Karlsruhe Reports in Informatics 2017,9
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Modular Verification of Information Flow Security in Component-Based Systems – Proofs and Proof of Concept

Simon Greiner, Martin Mohr, and Bernhard Beckert

2017

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Modular Verification of Information Flow Security in Component-Based Systems – Proofs and Proof of Concept

Simon Greiner, Martin Mohr, and Bernhard Beckert

{Simon.Greiner, Martin.Mohr, beckert}@kit.edu, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany

1 Introduction

Distributed component-based systems engineering is a method that allows to modularily implement large and complex systems where the functionality provided by one component as so-called services, depends on functionality provided by other components. Systems implemented according to the component-based system paradigm allow high scalability. In the paper *Modular Verification of Information Flow Security in Component-Based Systems* [8] we propose a sound and modular method that allows to identify and verify flows of information caused by single services. We show how those individual flows can be formalized as so-called *dependency clusters*, and how dependency clusters can be used as building blocks to modularily construct system-wide security specifications in a sound way.

This technical report accompanies the original paper and we assume the reader to be familiar with the original work. We provide here the formal proofs for the theorems discussed there. Further, we provide a detailed description of the proof of concept addressed in the original work.

In the next section, we provide the formal proofs for all theorems stated in [8]. In Section 3 we present the proof of concept for the method described in [8]. First, we describe in detail the system, the target security specification and the components which make up the system. Then we discuss the mapping of the formal model to programs written in Java Enterprise Edition. We then present how we identify and verify dependency clusters using two different methods, one based on JOANA, one based on KeY, and present the formalizations and proof obligations for the system-wide non-interference properties. Finally, we provide the evaluation of the proof of concept in the conclusion.

2 Proofs

We refer to the execution of a service, started in state σ and terminating in state σ' by $\langle \sigma; handler_{serv} \rangle \xrightarrow{t} \langle \sigma'; SKIP \rangle$, while t represents the trace, i.e. the list of messages communicated during service execution. To argue about properties of traces, we use a filter operator $t \rhd M$ yielding a trace which is equal to t after

removing all elements from t which are not in M. The empty trace is denoted by $\langle\rangle$, $\frown$ is the concatenation operation, and $\leq$ is the prefix operation on traces. We further introduce the placeholder $\Box$ and define a message m as invisible, i.e. its existence is specified high, iff $m \sim \Box$.

For a service to be non-interferent, it must not to reveal its execution, if it was called using an invisible message. Also, the service must not change the low part of the state if called with an invisible message, since subsequent service executions may provide information about this change as outputs and therefore could indirectly reveal the execution of the secret service call. This property is formalized in the following definition.

We call a service *visibility-preserving*, if it only produces invisible outputs after receiving invisible inputs and does not change the low part of the state.

Definition 1 (Visibility-preserving Service). *A service serv is visibility-preserving with respect to $\sim$ and $\approx$, iff*

$$\begin{aligned}&\forall \sigma, \sigma', t, t' \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma'\rangle \implies \\ &\quad (t \rhd \mathbb{I} \sim \langle\rangle \implies t \rhd \mathbb{O} \sim \langle\rangle) \\ &\wedge (t \frown t' \rhd \mathbb{I} \sim \langle\rangle \implies \sigma \approx \sigma')\end{aligned}$$

Definition 2 (Service Non-Interference(Definition 2 in [8])). *A Service serv is non-interferent with respect to $\sim$ and $\approx$, written serv $\in \mathtt{SNI}^{\approx}_{\sim}$ iff it is* visibility-preserving *with respect to $\sim$ and $\approx$ and*

$$\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2 \cdot \quad \sigma_1 \approx \sigma_2 \tag{1}$$

$$\wedge \langle handler_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle SKIP; \sigma_1'\rangle \wedge \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle SKIP; \sigma_2'\rangle \tag{2}$$

$$\implies \tag{3}$$

$$(t_1 \rhd \mathbb{I} \sim t_2 \rhd \mathbb{I} \implies \sigma_1' \approx \sigma_2') \tag{4}$$

$$\wedge (\forall t_1' \leq t_1, t_2' \leq t_2 \cdot \ t_1' \rhd \mathbb{I} \sim t_2' \rhd \mathbb{I} \implies \tag{5}$$

$$\exists t_1' \frown t_1'' \leq t_1, t_2' \frown t_2'' \leq t_2 \cdot \ t_1' \frown t_1'' \sim t_2' \frown t_2'') \tag{6}$$

A service started in two equivalent states (1) has to terminate (2) in equivalent states, if the input provided by the environment is equivalent for both runs (4). Implicitly, this condition encodes a well-behaving environment in the sense that we assume the environment to not leak any high information.

As a second condition it has to be ensured that the service does not leak information by providing non-equivalent output to the environment after receiving equivalent input. (5) to (6) ensure that t_1 and t_2 are equivalent up to the first non-equivalent input. For all prefixes of the two traces produced during execution which got provided equivalent input (5), the traces either are equivalent, or at least contain further events such that they can become equivalent (6).

Theorem 1 (Compositionality of dependency clusters (Theorem 3 in [8])). *Let $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$ be dependency clusters for a service serv. Then*

the composition
$(\sim_1, \approx_1) + (\sim_2, \approx_2) \;:=\; (\sim_1 \cap \sim_2, \approx_1 \cap \approx_2)$ *is a dependency cluster for serv.*

Proof (for Theorem 1). Assume *serv* is non-interferent with respect to $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$. Let $\sim := \sim_1 \cap \sim_2$ and $\approx := \approx_1 \cap \approx_2$.

