Combining Static and Dynamic Program Analysis Techniques for Checking Relational Properties

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Mihai Herda

aus Brașov, Rumänien

Tag der mündlichen Prüfung:	13.12.2019
Erster Gutachter:	Prof. Dr. Bernhard Beckert
Zweiter Gutachter:	Prof. Dr. Shmuel Tyszberowicz

Combining Static and Dynamic Program Analysis Techniques for Checking Relational Properties

Mihai Herda

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebenen Hilfen selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, _____

Mihai Herda

Acknowledgements

I thank Prof. Bernhard Beckert for giving me the opportunity to pursue a PhD in his research group and for his useful advice throughout my time as a PhD student.

I thank Prof. Shmuel Tyszberowicz for the great collaboration that we had, for his feedback, and for agreeing to be the second reviewer of this thesis.

I thank my current and former colleagues, Aboubakr Achraf El Ghazi, Thorsten Bormer, Stephan Falke, David Farago, Christoph Gladisch, Daniel Grahl, Sarah Grebing, Markus Iser, Michael Kirsten, Jonas Klamroth, Vladimir Klebanov, Marko Kleine Büning, Tianhai Liu, Simone Meinhart, Florian Merz, Jonas Schiffl, Christoph Scheben, Prof. Peter Schmitt, Prof. Carsten Sinz, Mattias Ulbrich, and Alexander Weigl, for the great collaboration and for the great discussions that we had during this time. Working with you is and has been a pleasure.

I thank the students Etienne Brunner, Stephan Gocht, Holger Klein, Daniel Lentzsch, Joachim Müssig, Joana Plewnia, Ulla Scheler, Chiara Staudenmaier, Benedikt Wagner, and Pascal Zwick, for their work that I had the pleasure to supervise.

I thank the researchers form the groups of Prof. Wolfgang Ahrendt, Prof. Reiner Hähnle, Prof. Heiko Mantel, Prof. Ralf Reussner, and Prof. Gregor Snelting with whom I had the pleasure to cooperate.

I thank the German Israeli Foundation (GIF), the German Research Foundation (DFG), and the Federal Ministry of Education and Research (BMBF) for financing my work.

Last but not least, I thank my family and friends for their continuous moral support.

Abstract

The topic of this thesis belongs to the research area of formal software verification. It deals with checking relational properties of computer programs (i.e., properties that consider two or more program executions). The thesis focuses on two specific relational properties: (1) noninterference and (2) whether a program is a correct slice of another. The noninterference property states that running a program with the same public input values will always produce the same public outputs, regardless of the secret inputs (e.g., a password). This means that the secret inputs do not influence public outputs. Program slicing is a technique to reduce a program by removing statements from it while preserving a specified part of its behavior, such as the value of a variable at a program location.

The thesis provides frameworks that allow the user to analyze the two properties for a given program. The thesis advances the state of the art in the area of formal verification of relational properties by providing novel approaches for the analysis on the one hand and by combining existing approaches on the other hand. The thesis contains two parts, each handling one of the two relational properties.

Noninterference. The framework for checking noninterference provides new approaches both for automatic test generation and program debugging, and it combines these new approaches with approaches based on deductive verification and dependency graphs. The first new approach provided by this framework allows for the automatic generation of noninterference tests. It allows the user to search for violations of the noninterference property, and it also provides a coverage criterion for the generated test suites that is appropriate for relational properties. The second new approach provided by the framework is a relational debugger for analyzing noninterference counterexamples. It employs well-known concepts from program debugging, and it

extends them for relational program analysis. To support the user in proving a noninterference property, the framework combines an approach based on dependency graphs with a logic-based approach that uses a theorem prover. Dependency-graph-based approaches compute the dependencies between various program parts and check whether the public output depends on the secure input. Compared with logic-based approaches, dependency-graph-based approaches scale better. However, because they over-approximate the dependencies in a program, they may report false alarms. Two further contributions of the framework are thus represented by two combinations of the logic-based and dependency-graph-based approaches: (1) the dependency-graph-based approach simplifies the program that is checked by logic-based approach, and (2) the logic-based approach proves that certain dependencies computed by the dependency-graph-based approach are over-approximations and can be discarded from the analysis.

Program slicing. The second part of the thesis discusses a framework for automatic program slicing. Whereas most state-of-the-art slicing approaches only perform a syntactical analysis of the program, this framework also considers the semantics of the program and can remove more statements from a program. The first contribution of the slicing framework consists of a relational verification approach that was adapted to check whether a slice is valid, (i.e., whether it preserves the specified behavior of the original program). The advantage of using relational verification is that it works automatically for two similar programs—which is the case when considering a slice candidate and an original program. Thus, unlike the few state-of-the-art slicing approaches that consider the semantics of the program, our approach is automatic. The second contribution of this framework is a new strategy for generating slice candidates by refining dynamic slices (i.e., slices which are valid for a single input) by using the counterexamples provided by the relational verifier.

