Using Theorem Provers to Increase the Precision of Dependence Analysis for Information Flow Control

Bernhard Beckert, Simon Bischof, Mihai Herda(✉), Michael Kirsten, and Marko Kleine Büning

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{beckert,simon.bischof,herda,kirsten}@kit.edu, marko@kleinebuening.de

Abstract. Information flow control (IFC) is a category of techniques for enforcing information flow properties. In this paper we present the *Combined Approach*, a novel IFC technique that combines a scalable system-dependence-graph-based (SDG-based) approach with a precise logic-based approach based on a theorem prover. The Combined Approach has an increased precision compared with the SDG-based approach on its own, without sacrificing its scalability. For every potential illegal information flow reported by the SDG-based approach, the *Combined Approach* automatically generates proof obligations that, if valid, prove that there is no program path for which the reported information flow can happen. These proof obligations are then relayed to the logic-based approach.

We also show how the SDG-based approach can provide additional information to the theorem prover that helps decrease the verification effort. Moreover, we present a prototypical implementation of the Combined Approach that uses the tools JOANA and KeY as the SDG-based and logic-based approach respectively.

Keywords: Information flow control · Noninterference
System dependence graph · Deductive verification

1 Introduction

When sensitive information leaks to unauthorized parties, it is often the result of bugs and errors introduced into the system during software development. An effective measure to reveal potential sources for such leakages are formal methods, which can provably detect any program instruction that may lead to such a violation of confidentiality. Whereas formal methods already experience successful applications for verifying functional properties, their adoption to security properties in larger programs still either lacks precision or scalability.

Noninterference. An established property guaranteeing confidentiality on code level is *noninterference*. Noninterference holds if no information flow from a

J. Sun and M. Sun (Eds.): ICFEM 2018, LNCS 11232, pp. 284–300, 2018.
https://doi.org/10.1007/978-3-030-02450-5_17

secret input (of *high* security) to a public output (of *low* security) of the system is possible, i.e., if and only if no secret input of a program may influence its public output. Research on secure information flow dates back to the works of Denning and Denning [5,6] and later Goguen and Meseguer [8]. In the following, we formally define the noninterference property that we want to prove using the approach described within this work. We distinguish high variables containing secret data, which should be protected, from low variables, which are publicly readable, and introduce the *low-equivalence* relation ($\sim_L$) to characterize program states that are indistinguishable for any potential attacker. A program state s is an assignment of values to program variables and program locations. We assume that the input of a program is included in the program's initial state and that the output of a program is included its final state. Two states, s and s', are low-equivalent iff all low variables in s have the same value as in s'.

Definition 1 (Noninterference). *A program P is noninterferent iff, for any initial states s_1 and s_2, the statement*

$$s_1 \sim_L s_2 \Rightarrow s_1' \sim_L s_2'$$

holds, where s_1' and s_2' are final program states after executing P in the initial states s_1 and s_2, respectively.

This means that two program executions starting in two low-equivalent states must terminate in two low-equivalent states. This guarantees that low outputs are not influenced by high inputs. Note that we restrict ourselves to terminating programs. In the rest of this work, we will refer to noninterference with respect to a given high input and a given low output simply as noninterference.

Existing Approaches. There exist various approaches and tools for checking the noninterference property of a program. In what follows, we describe the types of approaches, which we will combine in our approach.

Approaches that are based on *System Dependence Graphs* (SDGs) syntactically compute the dependences between program statements and check whether any low output syntactically depends on high input (see, for example, the JOANA tool [10]). Whereas this scales very well, such approaches overapproximate the actual dependences in the program, which may result in false alarms, since the analysis only works on the syntactical level of the program. For example, in a program such as "`l=l+h; l=l-h;`", a syntactical approach will identify a dependence between the variables `l` and `h`, even though there is in fact no (semantical) dependence between them.

Logic-based approaches (e.g., using dynamic logic for Java [3]), on the other hand, have a higher precision, i.e., they produce less false alarms, because they also consider the semantics of program statements. However, those approaches have a lower scalability. Using a logical proof calculus, the logic-based approaches' proof obligation is to show that the terminating states of two program executions are low-equivalent, assuming their two initial states are low-equivalent. False alarms only occur when the system fails to find a proof in the

allotted time even though the proof obligation is valid. Proving noninterference using this approach involves simultaneously checking all execution paths for two program executions. This makes the noninterference proof harder than proving a functional property as the number of execution paths to be checked is quadratic.