We first show that *serv* is visibility-preserving w.r.t. $(\sim, \approx)$. Select arbitrarily σ, σ', t, t' such that $\langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma'\rangle$ and $t \rhd \mathbb{I} \;\sim\; \langle\rangle$. By definition of $\sim$ and $\approx$, this implies $t \rhd \mathbb{I} \;\sim_1\; \langle\rangle$ and $t \rhd \mathbb{I} \;\sim_2\; \langle\rangle$. Since d_1 and d_2 are dependency clusters for *serv*, this means $t \rhd \mathbb{O} \;\sim_1\; \langle\rangle$ and $t \rhd \mathbb{O} \;\sim_2\; \langle\rangle$, and by definition of $\sim$ also $t \rhd \mathbb{O} \;\sim\; \langle\rangle$.

Now, we assume $t \frown t' \rhd \mathbb{I} \;\sim\; \langle\rangle$. Again, by definition of $\sim$ and $\approx$, this implies $t \frown t' \rhd \mathbb{I} \;\sim_1\; \langle\rangle$ and $t \frown t' \rhd \mathbb{I} \;\sim_2\; \langle\rangle$ and since d_1 and d_2 are dependency clusters $\sigma \approx_1 \sigma'$ and $\sigma \approx_2 \sigma'$ which implies by definition of $\approx$ also $\sigma \approx \sigma'$.

Therefore *serv* is visibility preserving w.r.t. $\sim$ and $\approx$.

Now we show non-interference.

Select $\sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2$ arbitrarily such that $\sigma_1 \approx \sigma_2$ and $\langle handler_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle SKIP; \sigma_1'\rangle$ and $\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle SKIP; \sigma_2'\rangle$

Now assume $t_1 \rhd \mathbb{I} \;\sim\; t_2 \rhd \mathbb{I}$, which, by definition of $\sim$ means $t_1 \rhd \mathbb{I} \;\sim_1\; t_2 \rhd \mathbb{I}$ and $t_1 \rhd \mathbb{I} \;\sim_2\; t_2 \rhd \mathbb{I}$.

Since *serv* is non-interferent w.r.t. $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$, we know that $\sigma_1' \approx_1 \sigma_2'$ and $\sigma_1' \approx_2 \sigma_2'$, and therefore by definition of $\approx$ also $\sigma_1' \approx \sigma_2'$.

With an argument of the same structure, we also get $\forall t_1' \le t_1, t_2' \le t_2 \cdot\; t_1' \rhd \mathbb{I} \;\sim\; t_2' \rhd \mathbb{I} \implies \exists t_1' \frown t_1'' \le t_1, t_2' \frown t_2'' \le t_2 \cdot t_1' \frown t_1'' \;\sim t_2' \frown t_2'')$

And therefore $d_1 + d_2$ is a dependency cluster for *serv*.

Theorem 2 ((Theorem 4 in [8])). *Let $\mathcal{C}$ be a callable set for service serv and $\mathbb{F}$ an assignable set for serv. A pair $(\sim_g, \approx_g)$ is a dependency cluster for serv if there is a dependency cluster $(\sim_{serv}, \approx_{serv})$ for serv such that, for all messages m, m' and states $\sigma, \sigma', \sigma_p, \sigma_p'$,*

$$\text{if } m \sim_g m' \text{ then } m \sim_{serv} m', \text{ and if } \sigma \approx_g \sigma' \text{ then } \sigma \approx_{serv} \sigma' \tag{7}$$

$$\text{if } m \sim_{serv} m' \text{ and } m \in \mathcal{C} \text{ then } m \sim_g m' \tag{8}$$

$$\text{if } \sigma \approx_g \sigma' \text{ and } \sigma_p \approx_{serv} \sigma_p' \text{ then } anon(\sigma, \mathbb{F}, \sigma_p) \approx_g anon(\sigma', \mathbb{F}, \sigma_p') \tag{9}$$

where $anon(\sigma, V, \sigma')$ yields a state σ_{anon} such that $\sigma_{anon}(v)$ evaluates to $\sigma'(v)$ if $v \in V$ and to $\sigma(v)$ otherwise.

Proof (for Theorem 2). Assume we have $\sim_g, \approx_g$ and $\sim_{serv}, \approx_{serv}$, such that conditions 7 to 9 hold.

We first show that *serv* is visibility-preserving with respect to $\sim_g, \approx_g$. Let σ, σ_p, t, t' such that $\langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma_p\rangle$, $t \rhd \mathbb{I} \sim_g \langle\rangle$. So, for all input messages m in t it holds that $m \sim_g \square$ and since condition 7 holds, it also holds $m \sim_{serv} \square$, and therefore $t \rhd \mathbb{I} \sim_{serv} \langle\rangle$. Since $(\sim_{serv}, \approx_{serv})$ is dependency cluster for *serv*, we know $t \rhd \mathbb{O} \sim_{serv} \langle\rangle$. Since $\mathcal{C}$ is callable set for *serv*, we know for all output messages m in t by Theorem 2, condition 8 we also know $m \sim_{serv} \square \implies m \sim_g \square$ and therefore $t \rhd \mathbb{O} \sim_g \langle\rangle$.

Further, we know $\sigma \approx_{serv} \sigma_p$ and since $\mathbb{F}$ is an assignable set, we know that $\sigma_p = anon(\sigma, \mathbb{F}, \sigma_p)$.

Since $\approx_g$ is an equivalence relation, we know $\sigma \approx_g \sigma$ and since $(\sim_{serv}, \approx_{serv})$ is a dependency cluster for $serv$, we know $\sigma \approx_{serv} \sigma_p$. With Theorem 2 condition 9, we get $anon(\sigma, \mathbb{F}, \sigma) \approx_g anon(\sigma, \mathbb{F}, \sigma_p)$, i.e. $\sigma \approx_g \sigma_p$.

Now we show equivalence of the post states: Assume $\sigma \approx_g \sigma'$ and furthermore $\langle handler_{serv}; \sigma\rangle \xrightarrow{t} \langle SKIP; \sigma_p\rangle$ and $\langle handler_{serv}; \sigma'\rangle \xrightarrow{t'} \langle SKIP; \sigma'_p\rangle$. Due to Theorem 2, 7, also $\sigma \approx_{serv} \sigma'$ and since $(\sim_{serv}, \approx_{serv})$ is a dependency cluster for $serv$, also $\sigma_p \approx_{serv} \sigma'_p$ holds. Since $\mathbb{F}$, we know that $\sigma_p = anon(\sigma, \mathbb{F}, \sigma_p)$ and $\sigma'_p = anon(\sigma', \mathbb{F}, \sigma'_p)$ and therefore with Theorem 2 condition 9 we know $\sigma_p \approx_g \sigma'_p$.

