Modular Verification of Information Flow Security in Component-Based Systems

Simon Greiner, Martin Mohr, and Bernhard Beckert

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

Abstract. We propose a novel method for the verification of information flow security in component-based systems. The method is (a) modular w.r.t. services and components, i.e., overall security is proved to follow from the security of the individual services provided by the components, and (b) modular w.r.t. attackers, i.e., verified security properties can be re-used to demonstrate security w.r.t. different kinds of attacks.
In a first step, user-provided security specifications for individual services are verified using program analysis techniques. In a second step, first-order formulas are generated expressing that component non-interference follows from service-level properties and in a third step that global system security follows from component non-interference. These first-order proof obligations are discharged with a first-order theorem prover. The overall approach is independent of the programming language used to implement the components. We provide a soundness proof for our method and highlight its advantages, especially in the context of evolving systems.
As a proof of concept and to demonstrate the usability of our method, we present a case study, where we verify the security of a system implemented in Java against two types of attackers. We apply the program verification system KeY and the program analysis tool Joana for analyzing individual services; modularity of our approach allows us to use them in parallel.

1 Introduction

Information flow (IF) security is a program property ensuring that certain information given as input to a system can only be observed by users of the system who are explicitly allowed to do so (and not by other users). Formal analysis of IF security requires specification of (a) which users shall be able to observe which information and (b) what outputs users can access and read. In practice, there is often more than one type of user – which we consider to be potential attackers – each requiring a specification and analysis. Component-based systems, in particular, are designed for re-use and, thus, are deployed in different environments where new user types have to be considered.

* This work was supported by the German Ministry for Education and Research within the framework of the project KASTEL_IoE in the Competence Center for Applied Security Technology (KASTEL) and by the german research foundation in the scope of the priority program "Reliably Secure Software Systems" (grants Sn11/12-1/2/3).

Overview of our method. We propose a novel method for the modular verification of IF security properties in component-based systems. Our method uses very expressive IF specifications, where information about arbitrary parts or combinations of input parameters can be declared to be secret as well as information about what service calls other users have initiated. The restricted programming paradigm used for component-based systems provides convenient compositionality properties, that we can use to verify non-interference properties in a modular way.

The first step of our proposed method is the specification (and verification) of service-local IF properties, which are independent of the user/attacker model. These service-local properties are modular and re-usable. We can apply tools with different precision and scalability properties for verification of each service-local specification, which allows us to improve scalability while maintaining precision of verification. Service-local specifications remain valid when other services are changed or new services are added. In the second step, we generate a system-wide specification from the service-local properties and verify that the service-local properties imply the system-wide IF properties. This is done by proving validity of a formula in first-order predicate logic, which is constructed from the specifications in a uniform way. In a third step, we show that the system-wide IF specification implies the required domain-motivated IF property w.r.t. particular user/attacker models.

Our method is tool-independent and allows the combination of different program analysis techniques – to be used in the first step. The second and the third step do not need program analysis but only a first-order theorem prover.

Proof of concept. As a proof of concept, we apply our method to component-based systems implemented in Java, using the program verification tool KeY and the program analysis tool JOANA for verification of service-local IF properties (first step). Further, we use the (Java-independent part of) KeY to prove validity of first-order formulas (second and third step). As an example, we specified and verified IF security of a web shop system consisting of several components w.r.t. two different types of attackers.

Related work. Non-Interference as a program property has its origins in the notion of *strong dependency* by Cohen [7] and the first definition of non-interference by Goguen and Meseguer [10]. The work in this paper is based on a line of work on non-interference for distributed interactive systems (which components can be considered to be) [25,6,26,12]. Work in [28] allows more expressive specifications than our framework, but does not provide compositionality results. Other recent work [2,21,22] discusses compositionality of concurrent threads with a shared state, which however is not present in components.

Approaches for analysis of event-based non-interference notions often use type systems and are limited to toy languages or abstract specification languages (e.g., [26,20,27]). Analysis for batch programs based on type systems (e.g., [3]) is typically limited to syntactical information, and therefore has limited precision. *Dependent types* [32,24] include semantic knowledge into the analysis and use SMT solvers as back ends. JiF [23,5] extends Java with security type systems,

which allows analysis of security properties in Java programs at compile time. We assume type-based analysis can be used as an additional technique for the first step of our method.

We use the JOANA tool, for program analysis in this work. Other slicing tools for Java include WALA [15], PIDGIN [16] and Indus [33]. Both WALA and PIDGIN employ program dependency graphs and especially PIDGIN could be a viable alternative in the context of this work, while Indus does not a provide pre-computed PDG, which we require here.