Our Contribution. In this paper we present the *Combined Approach*, a novel IFC technique that combines an SDG-based approach with a logic-based approach, and in consequence achieves a higher precision than the solely SDG-based approach. The Combined Approach analyzes the dependences from security violations reported by the SDG-based approach and proves the absence of these dependences (given the program actually is noninterferent) using a theorem prover. While the deduction steps in the theorem prover may require user interaction, we automatically generate its proof obligations from the reported dependences.

Furthermore, we reduce the verification effort by enriching the generated proof obligations with information obtained from the SDG-based approach. The information relayed to the theorem prover consists of information flow contracts for the called methods, (partial) loop invariants for loops inside the verified code, and preconditions generated by a points-to analysis.

Structure of the Paper. We organize the paper as follows. Sections 2 and 3 define the SDG-based and logic-based information flow analysis techniques, respectively, used by the Combined Approach. Section 4 presents the Combined Approach. A prototypical implementation of the Combined Approach is presented in Sect. 5. Related work is discussed in Sect. 6. Finally, Sect. 7 concludes.

2 SDG-Based Information Flow Control

SDG-based information flow analyses are purely syntactic, highly scalable, and sound. However, some of the reported noninterference violations may be false alarms. While Program Dependence Graphs (PDGs) [7] and System Dependence Graphs (SDGs) [15] have been developed during the eighties, their usefulness in the context of information-flow security has been first noticed by Snelting [24] in the nineties. Decades of research in this area have resulted in JOANA, a tool that statically analyzes Java programs of up to 100k lines of code for integrity and confidentiality [9,13]. We have used JOANA for a prototype implementation of our approach.

Without loss of generality, we use the JOANA tool as an example to explain the functionality of SDG-based information-flow analysis, but our approach applies to other SDG-based analysis tools as well. The following explanation is partially based on [14], where SDG-based analysis is also used as a preprocessing step (see Sect. 6). We define SDGs for deterministic inter-procedural programs with variable assignments, branchings, loops, and function calls. Moreover, we assume that a control-flow graph (CFG) for such programs exists.

The desired noninterference property is specified by annotating which program parts correspond to sensitive (high) information and the parts where public

(low) output occurs. Given these annotations, JOANA automatically builds an SDG for the program. An SDG is a directed graph consisting of interconnected Program Dependence Graphs (PDGs), where a PDG represents a single program procedure as a directed graph. More details on SDGs can be found in [15].

Nodes in the SDG represent program statements, conditions, or input parameters, and edges represent dependences between the nodes, i.e., an edge between two nodes exists if and only if the value or execution of one node may depend on the outcome of the other node. Whether an edge exists between two nodes in the SDG is determined syntactically by analyzing the control-flow graph of the analyzed program. There are roughly three types of edges in an SDG: (1) data dependence edges, representing possible direct dependences, (2) control dependences, which represent possible indirect dependences, and (3) interprocedural dependences, which represent dependences between nodes of different PDGs (other dependences have been introduced to support object orientation and multithreading, see [11]). Formal definitions for the three types of dependences can be found in [11, Chap. 2]. In the following, we give informal definitions.

A node n' is data-dependent on n if there is a program variable v that is used in n' and defined in n, and there is a path from n to n' in the CFG such that v is not redefined on any node between n and n' on that path. The standard definition of a control dependence between two nodes states that a node n' is control-dependent on a node n if the choice of the outgoing edge from n in the CFG determines whether node n' is reached. Note that it is undecidable whether a CFG path represents an actual execution path of the program, i.e., some paths in the CFG may represent executions that cannot actually take place. The CFG is thus an over-approximation of the actual program behavior. Since the dependences are defined using CFG paths, they too are an over-approximation of the actual (semantical) dependences in the program. In the rest of the paper, we refer to the program execution described by a CFG path as *execution path*. Note that an SDG path may also represent one or more actual execution paths.

Method calls are represented by special formal-in and formal-out nodes in the SDG. Formal-in nodes represent direct inputs that influence the method execution. These are the input parameters, used fields, and other classes called during execution and the class in which the method is executed. Moreover, formal-out nodes represent the influence of the method. In most cases, a formal-out node represents the method's return value. Other possibilities are that the method influences global variables, fields in other classes, or terminates with an exception.

```
int f(int x, int y) { return x; }        void caller() { ... f(a,b); ...}
```