Finally we show equivalence of the communicated traces: Assume $t_1 \leq t$ and $t'_1 \leq t'$ such that $t_1 \rhd \mathbb{I} \sim_g t'_1 \rhd \mathbb{I}$. Due to Theorem 2 condition 7, it also holds that $t_1 \rhd \mathbb{I} \sim_{serv} t'_1 \rhd \mathbb{I}$ and since $(\sim_{serv}, \approx_{serv})$ is dependency cluster for $serv$, there exists t_2, t'_2 such that $t_1 \frown t_2 \leq t$ and $t'_1 \frown t'_2 \leq t'$ and $t_1 \frown t_2 \sim_g t'_1 \frown t'_2$. Since $\mathcal{C}$ is a callable set for $serv$, we know for all m, m' in $t_1 \frown t_2$ and $t'_1 \frown t'_2$ respectively $m, m' \in \mathcal{C}$ and by Theorem 2 condition 8 we know $m \sim_{serv} m' \implies m \sim_g m'$ and therefore $t_1 \frown t_2 \sim_g t'_1 \frown t'_2$.

So combined, $(\sim_g, \approx_g)$ is dependency cluster for $serv$.

Theorem 3 ((Theorem 5 in [8])). *Let serv be a service that is non-interferent w.r.t.* $(\sim, \approx)$ *and* $\sim_w$ *a weakening of* $\sim$*. Then serv is non-interferent w.r.t.* $(\sim_w, \approx)$.

Proof (for Theorem 3). Follows directly from Definition 2. We strengthen the left hand side of the inner implication and weaken the right hand side.

3 Proof of Concept

In this section, we describe as a proof of concept the verification of security of a simple component-based web shop w.r.t. two attackers. The work flow is based on the results presented in [7] and [8].

We first introduce the web shop system, its interfaces with the environment, and the entities which are meant to interact with the system via the interfaces. After this, we describe the input information the entities may gain knowledge about and specify the outputs the entities have direct access to. This specification can be seen to be the result of a threat analysis and provides us with an attacker-motivated specification.

In Subsection 3.1 we describe the components the web shop consists of. We extract *dependency clusters* for each service provided by the system's components using our extension of the JOANA tool. We formalize additional dependency clusters manually where required and prove their validity using KeY. We combine the extracted and manually created dependency clusters to component-global information flow specifications and show that each service is non-interferent w.r.t. the global specification. Finally, we show that the attacker-motivated specifications are a weakening of the component-global non-interference specifications.

The full implementation of the web shop system, all specifications and proofs as well as the versions of the tools used to perform analysis can be found online at `https://formal.iti.kit.edu/greiner/sefm2017/`.

3.1 System Description

The system we analyze implements a simple web shop providing interfaces for selecting products, ordering of selected products, setting and reading customer information, e.g. name, address, and credit card information. Also, information can be retrieved from the system as necessary for a billing process and logistics of product delivery. The system requires a service from a bank in order to perform payment and access to a product database which stores prices of products. A full list of services provided and required by the web shop can be found in Figure 1.

Three roles of users are meant to interact with the system. The *customer* has access to the interfaces `AccountIF` and `CartIF` providing the services necessary for shopping and ordering. The *delivery department* can request information necessary for shipping products on interface `DeliverIF` and the *billing department* can request information necessary for creating bills after ordering. The *bank*, can access information via the required `BankIF` interface. A list of who can directly access which interfaces is shown in Figure 2. We assume that the described entities are the only groups interacting with the system. Further, we do not care about access control, since it is out of scope of this work. We assume that enforcement of access control is correct.

3.2 Confidentiality Specification

Each entity interacting with the system only needs a part of all input information in order to perform its duties. We limit our analysis to the roles *Customer*,

CartIF		
Service	Parameter	Return Value
`getCartContent`		`OrderElement[]`
`addToCart`	`int prodId, int amount`	`boolean`

AccountIF		
`orderElementsInCart`		`boolean`
`setName`	`char[] name`	
`setAdress`	`char[] adr`	
`setCCNr`	`int ccnr`	
`cvc`	`int cvc`	

BillingIF		
`getBillsToSend`	-	`Bill[]`

DeliveryIF		
`getDeliverySheets`	-	`DeliverySheet[]`

BankIF		
`makePayment`	`char[] name, int ccnr, int cvc, int amount`	`boolean`

ProductDBIF		
`getProductPrice`	`int prodId`	`int`

Fig. 1. Interfaces provided and required by the Webshop system

DeliveryDebt, and *BillingDebt*. We specify for each input which entity may gain knowledge about it. The decision is made on domain-level, meaning we decide who may know what based on whether we think from a domain point of view if the respective information is required by the entity. For example, the delivery department needs to know the name and the address of the customer as well as the products he ordered. However, it is irrelevant how much the customer paid for these products or what the number of the credit card used for payment is. On a more specific level, it is also irrelevant, whether the customer changed his credit card and how often he looked at the products already in the cart. The full list is shown in Figure 3.

Note especially that we use declassification for the credit card information. While the delivery department must not gain any knowledge about it, the billing department is meant to be able to print the last four digits of the credit card on the bill.

The security specification for each role considered as a potential attacker results from the combination of the confidentiality specification for the inputs and the accessibility specification for the outputs provided by the interfaces. We formalized the attacker specific security specification for the delivery department in the online available file `dc_Attacker_Delivery.key` in JavaDL.

