Integration of Static and Dynamic Analysis Techniques for Checking Noninterference

Bernhard Beckert[1], Mihai Herda[1][0000−0002−0142−1718], Michael Kirsten[1*][0000−0001−9816−1504], and Shmuel Tyszberowicz[2]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] Afeka Academic College of Engineering, Tel Aviv, Israel
{beckert, herda, kirsten}@kit.edu, tyshbe@tau.ac.il

Abstract. In this article, we present an overview of recent combinations of deductive program verification and automatic test generation on the one hand and static analysis on the other hand, with the goal of checking noninterference. Noninterference is the non-functional property that certain confidential information cannot leak to certain public output, i.e., the confidentiality of that information is always preserved.
We define the noninterference properties that are checked along with the individual approaches that we use in different combinations. In one use case, our framework for checking noninterference employs deductive verification to automatically generate tests for noninterference violations with an improved test coverage. In another use case, the framework provides two combinations of deductive verification with static analysis based on system dependence graphs to prove noninterference, thereby reducing the effort for deductive verification.

Keywords: Information-Flow Security · Deductive Verification · Software Testing · Program Slicing

1 Introduction

Generally, software developers primarily focus on functional requirements, even though non-functional requirements should also be at the center of attention. One paramount non-functional requirement is confidentiality. In this article, we target the preservation of confidentiality by the requirement that illegal *information flow* shall be avoided, i.e., we want to prevent situations where *high* (confidential) input leaks to *low* (public) output. This property is known as *noninterference* [14]. Intuitively, it requires that high input cannot interfere with low output. Thus, by observing the program's output, one cannot distinguish between different high inputs, i.e., if a program is executed twice with different high inputs but identical low inputs, then an attacker will observe identical behaviors (an attacker can observe low but not high information). Various approaches and tools for checking noninterference exist. Some have a high degree of automation,

* Corresponding author

yet produce many false alarms due to an over-approximation of the possible flows of information in the program. Others are more precise, but require more effort and user interaction. Approaches that are based on System Dependence Graphs (SDGs) syntactically compute the dependencies between the program's statements and check whether the low output potentially depends on the high input (see, e.g., Hammer and Snelting [18]). Whereas such approaches scale very well, they over-approximate the actual dependencies in a program, and may yield false alarms. *Logic-based* approaches (see, e.g., Beckert et al. [7] and Greiner and Scheben [32]) have a higher precision (i.e., they produce less false alarms), since they more precisely analyze the concrete semantics of the program statements. However, they have a lower scalability. In these approaches, one tries to formally prove that the terminating states of two arbitrary executions of the same program are low-equivalent (i.e., the low parts of the states have the same values), assuming that the 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. Since verifying noninterference using logic-based approaches requires to compare two program executions simultaneously, a quadratic number of program paths must be considered compared to functional verification. Approaches for *test generation* also require a quadratic number of test cases to achieve the same coverage as for a functional property.

In this article, we present an overview of recent combinations [6, 21, 22] of deductive program verification with automatic test generation on the one hand and static analysis on the other hand, with the goal of checking noninterference. The resulting *Noninterference Framework* can be used both for proving that a given program fulfills a given noninterference property and, also, for finding a counterexample that demonstrates a noninterference violation in case the noninterference property is not fulfilled. For the task of proving noninterference, the framework combines deductive program verification with static analysis, two complementary approaches with respect to precision and scalability, and thereby reduces the effort for the deductive verification. The task of finding counterexamples is achieved by combining deductive program verification with automatic test generation in order to improve the computation of the achieved test coverage. For a more extensive framework description, we refer to Herda [20].

The rest of this article is structured as follows. In Section 2, which is based on Herda et al. [21], we define the noninterference properties that are handled with the framework. In Section 3, which is based on Ahrendt et al. [4], Beckert et al. [6], and works by Herda et al. [21, 22], we present the used approaches. We provide an overview of the framework in Section 4, which is based on Herda [20]. Section 5, which is based on Herda et al. [21], presents an approach which uses deductive verification for automatic test generation. In Sections 6 and 7, which are based on Herda et al. [22] and Beckert et al. [6] respectively, we consider two approaches which combine SDG-based and logic-based approaches for proving noninterference. We discuss implementation aspects of the two combinations in Section 8, which is based on previous works [6, 20]. In Section 9, we present related work. Finally, we conclude in Section 10.

2 Information Flow Security

In this section, which is based on Herda et al. [21], we formally define the noninterference properties that can be checked by the Noninterference Framework. In this article, we consider only sequential and terminating programs. We introduce the *low-equivalence* relation $\sim_L$ that characterizes program states that are indistinguishable for an attacker with respect to a set L of low variables, where a program state s is an assignment of values to variables. We assume that the input of a program is included in the program's prestate and that the output of a program is part of the program's poststate. Hence, we define low-equivalence (Definition 1) and thereby noninterference (Definition 2) as follows:

Definition 1 (Low-equivalent states). *Two states s_1, s_2 are low-equivalent with respect to the set L of all low variables if and only if they assign the same values to low variables:*

$$s_1 \sim_L s_2 \Leftrightarrow \forall\, v \in L\ (v^{s_1} = v^{s_2})\ ,$$

where v^{s_i} denotes the value of the variable v evaluated in state s_i.

Definition 2 (Classical noninterference). *A program P is noninterferent if and only if, for any two initial states s_1 and s_2, the following holds:*

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

where s_1', s_2' are poststates after executing P in s_1 and s_2, respectively.

The classical noninterference property, as presented in Definition 2, requires that any two executions of the program that start in two states which are indistinguishable for the attacker will also terminate in two indistinguishable states. If this holds, then it is guaranteed that the high values of the prestates cannot influence the low values of the poststates.

This property, however, is often too strong for cases where it is acceptable that the attacker sees parts of the high values. A classical example is a login system in which an attacker can try out different combinations of user names and passwords. While the system does not immediately leak the user's password, an attacker can check whether particular combinations are correct, and thus obtain information about sensitive data. To allow such a case, albeit still forbidding cases in which the system outright leaks sensitive information to the attacker, we define (see Definition 3) the notion of noninterference with declassification (i.e., giving the attacker access to parts of the high information). For this, let *expr* be an expression in first order logic describing the high information that an attacker is allowed to know (we denote *expr*'s evaluation in a state s by $expr_s$).

Definition 3 (Noninterference with declassification). *Given a declassification expression expr, a program P is noninterferent if and only if we have for all initial states s_1, s_2 that*

$$s_1 \sim_L s_2\ \wedge\ expr_{s_1} = expr_{s_2} \Rightarrow s_1' \sim_L s_2'\ ,$$

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

For object-oriented programs, it is too restrictive to require that all low variables and heap locations are equal for the final states to be low-equivalent. Consider the case of programs that create new object references: a program will not necessarily create the same reference for different executions, even for exactly the same input. Beckert et al. [7] have developed a variation (Definition 4) of classical noninterference using a semantics that is based on an object isomorphism.

Definition 4 (Low-equivalence with isomorphism). *Two states s_1, s_2 are low-equivalent if and only if*

$$s_1 \sim_L^{\pi} s_2 \Leftrightarrow \forall\, v \in L\; (\pi(v^{s_1}) = v^{s_2})\ ,$$

where π is a bijective function on heap locations.