Listing 1. Method call

As example, for function `f` in Listing 1, we have two formal-in nodes for `x` and `y`, and one formal-out node for the return value of `f`. At each method call site, there are actual-in nodes representing the arguments and actual-out nodes representing the return values. For a given method site, each actual-in node corresponds to a formal-in node of the callee and vice versa; the same

holds for actual-out and formal-out nodes. Interprocedural dependences connect actual-in nodes to the corresponding formal-in nodes, and formal-out nodes to the corresponding actual-out nodes. For the call in Listing 1, there are actual-in nodes for `a` and `b`, corresponding to `f`'s formal-in nodes for `x` and `y`, respectively. The actual-out node representing the return value of `f` corresponds to the single formal-out node of `f`. For every method call we also have so-called *summary edges* in the SDG from any actual-in to any actual-out node of the method whenever the tool finds a flow from the formal-in to the formal-out node of the called method. In Listing 1, we have a flow in `f` from `x` to the return value, thus a summary edge is inserted at call site, namely from `a`'s actual-in to the single actual-out node. For a method involving many objects, there can be a huge number of actual-in and actual-out nodes and an even greater number of summary edges.

SDG-based information-flow analysis approaches, such as the one implemented by JOANA, detect illegal information flows through graph analysis, using a special form of conditional reachability analysis – *slicing* and *chopping* – at the SDG level. A forward slice of a node s consists of all nodes in SDG paths starting in s. Conversely, a backward slice of a node s consists of all nodes in SDG paths ending in s. A chop from a node s to a node t consists of all nodes on paths from s to t in the SDG and is commonly computed by calculating the backward slice for t, and then computing the forward slice for s within the subgraph induced by it. When the slicer or chopper encounters a method call site, it descends into the called method without ascending back up. However, this cannot miss any potential information flow, since for every flow through that method, a summary edge was inserted at the call site, which can be taken as a shortcut. JOANA reports a security violation whenever there exists a path from a node in the SDG that is annotated as `high` to a node annotated as `low`, i.e., when the chop of these two nodes is not empty. It has been proven that this approach may not miss any potential information flow, i.e., that JOANA is sound, and that any illegal information flow in the program can occur only in the execution paths determined by an SDG path from a high node to a low node [27]. Since the dependences in the SDG are over-approximations of actual dependences in the program, if no SDG path for the illegal flow is found, the program is guaranteed to be noninterferent. However, whenever there is an SDG path between a high input and a low output, the program may still be noninterferent.

3 Logic-Based Information Flow Control

Logic-based information flow analysis takes the semantics of the program language into account. The semantics of modern program languages provide a high degree of expressiveness, which must be considered when sources of illegal information leaks may be exploiting features of the program semantics. Logic provides a means for abstraction and can capture such features and moreover, using logical calculi, enables reasoning about their –direct or indirect, explicit or implicit– effects on any low program variables or locations. However, this requires a logical representation of the program together with the precise property we want

to prove. Using dynamic logic [4] together with symbolic values, we can express the functional property of *partial correctness* of a program P for a precondition ϕ and a postcondition ψ by the following formula:

$$\phi \rightarrow [P]\psi$$

This means that ψ holds in all possible states in which P terminates. Since we analyze only deterministic programs, this means that either P terminates and ψ holds afterwards, or the program never terminates. Since we restrict ourselves to terminating programs, we only need to prove partial correctness in the following. Applying a logical calculus with a deductive theorem prover, we can hence symbolically execute P and attempt to prove the formula.

On this basis, we state the noninterference property based on value independence for a high variable h, a low variable l and a program P in the following way:

Definition 2 (Noninterference as value independence). *When starting* $\mathtt{P}$ *with arbitrary values* $\mathtt{l}$*, then the value* r *of* $\mathtt{l}$ *– after executing* $\mathtt{P}$ *– is independent of the choice of* $\mathtt{h}$ *(note the order of the quantifiers).*

$$\forall \mathtt{l}\ \exists r\ \forall \mathtt{h}\ [P]\ r = \mathtt{l}$$

However, instantiating existential quantifiers hinders automation and requires user interaction. As a mitigation, [2] established a noninterference formalization based on self-composition, effectively reducing it to a safety property. Using self-composition, the noninterference property of a program P translates to a safety property of a new program which consists of P composed with a renaming of P.

Furthermore, we need to introduce the concept of state updates [1], which capture the effects of symbolically executing program statements. We denote updates by variable assignments enclosed by curly braces, which are applied to logical terms and formulae, and thus change the program state.

We can now, based on the low-equivalence in Definition 1 from Sect. 1, extend our formalization of noninterference in Definition 3.

