Combining Graph-Based and Deduction-Based Information-Flow Analysis

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 ensuring system security by enforcing information flow properties such as non-interference. Established IFC techniques range from fully automatic approaches with much over-approximation to approaches with high precision but potentially laborious user interaction. A noteworthy approach mitigating the weaknesses of both automatic and interactive IFC techniques is the hybrid approach, developed by Küsters et al., which – however – is based on program modifications and still requires a significant amount of user interaction.

In this paper, we present a combined approach that works without any program modifications. It minimizes potential user interactions by applying a dependency-graph-based information-flow analysis first. Based on over-approximations, this step potentially generates false positives. Precise non-interference proofs are achieved by applying a deductive theorem prover with a specialized information-flow calculus for checking that no path from a secret input to a public output exists. Both tools are fully integrated into a combined approach, which is evaluated on a case study, demonstrating the feasibility of automatic and precise non-interference proofs for complex programs.

1 Introduction

When sensitive information leaks to unauthorized parties, this is often a result from bugs and errors introduced in the system during software development. In order to prevent such a leakage, we analyze systems for the absence of illegal information flows, also defined as confidentiality [20]. An established approach for proving confidentiality for a system is to prove the *non-interference* property. Non-interference holds if there is no illegal information flow from a secret (high) input to a public (low) output of the system.

There are a variety of different techniques for checking non-interference. Within this work, we distinguish these techniques by their required user interaction and the precision of their checks. These are (a) fully automatic approaches based on type checking [23] or dependency graphs [7], which require no user interaction. Due to decidability problems, this automation comes at the cost of over-approximation and therefore can generate *false positives*. A false positive is a situation where the approach indicates an illegal information flow, even though there is none. Tools implementing such techniques are, e.g., JIF [18] and JOANA [14]. There are also (b) interactive techniques based on theorem provers that do not run automatically, but achieve more precise non-interference checks [3]. Interactive approaches do not generate false positives, but require a high degree of time- and user-interaction.

Küsters et al. developed a hybrid approach that combines an automatic dependency-graph analysis with a theorem prover to minimize the user-effort of non-interference proofs [16]. The hybrid approach attempts to show non-interference with the dependency-graph analysis tool and –if not successful– the user must extend the program such that the affected low output is overwritten independently of the high inputs. Hence, the proof task is divided into two parts, one equivalence proof for the theorem prover, and one non-interference check with the dependency-graph analysis. In this paper we provide an alternative way to combine both tools, thereby not requiring the user to modify the original program in any way, and thus automating large parts of the combination that requires user interaction.

Our contribution is a new combined approach that improves state-of-the-art approaches regarding the automation while maintaining the same level of precision. The combined approach attempts to show the non-interference property of a program using the dependency-graph tool. If the attempt fails, the reported leaks are disproved with the theorem prover. The communication between the two approaches is fully automated: the reported leaks are analyzed one by one and the information flow proof obligations necessary to disprove the leaks are generated and given to the theorem prover. The user can simply provide the program (with high sources and low sinks annotated) to our tool-chain and attempt to prove non-interference. The user does not need to analyze the output of the dependency-graph analysis tool and manually extend the program. The proof obligations for the theorem prover consist of code with generated specification that need to be proven in order to show that the reported leaks are false positives. However, the user might need to provide additional auxiliary specification (e.g., loop invariants) in order for the proof attempt to succeed, this is generally an undecidable problem.

We implemented this new combined approach for the Java programming language focusing on sequential and terminating programs. For the implementation, we chose JOANA [14] as the dependency-graph analysis tool and KeY [1] as the theorem prover, both based on state-of-the-art approaches. The approach is evaluated based on examples and a small case study for which JOANA by itself returns false positives. For these examples, a direct non-interference proof with KeY would require a high degree of user interaction in the form of specifications and proof-interactions. Based on these examples, we show that our approach can prove non-interference automatically but also indicate limits in automation.

We give definitions and background information on logic- and graph-based information-flow analysis underlying the IFC tools JOANA and KeY in Section 2.

In Section 3, we present our combined approach and its guaranteed properties. The implementation including the specification generation and heuristics for an efficient selection of proof obligations is described in Section 4, and evaluated on a number of case studies in Section 5. In Section 6, we discuss related work, and finally conclude in Section 7.