We assume that an attacker cannot see the exact reference address of an object and can compare object references only for equality (e.g., with the Java == operator). Thus, if the object structures in the two poststates are isomorphic, the attacker obtains the same results when comparing the object references for equality. Note that the properties from Definitions 3 and 4 can be combined in order to obtain noninterference with isomorphism and declassification.

3 Approaches of the Noninterference Framework

This section presents the approaches in the Noninterference Framework. We present SDG-based approaches (using Herda et al. [22]) in Section 3.1, logic-based approaches (using previous works [6,21]) in Section 3.2, and automatic test generation based on symbolic execution (using Herda et al. [21]) in Section 3.3.

3.1 SDG-based Approaches

While the concepts of Program Dependence Graphs (PDG) [13] and System Dependence Graphs (SDG) [23] have been developed during the eighties, their usefulness in the context of information flow security has been first noticed by Snelting [34] in the nineties. Without loss of generality, we use the JOANA tool [18] to explain the functionality of an SDG-based analysis. SDG-based information-flow analyses are purely syntactic, highly scalable, and sound. However, some of the reported noninterference violations may be false alarms. The desired noninterference property (as in Definition 2) is specified by annotating which program parts correspond to high information as well as the parts where low output occurs. JOANA automatically builds an SDG from these annotations.

The resulting SDG is a directed graph consisting of interconnected PDGs, where each PDG represents a single program procedure in the form of a directed graph. Nodes in the SDG represent program statements, conditions, or input parameters, and edges represent dependencies between the nodes (i.e., there is an edge between two nodes if and only if the value or execution of one node may depend on the outcome of the other node). Whether there is an edge between two nodes in the SDG, is determined syntactically by analyzing the control-flow graph of the program. There are three main types of edges in an SDG:

1. data dependency edges, which represent possible direct dependencies,
2. control dependencies, which represent possible indirect dependencies, and
3. interprocedural dependencies, which represent dependencies between nodes in different PDGs.

Formal definitions for the three types of dependencies can be found in Hammer [17, Chapter 2]. In the following, we give informal definitions. A node n' is data-dependent on a node n iff 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 Control Flow Graph (CFG) such that v is not redefined on any node between n and n' on that path. A node n' is control-dependent on a node n iff the choice of the outgoing edge from n in the CFG determines whether node n' is reached. Note that it is generally undecidable whether a CFG path represents an actual execution path in the program, i.e., some paths in the CFG may represent executions that cannot actually take place. Hence, the CFG is an over-approximation of the actual program behavior. Since the dependencies are defined using CFG paths, they are also an over-approximation of the actual dependencies in the program.

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 can be input parameters, used fields, other classes called during execution, or the class in which the method is executed. Formal-out nodes represent the influence of the method and can represent the method's return value, global variables, fields in other classes, or exceptions. 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 called method and each actual-out node to a formal-out node. Interprocedural dependencies connect actual-in nodes to the corresponding formal-in nodes, and formal-out nodes to the corresponding actual-out nodes. For every method call, there are also so-called *summary edges* in the SDG from any actual-in to any actual-out node of the method for which the tool finds a possible information flow from the corresponding formal-in to the corresponding formal-out node of the called method.

JOANA detects illegal information-flows through graph analysis, using a special form of conditional reachability analysis, so-called (back- and forward) *slicing* and *chopping*, at the SDG level. A *forward slice* of a node s consists of all SDG-nodes that can be reached from s. Conversely, a *backward slice* of a node s consists of all nodes on 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 first calculating the backward slice for t, and then computing the forward slice for s within the subgraph induced by the backward slice. 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). Wasserrab and Lohner [36] proved that no potential flow of information is missed, i.e., that JOANA is sound. Since the dependencies modeled in the SDG are in fact over-approximations of actual dependencies in the program, the program is guaranteed to be noninterferent if no SDG path

from a high input to a low output is found. However, the program may still be noninterferent, even though there is a path from a high input to a low output.

3.2 Logic-based Approaches

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 becomes important for analysis to deal with sources for illegal information-flow leaks which possibly exploit features of the program semantics. Logic provides a means for formalizing such features, and enables reasoning about their effects on any program variables or locations.

Dynamic logic [8] can express the functional property of *partial correctness* of a program P for a precondition ϕ and a postcondition ψ by the 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. Applying a logical calculus with a deductive theorem prover (e.g., KeY [3]), we can symbolically execute P and attempt to prove this formula. KeY allows for proofs of functional properties specified with the Java Modeling Language (JML) [27]. Furthermore, KeY has been extended to support the verification of noninterference for sequential programs [31,32]. The classical proof obligation (Definition 2) requires that two program runs that start in low-equivalent states will also terminate in low-equivalent states, i.e., given (simplified) in dynamic logic:

$$\Big(\underbrace{([P]\, out_l \doteq out_l^A)}_{\text{Execution A}} \wedge \underbrace{([P]\, out_l \doteq out_l^B)}_{\text{Execution B}}\Big) \rightarrow \Big(\underbrace{(in_l^A \doteq in_l^B) \rightarrow (out_l^A \doteq out_l^B)}_{\substack{\text{Low-equivalent prestates imply} \\ \text{low-equivalent poststates}}}\Big)$$

In the above proof obligation, the placeholder out_l represents the low output of program P. On the left hand side of the main implication, the proof obligation contains two executions of P: execution A, after which the low output out_l is equal to out_l^A, and execution B, after which the low output is equal to out_l^B. On the right hand side of the main implication, the proof obligation contains an implication stating that, if the two low inputs in_l^A and in_l^B are equal, then the low outputs out_l^A and out_l^B are equal as well. To support declassification as defined in Definition 3, the second implication of the proof obligation requires additionally that the declassified expressions are equal in the prestates of the two executions. To support object isomorphism as defined in Definition 4, the second implication of the proof obligation is enhanced with the predicate *newObjIso*:

$$(in_l^A \doteq in_l^B) \rightarrow \big(newObjIso(N, h^A, h^B) \wedge (N^A \doteq N^B \rightarrow out_l^A \doteq out_l^B)\big)$$

The predicate *newObjIso* accepts as parameters a list N of reference type expressions and the two heaps h^A and h^B that represent the two poststates of executions A and B. The predicate holds under the following three conditions:

1. Every reference expression in N is newly created in the poststates of both executions (A and B) of P.
2. Every reference expression in N has the exact same type in the post states of executions A and B.
3. If two reference expressions in N are equal in the poststate of one execution (A or B), then they must be equal in the poststate of the other execution.

These requirements ensure that the reference expressions in N are isomorphic in both poststates. If the reference expressions in N fulfill the *newObjIso* predicate, the proof obligation no longer requires their equivalence in the two poststates (this relaxation is done with the second implication). Note that the user-provided list N must also be part of the noninterference specification, in addition to the variables that must be low-equivalent before and after both executions.

3.3 Automatic Test Generation

On top of the deductive theorem prover and its calculi for finding functional (or non-functional, see Section 3.2) proofs, KeY was extended with an automatical test generation for functional properties [12]. In this section, which is based on [4], we give an overview of KeYTestGen, the current extension of KeY for automatic test generation. KeYTestGen uses symbolic execution and attempts to generate a test suite that achieves a high path coverage. In the following, we present the three steps taken by KeYTestGen to automatically generate tests.