Definition 3 (Noninterference as self-composition with state updates).

$$\begin{aligned} \forall in_l\ \forall in_h^1\ \forall in_h^2\ \forall out_l^1\ \forall out_l^2\ \{l := in_l\}(\\ \{h := in_h^1\}[P]\, out_l^1 = l \\ \wedge\ \{h := in_h^2\}[P]\, out_l^2 = l \\ \rightarrow\ out_l^1 = out_l^2) \end{aligned}$$

Therein, we have two executions of P, one where the (high) program variable h is renamed to in_h^1, and another one where it is renamed to in_h^2. The (low) output variable l is captured in the variable out_l^1 after the first execution and in variable out_l^2 after the second one. Finally, we need to prove that both outputs

out_l^1 and out_l^2 are equivalent in the final state and assume equivalent low inputs via the variable in_l. The self-composition formula can hence be enclosed with purely universal quantifiers over the renaming variables for input and output. When trying to prove noninterference for a program P, theorem provers can now skolemize these variables and greatly reduce the necessary user interaction.

Now, when dealing with object-orientation, it is sometimes too strict to require all (low) variables and locations in the final state to be equivalent. For this matter, [23] developed a variation of noninterference using a different semantics of low-equivalence based on an object isomorphism as defined in Definition 4. Therein, for any two states s_1 and s_2, and two isomorphisms π_1 and π_2, $\pi_1(o) = \pi_2(o)$ holds if o is observable in both states s_1 and s_2.

Definition 4 (Low-equivalence with isomorphism). *Two states s, s' are low-equivalent iff they assign the same values to low variables (with L denoting the set of all low variables in state s).*

$$s \simeq_L^{\pi} s' \quad \Leftrightarrow \quad \forall\, v \in L\, (\pi(v^s) = v^{s'})$$

The techniques described above together with this semantics are defined and implemented in the deductive program verification tool KeY for Java [1]. It furthermore allows for more efficient noninterference proofs using modularization via the *design-by-contract* concept with an extension of the Java Modeling Language (JML) [18]. Such a contract specifies the low program variables and locations for the initial and the final state of the specified program part. The proof obligation hence requires the low elements in the final state to depend at most on the low elements in the initial state. When using the semantics for object isomorphisms, these contracts may also contain a list of fresh objects to be included in the isomorphism.

In general, the problem is undecidable and verification sometimes requires some user interaction. KeY is capable of verifying noninterference for Java programs and covers a wide range of Java features. With this toolkit, powerful specification elements are given for proving noninterference, also allowing for declassification.

4 The Combined Approach

In the following, we describe our Combined Approach on the example of proving noninterference for a given program P. The first step of the Combined Approach consists of running the SDG-based analysis to check the noninterference property for P. If there is no illegal information flow for P, we need no further action as noninterference is guaranteed to hold. If – however – the automatic SDG-based approach detects an illegal information flow, we apply the second step of the Combined Approach in order to check whether this information flow is a false positive or a genuine leak. Since the SDG-based analysis is performed as the first step, the results provided by our approach are at least as good as those of the SDG-based analysis.

The SDG-based analysis creates an SDG that models the syntactic dependences between the program parts of P. However, as explained in Sect. 2, these dependences represent an over-approximation of the actual program dependences. The goal of the Combined Approach is to use a logic-based IFC approach to prove that certain syntactic dependences in the SDG do not represent real dependences. If all syntactic dependences between the high inputs and the low outputs reported by the SDG-based analysis are proven, using the logic-based approach, to not exist semantically, then the analyzed noninterference property is proven to hold for P. We assume that the SDG-nodes corresponding to high inputs and low outputs are annotated as high and low respectively. Let N_h denote the set of all nodes annotated as high, and N_ℓ the set of all nodes annotated as low.

The SDG-based approach then returns a set of *violations*. A violation is a pair (n_h, n_ℓ) of a high node $n_h \in N_h$ and a low node $n_\ell \in N_\ell$ such that there is a path from n_h to n_ℓ in the SDG of P. We then call the set of all nodes lying on a path from n_h to n_ℓ the *violation chop* $c(n_\ell, n_h)$. To keep the notation simple, we will also use $c(n_h, n_\ell)$ for the subgraph induced by those nodes. If the set of all violation chops, denoted by C_V, is empty, the SDG-based approach guarantees noninterference. If – however – there is a false positive, C_V contains at least one chop. The idea of the Combined Approach is then to validate each violation chop $c(n_h, n_\ell) \in C_V$ and attempt to prove that the chop does not exist on the semantic level in program P. We prove this by showing that each violation chop is interrupted (see Definition 5) with the help of a logic-based approach.