Zusammenfassung

Die vorliegende Dissertation ist im Bereich der formalen Verifikation von Software angesiedelt. Sie behandelt die Überprüfung relationaler Eigenschaften von Computerprogrammen, d.h. solche Eigenschaften, die zwei oder mehr Programmausführungen betrachten. Die Dissertation konzentriert sich auf zwei spezifische relationale Eigenschaften: (1) Nichtinterferenz und (2) ob ein Programm ein Slice eines anderen Programms ist. Die Nichtinterferenz-Eigenschaft besagt, dass die Ausführung eines Programms mit den gleichen öffentlichen Eingaben die gleichen öffentlichen Ausgaben produziert und dies unabhängig von den geheimen Eingaben (z.B. eines Passworts) ist. Das bedeutet, dass die geheimen Eingaben die öffentlichen Ausgaben nicht beeinflussen. Programm-Slicing ist eine Technik zur Reduzierung eines Programms durch das Entfernen von Programmbefehlen, sodass ein spezifizierter Teil des Programmverhaltens erhalten bleibt, z.B. der Wert einer Variablen in einer Instruktion in dem Programm.

Die Dissertation stellt Frameworks zur Verfügung, die es dem Nutzer ermöglichen, die obigen zwei Eigenschaften für ein gegebenes Programm zu analysieren. Die Dissertation erweitert den Stand der Technik in dem Bereich der Verifikation relationaler Eigenschaften, indem sie einerseits neue Ansätze zur Verfügung stellt und andererseits bereits existierende Ansätze miteinander kombiniert. Die Dissertation enthält jeweils einen Teil für die behandelten zwei relationalen Eigenschaften.

Nichtinterferenz. Das Framework zur Überprüfung der Nichtinterferenz stellt neue Ansätze für die automatische Testgenerierung und für das Debuggen des Programms zur Verfügung und kombiniert diese mit Ansätzen, die auf deduktiver Verifikation und Programmabhängigkeitsgraphen basieren. Der erste neue Ansatz ermöglicht die automatische Generierung von Nichtinterferenz-Tests. Er ermöglicht dem Nutzer, nach Verletzungen der

Nichtinterferenz-Eigenschaft im Programm zu suchen und stellt zudem ein für relationale Eigenschaften passendes Abdeckungskriterium für die generierten Test-Suites zur Verfügung. Der zweite neue Ansatz ist ein relationaler Debugger zur Analyse von Nichtinterferenz-Gegenbeispielen. Er verwendet bekannte Konzepte des Programm-Debuggens und erweitert diese für die Analyse relationaler Eigenschaften. Um den Nutzer beim Beweisen der Nichtinterferenz-Eigenschaft zu unterstützen, kombiniert das Framework einen auf Programmabhängigkeitsgraphen basierenden Ansatz mit einem auf Logik basierenden Ansatz, der einen Theorembeweiser verwendet. Auf Programmabhängigkeitsgraphen basierende Ansätze berechnen die Abhängigkeiten zwischen den unterschiedlichen Programmteilen und überprüfen, ob die öffentliche Ausgabe von der geheimen Eingabe abhängt. Im Vergleich zu logik-basierten Ansätzen skalieren programmabhängigkeitsgraphen-basierte Ansätze besser. Allerdings, können sie Fehlalarme melden, da sie die Programmabhängigkeiten überapproximieren. Somit bestehen zwei weitere Beiträge des Frameworks in Kombinationen von programmabhängigkeitsgraphen- und logik basierten Ansätzen: (1) der programmabhängigkeitsgraphen basierte Ansatz vereinfacht das Programm, das danach vom logik basierten Ansatz überprüft wird und (2) der logik basierte Ansatz beweist, dass einige vom Programmabhängigkeitsgraphen-basierten Ansatz berechnete Abhängigkeiten Überapproximationen sind und aus der Analyse entfernt werden können.

Programm-Slicing. Der zweite Teil der Dissertation behandelt ein Framework für das automatische Programm-Slicing. Während die meisten zum Stand der Technik gehörenden Slicing-Ansätze nur eine syntaktische Programmanalyse durchführen, betrachtet dieses Framework auch die Programmsemantik und kann dadurch mehr Programmbefehle entfernen. Der erste Beitrag des Frameworks besteht aus einem Ansatz zur relationalen Verifikation, der erweitert wurde, um die Korrektheit eines Programm-Slice nachzuweisen, d.h. dass es das spezifizierte Verhalten des Originalprogramms bewahrt. Der Vorteil der Benutzung relationaler Verifikation ist, dass sie auf zwei ähnlichen Programmen automatisch läuft – was bei einem Slice-Kandidaten und Originalprogramm der Fall ist. Somit, anders als bei den wenigen zum Stand der Technik gehörenden Ansätzen, die die Programmsemantik betrachten, ist dieser Ansatz automatisch. Der zweite Beitrag des Frameworks besteht aus einer neuen Strategie zur Generierung von Slice-Kandidaten durch durch die Verfeinerung von dynamischen Slices (für eine Eingabe gültigen Slices) mithilfe von der relationalen Verifikation gelieferte Gegenbeispiele.