Step 1: Constraint generation. The input to KeYTestGen is a Java method under test (MUT), together with a specified functional property. KeYTestGen loads the proof obligation for the specified MUT and symbolically executes the program. The Java code is transformed into updates, which are a compact representation of the statements' effects. Case distinctions (including implicit ones such as, e.g., whether or not an exception is thrown) in the program are reflected as branches of the proof tree. The symbolic execution is bounded by a number b provided by the user. Hence, loops in the program are unwound a maximum of b times. Each branch in the obtained proof tree represents a b-path (i.e., a path that goes through each cycle at most b times) in the CFG. At the end of the symbolic execution, a model of a leaf of the tree is both a model of the precondition and of the path condition of the b-path that corresponds to that proof tree branch.

Step 2: Test data generation. For generating a test, we first produce a concrete test input s which satisfies the test data constraint (i.e., a path condition together with the specification's precondition) obtained from the first step. For finding such models, we use the SMT solver Z3 [29]. The constraints from *step 1* are translated from KeY's Java first order logic [33] into the SMT-LIB 2 language [5] to be processed by Z3. The translation uses bounded data types (i.e., each data type only has a bounded number of instances), so that the SMT solver can find models much faster, but potentially misses some models for too small bounds.

Step 3: Code generation. In the third and final step, the JUnit test cases are generated. Each test case consists of a *test preamble*, a call of the MUT, and a call of the test oracle. The test preamble prepares the inputs for the MUT call, which are taken from the model found in the previous step. The MUT call in the test case is the same as in the proof obligation. The generated test oracle is a boolean method that checks the postcondition after the MUT was executed. Hence, each test suite contains only one oracle method for all test cases.

4 The Noninterference Framework

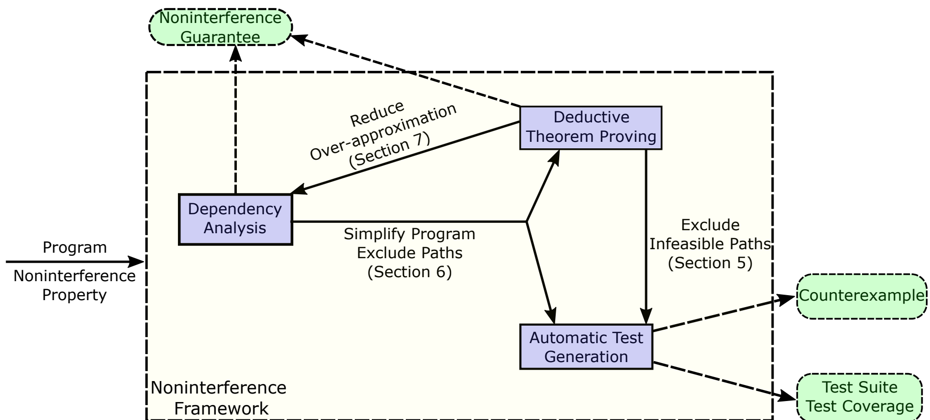

Figure 1: The Noninterference Framework

In this section, which is based on Herda [20], we present a framework (illustrated in Figure 1) for checking noninterference properties. The framework combines multiple contributions (presented in Sections 5 to 7) to the area of information-flow security. We provide a short overview of the framework and show how its individual approaches are integrated and cooperate. The framework encompasses the following three approaches: (1) *dependency analysis*, (2) *deductive theorem proving*, and (3) *automatic test generation*. Each of these methods has its strengths and weaknesses. By integrating them, the framework uses their strengths to mitigate their weaknesses. The framework handles the following two use cases. In the first one, the analyzed program does not fulfill the specified noninterference property, and the user is looking for a counterexample that showcases the noninterference violation. For this case, the framework provides an approach for automatic test generation, aiming to find noninterference violations. In the second case, the specified noninterference property is fulfilled by the analyzed program, and the user attempts to prove this property. For this case, the framework combines deductive theorem proving with dependency analysis. In the following, we give an overview of the contributions for the two use cases.

4.1 Finding Noninterference Counterexamples

For the case in which the user tries to find a noninterference violation, the framework provides an approach to automatically generate tests for noninterference properties, which is presented in Section 5. The approach serves two purposes. First, we can search for counterexamples of the analyzed noninterference property. If such a counterexample is found, the user gets two program inputs that are indistinguishable to the attacker but lead to two different low outputs, thus demonstrating a noninterference violation. Second, we can generate a noninterference test suite that achieves a certain test coverage. This is useful in the event that neither the program could be proved correct nor a counterexample could be found. Then, the user is provided with a test coverage value, which helps to assess the strength of the generated tests. The test generation approach uses deductive program verification to remove infeasible program execution paths from the test generation and from the computation of the achieved coverage. As shown in Section 6, the test generation also uses the results provided by the SDG-based analysis for removing parts of the program which are not relevant to the analyzed noninterference property.

4.2 Proving Noninterference

In case a user tries to prove that a given program fulfills a specified noninterference property, the framework combines two existing approaches: (1) dependency analysis (i.e., an SDG-based approach, see Section 3.1) and (2) deductive theorem proving (i.e., a logic-based approach, see Section 3.2). In order to make use of the advantages of both SDG-based and logic-based approaches, the framework combines these approaches in two ways. In the first combination, as presented in Section 6, the SDG-based approach is used to simplify the noninterference proof obligations for the deductive program verification. Those parts of the program that the SDG-method deems irrelevant to the noninterference property are removed before the program is analyzed with the logic-based approach. In the second combination, as shown in Section 7, the logic-based approach is used to increase the precision of the SDG-based approach by identifying certain SDG dependencies that are over-approximations and removing their corresponding edges from the SDG. Both combinations can be joined, i.e., the program simplification from Section 6 can be applied to simplify the deductive proof obligations for showing that a program dependency is over-approximated.

5 Test Generation for Noninterference Properties

This section presents the part of our Noninterference Framework that handles programs that violate the given noninterference property. We extend the approach for automatic test generation described in Section 3.3 to support the properties defined in Section 2. For a more extensive description of the automatic test generation, we refer to Herda et al. [21], on which this section is

based. In Section 5.1, we show how the properties from Section 2 can be tested. in Section 5.2, we explain how such tests are generated automatically. Finally, we extend existing coverage criteria such that they can be used for the generated noninterference test suites in Section 5.3.

5.1 Noninterference Tests

We begin by defining noninterference tests suitable for testing classical noninterference from Definition 2 by adapting functional tests from Ahrendt et al. [4].

A *noninterference test* can formally be described as a tuple $\langle s_1, s_2, Or \rangle$, consisting of two prestates representing *test inputs* s_1 and s_2, and an *oracle function* Or. The two prestates s_1 and s_2 are required to be low-equivalent according to either Definition 1 or 4. The oracle function $Or(s'_1, s'_2) \mapsto \{pass, fail\}$ checks whether the two poststates are low-equivalent. Thus, in order to perform a noninterference test, we execute the MUT twice, with inputs `in1` and `in2`, respectively (see Listing 1). Moreover, for a valid noninterference test, we require the two generated inputs to be low-equivalent. The oracle function then checks whether the two outputs, `out1` and `out2`, are also low-equivalent.

```
1 testMUT() {
2    in1 = generateInput(); // Execution 1 preamble
3    out1 = executeMUT(in1); // MUT call 1
4    in2 = generateLowEqInput(in1); // Execution 2 preamble
5    out2 = executeMUT(in2); // MUT call 2
6    oracle(out1,out2); // Oracle call #
7 }
```

Listing 1: Structure of a noninterference test

In order to check the noninterference properties from Section 2, we define the semantics for the functions `generateLowEqInput` and `oracle` from Listing 1 as follows: For the noninterference property from Definition 2, the method `generateLowEqInput` generates a second input that is low-equivalent according to the relation $\sim_L$ from Definition 1. The method `oracle` checks whether the two outputs are low-equivalent. For the property provided by Definition 3, `generateLowEqInput` generates a second input such that the two prestates are low-equivalent and, moreover, the results from evaluating the given expression in the two prestates are identical. The oracle function again checks the low-equivalence. Finally, for the property given by Definition 4, `generateLowEqInput` and `oracle` use the low-equivalence relation with isomorphism $\sim_L^\pi$.