2 Non-Interference

In order to prevent sensitive information from leaking to unauthorized parties, we analyze programs for the absence of illegal information flow. If such an analysis succeeds, we have shown that this program maintains confidentiality of the specified sensitive information with respect to the specified unauthorized parties. In general, the situation can be more complex, and not only two, but a multitude of different sensitivity levels may exist as, e.g., already expressed in the Bell-LaPadula security model [4] establishing access control mechanisms, and extended by the lattice-model thereby establishing a formal notion of information flow [8]. This emerged to more general techniques, denoted by information flow control (IFC), which check that no secret input channel of a given program may influence what is passed to a public output channel [10].

In its simplest form, i.e., there is no information flow, this is known as *non-interference*. With this generalized notion, it suffices to regard only two security or confidentiality levels, in the following referred to as *high* confidentiality and *low* confidentiality, as we abstract from the leakage itself, and instead analyze the program's potential for information leakage from a specified (information) *source* to a specified (information) *sink*. A sink specifies the (program) location, where an unauthorized party may be able to *observe* the potentially secret information, i.e., we call this a publicly observable location. We hence want to check that any two different executions of a program P with different secret inputs (i.e., coming from sources specified as high) i_h, i'_h and the same public input (i.e., coming from sources specified as low) i_ℓ must be indistinguishable in their publicly-observable output (i.e., sinks specified as low sinks). Note that this is stronger than dynamically searching for illegal flows during run-time, as when proven to be non-interferent, no illegal information flow is possible for *any* execution of the program. A given security lattice can hence also be specified as pairs of sources and sinks.

Within this work, we examine and make use of two different language-based types of techniques for IFC [21], namely dependency-graph-based techniques transforming the program into a graph and hence performing specialized graph traversal algorithms [12], as well as logic-based techniques based on a deductive theorem-prover approach symbolically executing the program twice and hence performing a logic-based calculus on the composition of both executions [6]. One main difference is that the dependency-graph analysis is done on the whole system, and the self-composition is done modularly for each involved method, allowing for reasoning about the whole system based on specialized method contracts. Note that we focus on techniques operating directly on either the program's source or byte code without the need for any manual program modifications. Language-based approaches, in our sense, refer to IFC techniques considering potential attackers being able to evaluate expressions, but not able to observe changes in the memory directly. Furthermore, we only consider deterministic sequential programs and do not regard concurrent flows.

2.1 SDG-based Approaches

JOANA is a tool for checking the non-interference property for a given program. It builds a system dependency graph (SDG) from the program code. A formal definition of an SDG is given in [9]. The nodes represent statements and the edges represent dependencies between those statements. JOANA is able to detect direct dependencies, which are also called data dependencies [11], and indirect or control dependencies [11]. Furthermore, there are special nodes, e.g., for method calls, field accesses and exceptions.

For a method we have special formal-in nodes and formal-out nodes. Formal-in nodes represent all direct inputs that influence the method execution. These are the input parameters, used fields, other classes that are called during execution and the class in which the method is executed. The formal-out nodes represent the influence of the method. In most cases the 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

For function `f` in Listing 1, we would 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, we have actual-in nodes representing the arguments and actual-out nodes representing the returned 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. For the call in Listing 1, there are actual-in nodes for `a` and `b`, corresponding to the formal-in nodes of `f` for `x` and `y`, respectively. We also have one actual-out node representing the return value of `f`, which corresponds to the single formal-out node of `f`. For every method call we also have so called *summary edges* [9] in the SDG from any actual-in node to any actual-out node of the method whenever the tool finds a flow between the corresponding formal-in to the formal-out node of the called method. In Listing 1, we have a flow in `f` from `x` to the result, so a summary edge is inserted at the call site, namely from the actual-in node representing `a` to the single actual-out node. For a complex method there can be a huge number of actual-in and actual-out nodes and therefore an even greater number of summary edges. For our combined approach, we focus on summary edges that belong to a *chop* between high and low and it thus is sufficient to regard only a smaller subset of these edges. A chop from a node s to a node t consists of all nodes on paths from s to t in the SDG. It is commonly computed by intersecting the backward slice for t with the forward slice for s. An example of an SDG generated from a program is given in Giffhorn's thesis on page 18 [9].