Work on tool combinations for IF analysis includes the RIFL language [9], a specification language for IF policies in programs. It is supported by several tools. However, in contrast to our work, RIFL does not provide a formal semantics for the specifications and does not support secrecy of messages. Küsters et al. propose the *hybrid approach* [18] for the verification of IF properties with declassification in batch programs. Their approach, however, relies on making provably ineffective changes to the program and is limited to the combination of two tools. SHRIFT [19] combines dynamic analysis on the operating system layer and static IF analysis to track information flows through multiple layers of abstraction efficiently and precisely. We limit the presentation here to static analysis tools, while in general, our method would allow dynamic analysis.

Paper outline. Next, we define the formal framework (Sect. 2), i.e., notions of service and component, and non-interference for components and services. Then, in Section 3, we introduce our concept of service-local specifications (first step). We show in Section 4 how they can be used to generate and verify system-wide properties (second step). And, in Section 5, we describe how domain-motivated IF specifications can be derived from system properties (third step). We discuss the proof of concept and the web shop example in Section 6. Finally, we conclude.

2 Formal Framework

We present the formalization of components, services, and composition of components, and we formally define non-interference in component-based systems. This formal framework, which we use as a basis for our method, is mainly taken from [12], where also proofs for the theorems can be found.

2.1 Components and Services

Components have a (private) state σ, which is a mapping from a set $\mathcal{V}$ of variables to a set $\mathbb{V}$ of values. A component's functionality is implemented by services, which are sequential, terminating, deterministic programs.

For each service *serv*, a dedicated initial channel $\mathit{Ini}(\mathit{serv})$ and a termination channel $\mathit{Fin}(\mathit{serv})$ is contained in the system's set $\mathbb{C}$ of channels. If the environment wants to call *serv*, it sends a message $m \in \mathbb{M}$, where $\mathbb{M} \subseteq \mathbb{C} \times \mathbb{V}^n$, on the initial channel $\mathit{Ini}(\mathit{serv}) \in \mathbb{C}$ and parameter values from $\mathbb{V}$ to the component. Then, the component executes the service starting in its current state and returns a message on $\mathit{Fin}(\mathit{serv})$ on completion. While a component executes a service,

```
Component Shop {
  int sum, check, payId;
  String prods;

  buy(prodId,price) {
    sum = sum+price;
    prods = prodId.prods;
    return prods; }
print(p) {
  if (check)
  return sum.prods.payId;
  else return 0; }
pay(ccnr,pin) {
  check = trans(sum,ccnr,pin);
  if (check) payId = ccnr%2;
  return 0; } }
```

Fig. 1. Shop component as running example (see Example 1)

all other service calls to that component are postponed, making the execution of a component a non-reentrant, sequential composition of service executions.

During execution, a service may call other services $serv'$ by sending a message on channel $Ini(serv')$. After the call, the service waits for the termination of $serv'$, making communication synchronous.

This computational model for component-based systems is rather restrictive. It is, nevertheless, consistent with practically used frameworks for implementation of component-based systems. We discuss, for example, how our model applies to the Java Enterprise Edition in Section 6. Also, assemblies as used in the .net framework have similar properties. And even relational databases can be considered components according to our definition.

Example 1. As a running example, we use the simple Shop component shown in Figure 1. The service `buy` receives a product id, adds it to the list `prods` of products in the cart, and increases the variable `sum` by the product's price. Service `pay` uses the service `trans` provided by the environment to perform payment with credit card number `ccnr`, the given pin and the sum of prices stored in the state. Service `print` prints the receipt with the paid sum, the products and last bit of the credit card used for paying, if payment was successful.

For messages on a channel α with parameter v, we write $\alpha?v$ (input message), $\alpha!v$ (output message), or $\alpha.v$ (direction irrelevant). The sets of all input messages and all output messages are denoted by $\mathbb{I}$ and $\mathbb{O}$, respectively. The empty trace of messages is denoted by $\langle\rangle$, and $\frown$ is the concatenation operation for traces.

Two components c and d are composed by synchronizing messages on services required by c and provided by d and vice versa. In the trace of the composition, messages resulting from communication between c and d can be observed by the environment as outputs.

2.2 Non-Interference

Intuitively, a component is non-interferent, i.e., it has no unwanted information flows, if an environment (or an attacker) observing the public (low) output of a component cannot distinguish between inputs which only differ on (high) secrets.

In the following, to distinguish between public (low) and secret (high) inputs and outputs, we use an equivalence relation $\sim \subseteq \mathbb{M} \times \mathbb{M}$. Messages which are equivalent w.r.t. $\sim$ must not be distinguishable by the environment, i.e., it is a secret which of two equivalent messages has been sent. This flexible formalization allows a very precise specification of what-declassification (see [29]).