5.2 Automatic Test Generation

We describe the approach for generating test suites for a given MUT and a specified noninterference property as defined in Section 2 by extending the approach for automatic test generation from Section 3.3. The main difference to functional properties is that two related program runs are necessary. The test generation approach offers two options that yield the following two scenarios: With the first option, the user searches for noninterference violations. In this

case, the constraints that are passed on to the model generator require the two poststates to be non-low-equivalent (i.e., to demonstrate a noninterference violation). This first option is useful in the event that the user has failed to prove that the program is noninterferent and suspects that the program violates the property. With the second option, the user generates a noninterference test suite with a high test coverage. In this case, the constraints do not restrict the poststates in any way. The second option is useful for finding errors given that both verification and counterexample generation have failed. Moreover, generating a high-coverage test suite is useful for finding violations introduced by the operating system or by the compiler. The coverage of the generated test suite can increase the user's confidence in the correctness of the program. The approach for automatic test generation works in three steps as described in the following.

Step 1: Constraint generation. The MUT specified with a noninterference property is loaded into KeY and symbolically executed (Section 3.3). At the end of the bounded symbolic execution, we obtain a proof tree where each leaf contains a pair of path conditions, one for each program execution. Thus, a model that satisfies the formulas in a leaf is also a model for the two path conditions represented by the leaf, and for the requirements that the prestates are low-equivalent and the poststates are non-low-equivalent. With the second option, the constraint requiring that the poststates violate the low-equivalence requirement is ignored.

Step 2: Test data generation. For test data generation, we can reuse the SMT-translation used to generate tests for functional properties (see Section 3.3). This step can also generate models that fulfill the declassification expressions and thus supports noninterference with declassification (see Definition 3).

Step 3: Code generation. As shown in Listing 1, a noninterference test contains two preambles and two calls of the MUT. For each call, an input configuration is set up and—after the second MUT call—the test oracle is called to decide whether the test was successful or not. For the two preambles, we create two isomorphic input states by duplicating the objects and values from the model. Thereby, we avoid that the second MUT execution affects the first MUT execution's results. The MUT and its surrounding code are taken from the JavaDL modality in the root node of the proof tree. It is important to use the surrounding code—rather than just the MUT's invocation—to ensure semantic equivalence of the actual and the symbolic execution of the code. The surrounding code typically consists of a *try/catch* block which allows the test oracle to decide what to do when an exception is thrown. Each test suite contains only one oracle method that is used for all tests. The oracle checks equality for low variables of primitive type and isomorphism for reference type variables. For references created during the MUT execution, the requirements of *newObjIso* (see Section 3.2) are checked. However, for references created in one of the preambles, only the second and third requirements of *newObjIso* are checked.

5.3 Coverage Criteria

We extend existing test coverage definitions in order to make them suitable for measuring the coverage of noninterference test suites, and we discuss the coverage provided by the test suites generated with our approach. Well-established coverage criteria such as *statement*, *branch*, and *path* coverage are defined on the CFG. Full statement coverage of a test suite requires each node in the CFG to be traversed during the execution of the test suite, whereas full branch coverage requires each edge to be traversed, and for full path coverage, each path must be traversed. In order to extend these coverage criteria to the noninterference test suites defined in this paper, we no longer use the standard CFG, but rather the self-composed CFG, to define the notions of *relational* statement, branch and path coverage. The self-composed CFG consists of two copies of the MUT's CFG, with renamed variables and with a directed edge from the end node of the first copy to the start node of the second copy.

Since a CFG may contain loops, the number of paths is possibly infinite. Hence, we only count b-paths (i.e., paths which pass through each cycle at most b times) on the self-composed CFG, and define the *relational bounded path coverage criterion* as #(covered b-paths) / #(b-paths). Our approach explicitly provides the required values for this coverage criterion: the number of generated tests represents the number of covered b-paths, while the number of leaves of the proof tree represents the total number of b-paths in the self-composed CFG. One problem with the relational bounded path coverage criterion is that typically a large portion of the program execution paths in the self-composed CFG are infeasible (i.e., the path condition for that execution path is unsatisfiable). It is generally undecidable whether a path is infeasible or not. Hence, if our test suite has a low coverage, we cannot know whether the paths for which no test was generated are infeasible or whether, given more time, our approach would be able to generate tests for those paths. For noninterference properties, the low-equivalence requirement for the two inputs may cause certain paths in the self-composed CFG to be infeasible. A more useful criterion is therefore the *relational bounded feasible path coverage*, defined as #(covered b-paths) / #(feasible b-paths) on the self-composed CFG. Even though we cannot compute the exact number of feasible b-paths, we can use the theorem prover to show that certain paths are infeasible, and the remaining paths—that could not be proved infeasible—constitute an over-approximation of the feasible b-paths. This over-approximated number is used to compute an under-approximation of the *relational bounded feasible path coverage* for a generated test suite.

An evaluation [21] done on a collection of benchmarks [16] containing both secure and insecure programs shows that relational bounded feasible path-coverage is an appropriate coverage criterion for noninterference properties: for high coverage values, we either find no violations for the secure programs or find at least one violation for most insecure programs.

6 Simplifying Programs for Testing and Verification

This section presents an approach which, given a program and a noninterference property, uses an SDG-based analysis to generate a simplified version of the original program. The simplified program is noninterference-equivalent (with respect to the given property) to the original program, and can be analyzed with a second, more precise approach. We use the logic-based approach (see Section 3.2) and automatic test generation (see Section 5) as a second approach. Thus, the simplification approach is used by our framework to simplify both searching for counterexamples and proving the property. For a more extensive description of the approach, see Herda et al. [22], on which this section is based. Section 6.1 presents an illustrative example that is also used in Section 7, Section 6.2 explains how simplified programs are generated, and Section 6.3 presents the advantages of the simplified programs for verification and test generation.

6.1 Running Example

Consider the method `secure` in Listing 2 with the parameter `high` as secret input and the method's result value as public output. Since the result value does not depend on the value of `high`, the method is noninterferent—which can be proven using deductive verification.

```
1  int secure(int high, int low) {
2    if (low == 5) {
3      low = identity2(low, high);
4    } else {
5      if (low == 2) {
6        low = identity1(low, high);
7      } else {
8        low = identity2(low, high);
9      }
10   }
11   return low;
12 }
```

```
13 int identity1(int low, int high) {
14     low = low + high;
15     low = low - high;
16     return low;
17 }
18
19 int identity2(int low, int high) {
20     return low;
21 }
```

Listing 2: Running example

The corresponding proof must handle nine symbolic execution paths: following Definition 2, we need to analyze two program runs, for each of which we distinguish three different cases, namely that the value of the parameter `low` is (a) 5, (b) 2, or (c) any other value. For this example, SDG-based techniques for checking noninterference will report a possible noninterference violation, as the called method `identity1` contains a *syntactic* dependency between its result value (that is assigned `low` afterwards) and the parameter `high`. This dependency, however, only affects the path for which the initial value of `low` is 2. Hence, the other two execution paths can be guaranteed to be noninterferent by the SDG-based approach.