Through graph analysis, namely through *slicing* and *chopping* on a syntactic level, [11] JOANA is able to detect an information flow. As with KeY, there are some specifications required. But in comparison to KeY, these are rather lightweight. The user must annotate which variables contain secure (high) or public (low) information. After these annotations have been made, JOANA can run the information flow analysis automatically. If the analysis returns that there is no illegal information flow, JOANA guarantees that the program is secure.

Before we give specific property definitions, we introduce the relation *low-equivalent* ($\sim_L$) for the term *state*. We base our definitions on Wasserab's thesis [24]. A state s is a program state, consisting of variable values and storage locations. We assume that the input of a program is included in the initial state and the output of a program is included in the final state. Two states s, s' are low-equivalent if all low variables have the same value.

We only regard sequential programs here. Thus, we want to prove a property called *sequential non-interference* or *classical non-interference* as shown in Definition 1 [24]. If for a sequential program, JOANA returns that there is no illegal information flow, sequential non-interference holds for that program [24]. Note that this definition is equivalent to Definition 4 and hence also guaranteed by non-interference proofs done with KeY.

Definition 1 (Sequential non-interference (SNI)). *Let P be a program. Let s, s' be initial program states, let $[\![P]\!](s), [\![P]\!](s')$ be the final states after executing P in state s resp. s'. Non-interference holds iff*

$$s \sim_L s' \Rightarrow [\![P]\!](s) \sim_L [\![P]\!](s')\,.$$

JOANA guarantees that for a program P it finds secure, if two initial states are low-equivalent then the final states, after executing P from each of the two initial states independently, are also low-equivalent. In case SNI is violated, JOANA generates at least one violation, and can calculate the respective violation chops as well.

2.2 Logic-based Approaches

When attempting to prove non-interference with respect to existing software programs, precision can only be attained by taking functional properties into account. For example, a program such as "`low = high * 0;`" can only be proven to be secure with knowledge about the functionality of `*`. Similarly, for proving non-interference of the program "`if (high) low = f1(); else low = f2();`", we need to verify that `f1` and `f2` compute the same value.

We start with the standard dynamic logic definition from [6], which defines non-interference as a problem of value independence (Definition 2). Dynamic program logics allows to reason about the program `P` as well as program variables `h` of high confidentiality, and `l` of low confidentiality. The predicate $\doteq$ is to be evaluated in the post-state of `P`.

Definition 2 (Non-Interference as value independence). *When starting* `P` *with arbitrary values* `l`*, then the value* r *of* `l` *– after executing* `P` *– is independent of the choice of* `h`*.*

$$\forall \mathtt{l}\ \exists r\ \forall \mathtt{h}\ \langle P \rangle\ r \doteq \mathtt{l}$$

Non-interference verification using self-composition. Amtoft et al. introduced an approach based on a Hoare-style logic, which formalizes non-interference as an "indistinguishability" relation on program states [2]. As such, the foremost *functional* verification task now becomes *relational* by comparing two runs of the same program, performed by a technique called *self-composition* as proposed, e.g., in [3,6]. Furthermore, we can abstract from a concrete location and instead talk about location sets. Based on the notion of *low-equivalence* as in Definition 3, we obtain the notion given in Definition 4, where low-equivalence refers to identity on all low variables [6,22]. Note that by self-composing the program to two instances, we got rid of the existential quantifier, thereby enabling automatic verification techniques as we avoid the difficult quantifier instantiation.

Definition 3 (Low-equivalence). *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*).*

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

Definition 4 (Non-Interference as self-composition). *Let* P *be a program and* L_1, L_2 *two sets of low variables. Then starting two instances of* P *in two arbitrary low-equivalent states (on arbitrary high values however) results in two final states that are also low-equivalent.*

$$s_1 \simeq_{L_1} s_1' \Rightarrow [P]s_2 \simeq_{L_2} [P]s_2'$$

These findings were extended by a fully compositional information-flow calculus for Java based on a deductive theorem prover for functional program verification [1,22]. It deals with object-oriented software by allowing for two different semantics, distinguishing on whether object creation is low-observable or not. For the new semantics, we assume that references are opaque, in particular object comparison can only be done via the operator `==`. Furthermore, we assume isomorphisms π_i on objects such that π_1 and π_2 are compatible, i.e., for an object o, $\pi_1(o) = \pi_2(o)$ holds if o is observable in both states s_1 and s_2. Then, low-equivalence can be generalized by Definition 5.

