developers in finding confidentiality leaks with less overhead and in earlier development stages.

References

[1] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, 2016, 408 pp.

[2] M. E. Kramer, M. Langhammer, D. Messinger, S. Seifermann, and E. Burger, "Change-driven consistency for component code, architectural models, and contracts," in *18th International Symposium on Component-Based Software Engineering*, ser. CBSE '15, Montréal, QC, Canada: ACM, 2015, pp. 21–26.

[3] S. Greiner and D. Grahl, "Non-interference with what-declassification in component-based systems," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 253–267.

[4] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS 4334. Springer-Verlag, 2007.

[5] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *ACM SIGSOFT Software Engineering Notes*, 31, no. 3, pp. 1–38, 2006.

[6] C. Scheben and S. Greiner, "Information flow analysis," in *Deductive Software Verification – The KeY Book: From Theory to Practice*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds. Cham: Springer International Publishing, 2016, pp. 453–471.

[7] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *International Conference on The Unified Modeling Language*, Springer, 2002, pp. 412–425.

[8] K. Katkalov, K. Stenzel, M. Borek, and W. Reif, "Model-driven development of information flow-secure systems with iflow," in *Social Computing (SocialCom), 2013 International Conference on*, IEEE, 2013, pp. 51–56.