Definition 5 (Unnecessary summary edge, Interrupted violation chop). *A summary edge $e = (a_i, a_o)$ is called* unnecessary *if there is no information flow from the formal-in node f_i to the formal-out node f_o corresponding to a_i and a_o, respectively.*

A violation chop is interrupted*, if we find a non-empty set S of unnecessary summary edges on this chop, such that after deleting the edges in S from the SDG, no path exists between the source and the sink of the violation chop.*

In order to show that a summary edge $e = (a_i, a_o)$ is unnecessary, a proof obligation is generated for the theorem prover of the logic-based approach. This proof obligation states that there is no information flow from the formal-in node f_i to the formal-out node f_o corresponding to the summary edge e (Sect. 5.1 contains a more precise description of the proof obligation). The proof is done for all possible contexts of the called method. If the proof is successful, we have proven that the summary edge was only inserted as a result of the over-approximation, and we can soundly delete this edge.

Note that for checking whether a violation chop is interrupted, we rely on the way the chopper works on method call sites: When deleting a summary edge, the chopper still finds the corresponding information flow in the called method because no dependence edges have been deleted there. However, since it does not ascend back to the caller and relies on the (now deleted) summary edge, the chopper proceeds in the caller as if it did not find that corresponding information flow.

```
Data: Set of violation chops S
Result: Noninterference guarantee or failed verification attempt
foreach Violation chop C_V ∈ S do
    Build queue Q of summary edges in C_V, ordered by heuristics;
    while C_V not interrupted and Q not empty do
        Pop summary edge e from Q;
        Generate proof obligation PO for proving that e is unnecessary;
        if PO proved with theorem prover then
            Delete e from C_V;
        end
    end
end
```

Algorithm 1. The Combined Approach

Our approach, shown in Algorithm 1, attempts to interrupt each violation chop in C_V. For each violation chop a summary edge is taken, the appropriate information flow proof obligation is generated for the method corresponding to the summary edge, and a proof attempt is made using the theorem prover. If the proof is successful, the summary edge can then be deleted from the SDG, based on Definition 5. The order in which the summary edges are checked is established by a heuristic which is explained towards the end of this section. Note that we only need to consider summary edges that belong to a *chop* between high and low. Thus, it is sufficient to regard only a smaller subset of all summary edges. We then check whether this violation chop is interrupted. In this case we can proceed to analyze the remaining violation chops until all of them are interrupted. In case the violation chop is still not interrupted, or the proof attempt is not successful, another summary edge from the violation chop is chosen. If we are able to interrupt every violation chop by deleting unnecessary edges, our approach guarantees noninterference.

Theorem 1 (Noninterference Combined Approach). *The Combined Approach guarantees noninterference.*

Proof. Let S be the set of unnecessary summary edges that interrupt a violation chop $c(n_h, n_\ell) \in C_V$. Using the logic-based approach, we have shown for each summary edge $e = (a_i, a_o) \in S$ that the actual-out node a_o does not depend on the actual-in node a_i of that summary edge. Since each path from n_h to n_ℓ contains one such summary edge we have in fact shown that the potential dependences from n_h to n_ℓ, represented by the violation chop, do not represent real dependences. The soundness of the SDG-based approach guarantees that there are no other potential dependences from n_h to n_ℓ than the ones in the chop. Thus, proving all violation chops to be interrupted proves that the program is noninterferent. □

Note that each violation chop is guaranteed to contain at least one summary edge, namely the one corresponding to the `main` method. Generating a proof

obligation for the `main` method – however – is equivalent to verifying the entire program with the theorem prover. In practice, however, programs are interprocedural and thus there are plenty of summary edges for our approach to check. Nevertheless, the verification of the `main` method with the theorem prover is still the worst case of our approach and can occur in case not enough summary edges of inner method calls can be proved to be unnecessary.

```
public int test(int high, int low) {
  int result = identity(high, low);
  return result;
}

public int identity(int h, int l) {
  l = l + h;
  l = l − h;
  return l;
}
```

Listing 2. Example program

For the example in Listing 2, when trying to show that there is no information flow from the parameter `high` to the return value of the method `test`, the SDG-based approach reports an illegal information flow, because the return value of the method `identity` is data-dependent on the parameter `h` of the same method. This is, however, a mere syntactic dependence and the reported violation is a false alarm. The reported violation chop contains only one path which contains the actual-in SDG-node representing parameter h and the actual-out SDG-node representing the return value of `identity`, connected by a summary edge as explained in Sect. 2. The Combined Approach automatically generates a proof obligation for the logic-based approach which states that the return value of `identity` does not depend on parameter `h`. By proving this, we also prove that the return value of the method `test` does not depend on the parameter `high` and thus show the noninterference of `test`. This simple example showcases a major advantage of our approach: the logic-based approach does not need to analyze the entire program, but only those parts that cannot be handled with the SDG-based approach.