Contents

List of Figures

Page

Part I

Introduction

1 Introduction

Software plays an increasingly important role in our society, as it is used everywhere, from transport and energy infrastructures to devices that we use in our daily life. One effect of this is the increasing importance of software correctness and security, as software faults can lead to dire consequences. Another effect is the increased size and complexity of software systems, which makes understanding the system and checking its correctness more difficult.

To check the correctness of computer programs with respect to a given property two types of approaches are currently used. On the one hand, there are dynamic approaches that check the program while running it. These approaches scale relatively well with the size of the analyzed programs. However, their coverage (i.e., the amount of inputs for which the property was checked) is relatively low. On the other hand, static approaches, that check the source code of the program without running it, offer high coverage but low scalability.

Until now, most effort was put into the development of approaches that check functional properties. These properties consider the relation of the inputs and outputs of a single program execution. There are, however, useful properties that consider the relation of inputs and outputs of two (relational properties) or more (hyper-properties) program executions. Examples of such properties are noninterference (introduced by Goguen and Meseguer [1982]) and whether a program is a valid slice of another program (program slicing was introduced by Weiser [1981]). When checking functional properties static approaches have a scalability problem and dynamic approaches have a coverage problem. For checking relational properties these problems are even more dire, because multiple program executions must be considered.

This thesis provides frameworks that allow the user to analyze two relational properties (noninterference and slice validity) for a given program. It advances the state of the art in the area of formal verification of relational properties by providing novel approaches on the one hand and by combining existing approaches in new ways on the other hand.

1.1 Goal

This thesis tackles the problem of analyzing relational properties of programs. Its goal is to build frameworks that combine the advantages of both static and dynamic program analysis techniques. The thesis focuses on two types of relational properties and provides a framework for each of them. The following relational properties are considered in this thesis:

1. In the research area of information flow security, it handles problem of checking the *noninterference* property (i.e., whether inputs with *high* security level influence outputs with *low* security level).

2. In the more general area of software engineering, it handles the problem of *program slicing*, which is a technique to reduce a program by removing statements from it while preserving a specified part of its behavior.

The first relational property, noninterference, addresses the confidentiality and integrity aspects of computer security. Thus, checking a system with respect to this property, allows us to evaluate the security of a software. The variations of the noninterference property which are handled in this thesis are defined in Section 2.1.

The second property, whether a program reduced by removing statements from it preserves a specified aspect of the original program behavior, addresses the problem of increasing size and complexity of software. The reduced program can be easier understood by the user or easier analyzed with a second approach. Program slicing is a powerful tool for challenges in software engineering such as code comprehension, debugging, refactoring and fault localization (see Binkley and Harman [2004]), as well as in information flow security (see Hammer and Snelting [2009]). We define what constitutes a correct program slice in Section 2.5.

The two properties which are considered in this thesis are highly representative in the area of relational property verification. Numerous other relational properties can be stated as a variation of one of the two properties (e.g., program equivalence can be formulated as a noninterference property in which all inputs have a high security level and all outputs a low security level). Thus, being able to analyze a software system with respect to these two properties and improvements on the state of the art in these areas are of great benefit. Furthermore, the approaches that are part of the two frameworks can be adapted for the analysis of other relational properties. This is especially true for the idea of automatically identifying those parts of the analyzed program which are irrelevant with respect to the analyzed property and discarding them from the analysis.

The goal is realized through the contributions presented in Section 1.3.

1.2 State of the Art

Before explaining the contributions of this thesis, we present a short overview of the state of the art in information flow security and program slicing. A more in-depth overview of related work can be found in Chapter 9.

Information Flow Security. There exist various static approaches and tools for checking the noninterference property (i.e., whether certain inputs with *high* security level influence outputs with *low* security level) of a program. Some of these approaches have a high degree of automation, yet they may produce false alarms, since they over-approximate the dependencies within a program. Other approaches are more precise, but require more effort and user interaction. In what follows, we describe some of the main approaches.

Approaches (e.g., Graf et al. [2016]) that are based on *system dependence graphs* (SDGs) syntactically compute the possible dependencies between the program statements and check whether the low output depends on the high input. Whereas they scale very well, such approaches over-approximate the actual dependencies in the program, as they work on the syntactical level of the program. This can result in false alerts. For a program such as "`l=l+h; l=l-h;`", where `l` has the security level low and `h` the security level high, a syntactical approach will identify a dependency between the variables `l` and `h` even though there is in fact no (semantical) dependency between them.

Approaches that are based on *type systems* (e.g., Lortz et al. [2014]) have the same scalability and precision as SDG-based approaches, as shown by Mantel and Sudbrock [2013]. Such approaches enforce noninterference by assigning a security type (e.g., high or low) to the program variables and then checking, using the rules of the type system, whether the expressions in the program conform to the type system.