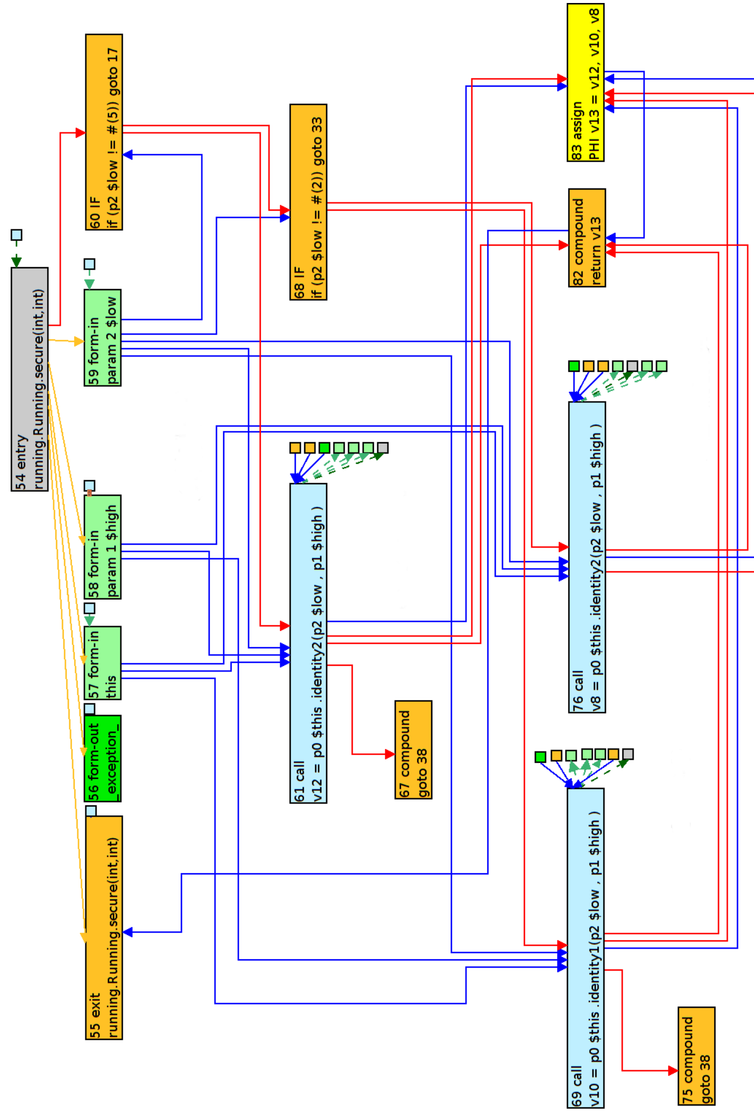

Figure 2: The SDG for the running example

6.2 Generation of the Simplified Program

In the following, we explain how we can simplify the program from Listing 2 in order to reduce the verification and testing effort. As seen in Section 3.1, SDG-based analyses are purely syntactic, highly scalable, and sound, but some of the reported violations may be false positives. However, even if there are false warnings, the SDG-based analysis allows to exclude execution paths and statements that are guaranteed not to affect have any effect on noninterference.

A further analysis then only needs to deal with those parts of the program that can potentially lead to an illegal leak according to the SDG-based analysis.

Figure 2 shows the SDG generated by JOANA for the method `secure` in the running example from Listing 2. The dependencies inside the three calls of the methods `identity1` and `identity2` are hidden, and only the method call nodes are shown. We use this simplification also in the following for describing paths in the SDG. Therein, node 58 represents the parameter `high` of the method `secure`, and node 55 the method's exit node. The two nodes are annotated as `high` and `low`, respectively. JOANA reports a security violation that contains an SDG path from the parameter `high` to the result value of the method `secure`, but successfully determines that there is no dependency between the parameter `high` and the result value of the two calls to the method `identity2`. Since the edges in the SDG represent all syntactical dependencies, JOANA cannot show the same for the call of `identity1`. Therefore, the SDG path $58 \rightarrow 69 \rightarrow 82 \rightarrow 55$ is reported as possible violation (the numbers correspond to the node ids in Figure 2). For the two execution paths where `identity2` is called, the absence of any illegal information flow is shown, and thus their analysis with the second, more precise, approach can be skipped.

Hence, we create a simplified program by excluding the parts already proved secure by JOANA. A simplified program is based on the backward slice of the low output and excludes some execution paths. We determine which paths to exclude by analyzing both the SDG of the entire program and the chop, to decide whether a branching node (e.g., a node representing an if-statement) must be true or false for an illegal information flow to occur, based on Definition 5.

Definition 5 (Analysis of branching nodes). *Let n_b be a conditional branching node in the backward slice with some node n_l corresponding to the low output. Let N_{true} be the set of successor nodes following the* true *branch of n_b in the CFG, and let N_{false} be the successor nodes following the* false *branch in the CFG. We define n_b to be a condition that* must be true *if the analyzed chop $C(n_h, n_l)$ contains nodes from N_{true} and no nodes from N_{false}. Conversely, n_b is defined as a condition that* must be false *if the chop contains nodes from N_{false} and no nodes from N_{true}.*

Consequently, given a high input and a low output corresponding to the SDG nodes n_h and n_l, respectively, and a branching node n_b that must be true (resp. false), any execution path of the original program on the false (true) branch of n_b will not lead to an illegal information flow from n_h to n_l. This allows us to exclude the execution paths that do not lead to an illegal information flow from the further analysis with a second approach. We exclude these paths by adding a special statement that disrupts the symbolic execution at the beginning of a false branch for a branching statement that *must be true* and at the beginning of a true branch for a branching statement that *must be false*. When the program is symbolically executed for verification and we reach a disruptive statement, the proof closes automatically for that branch. The test generation also immediately halts for that path once the symbolic execution reaches a disruptive statement. We can now define the simplified program by Definition 6 as follows.

Definition 6 (Simplified program). *Let n_h and n_l (corresponding to a high input and a low output, respectively) be two nodes in the SDG of a program P. The simplified program P_S consists of the following statements:*

1. *all statements whose SDG-nodes are in the backward slice $S_{bw}(n_l)$,*
2. *disruptive statements on branches that cannot lead to an illegal information flow (according to Definition 5).*

As shown in Herda et al. [22], the simplified program generated according to Definition 6 is noninterference-equivalent to the original program (i.e., the simplified program is noninterferent if the original program is noninterferent, and a counterexample for one program is also a counterexample for other program).