Proofs with the theorem prover are often performed fully automatically, but may sometimes need auxiliary specification and user interaction. Therefore, we want to minimize the theorem prover usage as much as possible. The order in which the summary edges of the violation chops are checked has a major impact on the performance of the Combined Approach. Ideally we want to avoid proof attempts of methods that do have an information flow or of very large methods that would overwhelm the theorem prover (for example the `main` method). In order to achieve these goals, we developed several heuristics for establishing the order in which we check the summary edges with the logic-based approach. A first category of heuristics searches the code for code patterns that are likely to cause false positives by the SDG-based approach. Such patterns include code that contains array handling, arithmetic operations, or code that can throw runtime exceptions. SDG-based approaches are particularly prone to report false positives for such code, because they neither distinguish between the different array fields nor do they take the values of variables and semantics of operators

into account. The second category of heuristics attempts to identify the methods that are likely to run through the theorem prover automatically. Earlier, we mentioned that it is difficult to create precise loop-invariants and thus methods without loops are assigned a higher priority. Additionally, depending on the tools used, we can exclude methods that contain programming language features that are not supported by the logic-based approach, or library methods from the analysis. A third category of heuristics tries to identify the methods that, if proven noninterferent, would bring the greatest benefit to the goal of proving the entire program noninterferent. We assign a high priority to summary edges which are *bridges* in the SDG, i.e., an edge whose removal from the SDG would result in two unconnected graphs. In case no bridge exists within the SDG, we prefer the method with the highest number of connections, i.e., the most often called method.

Due to its low scalability, the logic-based approach is more likely to handle methods that are deeper in the call graph (i.e., that call few other methods) than methods which are high in the call graph. However, the parts of the program that can disprove a reported security violation may be present in a high level method. In order to still be able to handle such cases, we automatically generate information flow contracts for the method calls occurring inside the analyzed method based on the results of the SDG-based analysis. These method calls have actual-in and actual-out SDG-nodes connected by summary edges. The generated information flow contracts state that the program parts corresponding to the actual nodes of the method call site depend at most on the actual-out nodes of the respective method call site. Due to the soundness of the SDG-based analysis, this information flow contract is also sound. However, the over-approximation done by the SDG-based analysis is also present in the contracts generated this way. Thus, using such contracts does not guarantee that the logic-based approach will successfully disprove the reported security violation, but it allows for an analysis of higher-level methods.

5 Implementation

We implemented[1] the Combined Approach using JOANA as the dependence-graph analysis tool and KeY as the theorem prover. In this section, we show how we generate the proof obligations for KeY in the form of specified Java code and also present the results of running the Combined Approach on a collection of examples that cannot be handled by JOANA alone.

5.1 Specification Generation

For the method corresponding to the summary edge selected by the heuristics, we generate an information flow method contract such that a successful proof would show that there is in fact no dependence from the formal-in to the formal-out node of the summary edge.

[1] Code available at https://git.scc.kit.edu/py8074/keyjoana.

Thus, in order to show that a summary edge $se(a_i, a_o)$ is unnecessary, we prove that there is no information flow between the corresponding formal-in node f_i and formal-out node f_o. In order to achieve this, we generate a JML specification for the appropriate method stating that f_o is determined by all formal-in nodes other than f_i, as explained in Definition 6. Note that the `determines` clause used in Definitions 7 and 8 is not part of the JML standard, and is only supported by KeY. The clause requires the expressions before the `by` keyword, evaluated in the post-state, to depend at most on the expressions after the `by` keyword, evaluated in the pre-state.

Definition 6 (Generation of the determines clause). *Let $se(a_i, a_o)$ be the summary edge to be checked, and let f_i and f_o be the formal nodes corresponding to the actual nodes a_i and a_o. Let L_i be a list of all formal-in nodes f_i' other than f_i of the method belonging to the call site of a_i and a_o. The following determines clause is added to the method contract:* `determines` f_o `\by` L_i.

Should the proof of this property succeed then it would show that f_o does not depend on f_i and therefore a_o does not depend on actual-in parameter a_i. Since there is no dependence between a_i and a_o the summary edge can be safely deleted from the violation chop.