Logic-based approaches (e.g., Beckert et al. [2013]), on the other hand, have a higher precision (i.e., they produce less false alarms) because they also consider the semantics of the program statements. However, those approaches have a lower scalability. False alarms occur when the system fails to find a proof in the allotted time even though the proof obligation is valid. Proving noninterference using this approach is harder than proving a functional property. This is because the number of execution paths for which the property needs to be checked is quadratic when considering two program executions.

Dynamic approaches based on *runtime monitoring* (e.g., Guernic [2007]) have been developed. A monitor checks every program instruction before its execution. If it determines that the instruction reveals some high information, then the execution of that instruction is either prevented, or the instruction is replaced with a safe instruction which is then executed. However, the precision of sound dynamic techniques for checking noninterference is limited,

and they may report more false alarms than (precise) static techniques, as shown by Russo and Sabelfeld [2010].

Approaches for automatic test generation for the noninterference property have also been developed (e.g., Milushev et al. [2012]). However, they do not support heap data structures. They also provide no solution for the case in which counterexamples could not be found due to timeouts.

Program Slicing. Most existing slicing approaches are only syntactical (i.e., they do not take the semantics of the various program operations into account). The problem with syntactic approaches is that the obtained slices are too large and often not useful. The SDG plays a prominent role in syntactic slicing; approaches based on it have been surveyed by Xu et al. [2005] and by Tip [1994]. On the other hand, existing approaches that do consider the program semantics are not fully automatic and require auxiliary specifications from the user. For example, precomputed or user-provided functional loop invariants are used in Barros et al. [2012]. Other program slicing approaches (e.g., by Jaffar et al. [2012a]) that consider the program semantics can automatically handle loops by using techniques like abstract interpretations to over-approximate the values of the program variables. However, the over-approximated values lead to larger slices.

1.3 Contributions

This thesis has two major contributions in the form of two frameworks—one for analyzing noninterference properties (contribution **C1.1**) and one for slicing programs (contribution **C2.1**). The two frameworks and the contributions they provide are illustrated in Figure 1.1. Both frameworks combine static and dynamic program analysis techniques in order to analyze a program with respect to one of the two relational properties.

The noninterference framework assists the user in two cases. In the first case, it handles the situation in which the program violates the specified noninterference property. For this, the framework provides an approach for automatic test generation for noninterference properties (contribution **C1.2**) and an approach for debugging (contribution **C1.3**) a program with respect to a noninterference property. These two approaches assist the user in finding and understanding noninterference violations (i.e., noninterference counterexamples). Furthermore, it provides an approach (contribution **C1.4**) for simplifying the analyzed program that reduces the necessary effort for generating and executing the noninterference tests.

In the second case, it handles the situation in which the program fulfills the specified noninterference property, and the user tries to prove this. The framework combines an SDG-based and a logic-based approach in two ways. On the one hand, the approach for simplifying programs (contribution **C1.4**)

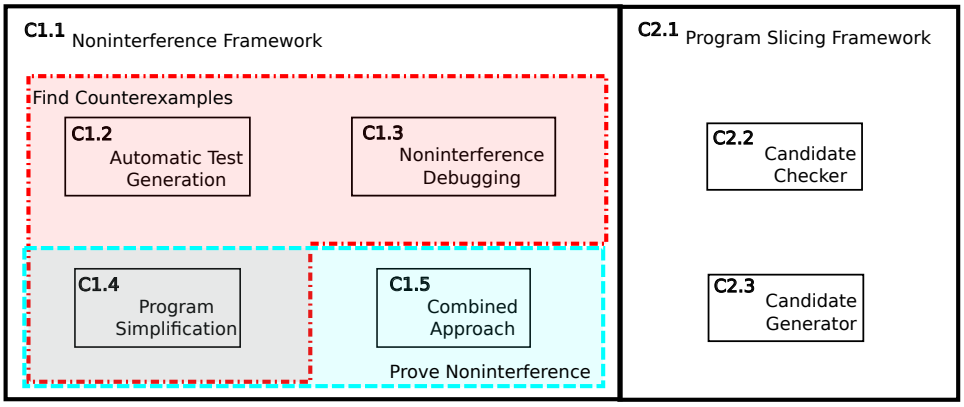

Figure 1.1: Contributions of this thesis

is used again, as it is also able to reduce the effort needed to verify a program with respect to a noninterference property. On the other hand, as part of a Combined Approach (contribution **C1.5**) a logic-based approach is used to reduce the over-approximation of an SDG-based approach.

The framework for program slicing combines static and dynamic program analysis techniques to automatically reduce a program by removing statements from it such that a specified part of the program's behavior is preserved. The slicing framework is composed of two components which interact with each other. The first component, the candidate checker (contribution **C2.2**), is an adapted relational verifier (a static technique, see Section 2.7) that checks whether a slice candidate obtained by removing statements from a program is a valid slice (i.e., it preserves the required behavior). The second component, the candidate generator (contribution **C2.3**), uses a dynamic technique to check which program instructions contributed to the preservation of the specified behavior for a single program run. This information is then used to generate slice candidates.