Role	*Accessable IF*	*Description*
Customer	`AccountIF`, `CartIF`	Browses Products and orders
DeliveryDept	`DeliveryIF`	Department responsible for making packages and sending ordered products
BillingDebt	`BillingIF`	Department responsible for printing bills, sending them to the customers
Bank	`BankIF`	Financial Institute responsible for performing payments

Fig. 2. Roles interacting with the system

3.3 Components of the System

The web shop system consists of five components interacting with each other. Each component is responsible for a part of the functionality and together, they provide the functionality to the environment, i.e. the entities described above.

The system consists of the components `Cart`, `Account`, `OrderDB`, as well as `Billing`, and `Delivery`, and the components are connected via the interfaces `CartToAccountIF`, `OrderToAccountIF`, `BillingToAccountIF`, and `DeliveryToAccountIF`. The services specified with these interfaces are shown in Figure 4. The `Cart` component is responsible for managing the shopping process before the customer actually orders the products. The `Account` component serves as the central business logic for all processes happening after the customer decided to place an order. `OrderDB` keeps track of the performed orders, billed orders and delivered orders. `Billing` and `Delivery` collect information from the account, as necessary for the respective departments for performing their tasks. The components re-organize collected data such that the delivery and billing departments get the relevant information according to the business process.

3.4 Mapping from formal model to JavaEE

Our formal framework for component-based systems as presented in [7,8] can be instantiated for a large subset of components implemented in the Java Enterprise Edition (JavaEE) [3]. The only exception are so-called message-driven beans and user-defined synchronization for so-called singleton beans, a special synchronization model which should only be used in an implementation very cautiously. Whether these exceptions occur in a program can be checked syntactically.

JavaEE is a framework for implementing component-based systems in Java. Components are implemented as annotated Java objects, called *Enterprise Java Beans* (EJB), providing annotated methods, i.e., services. EJBs are executed by an application container, a middleware that allows easy scalability of JavaEE applications. EJBs come in four different shapes. Stateless beans are freshly created each time one of their services is called, stateful session beans manage a state over the lifetime of a user session. Stateless and stateful session beans are

	Customer	DeliveryDept	BillingDebt
`CartIF.getCartContent`			
`call`	✓	X	X
`CartIF.addToCart`			
`call`	✓	✓	✓
`prodId`	✓	✓	✓
`amount`	✓	✓	✓
`AccountIF.orderElementsInCart`			
`call`	✓	✓	✓
`AccountIF.setName`			
`call`	✓	✓	✓
`name`	✓	✓	✓
`AccountIF.setAdress`			
`call`	✓	✓	✓
`adr`	✓	✓	✓
`AccountIF.setCCNr`			
`call`	✓	X	✓
`ccnr`	✓	X	%10000
`AccountIF.setCVC`			
`call`	✓	X	X
`cvc`	✓	X	X
`BillingIF.getBillsToSend`			
`call`	✓	✓	✓
`DeliveryIF.getdeliverySheets`			
`call`	✓	✓	✓
`BankIF.makePayment`			
`result`	✓	✓	✓
`ProductDBIF.getProductPrice`			
`result`	✓	X	✓

Fig. 3. Specification of the inputs the roles may gain knowledge about (Represents the *low* inputs per role)

Service	Parameter	Returnvalue
CartToAccountIF		
getCartElementsForOrder		OrderElement[]
OrderToAccountIF		
makeNewOrder	OrderElement[] oe	boolean
getOrdersToBill		Order[]
getUnshippedOrders		Order[]
getAllOrders		OrderHistory
BillingToAccountIF		
getBillingAdress		char[]
getBillingCreditCard		int
getBillingName		char[]
getOrdersToBill		Order[]
DeliveryToAccountIF		
getOrdesForDelivery		Order[]
getName		char[]
getAdress		char[]

Fig. 4. Internal interfaces of the webshop system

the main components implementing the business logic in a JavaEE application. Singleton beans are instantiated exactly once per application, and message-driven beans can be called asynchronously and do not provide return values. We do not consider message driven beans.

The application container takes serialized requests for remote service calls, translates the message into a method call to the respective bean and returns the return value as a serialized object, similar to messages in our formal model. Since writing static fields by EJBs is disallowed, each bean's state is independent from the state of other beans. The application container also is responsible that services of one bean are not executed concurrently, and JavaEE prohibits thread creation and management by beans. Only singleton beans can be specified to be re-entrant using user-provided annotations. We do not consider singleton beans with user-defined synchronization. Finally, the application container is responsible for access control such that bean implementations can assume that services are only called by authorized users.

We consider the restrictions we make not to be fundamental, because singleton beans with user-defined synchronization should be rarely used in practice. Message-driven beans may be more common in practice, however we assume our theoretical framework to actually also hold for them, although, we do not provide a formal proof.

In contrast to our formal model, Java is an object oriented programing language, where the state contains a heap that maps objects and fields to values. The KeY tool [1] formalizes the state in a Java program such that it contains one variable of type heap and other variables for local variables, parameters, and the return value of methods. Since our specification allows to specify equivalence

relations using expressions, we can express declassification and navigation over field access as side-effect-free expressions in the underlying logic. In [11], a similar approach was used for specification of non-interference in Java batch programs without message passing.

For simplicity, we assume each public method of the class under analysis to be a service provided to the environment. While this assumption does not limit the applicability of our approach, it frees us from implementing support for JavaEE specific annotations in our analysis tools.

We apply our method using two different toolings to verify dependency clusters for services according to the first step of our method. The first tool is an extension of the theorem prover KeY which uses symbolic execution to verify properties of Java programs. KeY allows very precise verification of properties, especially dependency clusters with declassification. KeY supports a major subset of Java 6, but not JavaEE annotations. The second approach is an extension of the JOANA tool, a tool analyzing information flow in Java programs using program dependency graphs (PDG). While this static analysis cannot make use of semantic information for parameters and states, JOANA is fully automatic. JOANA supports full Java byte code, except reflection. Thus, we gain language support for a major subset of Java for free in the context of this work. Finally, we use the Java-independent part of the KeY tool to prove the first-order formulas that are constructed in the second and third step of our method.