Definition 5 (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 calculus and means for specification are implemented in the KeY system [1]. KeY is a deductive theorem prover for Java programs based on JavaDL, a first-order dynamic logic for Java, which allows to reason directly about Java programs on a language-level with an explicit heap variable and changes to the program state translated into so-called *updates* operating on the heap. Thereby, the program can be symbolically executed directly in the logic.

In JavaDL, we can express non-interference based on Definition 4 using heap variables within update operations as given in Definition 6. The updates as a means to change program states are denoted by curly braces.

Definition 6 (Non-Interference as self-composition in JavaDL).

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

The postcondition can be weakened by only proving the variables to be equal up to isomorphism.

The KeY system proves non-interference or other program properties modularly on the program code on Java method level, specified using method contracts as well as auxiliary specifications such as loop invariants inside the method by the modelling language JML*. The formulation of these specifications always depends on the outcome to be proven and describes, e.g., the non-interference property of the program. After the specification is complete, KeY transforms it into equivalent formulas in Dynamic Logic and performs a proof using the sequent calculus. In general, the problem is undecidable and verification sometimes requires some user-interaction. KeY is capable of verifying non-interference for Java programs and covers a wide range of Java features. Proofs are constructed in a precise manner based on a deductive rule base with the possibility of inspecting the proof tree later-on.

Non-interference specification. Information-flow properties are specified in KeY using an extension of the Java Modeling Language (JML) [17], thereby introducing special *determines* clauses for expressing a fine-grained information flow control [22]. These constructs can be used for modular specifications on the method level as well as for enhancing loop invariants for the self-composition of loop statements and block contracts for the self-composition of arbitrarily chosen blocks of statements enclosed by curly braces.

The central specification elements for IFC purposes consist of the two keywords `determines` and `\by` both followed by a comma-separated list of JML expressions. The `determines` clause states that the JML expressions found after the `determines` keyword depend *only* on the JML expressions found after the `\by` keyword. This can furthermore be followed by the keyword `\new_objects` for specifying fresh objects to be included in the isomorphism. With this toolkit, powerful specification elements are given for proving non-interference, also allowing for declassification.

3 The Combined Approach

In this section, we present our approach that combines the advantages of precise logic-based approaches based on theorem provers, such as KeY, and automatic SDG-based approaches using graph-traversal algorithms, such as JOANA. We argue that our approach gurantees the SNI property for a given program and specified sources and sinks.

In the following, we describe our combined approach on the example of proving non-interference for a given program P. In Section 2.1, we established that SDG-based IFC techniques can detect any illegal information flow. Hence, if the SDG-analysis indicates that there is no illegal information flow for the program P, we need no further action as it is guaranteed that non-interference holds. The combined approach is used in case the automatic SDG-based approach detects an illegal information flow and we want to check whether this information flow is a false positive or a genuine leak.

For the information flow check, we first create a system-dependency graph (SDG) as defined in [9]. The created SDG is over-approximated and thus may contain edges which do not represent an actual flow in the program, hence potentially leading to false positives. Our approach assumes that the SDG nodes corresponding to high inputs and low outputs are annotated as high and low respectively. Furthermore, N_h denotes the set of all nodes annotated as high, and N_ℓ the set of all nodes annotated as low. There is an illegal information flow if information may flow from a node that is annotated as high to a node that is annotated as low. If any set of N_h or N_ℓ is empty, there is no illegal information flow.

After the SDG has been annotated by the user, the automated tool runs an information flow check. This check returns a set of *violations.* A violation is a pair (n_h, n_ℓ) of a high node $n_h \in N_h$ (secret source) and a low node $n_\ell \in N_\ell$ (public sink) such that there is a path from n_h to n_ℓ. 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. The set of all violation chops is denoted by C_V. If this set is empty, the SDG-based approach guarantees non-interference, independently from our approach. 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 try to prove it does not exist on the semantic level in program P. We show this by verifying each chop to be interrupted (see Definition 7) with the help of a theorem prover.