In the following we present each contribution of this thesis.

1.3.1 Information Flow Security

C1.1: A framework for checking noninterference. This contribution consists of a framework combining SDG-based and logic-based approaches for proving noninterference. If these approaches fail, the framework provides the user with the option of automatically generating noninterference tests. The user can also analyze noninterference counterexamples with a noninterference debugger. The novelty of the framework lies in the fact that it provides a set of tools for both statically analyzing noninterference properties and for searching and analyzing counterexamples. The framework uses the contributions **C1.2**, **C1.3**, **C1.4**, and **C1.5**, which we describe in the following.

C1.2: Automatic test generation for noninterference properties. This contribution consists of an approach for generating tests for noninterference properties. Improving on the state of the art, we show how noninterference properties with declassification for object oriented programs can be tested, and we extend existing test coverage criteria to measure the coverage of noninterference test suites. This contribution was previously published in Herda et al. [2019c]. The test generation approach for noninterference properties is an extension of the test generation approach for functional properties which was published in Ahrendt et al. [2016b].

C1.3: An approach for analyzing noninterference violations. This contribution consists of a novel technique for analyzing counterexamples of noninterference properties. It employs well-known concepts from program debugging, and it extends them for relational properties such as noninterference. Furthermore, it provides novel features such as such as *relational watch expressions* and *relational conditional breakpoints*. This contribution was previously published in Herda et al. [2019a].

C1.4: Using dependency analysis to simplify programs. This contribution consists of an approach that uses SDG-based static analysis to create simplified programs that are noninterference equivalent to the original program. The simplification is done by removing program statements and branches which are irrelevant with respect to the analyzed noninterference property. The simplified program can then be analyzed using a second approach. We use deductive verification and automatic test generation as a second approach. The improvement on state of the art lies in the fact that the approach goes beyond simple program slicing—it identifies program paths that are irrelevant to the noninterference property and can be ignored by the second approach. This contribution was previously published in Herda et al. [2018].

C1.5: Using theorem provers to increase the precision of SDG-based approaches. This contribution consists of the *Combined Approach*, which combines deductive verification with SDG-based static analysis and has the goal of increasing the precision of the SDG-based static analysis. For each possible noninterference violation reported by the SDG-based approach, the *Combined Approach* automatically generates proof obligations that state that certain reported program dependencies are not real (because they are over-approximated). These proof obligations are then relayed to a theorem prover, which employs a more precise analysis. The improvement on state of the art lies in the fact that the interaction between the two approaches is fully automated. This contribution was previously published in Beckert et al. [2017a] and Beckert et al. [2018a].

1.3.2 Program Slicing

C2.1: A framework for precise and automatic programs slicing. This contribution consists of an easily extensible framework for precise and automatic program slicing. The framework consists of two components which interact with each other. The first component is the candidate generation engine, which removes statements in the program according to a given strategy and sends the obtained slice candidate to the second component. The second component is a relational verifier that was adapted for the verification of the validity of slice candidates. The relational verifier transmits one of three possible answers to the candidate generation engine: (1) the candidate is a valid slice, (2) the candidate is not valid along with an input that leads to a violation of the slice property, or (3) a timeout. The framework uses the contributions **C2.2** and **C2.3**, which we describe in the following. The novelty of the framework lies in the fact that it allows for precise program slicing and takes the semantics of the program into consideration without requiring any auxiliary specification like loop invariants.

C2.2: Adapting a relational verifier to check the validity of slice candidates. This contribution consists of an approach for verifying the validity of a slice candidate. We adapt a relational verifier to show that a slice candidate is a valid slice of the original program. The novelty lies in the fact that the adaptation allows the slicing approach to be automatic while also considering the program semantics.

C2.3: Using dynamic slicing to generate slice candidates. This contribution consists of a novel slice candidate generation strategy that uses a dynamic slicer to generate an initial slice candidate which is then iteratively refined using the counterexamples provided by the relational verifier.

Contributions **C2.1**, **C2.2**, **C2.3** were previously published in Beckert et al. [2017b], Beckert et al. [2019d], and in Beckert et al. [2019c].

1.4 Relevant Publications

In this section we list for each chapter the previous publications by the author in which the contributions of the respective chapter were presented. An overview of these publications is shown in Table 1.2. All publications (including the ones that do not contain any results presented in this thesis) of the author are listed in Appendices B.1–B.4.

Table 1.2: Overview of relevant publications

Chapter 4	Ahrendt et al. [2016b]
	Herda et al. [2019c]
Chapter 5	Herda et al. [2019a]
Chapter 6	Herda et al. [2018]
Chapter 7	Beckert et al. [2017a]
	Beckert et al. [2018a]
Chapter 8	Beckert et al. [2017b]
	Beckert et al. [2019d]
	Beckert et al. [2019c]

Chapter 4

- Ahrendt et al. [2016b], which is a chapter in the book about the KeY theorem prover (see Ahrendt et al. [2016a]) that was published in 2016.
- Herda et al. [2019c], which was presented at the Software Verification and Testing track at the *ACM Symposium of Applied Computing* in 2019.