The chop reported by JOANA for the running example contains the nodes 58, 69, 82, and 55 (Figure 2). In the backward slice of the low output (i.e., of node 55), there are two branching nodes, nodes 60 and 68, corresponding to the two if-statements in the example program. Upon analyzing the two branching nodes, we automatically determine that, if an illegal information flow was possible, the first if-statement would need to take the *false* branch and the second if-statement would need to take the *true* branch. The program in Listing 3 is the simplified program of the running example. While in general the backward slice of the return statement may be much smaller than the original program, it contains the entire program from our example. Nevertheless, our approach determines that the paths which lead to the call of `identity2` cannot lead to an illegal information flow, thereby adding two occurrences of the statement `disruptExecution()`. This statement stops symbolic execution in the event of verifying the running example or generating tests for it.

```
1  int secure(int high, int low) {
2    if (low == 5) {
3      disruptExecution();
4      low = identity2(low, high);
5    } else {
6      if (low == 2) {
7        low = identity1(low, high);
8      } else {
9        disruptExecution();
10       low = identity2(low, high);
11     }
12   }
13   return low;
14 }
```

Listing 3: Simplified program for the running example

6.3 Verification and Testing of the Simplified Program

When using KeY to prove noninterference for the running example, we require 771 rule applications. Once the symbolic program execution is finished, there are still nine proof branches, one for each combination of paths in the two program runs as explained in Section 6.1, with open goals that remain to be closed. The verification of the simplified program in Listing 3 still requires 511 rule applications. For other examples, the verification effort may be further reduced

by removing program statements through slicing. Therein, the symbolic execution halts when one of the two program runs reaches a path that has already been deemed secure by JOANA, and the open goal of the corresponding proof branch is automatically closed. Thus, of the nine open proof goals remaining after symbolic execution, eight are trivially closed—in some cases even before their symbolic execution is finished.

The running example showcases how the SDG-based approach can assist verification by excluding both statements and execution paths from the program. The exclusion of execution paths is especially useful when proving noninterference. For an original program with n execution paths, the verification process must prove that the noninterference property holds for n^2 paths. When adding a single disruptive statement, the number of paths that need to be verified in the simplified program drops to a number between $(n-1)^2$ and $n^2/4$ (depending on whether the affected condition is the top level or not). Thus, the number of execution paths that need to be analyzed with the theorem prover can drop to a quarter of those in the original program.

The running example contains three execution paths. Thus, the automatic test generation approach (see Section 5) will attempt to generate 3^2 input pairs. However, only three pairs are low-equivalent as the branch that is taken gets determined by the low input. Consequently, the automatic test generation approach generates a test suite with three noninterference tests. When running the automatic test generation approach on the simplified program, only one test will be generated for the case in which both `low` inputs have the value 2. This is due to the inserted statements in the simplified program that disrupt symbolic execution. For the simplified program, the automatic test generation approach attempts (and succeeds) to generate a noninterference test only once, compared to the above-mentioned nine attempts for the original program. Hence, we have eight test generation attempts (calls to the SMT solver) that cannot lead to a noninterference property violation and are hence soundly skipped.

Besides the exclusion of execution paths as in the case of the running example, using the simplified program reduces the verification and testing effort also due to the removal of program statements from the original program.

7 The Combined Approach

The final part of the Noninterference Framework is the *Combined Approach* (see Beckert et al. [6], on which this section is based, for a more extensive description). The Combined Approach also integrates an SDG-based with a logic-based approach and attempts to prove noninterference. A comparison between the Combined Approach and the approach from Section 6 can be found in Section 8.

For a given program P, the first step of the Combined Approach is to run the SDG-based analysis in order to check the noninterference property (according to Definition 2) for P. If this step already clears out any illegal information flow for P, we need no further action as noninterference is guaranteed to hold. If, however, the automatic SDG-based approach detects a potential illegal informa-

tion flow, we continue with the second step of the Combined Approach which checks whether this information flow is a false positive or a genuine leak. Since the SDG-based analysis is the first step, the results provided by our approach are at least as precise as those of the SDG-based analysis. The second and final step of the Combined Approach is to apply a logic-based approach which tries to prove that certain syntactic dependencies in the SDG do not represent actual (semantic) dependencies. If all syntactic dependencies between high inputs and low outputs reported by the SDG-based analysis are proved to not represent semantic dependencies using the logic-based approach, then the analyzed noninterference property is proved to hold for P.

Formally, the Combined Approach (shown in Algorithm 1) works as follows. Let N_h denote the set of all nodes annotated as high, and N_ℓ the set of all nodes annotated as low. Moreover, we define for $n_h \in N_h$ and $n_\ell \in N_\ell$ a *violation* as a pair (n_h, n_ℓ) for which there is a path from n_h to n_ℓ in the SDG of P. We call the set of all nodes on a path from n_h to n_ℓ the *violation chop* $c(n_h, n_\ell)$. The result of the SDG-based approach is hence a set of violations (if this set is empty, noninterference is already guaranteed) for which the Combined Approach validates each violation chop and attempts to prove that it does not represent a semantic dependency in program P. This is done by attempting to show that the chop is interrupted (see Definitions 7 and 8) using a logic-based approach.

Definition 7 (Unnecessary summary edge). *A summary edge $e = (a_i, a_o)$ is called* unnecessary *if and only if there is no potential information flow from the formal-in node f_i to the formal-out node f_o corresponding to the actual-in node a_i and the actual-out node a_o, respectively.*

Definition 8 (Interrupted violation chop). *A violation chop is* interrupted *if we find a non-empty set S of unnecessary summary edges on this chop such that upon deletion of 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. The proof is done for all possible contexts of the called method. If the proof is successful, it is guaranteed that the summary edge was only a result of the over-approximation and can thus be soundly deleted.

As described above, the Combined Approach (see Algorithm 1) attempts to prove each violation chop to be interrupted. For each violation chop, we pick a summary edge, generate the appropriate information flow proof obligation for the corresponding program method, and perform a proof attempt with the theorem prover. If the proof succeeds, the summary edge can, by Definition 8, be safely deleted from the SDG. The choice of the summary edge in the violation chop, containing potentially multiple summary edges, is established by a given heuristic. Note that we only need to consider summary edges that belong to a violation chop and it is hence sufficient to regard only a smaller subset

```
Data: A set S of violation chops
Result: A set T of interrupted violation chops
foreach Violation chop C_V ∈ S do
    Build queue Q of summary edges in C_V, ordered by given heuristic;
    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
    if C_V interrupted then
        Add C_V to T;
    end
end
return T
```

Algorithm 1: The Combined Approach

of all summary edges. When—upon a successful proof attempt—we delete the corresponding summary edge, we check whether the deletion makes this violation chop interrupted. In this case, we can proceed with the remaining violation chops, attempting to make all of them interrupted. In case either the violation chop is not interrupted after deleting the summary edge or the proof attempt is not successful, we choose the next—following the given heuristic—summary edge from the violation chop. If we are able to interrupt every violation chop by deleting unnecessary edges, the Combined Approach guarantees noninterference.

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 noninterference of the entire program with the theorem prover. In practice, however, programs are often inter-procedural, and there are plenty of (different) summary edges for our approach to check. Nevertheless, it may happen in the worst case that we need to verify the `main` method with the theorem prover. This worst case scenario for our approach occurs in the event that not enough summary edges from inner method calls can be proved to be unnecessary.

For the example in Listing 2, we attempt to show that there is no potential information flow from the parameter `high` to the method's result value `secure`. When applying the SDG-based approach, an illegal information flow is reported, because the method `identity1`'s result value syntactically depends on the parameter `high` of the same method. This reported violation, however, is a false alarm. The violation chop consists of only one path that contains for a summary edge which connects (as explained in Section 3.1) the actual-in SDG-node for the parameter `high` and the actual-out SDG-node for `identity1`'s result value. Proceeding with the Combined Approach, we automatically generate a proof obligation for the theorem prover, stating that the result value for `identity1` does not semantically depend on parameter `high`. Upon the attempt to prove

the statement, we also prove that the result value of the method `secure` does not depend on the parameter `high` and thus show the noninterference for the method `secure`. 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 by the SDG-based approach.