Example 2. Continuing from Example 1, the relation $\sim$ may be defined for the Shop component by: $Ini(pay)?(ccnr, pin) \sim Ini(pay)?(ccnr', pin')$ iff $ccnr\%2 = ccnr'\%2$, stating that the last bit of the information stored in parameter *ccnr* is low, while parameter *pin* is considered to contain high information. The definition $Fin(pay)!r \sim Fin(pay)!r'$ iff $r = r'$ expresses the return value of *pay* to be low.

Apart from communicated values, the mere existence of communication can contain information. All results provided in the remainder also hold in the case when the specification of high message existence is possible.[1]

Equivalence of messages raises a natural notion of equivalence of message traces, for which we overload $\sim$: Two traces t and t' are equivalent if their projection on the equivalence classes of messages implied by $\sim$ are equal. Then, non-interference for components can be defined as follows:

Definition 1 (Component non-interference). *A component c is non-interferent w.r.t. an equivalent relation $\sim$ on messages if, for all message traces that can be produced by c in some environment, an equivalent trace is produced by c in any environment supplying equivalent inputs.*

In [12], environments are formalized as functions providing for each observation of a component's behavior some input for the component. We omit here this more formal consideration of non-interference and refer to the original work. We do need, however, the following compositionality result:

Theorem 1 (Non-interference compositionality). *If components c and d are non-interferent w.r.t. $\sim$, then the composition of c and d is non-interferent w.r.t. $\sim$.*

We want to verify non-interference of components in a modular way, i.e., by first analyzing individual services. Thus, we need to ensure that one service does not break the non-interference of another service. We must check that no service returns a high value stored in the state by another service. For that purpose, we use an equivalence relation $\approx \subseteq \mathbb{S} \times \mathbb{S}$ over states to define the low part of a state: Two states are equivalent w.r.t. $\approx$ iff they only differ on the secret (high) part of the state. Now, we can define non-interference for services as follows:

Definition 2 (Service non-interference). *A service serv is non-interferent with respect to $\sim$ and $\approx$ iff, for all pre-states σ_1, σ_2, all post-states σ_1', σ_2', and all traces t_1, t_2 such that serv started in σ_i terminates in σ_i' by communicating trace t_i, the following holds:*

1. *If $\sigma_1 \approx \sigma_2$ and t_1 and t_2 have equivalent input messages, then $\sigma_1' \approx \sigma_2'$.*

[1] We omit discussion of high messages here, and refer to [13].

2. *If $\sigma_1 \approx \sigma_2$ and t'_1, t'_2 are prefixes of t_1, t_2 with equivalent inputs, then there exist longer prefixes $t'_1 \frown t''_1$ of t_1 and $t'_2 \frown t''_2$ of t_2 such that $t'_1 \frown t''_1 \sim t'_2 \frown t''_2$.*

Condition 1 ensures that the service, when started in equivalent pre-states and provided with equivalent inputs, terminates in equivalent post states, i.e., no high information is written to the low part of the state. Condition 2 ensures that all outputs created by the service are equivalent if the pre-states and all inputs previously provided to the service are equivalent, i.e., no high information is sent as output to the environment.

Non-interference for services as defined above is termination-insensitive. While generally this is a weak non-interference property, that is not relevant here, since we assume every service to terminate. Components, on the other hand, never terminate: After termination of a service, the component continues to offer all its services to the environment.

The following theorem states that non-interference for components can be verified by first proving non-interference for services.

Theorem 2 (Compositionality of Services). *A component c is non-interferent w.r.t. $\sim$ (Def. 1) if there exists an equivalence relation $\approx$ on states such that all services provided by c are non-interferent w.r.t. $\sim$ and $\approx$ (Def. 2).*

Note that Theorem 2 requires all services to be non-interferent w.r.t. the *same* relations $(\sim, \approx)$. Moreover, Theorem 1 requires components to be non-interferent w.r.t. the *same* relation $\sim$ to derive non-interference of their composition. In the following sections we describe a method for generating appropriate system-global relations from service-local relations.

3 Service-Local Non-Interference Specification

In the first step of our method, we specify and verify information-flow (resp. non-interference) at the level of individual services. We do this in a modular way such that verified properties of services imply properties of the overall system. Modularity is essential as it is very tedious to find and formalize system-wide IF properties for a realistic system and, moreover, properties change whenever a system is modified or deployed in a new context.

As a concept for modular service-local specification, we introduce *dependency clusters*. Whether a dependency cluster is valid, i.e. whether the specification it represents is satisfied by a service, only depends on the service's implementation and not on the environment or other services. Moreover, existence of some information flow does not depend on the system-wide attacker model (but only whether the flow is harmful). We will use dependency clusters as building blocks for system-wide specifications in the second step of our method.