Definition 7 (Unnecessary summary edge, Interrupted violation chop). *A summary edge $e = (a_i, a_o)$ is called unnecessary if we can prove with a theorem prover that, in the context of the SDG, there is no 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. This proof obligation states that there is no information flow from f_i to f_o. The proof is done for the method corresponding to the summary edge e and is generally done for all possible contexts. Additionally, results from software analyses done by the SDG-based approach (e.g., points-to analysis) are used to generate a precondition for the analyzed method thus increasing the likelihood of showing non-interference for that method and interrupting the violation chop.

Our approach 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. Our non-interference transformation directly converts the summary edge information to a specification for KeY. If the proof is successful, the summary edge can then be deleted from the SDG based on Definition 7.

Note that this is possible as KeY's (object-sensitive) non-interference property is at least as strict as SNI (Definition 1). This however only holds without the opaqueness assumption, i.e., only for KeY's standard non-interference semantics based on Definition 3 and not Definition 5. If this obligation is chosen, low-equivalence of states from Definition 1 matches low-equivalence of heap locations from Definition 6. In conclusion, we can state that KeY's non-interference property is equivalent to SNI (Definition 1). This implies Theorem 1.

Theorem 1 (Non-Interference Combined Approach). *The combined approach guarantees sequential non-interference.*

We then check whether this violation chop is interrupted, in which case we can proceed to analyse the remaining violation chops until all of them are interrupted. If the violation chop is still not interrupted, or in case 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 non-interference.

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.

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. We hence developed a number of heuristics for choosing the order in which the edges are checked by the theorem prover.

4 Implementation

We implemented the combined approach using JOANA as the dependency-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 describe the heuristics choosing the summary edges that are to be analyzed by KeY.

4.1 Method Contracts

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 dependency between the formal in and formal out node of the summary edge.

Thus, 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 8.

Definition 8 (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 dependency between a_i and a_o the summary edge can be safely deleted from the violation chop.

In order to avoid some false positives, 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 9, thus transferring information about the context from JOANA to KeY and increasing the likelihood of a successful proof.

Definition 9 (Generation of preconditions). *Let o be a reference and P_o its points-to set. We generate the following precondition:* $\bigvee_{o' \in P_o} o = o'$

4.2 Loop Invariants

In Section 3, we stated that the proof with the theorem prover can cost a lot of time- and user-effort. The theorem prover needs auxiliary specification like loop invariants or frame conditions. The method contracts generated as described in the previous section 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 [15, 19]. These approaches focus on functional loop-invariants and do not consider information flow loop-invariants.

The determines clause, described in the previous section, 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 10 shows what determines clauses are generated for loops invariants:

Definition 10 (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 analysing the SDG.

4.3 Heuristics

The order in which the summary edges of in the violation chops are checked determine the performance of the combined approach. Ideally we would 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.

A first category of heuristics searches the code for three patterns that are likely to cause false positives by the SDG-based tool . The first pattern focuses on the problem of array-handling. The tool considers the array to be one syntactical construct and ignores the indexes. Thus, for Listing 2, tools like JOANA would detect an information flow from `high` to the *return value*. Thus we consider methods containing array accesses to be more likely to cause a false positives and assign a higher priority to them.

The second pattern in Listing 3 considers infeasible path conditions. Through purely syntactical slicing, it is not possible to detect that there cannot be an illegal information flow in the example below. The current implementation finds simple excluding statements, like “`x < = 0`” and “`x > 0`”. While the heuristic itself does not check wheter a method contains infeasible paths it does assign a higher priority to methods containing complex path conditions.

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. Furthermore, the method should have as few as possible references to other classes and methods.

A third category of heuristics tries to identify the methods that, if proven non-interferent, would bring the greatest benefit to the goal of proving the entire program non-interferent. 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 [5].

In the case that there is no bridge, we prefer the method with the highest number of connections i.e. the most often called method.

5 Evaluation

The evaluation is two-parted. First the combined approach was evaluated based on examples that generate false positives for SDG-based tools like JOANA. Second, we applied the combined on a case study based on a simple e-voting system. The simple e-voting system was taken from the information flow examples of the KeY system. Both evaluations were tested on a standard PC (Core i7 2.6GHz, 8GB RAM) and outline the advantages and limitations of the combined approach compared to the state-of-the-art.