As a case study, we implemented the simple web shop system explained above as a JavaEE program. In total, the components implement 21 services with about 130 lines of code combined (not counting specifications, interface declarations, and comments). In the following, we describe how we instantiated our method to verify security for the web shop program.

3.5 Automatically derived Dependency Clusters in Components

To automatically derive dependency clusters, we use *program dependency graphs* (PDGs), a language-independent representation of the dependencies between the statements and expressions of a program. To build and use PDGs for our purposes, we use JOANA [6,9], a state-of-the-art information flow analysis tool based on PDGs. JOANA is most suited for batch programs with a single entry point. Recent work on JOANA includes component-based architectures, like Android [10] and modularization of program dependency graphs (PDG) [5].

The nodes of a PDG represent statements and expressions, while edges model the dependencies between them. The most important kinds of dependencies are *data dependencies*, (a statement using a value produced by another statement) and *control dependencies* (a statement or expression controlling whether another statement is executed or not). Figure 5 shows a small code snippet and the corresponding part of its PDG. For inter-procedural and object-oriented languages such as Java, PDGs model – in addition to statements and expressions – also parameter passing and dependencies through the heap by incorporating appropriate nodes and dependencies.

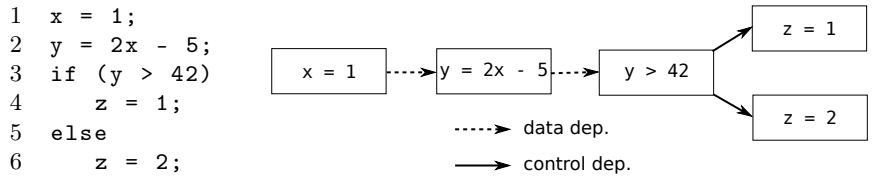

Fig. 5. A code snippet and its PDG

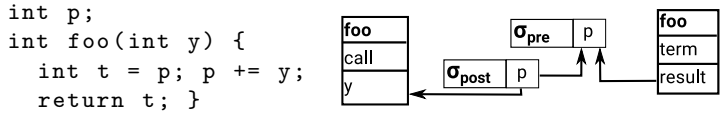

Fig. 6. A small example service and the graph produced by slicing its PDG

Given a node n of the PDG, the *backwards slice* of n contains all nodes from which n is reachable in the PDG. For sequential programs, it has been shown [13] that a node not contained in the backwards slice of n cannot influence n, hence PDG-based slicing on sequential programs guarantees *non-interference* [4]. This makes PDGs suitable for our purposes: By computing slices, we obtain a graph like the one shown in Figure 6. To extract dependency clusters from this graph, we in principle perform a reachability analysis, but additionally adhere to the fact that for precision reasons, JOANA distinguishes between the pre-state and the post-state. Hence, if during the reachability analysis we arrive at a node belonging to the pre-state, we also have to consider the dependencies of the corresponding node in the post-state. This ensures that also those information flows are covered which occur only if a service is executed multiple times. In our example, `p` in the pre-state depends on `p` in the post-state, so once we consider the dependencies of `p` in the pre-state, we also have to consider what `p` in the post-state depends on.

We used JOANA to extract dependency clusters from the implementation of each component. In total, we gain this way 245 dependency clusters for `Account`, 27 for `Billing`, 36 for `Cart`, 21 for `Delivery`, and 151 for `OrderDB`. Each of these dependency clusters, we (automatically) formalize in JavaDL as equivalence relations.

Note that the dependency clusters in the context of this work are a mere technicality, which we require to show security of our system against an attacker. We therefore do not discuss any semantic meaning for the different clusters.

3.6 Fine-Grained Dependency Clusters

The dependency clusters extracted automatically from the implementation are not sufficient to show security against an attacker. We require additional dependency

clusters with higher precision, which we formalize manually. We use the program analysis capabilities of the theorem prover KeY to verify manually identified, precise dependency cluster. KeY is a theorem prover designed for the verification of properties in Java programs against specifications formalized in the Java Modeling Language (JML) or Java Dynamic Logic (JavaDL). The KeY system was previously used for verification of non-interference properties in Java batch programs without events [11] and implements support for better scalability of proofs and object-orientation [12,2]. Our implementation is influenced by these previous results. For a full account of KeY and JavaDL, we refer to [1,14].

JavaDL contains function symbols for local program variables, instance and static fields, method parameters, and operations of primitive Java data types. In addition to general functions and predicates known from first-order logic, JavaDL also contains constructs for programs which allow symbolic execution.

We extend JavaDL, and therefore KeY, by elements representing events. We use a constructor function with parameters indicating whether the event is an initial or terminating event, whether it is an input or output, the component providing the called service, a function symbol identifying the service, a sequence of the communicated parameters or return values, and the heap required for evaluating expressions over the parameters. The method, the calling object, the direction, and the object type in combination represent the channel according to our formal model. The formal definition can be found in Figure 7.

```
event(Calltype ct, CallDirection cd, Object o, Method m, Seq s, Heap h)
```

Fig. 7. Constructor Function for Events

We use a static ghost variable, i.e., a specification-only variable, to record the history of events passed during execution of a service. When during symbolic execution the service is called, the method contract is applied instead of actually symbolically executing the service. We formalize the general assumptions ensured by the application container according to JavaEE as method contracts. This includes that service calls do not change the local state of the calling component except for object creation during deserialization of return values, and that the history is extended by one call event and one termination event. Further, we do not require the calling component to ensure the object invariant of the called component to hold. A method contract for service `Bank.trans()` is shown in Figure 8.

We verified all dependency clusters necessary for verification of security of the program in the proof of concept, which we could not automatically derive using JOANA, as described in the previous section.