Intuitively, a dependency cluster is a set of message parameters and state variables (resp. more complex expressions) whose values in the post-state of the service only depends on their values in the pre-state. Thus, if the cluster is used to specify which information is public (low), then the service is indeed non-interferent.

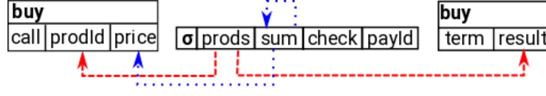

Fig. 2. Two dependency cluster (dashed and dotted arrows) of the service `buy`. The arrows illustrate dependencies between the state, parameters and the return value.

Example 3. In the Shop component shown in Figure 1, the return value of service *buy* depends (only) on the value of parameter *prodId* and on the pre-state value of *prods* (the dashed arrows in Figure 2 illustrate these dependencies). Thus the return value, *prodId*, and *prods* form a valid dependency cluster. A second cluster is formed by the parameter *price* and the state variable *sum* (dotted arrows).

As described in the previous section, we use equivalence relations on messages and states to formalize which information is considered public (low). Thus, more formally, a dependency cluster is a pair $(\sim, \approx)$; it is valid for some service if that service is non-interferent w.r.t. $(\sim, \approx)$:

Definition 3 (Dependency cluster). *A pair* $(\sim, \approx)$ *of equivalence relations is a* dependency cluster *for a service serv if serv is non-interferent w.r.t.* $(\sim, \approx)$.

For example, the universal relations, defined by $m_1 \sim m_2 \Leftrightarrow \mathit{true}$ and $\sigma_1 \approx \sigma_2 \Leftrightarrow \mathit{true}$, form a trivial dependency cluster for all services. This cluster defines all inputs, outputs and the entire post-state to contain high information (nothing is low). At the other extreme, the dependency cluster defined by $m_1 \sim m_2 \Leftrightarrow m_1 = m_2$ and $\sigma_1 \approx \sigma_2 \Leftrightarrow \sigma_1 = \sigma_2$ is also valid for all services. It declares all inputs, outputs and the entire state to only contain low information. In practice, of course, one needs to find clusters that are valid without being trivial.

Several dependency clusters for the same service *serv* are compositional in the sense that their intersection is again a dependency cluster *serv* (the formal proof, together with all proofs for this paper, can be found in [13]):

Theorem 3 (Compositionality of dependency clusters). *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.*

Intuitively, intersecting relations has the effect that equivalence classes become smaller and, thus, more information is considered low. Interestingly, according to Theorem 3, a composition of dependency clusters considers more outputs to be low, i.e., allows less flows than the individual clusters. At the same time, the composition is less restrictive than a mere conjunctive combination of the two individual clusters: Assume, for example, that one dependency cluster allows only flows from state variable a to itself (i.e. if $\sigma_{pre}(a) = \sigma'_{pre}(a)$ then $\sigma_{post}(a) = \sigma'_{post}(a)$), and the other allows only flows b to itself (i.e. if $\sigma_{pre}(b) = \sigma'_{pre}(b)$ then $\sigma_{post}(b) = \sigma'_{post}(b)$). Their intersection (i.e. if $\sigma_{pre}(a) = \sigma'_{pre}(a) \wedge \sigma_{pre}(b) = \sigma'_{pre}(b)$ then $\sigma_{post}(a) = \sigma'_{post}(a) \wedge \sigma_{post}(b) = \sigma'_{post}(b)$) additionally allows flows from a to b and vice versa, for example the program $a = a + b$.

As a formalism for defining dependency clusters and, thus, for specifying information flow properties, we introduce the following notation: Each dependency cluster is given as a pair $(\mathit{LowIO}, \mathit{LowState})$ of lists, specifying $\sim$ resp. $\approx$. The elements of LowIO are of the form $c.e$, where c is an initial or termination channel and e is an expressions over the parameters or the return values of the service. Two messages on channel c are equivalent iff, for all $c.e \in \mathit{LowIO}$, e evaluates to the same value for the two messages. Similarly, the elements of $\mathit{LowState}$ are expressions over the state variables. Two states are equivalent, if the expressions evaluate to the same values in both states. Intuitively, the two lists LowIO and $\mathit{LowState}$ describe what information is to be considered low. Thus, state variables, parameters, and channels not mentioned in the lists are secret (high).

The above notation can be used to define dependency clusters for services but also to specify global information-flow properties for components and systems.