To increase its precision, JOANA uses a points-to analysis which keeps track of the objects a reference o may point to (the points-to set of o). This information is useful, since it may show that two references cannot be aliased. We use the results of the points-to analysis to generate preconditions for the method contracts, as shown in Definition 7, thus transferring information about the context from JOANA to KeY and increasing the likelihood of a successful proof.

Definition 7 (Generation of preconditions). *Let o be a reference and P_o its points-to set. We generate the following precondition:* $\bigvee_{o' \in P_o} o = o'$

The method contracts generated this way are necessary for proving a summary edge is unnecessary, however in the general case they are not sufficient for a successful proof. If the method contains loops of any kind, the theorem prover needs loop-invariants. The automatic generation of loop-invariants is an active research field, see for example [16,21]. These approaches focus on functional loop-invariants and do not consider information flow loop-invariants.

The determines clause generated for method contracts, can be used to specify the allowed information flows of a loop. The determines clause generated for a loop invariant is similar to the one for method contracts. Because the variables from the formal-in and formal-out nodes may not directly occur in the loop some adjustments are necessary. Definition 8 shows what determines clauses are generated for loop invariants:

Definition 8 (Generation of the determines clause for loop invariants). *Let $se(a_i, a_o)$ be the summary edge to be checked, and let f_i and f_o be the formal nodes corresponding to the actual nodes a_i and a_o. Let L_i be a list of all formal-in nodes f_i' other than f_i of the method belonging to the call site of a_i and a_o. Let V_i be the set of all variables in the loop and let I_i be a list of variables in the*

method that influence f_o. The following determines clause is added to the loop invariant: `determines` f_o, V_i `\by` L_i, I_i.

Note that the sets V_i and I_i can be constructed by analyzing the SDG.

5.2 Evaluation

We considered eleven examples, which cover different program structures and reasons for false positives. Each of these examples is not solvable by automated graph based approaches like JOANA. In Table 1 we have listed the eleven examples. The evaluation is split into automatic mode and interactive mode. In the automatic mode, an attempt is made to prove the generated proof obligations automatically. In the interactive mode, the theorem prover is called for all proof obligations in interactive mode. In this mode, the user can perform automatic or interactive steps and can add auxiliary specification. The column *KeY Calls* represents the number of times KeY was called to show that a summary edge is unnecessary. As can be seen in the table, in interactive mode sometimes fewer calls to KeY are necessary, as the user can better recognize which summary edges are more likely to be successfully proven as unnecessary.

The eleven examples are again divided into two groups. First, there are individual methods that cause false positives. In the method `Identity`, the high value is added and subtracted to the low variable such that the low value remains the same. There is a dependence from high to low on a syntactical level, but in reality there is none. In the method `Precondition` there is an if-condition that can never be true and the method `Excluding Statements` contains if-statements that can not both be true at the same program execution. The example `Loop Override` contains a loop which overrides the low value in the last loop execution. For this example the noninterference loop-invariant was not enough for an automated proof and further functional information had to be given by the user. The last simple method `Array Access` contains array handling code. The second group consists of programs that include these problems in different program structures. Based on the possible SDG, we regard simple flows, branching, nested summary edges and a combination of all.

The example programs are in the range of 5 to 30 lines of code. They show that the combined approach can prove programs automatically for which JOANA would generate false positives.

6 Related Work

There exist many different approaches for proving noninterference. A survey on approaches for IFC is found in [22]. In what follows, we describe some approaches that are similar to ours.

The *Hybrid Approach* [17] also aims to combine automatic dependence-graph analysis and theorem proving. The user first attempts to show noninterference using JOANA. If the user suspects the reported violation to be a false alarm, he

Table 1. List of examples

Program	*Automatic Mode*			*Interactive Mode*	
	Provable	KeY Calls	Time	Provable	KeY Calls
Individual methods					
Identity	Yes	1	5 s	Yes	1
Precondition	Yes	1	5 s	Yes	1
Excluding statements	Yes	1	5 s	Yes	1
Loop override	No	1	7 s	Yes	1
Array access	Yes	1	6 s	Yes	1
Whole programs					
KeY example	Yes	1	7 s	Yes	1
Single flow	Yes	1	6 s	Yes	1
Branching	Yes	2	10 s	Yes	2
Nested methods	Yes	2	10 s	Yes	2
Mixture	Yes	4	19 s	Yes	3
Mixture with loops	No	7	20 s	Yes	5

must identify the cause of the alarm and extend the program such that the low output is overwritten with a value that does not depend on the high input. The extended program is rechecked by JOANA, and if deemed noninterferent, KeY is used to show that the extended program computes the same low output as the original. This approach improves the precision provided by JOANA. However, there is no assistance in finding the causes of the false alarms, and the program extension must be done manually.