Some of the manually created dependency clusters are necessary due to imprecision of JOANA. We assume some of these imprecisions can be fixed in a more mature, non-prototype version. Other manually specified dependency clusters, especially those handling the declassification of the credit card information, can not

```
/*@ public normal_behaviour
  @ requires true;
  @ ensures
  @  Main.hist == \old(Main.hist) +
  @       event(servcall, out, this, Bank_trans,
  @                                   (s,c,p), heap)+
  @       event(servterm, in, this, Bank_trans,
  @                                  (\result), heap);
  @ modifies Main.hist; */
public /*@ helper */ int trans(int a);
```

Fig. 8. Method contract of service `Bank.trans()` (Simplified notion)

be found by JOANA. We require semantic knowledge in order to show correctness of these dependency clusters, which JOANA can not deal with by construction.

We therefore *guess* additional 3 dependency clusters for services provided by `OrderDB`, 3 for a service provided by `Cart`, 1 for the service provided by `Delivery`, and 15 for services provided by `Account`. Each of these clusters is formalized in JavaDL. An example of the formalization of a dependency cluster in JavaDL is shown in Figure 9. Formalization of predicates representing equivalence relations is uniform and can be generated automatically, given a specification for the dependency cluster. The full definition of the dependency clusters and their formalization can be found in the online available material.

Since these dependency clusters are manually created and it is unreasonable to assume that we did not make mistakes when identifying them, we have to formally show that they are indeed dependency clusters. We therefore have to show the proof obligation which we gain from Definitions 2 and 3 in [8]. We apply the KeY tool to verify the resulting JavaDL formula for each manually created dependency cluster.

We formalize the proof obligations from the first step of our method directly as a JavaDL formula. A simplified formalization for a service `foo` can be found in Figure 10. Lines 1 to 7 formalize the first execution of the service, storing the post state in variable `heapAtPost_A`. Line 12 indicates the repetition of a similar formula formalizing the second execution.

The variable `heapAtPre_A` represents the pre-state of one execution of the service. In Line 1, the pre-state of the execution is stored to the program variable `heap`, which is used as the program heap during symbolic execution. Line 3 states some technical welldefinedness properties of the heap, which are out of context for the presentation here. In Line 5 we ensure that the history of the program execution starts with the event calling the service under analysis. The actual symbolic execution is performed in Line 7 which is followed by adding the termination event to the history and storing the heap after execution to a logic variable `heapAtPost_A`. The dots thereafter indicate a copy of the formula discussed so far and replacing `A` by `B`, representing the second execution of the service.

```
equivEvent_Account_getAdress_dc_1(
        event(t1, dir1, callee1, m1, p1, h1),
        event(t2, dir2, callee2, m2, p2, h2)) :=
 (invEvent_Account_getAdress_dc_1(
                event(t1, dir1, callee1, m1, p1, h1)) &
 invEvent_Account_getAdress_dc_1(
                 event(t2, dir2, callee2, m2, p2, h2)) )
 |
 ((t1 = t2 & m1 = m2 &
   !invEvent_Account_getAdress_dc_1(
                  event(t1, dir1, callee1, m1, p1, h1)) &
   !invEvent_Account_getAdress_dc_1(
                  event(t2, dir2, callee2, m2, p2, h2)))
   &((m1 = DeliveryToAccount_getAdress
                              & t1 = servcall & (true)) |
     (m1 = DeliveryToAccount_getAdress
                              & t1 = servterm & (true) &
       (\forall i0; (
          ((0<=i0 & i0<length(char[]::seqGet(p1, 0))))
          -> (seqGet(p1,0)[i0])@h1 =
             (seqGet(p2,0)[i0])@h2)))
    ))
```

Fig. 9. Formalization of a dependency cluster in JavaDL (call is visible and all elements of return array are low)

The actual proof obligation starts in Line 14. `wellformedListCoop` is a predicate ensuring that all service terminations represented in the history have the same visibility as the previous service call. The predicate `coopListEquiv` formalizes that for both executions of the service all visible events representing a service termination are equivalent, if the respective service calls are equivalent. The two predicates in combination formalize cooperative environments. We ensure that the service under analysis is called with two equivalent initial events (Line 18) and in equivalent pre-states (Line 21).

If these assumptions hold, it has to be shown (Line 23) that the two executions produce equivalent traces, and the service terminates in equivalent post-states (Line 25). The predicate `equivHistory` and state equivalence depend on the specification of dependency clusters. Their full definitions can be found in the online available material. We can verify formulas as in Figure 10 using KeY.

3.7 Component Specifications

To show secure information flow for an entire component, we have to find a component-global non-interference specification according to Theorem 2. However, we additionally want to show security for the entire system, which is composed

```
    {heap:=heapAtPre_A}
      < ... >
3     & self_A.<inv>@heap
      & Main.hist@heap = <(event(servcall, in, self_A,
                              Shop_buy, <a_A, b_A, c_A), heap)>
      & \[{ result_A =
            self_A.buy(prodId_A, price_A)@Shop; }\]
8       {Main.hist := seqConcat(Main.hist,
              <event(servterm, out, self_A, Shop_buy,
                    seqSingleton(result_A), heapAtPost_A)>)}
        (selfAtPost_A = self_A & heapAtPost_A = heap))
  & < ... second execution ... >
13    ->
      (wellformedListCoop(Main.hist@heapAtPost_A)
       & wellformedListCoop(Main.hist@heapAtPost_B)
       & coopListEquiv(filterVisible(Main.hist@heapAtPost_A),
                       filterVisible(Main.hist@heapAtPost_B) )
18     & equivEvent(Main.hist@heapAtPost_A[0],
                    Main.hist@heapAtPost_B[0])
       & self_A.dc1_buy@heapAtPre_A =
         self_B.dc1_buy@heapAtPre_B
     ->  (equivHistory(Main.hist@heapAtPost_A,
23                     Main.hist@heapAtPost_B)
          & self_A.dc1_buy@heapAtPost_A =
            self_B.dc1_buy@heapAtPost_B
          )
       )
```

Fig. 10. JavaDL version of the proof obligation for a dependency cluster for service `buy` (simplified)

of several components. From Theorem 2 in [7], we know that non-interference is compositional for components with synchronized communication.