Example 4. A component-global information-flow specification for the Shop example (Figure 1) may be given by:

$$\begin{aligned} \mathit{LowIO}_1 &= \langle \mathit{Ini}(\mathit{buy}).(\mathit{prodId}, \mathit{price}),\ \mathit{Fin}(\mathit{buy}).(r),\ \mathit{Fin}(\mathit{print}).(r), \\ &\qquad \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}\%2),\ \mathit{Fin}(\mathit{pay}).(r),\ \mathit{Fin}(\mathit{trans}).(r)\rangle \\ \mathit{LowState}_1 &= \langle \mathit{prods},\ \mathit{sum},\ \mathit{check},\ \mathit{payId}\rangle \end{aligned}$$

We use declassification for the credit card number expressing that (only) the last bit of the contained information is low. We can apply similar expressions for state variables.

The first dependency cluster in Figure 2 (dashed line) may be defined by $\mathit{LowIO}_2 = \langle \mathit{Ini}(\mathit{buy}).(\mathit{prodId}), \mathit{Fin}(\mathit{buy}).(r)\rangle$ and $\mathit{LowState}_2 = \langle \mathit{prods}\rangle$, and the second dependency cluster (dotted line) by $\mathit{LowIO}_3 = \langle \mathit{Ini}(\mathit{buy}).(\mathit{price})\rangle$ and $\mathit{LowState}_3 = \langle \mathit{sum}\rangle$.

The expressiveness of the list notion depends on the expressions allowed to occur in the lists. We do not define a particular language here but assume computability of the expressions. In practice, the concrete language will depend on the tools for verification of dependency clusters. Heavy-weight methods like theorem provers can deal with more expressive languages while light-weight tools like PDG- or type-based systems may support a limited subset.

Using the list notion, the composition of dependency clusters (see Theorem 3) can be constructed by concatenating the respective lists.

Example 5. The composition of the two dependency clusters from Example 4 can be written as $\mathit{LowIO}_4 = \langle \mathit{Ini}(\mathit{buy}).(\mathit{prodId}, \mathit{price}),\ \mathit{Fin}(\mathit{buy}).(r)\rangle$ and $\mathit{LowState}_4 = \langle \mathit{prods}, \mathit{sum}\rangle$.

According to Theorem 3, it is sufficient to show for two specifications independently that they are dependency clusters in order to gain a composed, potentially more complicated, specification. Ideally, one uses an analysis method to identify simple dependency clusters which describe information flows inherent to the implementation of a service. More complicated clusters, which are necessary to compare information flow to a security policy, can then be constructed by

composing these simple dependency clusters, which may be verified separately by different tools. This makes dependency clusters convenient building blocks for complex information flow specifications for services.

Since we allow declassification to be used in our specifications there are infinitely many potential dependency clusters for each service. The first step of the method proposed in this work is identifying useful dependency clusters for each service. In Section 6, we show two concrete approaches for identification. The first approach is to manually specify dependency clusters and verify them using a program verification tool. This is especially useful for declassification, when analysis with high precision is required for analysis. In the second approach, we use an automatic, less precise program analysis tool which directly creates a set of all dependency clusters it can find.

4 Dependency Clusters and Components

In the second step of our method, we compose dependency clusters of all services and thus gain component- and system-wide non-interference specifications. While dependency clusters for the same service are compositional, dependency clusters for different services are not, hence we have to show that composed dependency cluster of different services are consistent.

Since dependency clusters are service-local specifications, each dependency cluster will most likely mention at most the part of the state relevant for the service and the messages sent and received by the service. Consider, for example, the service *buy* in Figure 1: We have defined several dependency clusters for *buy*; but, none of these clusters mentions the variable *check*.

An approach in program analysis to deal with irrelevant parts of states is *framing* [17]. Framing uses an abstract description of an upper bound of relevant variables for a particular service and of the other services it requires. An *assignable set* describes the variables that a service may at most change. Indirectly, this specifies that the value (and security level) of all variables not in the set remains unchanged. A set $\mathbb{F} \subseteq \mathcal{V}$ is an *assignable set* for a service *serv* iff, for all executions of *serv*, $v \notin \mathbb{F}$ implies $\sigma(v) = \sigma'(v)$ (σ, σ' are the pre- and post-state).

Similar to the assignable sets, a *callable set* is a list of services which can at most be called by a service. $\mathcal{C} \subseteq \mathcal{S}$ is a *callable set* for service *serv* if all traces produced by execution of *serv* at most contain messages on initial and termination channels of the services in $\mathcal{C}$.

Example 6. In the Shop component, an assignable set for *buy* is $\{sum, prods\}$. And $\{\}$ is an assignable set for *print*. The empty set is a callable set for both *buy* and *print*. A callable set for *pay* is $\{trans\}$.