In Ahrendt et al. [2016b] the extension of the KeY theorem prover for automatic test generation for functional properties is presented. This test generation approach is extended in Herda et al. [2019c] to support noninterference properties. Chapter 4 is based on Herda et al. [2019c].

Chapter 5

- Herda et al. [2019a], which was presented at the *Workshop on Program Equivalence and Relational Reasoning* in 2019.

Herda et al. [2019a] is a tool paper which presents DIbugger, a debugger for relational properties. DIbugger constitutes the implementation of the approach for analyzing noninterference counterexamples and is presented in Chapter 5.

Chapter 6

- Herda et al. [2018], which was presented at the *International Conference on Tests and Proofs* in 2018.

In Herda et al. [2018] an approach is presented that uses an SDG-based analysis to create simplified programs with respect to a noninterference property. The simplified programs are noninterference equivalent to the original program and can be analyzed with respect to the noninterference property with a second approach. As a second approach we use deductive verification and automatic test generation. Chapter 6 is based on Herda et al. [2018].

Chapter 7

- Beckert et al. [2017a], which was presented at the *Workshop on Hot Issues in Security Principles and Trust* in 2017.
- Beckert et al. [2018a], which was presented at the *International Conference on Formal Engineering Methods* in 2018.

In Beckert et al. [2017a] and in its extension, Beckert et al. [2018a], a Combined Approach is presented which uses a deductive theorem prover to show that certain program dependencies computed by an SDG-based approach are over-approximations. The precision of the SDG-based approach for proving noninterference is thus improved. Chapter 7 is based on Beckert et al. [2018a].

Chapter 8

- Beckert et al. [2017b], which was presented at the *International Conference on Integrated Formal Methods* in 2017.
- Beckert et al. [2019d], which is a technical report that was published in 2019.
- Beckert et al. [2019c], which was presented at the *International Conference on Software Engineering and Formal Methods* in 2019.

In Beckert et al. [2019c], which is an extension of Beckert et al. [2019d], a framework that uses relational verification for automatic and precise program slicing is presented. In Beckert et al. [2017b] the implementation of the slicing framework, the tool SemSlice, is presented. Chapter 8 is based on Beckert et al. [2019c].

1.5 Outline

The use cases for the analysis of the two properties (noninterference and slice validity) are fundamentally different. In the case of noninterference, the user wants a definitive answer on whether the program fulfills the property. If the approaches he uses do not guarantee noninterference, the user tests and debugs the program to find possible violations. Therefore, a framework for the analysis of noninterference properties needs to employ highly precise, possibly interactive, approaches as well as runtime analysis techniques such as testing and debugging.

For program slicing, on the other hand, we know that any program is a slice of itself. Thus, we are always guaranteed to have a correct slice. When employing a program slicing technique, the user is far less motivated to further investigate whether more statements can be removed, and he is more

likely to accept the slice that was found by the employed approach. Thus, a framework for program slicing needs to use fully automatic techniques.

The different application scenarios lead to different approaches that need to be used when analyzing a program with respect to the two properties. Thus, the two applications scenarios motivate the structure of this thesis—each of the two frameworks is presented in a different part of the thesis. In Part II we present the framework for checking noninterference properties which, on the one hand, provides the user with the means to analyze certain parts of the program precisely (and even interactively if needed) and, on the other hand, to test and debug the program in case it cannot be proved that the property holds for the program. In Part III we present the framework for automatic program slicing which does not require any input from the user other than the specification with respect to the behavior that must be preserved in the reduced program.

1.5.1 Part 1—Introduction

In Chapter 2 we introduce the notions that are needed to understand the rest of the thesis. We start by explaining the fundamental information flow security notions in Section 2.1. We then introduce the Dynamic Logic for Java (JavaDL), the KeY theorem prover, and the Java Modeling Language (JML) in in Section 2.2. We present the extensions for specifying and proving noninterference properties with JML and KeY in Section 2.3. Then, in Section 2.4, we introduce the extension of KeY for automatic test generation for functional properties. We continue with an introduction to program slicing in Section 2.5 and show in Section 2.6 how program slicing based on the SDG is used in the JOANA tool to prove noninterference properties. Finally, in Section 2.7 we introduce relational verification and the LLRêve tool. Chapter 2 does not contain any contribution of this thesis.

1.5.2 Part 2—Information Flow Security

Chapter 3 presents the framework for checking noninterference that constitutes contribution **C1.1**. The chapter presents a general overview of the framework, which is presented throughout Part II and explains how contributions **C1.2**, **C1.3**, **C1.4**, and **C1.5** interact with each other.