We therefore aim for a system-wide non-interference specification for messages and a component-wide non-interference specification for states. We construct our specification the following way. Given components $c1, ..., cn$ with services $s1ci, ..., smci$ for the component ci. For each service sk of component ci, we select a set of dependency clusters $(\sim^1_{skci}, \approx^1_{skci}), ..., (\sim^p_{skci}, \approx^p_{skci})$. We construct a dependency cluster for each service by intersecting the selected dependency clusters $(\sim^{loc}_{skci}, \approx^{loc}_{skci}) := (\sim^1_{skci}, \approx^1_{skci}) + ... + (\sim^p_{skci}, \approx^p_{skci})$. For each service the JavaDL formalization of the service's composed dependency cluster can be found online in the `dc_Classname_methodname_aggregate1.key` files for each method. Since each element of this composition is a dependency cluster of the respective service, we know according to Theorem 1 that the composition is a dependency cluster for the service as well. No additional property has to be shown.

```
equivEvent_Account_getAdress_aggregate(
              event(t1, dir1, callee1, m1, p1, h1),
              event(t2, dir2, callee2, m2, p2, h2)) :=
(invEvent_Account_getAdress_dc_aggregate1(
               event(t1, dir1, callee1, m1, p1, h1)) &
 invEvent_Account_getAdress_dc_aggregate1(
               event(t2, dir2, callee2, m2, p2, h2)) )
| ((t1 = t2 & m1 = m2 &
   !invEvent_Account_getAdress_dc_aggregate1(
                event(t1, dir1, callee1, m1, p1, h1)) &
   !invEvent_Account_getAdress_dc_aggregate1(
                event(t2, dir2, callee2, m2, p2, h2))) &
  equivEvent_Account_getAdress_dc_1(
                  event(t1, dir1, callee1, m1, p1, h1),
                  event(t2, dir2, callee2, m2, p2, h2))
 & equivEvent_Account_getAdress_dc_2(
                  event(t1, dir1, callee1, m1, p1, h1),
                  event(t2, dir2, callee2, m2, p2, h2))
 & equivEvent_Account_getAdress_dc_3(
                  event(t1, dir1, callee1, m1, p1, h1),
                  event(t2, dir2, callee2, m2, p2, h2))))
```

Fig. 11. Aggregation of three dependency clusters for a service in JavaDL

Then, we construct for each component a component-global equivalence relation for states by intersecting the equivalence relation of the composed dependency clusters: $\approx^{g}_{ci} := \approx^{loc}_{s1ci} \cap ... \cap \approx^{loc}_{smci}$. Similarly, we retrieve a system-global equivalence relation for messages by intersecting all equivalence relations for messages of the services: $\sim^{sys} := \sim^{loc}_{s1c1} \cap ... \cap \sim^{loc}_{smc1} \cap ... \cap \sim^{loc}_{s1cn} \cap ... \cap \sim^{loc}_{spcn}$.

We have to show for each service that it is non-interferent w.r.t. this global non-interference specification according to Theorem 2. The service local dependency cluster, called $(\sim_{serv}, \approx_{serv})$ in the theorem is for each service the composed cluster $(\sim^{loc}_{skci}, \approx^{loc}_{skci})$. For the component-global tuple, called $(\sim_g, \approx_g)$, we use $(\sim^{sys}, \approx^{g}_{ci})$ for all services implemented by component ci. An example for the JavaDL formalization of an aggregated dependency cluster is shown in Figure 11. We formalize again the proof obligation implied by Theorem 2 as JavaDL formula.

We would like to consider here two specific predicates and two fields more closely. The predicate `equiv_g(Event, Event)` encodes the component-global equivalence for messages. It is provided by the construction explained above. The predicate `equiv_l(Event, Event)` represents the service-local equivalence specification gained by intersecting dependency clusters for a service. We define it directly as the intersection:

```
equiv_l(e1, e2) := equiv_foo_dc1(e1, e2) & equiv_foo_dc2(e1, e2)
```

We use ghost fields like `dc1_foo` directly declared as JML specifications to formalize equivalence relations for states. Equivalence relation for the state

```
    (equiv_g(event(ct1, cd1, self_A, met1, param1, heap1),
             event(ct2, cd2, self_A, met2, param2, heap2))
     -> equiv_l(event(ct1, cd1, self_A, met1, param1, heap1),
                event(ct2, cd2, self_A, met2, param2, heap2))
 5  )
   & (self_A.dc_global@heap1 = self_A.dc_global@heap2
     ->
     self_A.dc_local@heap1 = self_A.dc_local@heap2)
   & ((equiv_l(event(ct1, cd1, self_A, met1, param1, heap1),
10           event(ct2, cd2, self_A, met2, param2, heap2)) &
       \elementOf(met1, self_A.callable_foo[j]@heap1) &
       \elementOf(met2, self_A.callable_foo[j]@heap2))
     -> equiv_g(event(ct1, cd1, self_A, met1, param1, heap1),
                event(ct2, cd2, self_A, met2, param2, heap2)))
15 & ((self_A.dc_global@heap1 = self_A.dc_global@heap2 &
       self_A.dc_local@heap1p = self_A.dc_local@heap2p)
       ->
       self_A.dc_global@anon(heap1,
                             self_A.assignable_foo@heap1, heap1p)
20     = self_A.dc_global@anon(heap2,
                             self_A.assignable_foo@heap2, heap2p))
```

Fig. 12. Proof obligation for Theorem 2

according to the global specification is directly encoded as the intersection of the local equivalence relations for the state. In a similar way, the equivalence relation for the state according to the global specification is directly encoded as a list of expressions in the field `dc_global`. The low-part of the state according to the local specification is given as the concatenation of the state equivalence relation of dependency clusters, for example as `dc_local := dc1_foo . dc2_foo`, where . is the concatenation operation on sequences. The callable set of the service `foo` is encoded as a list of function symbols in field `callable_foo`, and the assignable set is a given as a set of locations (the Java equivalent of variables) encoded as a field `assignable_foo`. We used JOANA to identify both of these sets for each service.