SDG-based approaches can also be used to identify program statements that do not contribute to a potential information flow or program execution paths that are guaranteed to not lead to an illegal information flow. This is done in [14], where the SDG-based approach is used to generate a simplified program that can then be more easily verified or tested. The approach is orthogonal to the Combined Approach presented in this paper, and the two approaches can be combined by using the approach in [14] to simplify the program for which we attempt to show that a summary edge is unnecessary.

Another combination of SDG-based approaches and theorem provers is by checking the satisfiability of the path conditions for the execution paths determined by the reported security violation [12,25]. If a path condition is unsatisfiable, then that execution path cannot lead to an illegal information flow.

Another class of approaches for information flow control are based on *type systems* [19,26]. They can have the same scalability and precision as SDG-based approaches [20], though most type systems have higher scalability but lower precision. They enforce secure information flow by assigning a security type (e.g.,

high or low) to the program variables and then checking whether the expressions in the program conform to the type system.

7 Conclusion

In this work, we introduced a new combined approach to prove noninterference with less user interaction while keeping the same precision. Our approach combines an automated SDG-based technique with a deductive theorem prover. We demonstrated that the noninterference properties guaranteed by the two tools are compatible and, thus, that our approach is sound. The Combined Approach has been developed tool-independently, but implemented and evaluated on a selection of examples as well as a small case study. Although the programs covered in our evaluation do not exceed 100 lines of code and could – as such – also be proven without the help of SDG-based IFC, they could – however – also be embedded in much bigger programs, which – as such – may be clearly too big for the analysis with a theorem prover.

Acknowledgements. We are grateful to the student Holger Klein for implementing the prototype. This work was supported by the German Research Foundation (DFG) under the project DeduSec (BE 2334/6-3) in the priority program "Reliably Secure Software Systems" (RS^3, SPP 1496).

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. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17 2004, pp. 100–114. IEEE Computer Society (2004)
3. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013. LNCS, vol. 8901, pp. 19–37. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14125-1_2
4. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32004-3_20
5. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
6. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
7. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
8. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy (SP), pp. 11–20 (1982)

9. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs - a practical guide. In: Wagner, S., Lichter, H. (eds.) Conference on Programming Languages (ATP). LNI, vol. 215, pp. 123–138. Springer, Heidelberg (2013)
10. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs-a practical guide. In: Wagner, S., Lichter, H. (eds.) Software Engineering, Fachtagung des GI-Fachbereichs Softwaretechnik. LNI, vol. 215, pp. 123–138. GI (2013)
11. Hammer, C.: Information flow control for java - a comprehensive approach based on path conditions in dependence graphs. Ph.D. thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012049
12. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: Symposium on Secure Software Engineering, pp. 87–96 (2006)
13. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Sec. **8**(6), 399–422 (2009)
14. Herda, M., Tyszberowicz, S., Beckert, B.: Using dependence graphs to assist verification and testing of information-flow properties. In: Dubois, C., Wolff, B. (eds.) TAP 2018. LNCS, vol. 10889, pp. 83–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92994-1_5
15. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
16. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Baader, F., Baumgartner, P., Nieuwenhuis, R., Voronkov, A. (eds.) Deduction and Applications, 23-28 October 2005. Dagstuhl Seminar Proceedings, vol. 05431. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
17. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) 28th Computer Security Foundations Symposium (CSF), pp. 305–319. IEEE Computer Society (2015)
18. Leavens, G.T., Kiniry, J.R., Poll, E.: A JML tutorial: modular specification and verification of functional behavior for Java. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 37–37. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_6
19. Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., Weber, A.: Cassandra: towards a certifying app store for Android. In: ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM), pp. 93–104. ACM (2014)
20. Mantel, H., Sudbrock, H.: Types vs. PDGs in information flow analysis. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 106–121. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38197-3_8
21. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symbolic Comput. **42**(4), 443–476 (2007)
22. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. A. Commun. **21**(1), 5–19 (2006)
23. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31762-0_15

24. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 332–348. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_51
25. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. Softw. Eng. Methodol. **15**(4), 410–457 (2006)
26. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
27. Wasserrab, D., Lohner, D.: Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In: Aderhold, M., Autexier, S., Mantel, H. (eds.) Verification Workshop (VERIFY). EPiC Series in Computing, vol. 3, pp. 141–155 (2010)