Chapter 4 handles the problem of testing noninterference properties of object oriented programs. We explain how noninterference tests can be used to check four variations of the noninterference property in Section 4.2 and then present an approach based on symbolic execution for the automatic generation of noninterference tests in Section 4.3. We extend existing test coverage criteria and discuss their suitability for noninterference test suites in Section 4.4. An implementation of the approach using the KeY theorem prover is evaluated in Section 4.5. Chapter 4 describes contribution **C1.2**.

Chapter 5 presents an approach for analyzing noninterference violations. The requirements and assumptions made for the design of this approach are described in Section 5.2. The approach is realized in the form of a noninterference debugger, which is presented in Section 5.3. Its usefulness is demonstrated on two examples in Section 5.4. Chapter 5 describes contribution **C1.3**.

Chapter 6 presents an approach that uses SDG-based program analysis to simplify programs with respect to a noninterference property. The approach is presented using an example that is shown in Section 6.2. Then, in Section 6.3 it is shown how possible noninterference violations reported by an SDG-based approach are used to create simplified programs. These simplified programs can then be handled with a second approach to prove or disprove the reported security violation. We discuss using (1) deductive verification (in Section 6.4) and (2) automatic test case generation (in Section 6.5) as the second approach. The challenges of implementing the approach using JOANA as the SDG-based approach and KeY as the theorem prover and test case generator are discussed Section 6.6. Chapter 6 describes contribution **C1.4**.

Chapter 7 presents the Combined Approach, which, like the approach in Chapter 6, also combines an SDG-based approach with a precise logic-based approach. For every potential noninterference violations reported by the SDG-based approach the Combined Approach, presented in Section 7.2, automatically generates proof obligations that state that certain program dependencies reported by the SDG-based approach are over-approximations. Section 7.3 presents a prototypical implementation of the Combined Approach that uses the tools JOANA and KeY as the SDG-based and logic-based approach respectively. The Combined Approach is evaluated in Section 7.4 and possible optimizations to it are discussed in Section 7.5. Chapter 7 describes contribution **C1.5**.

1.5.3 Part 3—Program Slicing

Chapter 8 presents a framework for automatic program slicing which uses relational verification. Section 8.2 formally defines the programs that are handled by the slicing framework and what constitutes a valid slice for a given program. In Section 8.3 a relational verifier is adapted to check whether a slice candidate obtained by removing instructions from a program is indeed a valid slice. Based on this, in Section 8.4, a framework for precise and automatic program slicing is presented. As part of this framework, three strategies for the generation of slice candidates are presented. It is shown how dynamic slicing approaches can be used to generate slice candidates that are then refined using the counterexample provided by the relational verifier. Section 8.5 discusses the evaluation results and the strengths and weaknesses of slicing approaches that are based on the framework. Chapter 8 describes contributions **C2.1**, **C2.2**, and **C2.3**.

1.5.4 Part 4—Related Work and Conclusion

Chapter 9 presents related work with respect to the contributions in both information flow security (in Section 9.1) and program slicing (in Section 9.2).

Chapter 10 gives a summary of the thesis in Section 10.1 and discusses in Section 10.2 possible ways in which the two frameworks can be further developed along with research questions that should be investigated.

2 Foundations

2.1 Information Flow Security

This section is based on Herda et al. [2019c], and it describes the noninterference properties that are handled in this thesis. It is needed to understand Chapters 3, 4, 5, 6, and 7.

Information flow security addresses the problem of whether certain parts of a program exercise an influence on other parts. The main property analyzed is related to confidentiality, and requires the avoidance of illegal *information flows* (i.e., situations where *high*/confidential input is leaking to *low*/public output). This property is known as *noninterference* and was introduced by Goguen and Meseguer [1982]. Intuitively it requires that the high input does not interfere with the low output. Thus, by observing the program's low output one cannot distinguish between different high inputs (i.e., if a program is executed twice—with different high inputs but identical low inputs, an attacker will observe identical behaviors). Note that an attacker can observe low information but not high information. For example, if the credit card number is the high input, then unauthorized viewers (e.g., people working in the company's warehouse) should not be able to observe this information—directly or indirectly.

Attacker model. The attacker considered in this thesis is able to provide low inputs and run the analyzed programs with these low inputs and observe the low outputs of the programs. In this work we consider only sequential, deterministic, and terminating programs. Thus, the attacker observes the low outputs of the program executions and tries to deduce information on the high inputs of that program.

Noninterference properties. In the following we formally define the properties which are supported by the approaches in the framework for checking noninterference properties, which is described in Chapter 3. We introduce the *low-equivalence* relation ($\sim_L$) to characterize program states that are indistinguishable for an attacker, where a program state s is an assignment of values to variables. We assume that the input of a program is included in the prestate and that the output of a program is included in the poststate.

Definition 2.1 (Low-equivalent states)**.** Two states—s_1 and s_2—are low-equivalent with regard to the set of all low variables L iff they assign the same values to low variables:

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

Definition 2.2 (Classical noninterference)**.** A program P is noninterferent iff for all prestates s_1 and s_2

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

where s_1' and s_2' are poststates after executing P in the initial states s_1 and s_2, respectively. Note that because we only consider deterministic and terminating programs, s_1' and s_2' are uniquely determined.