We can use known dependency clusters, assignable, and callable sets for some service *serv* to check if *serv* is non-interferent w.r.t. a component-global specification as follows:

Theorem 4. *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 $\sigma, \sigma', \sigma_p, \sigma'_p$,*

$$\text{if } m \sim_g m' \text{ then } m \sim_{serv} m', \text{ and if } \sigma \approx_g \sigma' \text{ then } \sigma \approx_{serv} \sigma' \quad (1)$$

$$\text{if } m \sim_{serv} m' \text{ and } m \in \mathcal{C} \text{ then } m \sim_g m' \quad (2)$$

$$\text{if } \sigma \approx_g \sigma' \text{ and } \sigma_p \approx_{serv} \sigma'_p \text{ then } anon(\sigma, \mathbb{F}, \sigma_p) \approx_g anon(\sigma', \mathbb{F}, \sigma'_p) \quad (3)$$

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.

Condition (1) states that input messages that are equivalent w.r.t. the component-global relation must also be equivalent w.r.t. the service-local relation, and that if two states are equivalent w.r.t. the global state relation, then they must also be equivalent w.r.t. the service-local relation. Indirectly, this ensures that, if all other services provided by a component ensure equivalence w.r.t. the global equivalence relation for their post state, then *serv* is guaranteed to be executed in pre-states which are equivalent w.r.t. the service-local specification.

In Condition (2), we use $m \in \mathcal{C}$ as abbreviation for m being an initial or terminating message for a service in $\mathcal{C}$. The condition guarantees that all output messages of a service are equivalent globally if they are service-locally equivalent and the messages can actually be communicated during execution of the service. In a similar fashion, Condition (3) guarantees that the parts of the post-states, which are actually changed by the service, are changed such that they are also equivalent w.r.t. the component-global state-equivalence relation.

Note that the condition to be checked according to Theorem 4 can be formalized in first-order predicate logic, if all expressions in the list notion are first-order. (Which we assume to be sufficiently expressive in practice)

Example 7. Reconsidering Example 4, we can use Theorem 4 to show that $LowIO_1, LowState_1$ is a dependency cluster for the service *buy*, since the service-local specification $LowIO_4, LowState_4$ is a dependency cluster for *buy*, as we have seen in the previous section, the expressions mentioning *prods* and *sum* are identical and *check* and *payId* are not in the assignable set. A similar argument holds for the events and the callable set.

It is not necessary to analyze the actual implementation of *buy* if the service-local specification, the assignable set, and the callable set are given.

The second step of our method creates the global non-interference specifications for a system. We consider identification of assignable and callable sets an orthogonal problem to the framework presented here and assume a useful (i.e., small) assignable set and callable set for each service to be given. (In our proof of concept, we automatically generated them with JOANA.) Further, we assume a set of dependency clusters $\{(\sim_{ij1}, \approx_{ij1}), \ldots, (\sim_{ijk}, \approx_{ijk})\}$ for each service s_j provided by component c_i in the system has been specified and verified in the first step of our method. We create a system-global equivalence relation over

messages by intersecting all equivalence relations in the set: $\sim_{sys} = \bigcap_{i,j,k} \sim_{ijk}$. We also create a component-global equivalence relation over states for each component: $\approx_i = \bigcap_{j,k} \sim_{ijk}$. For each service s_j provided by component c_i we prove the first-order formula gained from Theorem 4 with $\sim_g = \sim_{sys}$, $\approx_g = \approx_i$, and $\sim_{serv} = \bigcap_k \sim_{ijk}$, $\approx_{serv} = \bigcap_k \approx_{ijk}$.

The constructed formula is first-order. While each of the formulas is, as we can expect, rather large for a realistic system, big parts trivially evaluate to *true* or *false* because the callable and assignable sets are typically very small compared to the overall system.

Theorem 4 makes dependency clusters very useful for evolving components, since the need for actual program analysis is minimized.

Example 8. Assume that the Shop component from Figure 1 is re-used in a new context where, due to a changed use case, it is required that the last four digits of the credit card number are low (instead of the last bit). To realize this, the implementation of service *pay* is changed: Line 2 is replaced by "`if (check) payId=ccnr-(ccnr/10000)*10000;`". Since the code has changed, the dependency clusters for *pay* have to be re-verified. But the dependency clusters for all other services can be re-used without program analysis when the first order proof for step 2 is repeated.

In a second case of evolution, we assume that context remains the same and the implementation is optimized without changing the functionality. Line 2 is now replaced by "`if (check) payId = ccnr%10000;`". Again, since the code has changed, the dependency clusters for *pay* have to be verified, but since the service's behavior is not changed, no new dependency clusters have to be identified and proofs from step two of our method are still valid.

5 Weakening Specifications

In the third and last step of our method, we show that the system-wide non-interference specification gained from step two implies security of the system against an attacker. The specifications we gain by analyzing dependencies in services do not necessarily match a security policy provided by a domain expert for the system under analysis. While the first two steps of our method provide us with a specification reflecting the actual behavior of the program, the specification from a domain expert is the result of a threat analysis for a system and its context.