5.1 List of Examples

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.

```java
int[] array = new int[2];
array[0] = high;
array[1] = 3;
return array[1];
```

Listing 2. Array-handling

```
if (x > 0){ y = high; }
if (x <= 0){ low = y; }
return low;
```

Listing 3. Excluding statements

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. On a syntactical level there is dependency from high to low 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 non-interference 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* describes the problem described in Section 4.3, it represents the handling of data structures. 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 it all.

Table 1. List of examples

	Automatic Mode			*Interactive Mode*	
Program	**Provable**	**KeY Calls**	**Time**	**Provable**	**KeY Calls**
Individual Methods					
Identity	Yes	1	5 sec.	Yes	1
Precondition	Yes	1	5 sec.	Yes	1
Excluding Statements	Yes	1	5 sec.	Yes	1
Loop Override	No	1	7 sec.	Yes	1
Array Access	Yes	1	6 sec.	Yes	1
Whole Programs					
KeY example	Yes	1	7 sec.	Yes	1
Single Flow	Yes	1	6 sec.	Yes	1
Branching	Yes	2	10 sec.	Yes	2
Nested Methods	Yes	2	10 sec.	Yes	2
Mixture	Yes	4	19 sec.	Yes	3
Mixture with Loops	No	7	20 sec.	Yes	5

The example programs are in the scope of 5 to 30 lines of code. They show that the combined approach can prove programs automatically for which JOANA would generate false positives and KeY would require a significant amount of user interaction.

5.2 Case Study - E-Voting

In addition to the outlined examples a small case study has been conducted. The code used in this case study is attached in Appendix A (Listing 5) and represents a simple implementation of a voting system. The vote of every voter is read and sent over a simulated network. If the read vote is not valid, then 0 is sent instead to indicate abstention. The votes itself and whether a vote is valid is secret. All variables starting with `low` (e. g. `low_sendSuccessful`) are annotated as low and the `high_inputstream` is annotated as high.

In the first step of our combined approach, we use JOANA to analyze the program code based on the mentioned annotations. The SDG-based approach finds 14 violations. All these violations are false positives that occur due to the over-approximation of the SDG. Specifically, they occur because the condition `isValid(high_vote)`, which is high, controls which assignment to `low_sendSuccessful` is executed, so JOANA assumes that this variable depends on a secret input. In reality, the values assigned to `low_sendSuccessful` do not depend on which branch is taken, since they only depend on `low_outputStreamAvailable`, which remains constant during a fixed execution. All violations are different chops from the high input stream `high_inputstream` to the different low output streams at different locations, including one exception that can be thrown when assigning `low_numOfVotes` to a value.

The combined approach tries to validate these chops bottom up and interrupt them if possible. First the heuristic looks for smaller methods like `sendVote(int x)`, `inputVote()` and `isValid(int high_vote)` but all three of them are not secure in regards to our specification and thus cannot be proven secure with the KeY system. The approach then looks at the top-level method `secure_voting()`.

For the method given in Listing 4 our approach is able to generate most of the specifications automatically. The JML method contract can be generated automatically. For the loop-invariant, the current approach is not able to specify functional properties such as the frame condition or the term that is decreasing every loop-run. In Listing 4, the boxed commands must be added manually for a sufficient loop-specification. For the KeY to automatically prove the method contract two *block-contracts* are necessary. These are auxiliary specifications for a group of statements that provide supplementary information to the prover. For the method `secure_voting()` the information flow specification has to be copied manually for both if-blocks as shown in Appendix A.

This specification can then be proven with KeY for the first violation. All other 13 violations are running through the same top-level method and thus are satisfied by the same proof. Thus, we showed that our combined approach can automatically find the causes of false positives by the SDG-based tool and generate the necessary proof obligations in order to disprove the reported leaks.

```
/*@ normal_behavior
  @     determines low_outputStream, low_outputStreamAvailable,
  @                low_NUM_OF_VOTERS, low_numOfVotes,
  @                low_sendSuccessful \by \itself;
  @*/
void secure_voting() {
    /*@ loop_invariant  0 <= i && i <= low_NUM_OF_VOTERS;
      @  loop_invariant \invariant_for(this);
      @ determines low_outputStream, low_outputStreamAvailable,
      @            low_NUM_OF_VOTERS, low_numOfVotes,
      @            low_sendSuccessful, i \by \itself;
      @ decreases  low_NUM_OF_VOTERS - i;
      @*/
    for (int i = 0; i < low_NUM_OF_VOTERS; i++) {
        ...
    }
    publishVoterParticipation();
}
```

Listing 4. Method `secure_voting()` with loop invariant

6 Related Work

There exist many different approaches for proving non-interference. In Section 2.2 and Section 2.1 we have introduced logic-based and SDG-based approaches. In addition, we will discuss two other possibilities for proving non-interference.