The evaluation of the Combined Approach [6] shows that noninterference can be proved for programs for which the SDG-based approach—on its own—lacks the necessary precision and the logic-based approach—on its own—does not need to analyze the entire program.

8 Discussion

In Section 8.1, which is based on Herda et al. [22], we discuss the challenges of using the logic-based theorem prover KeY and the SDG-based static analysis tool JOANA for implementing the approaches from Sections 6 and 7. In Section 8.2, which is based on Herda [20], we compare the approaches in the Noninterference Framework and show how they can be integrated.

8.1 JOANA and KeY

For JOANA and KeY's challenges in our framework, we identified differing supports of Java features, possibly nonexecutable slices produced by JOANA, and differing program entry points (i.e., modular versus whole-program analysis).

Java features. While JOANA supports full Java except for reflection, KeY supports sequential Java programs only, as well as only a limited set of Java features. KeY also requires that either the source code or the method contracts of library methods are available. The implementation of the two approaches from Sections 6 and 7 using JOANA and KeY thus supports the same Java subset that KeY supports. Another challenge lies in the fact that the two tools do not work on the same programming language (respective level). While JOANA works on Java bytecode that is in a single static assignment (SSA) form, KeY works on Java source code. For the soundness of our implementation, we make the (arguably quite reasonable) assumption that the compilation of a Java program into bytecode does not change the noninterference properties of the program. Moreover, this also raises the issue of mapping byte code statements to source code statements. We are able to determine the line in the source code (which can contain more than one source code statement) from which a byte code statement originates. However, a source code statement can be compiled into more than one byte code statement and, due to the SSA form, for some source code statements, we may not even have a corresponding byte code statement. The simplified program from Definition 6 is thus impossible to generate using these tools. Instead, we generate an over-approximation of the simplified program by removing a line in the source code only if (1) the SDG contains a node corresponding to a byte code statement originating from that line and (2) no such node exists in the backward slice of the low output. In order to avoid multiple

source code statements on the same line, we first preprocess the source code and transform it so that it contains only one statement per line.

Slice executability. Another implementation challenge is the fact that SDG-based forward and backward slicing—as done by JOANA—can result in a program that is not executable or may handle jump statements such as `goto`, `break`, or `continue` incorrectly. This is not a problem for JOANA, since it does not need to generate any code for the analysis. For the second step of the approaches from Section 6, i.e., verification or test generation, however, having a program that can actually be executed is of great importance. This is a problem when implementing the approach using the slicers provided by JOANA off-the-shelf, but our approach is nevertheless feasible, since (as stated in Hammer [17, Chapter 2]) various solutions have been proposed that enhance SDG-based slicing and enable the generation of executable program slices [1,2,19]. For our implementation, we go about this problem by generating an over-approximation of the simplified program and by restricting ourselves to programs without jump statements. Thus, we obtain executable slices by (a) preserving lines that contain certain types of statements such as constructors or static initializers, and by (b) supporting only programs without jump statements.

Modular and whole-program analysis. Another issue with combining the two tools is the fact that, on the one hand, the analysis performed by KeY is done modularly at method level, and the results hold for any prestate that fulfills the precondition. JOANA, on the other hand, performs a whole-program analysis where the entire program is checked starting from an entry point method—in most cases the `main` method. If the goal is to verify the whole program, the approaches described in Sections 6 and 7 do not need any modification regarding this difference. For proving individual program methods, however, the SDG-based approach must analyze the given method without any context information. This is done in our implementation by adding an artificial main method as entry point for JOANA's analysis that only calls the method for which noninterference should be proven. However, this is no problem, as the SDG-based analysis can be implemented in such a way that any other method can serve as entry point.

8.2 Combinations of JOANA and KeY

The Combined Approach from Section 7 is well-suited for programs where the part that cannot be handled by the SDG-based approach is concentrated in a called method. If, however, syntactic dependencies between high input and low output are spread throughout the program, the whole program needs to be verified and no simplification is made. In such cases, the approach from Section 6 can still benefit from the SDG-based analysis. Using that approach, individual statements that have no effect on a potential noninterference violation can be removed, while the Combined Approach works on method-level granularity.

The Combined Approach and the simplification approach from Section 6 are orthogonal. In fact, the program fragment analyzed with the Combined Approach can be further reduced by applying the approach from Section 6. By simplifying the program (according to Definition 6), we can remove statements

and execution paths that are not relevant with respect to the possible dependency represented by the analyzed summary edge. Since the program that is simplified in this way is noninterference-equivalent to, i.e., conforms to the same noninterference properties as, the original program fragment, the soundness of the Combined Approach is not affected by this simplification.

9 Related Work

A multitude of research has been done on information flow security, dating back to the works by Denning and Denning [9, 10] and later by Goguen and Meseguer [14]. A survey on approaches for proving noninterference can be found in works by Sabelfeld and Myers [30]. In the following, we elaborate on some approaches that are similar to ours.

The work by Küsters et al. [25] also aims to achieve the best of both worlds—automatic analysis and interactive techniques for proving noninterference—by combining an automatic SDG-based analysis and a logic-based approach, i.e., a theorem prover. This approach (called *Hybrid Approach*) first attempts to prove noninterference using an SDG-based approach, e.g., an analysis by JOANA. If this attempt fails, the user must identify the cause of a possible false alarm and extend the program such that the affected low output is overwritten with a value that does not depend on the high input. The extended program is then checked by JOANA and, if the modified program is shown to be noninterferent, then—in the next step—a logic-based approach (using, e.g., the theorem prover KeY) is used to show that the extended program is equivalent to the original program (i.e., that the extended program results in the same low output). The Hybrid Approach improves the precision provided by JOANA and reduces the verification with KeY to purely functional proofs. However, the communication between the tools is done completely manually by the user and there is no assistance when searching for the causes of the false alarms. The approach does not utilize the results provided by the SDG-based analysis tool that could both discard those program parts that are irrelevant regarding the analyzed noninterference property and simplify the program to be verified. In fact, applying the program extension above makes the program to be verified in the second step even more complex.

Another combination of SDG-based and logic-based approaches is to check the satisfiability of path conditions for the execution paths that are determined by the reported security violation [17, 35]. If a path condition is unsatisfiable, then the respective execution path cannot lead to an illegal information flow. A program input that satisfies the path condition serves as a "witness", and the user can thereby analyze the program execution with that single input (the witness) and check whether an illegal information flow may occur. However, this is a difficult task, especially for indirect dependencies. For our work, the noninterference tests that we generate have two inputs and show the illegal information flow more clearly.

Moreover, there are also other approaches for automatic test generation that check noninterference. However, they do not support all the properties from Section 2. Le Guernic [26] proposed a sound information-flow testing mechanism based both on standard testing techniques and on a combination of dynamic and static analysis. Once a path coverage property is achieved, an argumentation regarding noninterference can be established. Furthermore, Milushev et al. present a tool that uses symbolic execution in combination with a form of self-composition for noninterference testing of C programs [28]. A logic-based approach to detect and generate exploits for information flow properties which presents them as JUnit tests is described by Do et al. [11]. Finally, information flow test case generation was also done by Gruska and Hriţcu et al. [15, 24].