In particular, the equivalence relation over messages we gain from the first two steps in our method may be stricter than necessary. We can relax the relation without harm by accepting low input where high input is expected, and we can allow the environment to treat low output of the component as high output.

Definition 4 (Specification weakening). *An equivalence relation $\sim_w$ is a weakening of $\sim$ iff*

- *for $m_1, m_2 \in \mathbb{I}$: $m_1 \sim_w m_2$ implies $m_1 \sim \;\; m_2$,*
- *for $m_1, m_2 \in \mathbb{O}$: $m_1 \sim \;\; m_2$ implies $m_1 \sim_w m_2$*

Example 9. Consider the simple Shop component from Figure 1 with the changes discussed in Example 8 in the previous section. When it is deployed, the domain expert may provide a specification expressing that the cashier may know the last five digits of the credit card number. The specification we gained from bottom-up program analysis, however, provides a stricter specification allowing at most the last four digits to be visible to the cashier. In this case, we can nevertheless use our bottom-up specification as an argument for security as the environment-specific IF-property is a weakening of the bottom-up specification.

Theorem 5. *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.* $(\sim_w, \approx)$*.*

Theorem 5 can easily be extended to components. If all services are non-interferent w.r.t. $(\sim, \approx)$, they also are non-interferent w.r.t. $(\sim_w, \approx)$ and therefore the component is non-interferent w.r.t. $(\sim_w, \approx)$ according to Theorem 2. This implies, for example, that the evolved Shop component from Example 8 is secure in the new environment from Example 9, although the required and the verified IF properties differ.

The third and last step of our method consists of showing that the security policy provided by the domain expert, which represents the actual security requirement, is a weakening of the system-global equivalence relation $\sim_g$ from the second step. Note that the proof obligation implied by Theorem 5 again can be shown using first-order logic and does not require program verification.

On first sight Theorem 5 seems to be a technicality. However, the theorem serves as an important connection between bottom-up specifications, which our method provides, and top-down specifications, gained from context- and attacker-motivated analysis. It frees the systems engineer from finding non-interference specifications for already implemented components which exactly fit the domain-driven idea of secrecy. Thus it serves as a glue which allows flexibility when bringing together domain expertise and context-independent program analysis.

6 Proof of Concept: Verifying JavaEE Implementations

We outline in this section the instantiation of our formal framework for component-based systems (Sect. 2) for a large subset of components implemented in the Java Enterprise Edition (JavaEE) [8], a framework for implementing Component-based Systems in Java. For a full discussion, including all sources and proofs the interested reader is referred to [13].

As a case study, we implemented a simple web-shop consisting of five components. We use the tools KeY and JOANA for verification and analysis of security of the case study against two attackers.

Verification of Dependency Clusters for Services. 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 [30,31,4]. For a full account of KeY and JavaDL, we refer to [1].

We extend JavaDL by events as part of the domain of the logic. We use a static ghost variable, i.e., a specification-only variable, to record the history of events passed during execution of a service. We formalize the general assumptions ensured by the application container according to JavaEE, e.g. no shared heap between components, as method contracts. We formalize proof obligations from the first step of our method and equivalence relations directly in JavaDL.

Automatically Deriving Service Dependency Clusters. To automatically derive dependency clusters, we use *program dependency graphs* (PDGs), a language-independent graph-representation of the dependencies between the statements and expressions of a program. We use the state-of-the-art information flow analysis tool Joana [11,14] to build and use PDGs for our purposes.

PDGs guarantee sequential non-interference [34] in the sense that a node n cannot influence a node n' if n cannot reach n' in the PDG. Hence, in order to obtain a dependency cluster, it suffices to perform reachability analysis on the PDG. We applied Joana to all services in our proof of concept and extracted the majority of all used dependency clusters automatically. Then, we automatically formalized the extracted dependency clusters as JavaDL predicates uniformly to the dependency clusters verified with KeY.

Checking Component-Global Dependency Clusters. In the second step, we re-use formalizations of the equivalence relations from the first step of our method to compose service-local to component-global dependency cluster, formalize Theorem 4 directly in JavaDL and use KeY for the proof. Finally, we used KeY to verify in the third step for each attacker that the attacker-related information-flow specification is a weakening of the specification from step 2, again directly encoded in JavaDL.

Evaluation. We identified 480 dependency clusters in the components of the web shop program with Joana automatically and manually specified and verified 21 dependency clusters with KeY, for which Joana was not sufficiently precise. Verification for the first attacker took about six days, while the main bottlenecks were specification and verification of functional support specifications, as well as manual interaction during verification of proof obligations in steps two and three of our method. Verification for the second attacker only took about one day, since we could make heavy re-use of the specifications for the first attacker.