6.1 The Hybrid Approach

The hybrid approach by Küsters et al. [16] is the work most related to our approach. It combines the same type of tools, i.e. an automatic dependency-graph analysis with a theorem prover, in an attempt to prove non-interference for a given program with minimized user-effort. The hybrid approach attempts to show non-interference with the dependency-graph analysis tool first. If the attempt does not succeed, the user must identify the possible cause of the false positive and extend the program such that the affected low output is overwritten with a value that does not depend on the high inputs. The extension is not allowed to change the state of the original program, it is allowed to use an extended state and is only allowed to read and overwrite variables from the original program. The extended program must be shown to be non-interferent with the dependency-graph analysis tool. In the next step, the theorem prover is used to show that the extended program is equivalent to the original program (modulo the extended state).

Similarly to our approach, the SDG-based tool is called first and if it does not prove non-interference, further action is taken. Unlike our case, the user has to analyze the program and the output of the SDG-based tool carefully in order to find out whether the reported flow is a false positive or not. The user then has to extend the program such that the low output is overwritten with a value such that the SDG-based tool successfully shows non-interference and then use the theorem prover to prove that the extended program is equivalent to the original one. In our approach, the interaction between the SDG-based tool and the theorem prover is automatic, the user needs to provide functional auxiliary specification when necessary.

6.2 Path Conditions

Path conditions [13] are another example of how SDG-based tools can be combined with more precise approaches, in this case constraints solvers, to increase precision. For a violation found by such an analysis, a path condition is a necessary condition that an information flow exists from the source to the sink of the violation. Path conditions can be computed automatically. In the program "`int y = high; if (x < 0) low = y;`", the path condition for an information flow from `high` to `low` would be `x < 0`. A constraint solver can then be used to generate a satisfying assignment for the path condition, which is a potential witness for an illegal flow. If the path condition is not satisfiable, then one can conclude that the violation was a false alarm.

Thus SDG-based non-interference analysis can be improved by path conditions. But it is important to state that the generation of path conditions is non-trivial [13]. The generation and checking of path conditions is not feasible for huge programs and is therefore not fully included in SDG-based tools like JOANA.

6.3 Type Systems

A well established technique is the information flow analysis based on security type systems. Type systems usually use syntactic rules to assign security types, typically low and high, to expressions and statements of a given program. If the program is typeable, the non-interference property holds. Examples of such a security type system are given in [21] or in [23].

The advantages of security type systems is that there is a clear separation between the rules and the concrete program execution. Furthermore, soundness proofs and the verification of a program with type systems are very fast. The disadvantages on the other hand are possible false positives and limitations of type systems. Most type systems are neither flow-, context- nor object sensitive, which degrades precision. Also, there exist languages like *separation logic* for which there is no known type system available.

7 Conclusion and Future Work

In this paper we introduced a new combined approach to prove non-interference 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 non-interference 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. Thereby, our evaluation demonstrates promising results for complex programs and we are confident that much bigger programs are in reach.

An extended case study, covering programs too big to be checked by a theorem prover alone, is planned. For future work, the heuristics can be improved by integrating an SMT solver in order to enhance the recognition of excluding statements or further excluding program structures. The user-effort of the approach can be further minimized by automating the generation of functional loop invariants. Furthermore, the approach itself can be extended to also cover non-sequential programs and declassification.