The formula in Figure 12 is a straight forward encoding of Theorem 2 in JavaDL. In Line 1 the proof obligation requires the message equivalence of the local specification to be implied by the global one. Similar, in Line 6 the same implication is required for the low-part of the state. This is followed by the reverse direction, i.e. global equivalence of messages and states are implied by the local specification under consideration of the assignable and callable sets. Line 9 requires that a message either cannot be communicated by the service, i.e. the channel is not an element of the encoding of the callable set `self_A.callable_foo[j]@heap1`, or the message is equivalent according to the global specification. Similarly, Line 15 formalizes that if the service started in two globally equivalent states, results after execution in states `heap1p` and

`heap2p`. If the two post-states are equivalent according to local specification, then the resulting state, expressed by anonymization, is also locally equivalent.

We have to do these proves twice for each service, since we show security against two attackers with different security specifications. When combining local dependency clusters to global ones, we heavily reuse dependency clusters for both system-wide specifications.

Since we achieved to prove the formula for all services provided by the components, we know that all components are non-interferent w.r.t. $\sim_g$, as well as, according to Theorem 2 in [7], the web shop system.

3.8 Attacker Weakening

It is left to show that the system is secure against the attacker-motivated specifications we identified in the beginning. We use a formalization of Theorem 3 to show that the two attacker specifications are a weakening of our two system-wide equivalence relations over messages $\sim_g$ again using KeY.

In combination, we have shown that our web shop system is secure against two attackers from the delivery department and the billing department.

4 Conclusion

We have shown security of a web shop system consisting of five components to be secure against two attackers who have the abilities of somebody working in the delivery department and the billing department respectively. They may know the information supposed to be known, since it is required for the business processes in the departments. We made heavy use of the results of the framework as presented in the main paper. Especially reusing dependency clusters for both attackers was useful, as was the modularization of the proof obligations into small, service-local problems.

We identified 480 dependency clusters in the components of the web shop program with JOANA automatically. In addition, we manually specified 21 dependency clusters, for which JOANA was not sufficiently precise and verified them with KeY. While the proof process was very fast and mostly automatic, we had to provide support specifications, like loop invariants, a time-consuming task in any theorem-prover based program verification task.

The manually verified dependency clusters are distributed over three different components, while for each service at most two dependency clusters had to be specified manually. This shows that in the case of evolution of a component, i.e., changing the implementation of a service, only few dependency clusters have to be re-verified and even less with manual interaction using KeY.

We generated the system-wide and component-wide specifications according to step two of our method by manually selecting the relevant dependency clusters and automatically generating the JavaDL proof obligation according to Theorem 2. And finally we automatically generated proof obligations according to Theorem 3, one for each of the two attacker types considered. We performed 42

proofs according to Theorem 2 and two proofs according to Theorem 3. During verification of the proof obligations with KeY in step two and three, manual interaction was necessary at many points, since the proof requires several case distinctions. KeY is, as a program verification tool, not optimized for these kinds of proof obligations and we assume a high degree of automation if better suited tools are used for this task, for example SMT solvers.

We first applied all steps of our method for one attacker and then for the other, which, in total, took about one week of work for verification. Program analysis with Joana basically does not take any time, and manually specifying the correct missing dependency clusters for the first attacker took about two days of the total effort, including repeated attempts to show step two and three of our method. We required about four days for the verification and specification of functional support specifications for the program and verification of the manually specified dependency clusters in the first step. However, we would like to stress that verifying the security of the second attacker only took about one day of the total effort, since we could re-use previously verified dependency clusters as well as already existing functional specifications.

We have shown with this proof of concept that it is possible to use our framework to verify security of component-based systems w.r.t. to an attacker. Our tool chain is based on the theorem prover KeY and an extension of Joana, using program dependency graphs for program analysis. In the combination we have also shown that dependency clusters are an interesting way of loosely coupling different tools with different weaknesses and strengths to co-operatively show information flow properties in component-based systems.

As a result, we find that the more automation for identifying dependency clusters can be achieved, the more our method scales. Sometimes, it might be reasonable to omit program verification for some services or components and instead use testing approaches or code inspection for validation of some dependency clusters. Also, it would be interesting, if we could automatize the selection of dependency clusters, identified in step 1, which are relevant for verification in step 2 and 3, e.g., using SMT solvers.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice. Springer (2016)
2. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: LOPSTR (2013)
3. EJB 3.1 Expert Group: JSR 318: Enterprise JavaBeans, Version 3.1. Sun Microsystems (2009), `https://jcp.org/en/jsr/detail?id=366`, accessed 31/08/2016
4. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Security and Privacy (1982)
5. Graf, J.: Information Flow Control with System Dependence Graphs – Improving Modularity, Scalability and Precision for Object Oriented Languages. Ph.D. thesis, Karlsruher Institut für Technologie (2016)

6. Graf, J., Hecker, M., Mohr, M.: Using joana for information flow control in java programs - a practical guide. In: ATPS (Feb 2013)
7. Greiner, S., Grahl, D.: Non-interference with what-declassification in component-based systems. In: CSF (2016)
8. Greiner, S., Mohr, M., Beckert, B.: Modular verification of information flow security in component-based systems. In: SEFM (2017), accepted
9. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security (Dec 2009)
10. Mohr, M., Graf, J., Hecker, M.: Jodroid: Adding android support to a static information flow control tool. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering, Dresden, 2015. CEUR Workshop Proceedings (2015)
11. Scheben, C., Schmitt, P.H.: Verification of information flow properties of java programs without approximations. In: FoVeOOS (2011)
12. Scheben, C., Schmitt, P.H.: Efficient self-composition for weakest precondition calculi. In: Formal Methods (2014)
13. Wasserrab, D., Lohner, D.: Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In: VERIFY (2010)
14. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)