10 Conclusion

We presented an overview of recent combinations of deductive verification and automatic test generation on the one hand, and static analysis on the other hand, together establishing our *Noninterference Framework*.

In a first use case, we presented an approach that uses deductive verification in order to automatically generate tests with the goal of finding noninterference violations. Deductive verification allows for the systematic generation of noninterference tests for all pairs of program execution paths and can identify pairs of execution paths that may never lead to a counterexample. This combination thus improves the computation of the achieved test coverage for noninterference.

In a second use case, we described two variants of an approach used for proving noninterference properties that combines deductive verification with an SDG-based static analysis. Deductive verification and SDG-based static analysis are complementary approaches. The highly-scalable SDG-based approach is used in a first step to show that large parts of the analyzed program cannot lead to a noninterference violation. In the second step, we apply the precise deductive verification approach for noninterference on the remaining program parts. This combination thus reduces the effort for the deductive verification.

References

1. Abadi, A., Ettinger, R., Feldman, Y.A.: Fine slicing - theory and applications for computation extraction. In: de Lara, J., Zisman, A. (eds.) 15th International Conference on Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science, vol. 7212, pp. 471–485. Springer (2012). https://doi.org/10.1007/978-3-642-28872-2_32
2. Agrawal, H.: On slicing programs with jump statements. In: Sarkar, V., Ryder, B.G., Soffa, M.L. (eds.) SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 302–312. ACM (1994). https://doi.org/10.1145/178243.178456
3. 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). https://doi.org/10.1007/978-3-319-49812-6

4. Ahrendt, W., Gladisch, C., Herda, M.: Proof-based test case generation. In: Ahrendt et al. [3], chap. 12, pp. 415–451. https://doi.org/10.1007/978-3-319-49812-6_12
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010), http://smt-lib.org/papers/smt-lib-reference-v2.0-r12.09.09.pdf, available at www.smt-lib.org
6. Beckert, B., Bischof, S., Herda, M., Kirsten, M., Kleine Büning, M.: Using theorem provers to increase the precision of dependence analysis for information flow control. In: Sun, J., Sun, M. (eds.) Formal Methods and Software Engineering (ICFEM). pp. 284–300. Springer (2018). https://doi.org/10.1007/978-3-030-02450-5_17
7. 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.) Logic-Based Program Synthesis and Transformation (LOPSTR). Lecture Notes in Computer Science, vol. 8901, pp. 19–37. Springer (2013). https://doi.org/10.1007/978-3-319-14125-1_2
8. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) Security in Pervasive Computing (SPC). Lecture Notes in Computer Science, vol. 3450, pp. 193–209. Springer (2005). https://doi.org/10.1007/978-3-540-32004-3_20
9. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056
10. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
11. Do, Q.H., Bubel, R., Hähnle, R.: Automatic detection and demonstrator generation for information flow leaks in object-oriented programs. Computers & Security **67**, 335–349 (2017). https://doi.org/10.1016/j.cose.2016.12.002
12. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) First International Conference on Tests and Proofs (TAP), Revised Papers. pp. 169–188. Springer (2007). https://doi.org/10.1007/978-3-540-73770-4_10
13. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. Transactions on Programming Languages and Systems **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041
14. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy (S & P). pp. 11–20. IEEE (1982). https://doi.org/10.1109/SP.1982.10014
15. Gruska, D.P.: Information flow testing. Fundamenta Informaticae **128**(1-2), 81–95 (2013). https://doi.org/10.3233/FI-2013-934
16. Hamann, T., Herda, M., Mantel, H., Mohr, M., Schneider, D., Tasch, M.: A uniform information-flow-security benchmark suite for source code and bytecode. In: Gruschka, N. (ed.) Nordic Conference on Secure IT Systems (NordSec). Lecture Notes in Computer Science, vol. 11252, pp. 437–453. Springer (2018). https://doi.org/10.1007/978-3-030-03638-6_27
17. Hammer, C.: Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs. Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Germany (2009). https://doi.org/10.5445/KSP/1000012049

18. 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). https://doi.org/10.1007/s10207-009-0086-1
19. Harman, M., Lakhotia, A., Binkley, D.: Theory and algorithms for slicing unstructured programs. Information and Software Technology **48**(7), 549–565 (2006). https://doi.org/10.1016/j.infsof.2005.06.001
20. Herda, M.: Combining Static and Dynamic Program Analysis Techniques for Checking Relational Properties. Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Germany (2020). https://doi.org/10.5445/IR/1000104496
21. Herda, M., Tyszberowicz, S., Müssig, J., Beckert, B.: Verification-based test case generation for information-flow properties. In: 34th SIGAPP Symposium on Applied Computing (SAC). pp. 2231–2238. ACM (2019). https://doi.org/10.1145/3297280.3297500
22. Herda, M., Tyszberowicz, S., Beckert, B.: Using dependence graphs to assist verification and testing of information-flow properties. In: Dubois, C., Wolff, B. (eds.) 12th International Conference on Tests and Proofs (TAP). Lecture Notes in Computer Science, vol. 10889, pp. 83–102. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_5
23. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. Transactions on Programming Languages and Systems **12**(1), 26–60 (1990). https://doi.org/10.1145/77606.77608
24. Hriţcu, C., Lampropoulos, L., Spector-Zabusky, A., Azevedo de Amorim, A., Denes, M., Hughes, J., Pierce, B.C., Vytiniotis, D.: Testing noninterference, quickly. Journal of Functional Programming **26**, 1–62 (2016). https://doi.org/10.1017/S0956796816000058
25. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. In: Computer Security Foundations Symposium (CSF). pp. 305–319. IEEE (2015). https://doi.org/10.1109/CSF.2015.28
26. Le Guernic, G.: Information flow testing. In: Cervesato, I. (ed.) 12th Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security (ASIAN). pp. 33–47. Springer (2007). https://doi.org/10.1007/978-3-540-76929-3_4
27. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
28. Milushev, D., Beck, W., Clarke, D.: Noninterference via symbolic execution. In: Giese, H., Rosu, G. (eds.) Formal Techniques for Distributed Systems. pp. 152–168. Springer (2012). https://doi.org/10.1007/978-3-642-30793-5_10
29. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
30. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
31. Scheben, C.: Program-level Specification and Deductive Verification of Security Properties. Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Germany (2014). https://doi.org/10.5445/IR/1000046878

32. Scheben, C., Greiner, S.: Information flow analysis. In: Ahrendt et al. [3], chap. 13, pp. 453–471. https://doi.org/10.1007/978-3-319-49812-6_13
33. Schmitt, P.H.: First-order logic. In: Ahrendt et al. [3], chap. 2, pp. 23–47. https://doi.org/10.1007/978-3-319-49812-6_2
34. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Cousot, R., Schmidt, D.A. (eds.) Third International Symposium on Static Analysis (SAS). Lecture Notes in Computer Science, vol. 1145, pp. 332–348. Springer (1996). https://doi.org/10.1007/3-540-61739-6_51
35. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. Transactions on Software Engineering and Methodology **15**(4), 410–457 (2006). https://doi.org/10.1145/1178625.1178628
36. 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.) International Verification Workshop (VERIFY@IJCAR). EPiC Series in Computing, vol. 3, pp. 141–155. EasyChair (2010). https://doi.org/10.29007/nnzj