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, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
2. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3148, pp. 100–115. Springer (2004)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE. pp. 100–114. IEEE (2004)
4. Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations. Tech. rep., DTIC Document (1973)
5. Bollobás, B.: Modern graph theory, vol. 184. Springer Science & Business Media (2013)
6. Darvas, Á., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow, pp. 193–209. Springer (2005)
7. van Delft, B.: Abstraction, objects and information flow analysis. Ph.D. thesis, Chalmers University of Technology, Goeteborg, Sweden (2011)
8. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
9. Giffhorn, D.: Slicing of Concurrent Programs and its Application to Information Flow Control. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)

10. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984. pp. 75–87. IEEE Computer Society (1984)
11. Graf, J., Hecker, M., Mohr, M.: Using joana for information flow control in Java programs-a practical guide. In: Software Engineering (Workshops). pp. 123–138 (2013)
12. Hammer, C.: Experiences with pdg-based IFC. In: Massacci, F., Wallach, D.S., Zannone, N. (eds.) Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5965, pp. 44–60. Springer (2010)
13. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: IEEE International Symposium on Secure Software Engineering. pp. 87–96 (2006)
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 8(6), 399–422 (2009)
15. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2006)
16. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. Proceedings of the Computer Security Foundations Workshop 2015-Septe, 305–319 (2015)
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
18. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. 9(4), 410–442 (2000)
19. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443–476 (2007)
20. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE pp. 214–227 (1999)
21. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2003)
22. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: International Conference on Formal Verification of Object-Oriented Software. pp. 232–249. Springer (2011)
23. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of computer security 4(2-3), 167–187 (1996)
24. Wasserrab, D.: From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (Oct 2010), http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678

A Source Code for E-Voting Case Study

```
/**
 * Information flow example.
```

```java
 * The example is a toy implementation of a voting process. The vote of
 * every voter is read and sent over a not further modelled network. If
 * the read vote is not valid, then 0 is sent instead to indicate
 * abstention. The votes itself and whether a vote is valid is secret.
 * At the end the participation is output.
 * Without the optimizations described in the FM-Paper the verification
 * of the method secure_voting() with the help of self-composition is
 * very expensive or even infeasible.
 *
 * @author Christoph Scheben
 */
public class Voter {
    public static int low_outputStream;
    public static boolean low_outputStreamAvailable;
    private static int high_inputStream;

    public static final int low_NUM_OF_VOTERS = 763;
    public static int low_numOfVotes;
    public boolean low_sendSuccessful;

    private boolean high_voteValid;

    public static void main(String[] args) {
        Voter v = new Voter();
        v.secure_voting();
    }

    /*@ normal_behavior
      @     determines low_outputStream, low_outputStreamAvailable,
      @                low_NUM_OF_VOTERS, low_numOfVotes,
      @                low_sendSuccessful \by \itself;
      @*/
    void secure_voting() {
        /*@ loop_invariant 0 <= i && i <= low_NUM_OF_VOTERS
          @             && \invariant_for(this);
          @ determines low_outputStream, low_outputStreamAvailable,
          @            low_NUM_OF_VOTERS, low_numOfVotes,
          @            low_sendSuccessful, i \by \itself;
          @ decreases low_NUM_OF_VOTERS - i;
          @*/
        for (int i = 0; i < low_NUM_OF_VOTERS; i++) {
            int high_vote = inputVote();
            /*@ normal_behavior
              @     determines low_outputStream,
              @                low_outputStreamAvailable,
              @                low_NUM_OF_VOTERS,
              @                low_numOfVotes,
              @                low_sendSuccessful \by \itself;
              @*/
            {
                if (isValid(high_vote)) {
                    high_voteValid = true;
                    low_sendSuccessful = sendVote(high_vote);
                } else {
                    high_voteValid = false;
                    low_sendSuccessful = sendVote(0);
                }
            }
            /*@ normal_behavior
              @     determines low_outputStream,
              @                low_outputStreamAvailable,
              @                low_NUM_OF_VOTERS, low_numOfVotes,
              @                low_sendSuccessful \by \itself;
              @*/
            {
                low_numOfVotes =
                    (low_sendSuccessful ?
                        low_numOfVotes + 1 : low_numOfVotes);
             }
        }
        publishVoterParticipation();
    }

    int inputVote() {
        return high_inputStream;
    }

    boolean sendVote(int x) {
        if (low_outputStreamAvailable) {
            // encrypt and send over some channel
            // (not further modeled here)
            return true;
        } else {
            return false;
        }
    }

    boolean isValid(int high_vote) {
        // vote has to be in range 1..255
        return 0 < high_vote && high_vote <= 255;
    }

    void publishVoterParticipation() {
        low_outputStream =
            low_numOfVotes * 100 / low_NUM_OF_VOTERS;
    }
}
```

Listing 5. Source Code