As a result, we find that KeY is not optimized for proof obligations gained during step 2 and 3 of our method and we assume a high degree of automation if better suited tools are used for this task, for example SMT solvers. Further, we observed that re-using support specifications and dependency clusters for the second attacker made the proof process considerably easier and faster.

7 Conclusion

We introduced dependency clusters as a novel specification approach for information flows caused by a single service in a component-based system. Each

specification is independent from other services in the system and the context, which makes dependency cluster very modular and highly re-usable building blocks for system specifications. Further, we introduced a novel method for constructing system-wide security specifications, where verification of dependency clusters at service-level is the only step requiring program analysis. Proof obligations in the second and third step are first-order formulas, which ensure consistency of the constructed specification w.r.t. an attacker-motivated specification.

For each step, we provide a soundness proof. Moreover, in a proof of concept, we show that our method can be instantiated for JavaEE programs and, for example, is usable for a small but realistic system. For verification of dependency clusters we used the KeY tool and JOANA, and verified the proof obligations for step two and three with KeY, re-using dependency cluster formalizations from the first step. The proof of concept especially showed the re-usability of dependency clusters for different types of attackers.

As future work, we plan to implement native JavaEE support for the KeY tool, a specification language for dependency clusters in JML, as well as proof management within the tool. It would also be very interesting if other program analysis methods could be extended to support our notion of non-interference and if some steps in our method could be further automatized.

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. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015 (2015)
3. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In: ESOP (2007)
4. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: LOPSTR (2013)
5. Chong, S., Vikram, K., Myers, A.C., et al.: Sif: Enforcing confidentiality and integrity in web applications. In: USENIX Security. vol. 7 (2007)
6. Clark, D., Hunt, S.: Non-interference for deterministic interactive programs. In: Formal Aspects in Security and Trust (2009)
7. Cohen, E.: Information transmission in computational systems. SIGOPS Oper. Syst. Rev. (Nov 1977)
8. 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
9. Ereth, S., Mantel, H., Perner, M.: Towards a common specification language for information-flow security in rs3 and beyond: Rifl 1.0 - the language. Tech. Rep. TUD-CS-2014-0115, TU Darmstadt (2014)
10. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Security and Privacy (1982)
11. Graf, J., Hecker, M., Mohr, M.: Using joana for information flow control in java programs - a practical guide. In: ATPS (Feb 2013)

12. Greiner, S., Grahl, D.: Non-interference with what-declassification in component-based systems. In: CSF (2016)
13. Greiner, S., Mohr, M., Beckert, B.: Modular verification of information flow security in component-based systems – proofs and proof of concept (2017)
14. 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)
15. IBM Research: T.J.Watson Library for Analysis (WALA), `http://wala.sf.net`, `http://wala.sf.net`
16. Johnson, A., Waye, L., Moore, S., Chong, S.: Exploring and enforcing security guarantees via program dependence graphs. In: PLDI (Jun 2015)
17. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions (2006)
18. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of java programs. In: CSF (July 2015)
19. Lovat, E., Fromm, A., Mohr, M., Pretschner, A.: SHRIFT – System-Wide HybRid Information Flow Tracking. In: ICT Systems Security and Privacy Protection (2015)
20. Mantel, H.: Possibilistic Definitions of Security — an Assembly Kit. In: CSFW (2000)
21. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: CSF (2011)
22. Murray, T.C., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: CSF (2016)
23. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1999)
24. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: IEEE Security and Privacy (May 2011)
25. O'Neill, K.R., Clarkson, M.R., Chong, S.: Information-flow security for interactive programs. In: CSFW (Jul 2006)
26. Rafnsson, W., Hedin, D., Sabelfeld, A.: Securing interactive programs. In: CSF (2012)
27. Sabelfeld, A., Mantel, H.: Static Confidentiality Enforcement for Distributed Programs. In: Static Analysis. Lecture Notes in Computer Science (2002)
28. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. Higher-Order and Symbolic Computation (2001)
29. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. (Oct 2009)
30. Scheben, C., Schmitt, P.H.: Verification of information flow properties of java programs without approximations. In: FoVeOOS (2011)
31. Scheben, C., Schmitt, P.H.: Efficient self-composition for weakest precondition calculi. In: Formal Methods (2014)
32. Sheldon, M.A., Gifford, D.K.: Static dependent types for first class modules. In: Proceedings of the 1990 ACM conference on LISP and functional programming. ACM (1990)
33. Venkatesh-Prasad Ranganath et al.: Indus, `http://indus.projects.cs.ksu.edu/`, last visited on 2017-02-01
34. Wasserrab, D., Lohner, D.: Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In: VERIFY (2010)