The classical noninterference property, as presented in Definition 2.2, requires that two executions of the program that start in two states which are indistinguishable for the attacker also terminate in two indistinguishable states. If this holds, then the high values of the prestates do not influence the low values of the poststates. This property, however, can be too restrictive for the cases in which it is acceptable for the attacker to see parts of the high data.

A classical example for this is a login system in which an attacker can try out different combinations of user names and passwords. While the system is not allowed to leak out a user's password, the attacker can find out whether these combinations are correct or not. Thus, he obtains information about the sensitive data. To allow such a case but still forbid cases in which the system outright leaks sensitive information to the attacker, we define (in Definition 2.3) the notion of noninterference with declassification (i.e., the release of part of the high information). For this, let $expr_s$ be a user-provided expression in first order logic that is evaluated in state s and describes the high information that an attacker is allowed to know.

Definition 2.3 (Noninterference with declassification)**.** Considering the declassification expression *expr*, a program P is noninterferent iff for all initial states s_1 and s_2

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

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

In Definition 2.3 we use the low-equivalence relation from Definition 2.1. For the login example, assuming the user names and passwords are respectively stored in two arrays—users and passwords—the expression can be, for instance:

$$\exists\, \text{int}\, i \wedge \text{users[i]} \doteq \text{user}_{\text{in}} \wedge \text{passwords[i]} \doteq \text{password}_{\text{in}},$$

meaning that the attacker is allowed to find out whether the user name and password provided as input to the program match a stored combination of user name and password.

When dealing with object-oriented programs, it is sometimes too restrictive to require all low variables in the two poststates to be equal. This is the case in programs that create new object references: a program will not necessarily create the same reference if executed twice, even for exactly the same input. Therefore, the two poststates that are analyzed at the end of the two program executions that started in two low-equivalent prestates will most likely not be low-equivalent. To tackle this issue, Beckert et al. [2013] have developed a variation of noninterference using a different semantics of low-equivalence, based on object isomorphism.

Definition 2.4 (Low-equivalence with isomorphism)**.** Two states s_1 and s_2 are low-equivalent iff

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

with L denoting the set of all low variables in state s and π a bijective function between states.

The idea is that an attacker cannot see the exact reference address of an object reference and can only compare object references using the `==` operator. Thus, if the object structures in the two poststates are isomorphic, then the attacker will obtain the same results when comparing the object references with the `==` operator.

The specification and verification of the noninterference properties defined in this section are supported by the program verification tool KeY (as explained in Section 2.3). Furthermore, the properties from Definitions 2.3 and 2.4 can easily be combined.

2.2 JavaDL and the Theorem Prover KeY

In this section we introduce JavaDL—a first order dynamic logic for Java—and KeY—a deductive program verification tool for Java. This section is needed to understand Chapters 4, 6, 7, and 3.

2.2.1 JavaDL

Dynamic logic, introduced by Pratt [1976] (also see Harel et al. [2002]), is a multi-modal logic used to express properties of computer programs and reason about them. A dynamic logic for a subset of sequential Java programs was introduced by Beckert [2001] and extended by Schmitt et al. [2011] to support heaps. In this section we present the syntax and semantics of JavaDL, and then we present a calculus for it. The definitions in this section are based on the ones presented in [Weiß, 2011, Chapter 5]. An even more extensive description of JavaDL can be found in Beckert et al. [2016b].

JavaDL syntax. We begin by defining JavaDL signatures (Definition 2.5), we then define program fragments (Definition 2.6), and we define the JavaDL syntax (Definition 2.7).

Definition 2.5 (JavaDL Signature)**.** A JavaDL signature Σ is a tuple $(\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$ consisting of:

- a set τ of types
- a partial order $\preceq$ on τ, called the subtype relation
- a set V of (logical) variables,
- a set PV of program variables,
- a set F of function symbols,
- a set $F_u \subseteq F$ of unique function symbols,
- a set P of predicate symbols,
- a static typing function α such that $\alpha(v) \in \tau$ for all $v \in V \cup PV$ such that $\alpha(f) \in \tau^* \times \tau$ for all $f \in F$, and such that $\alpha(p) \in \tau^*$ for all $p \in P$ (where τ^* denotes the set of arbitrarily long tuples of elements of τ), and
- a Java program *Prg*.

Note that we restrict ourselves to single-threaded Java programs. All JavaDL signatures are required to contain some function and predicate symbols that are needed to formalize certain aspects of Java programs (e.g., heaps using the theory of arrays, as introduced by McCarthy [1993]). We do not present them here, but refer to [Weiß, 2011, Chapter 5]. In the following we define program fragments in the context of a Java program *Prg* that is part of a JavaDL signature.

Definition 2.6 (Program Fragments)**.** A program fragment p in the context of a program *Prg* is a sequence of Java statements, where there are local variables $a_1, \ldots, a_n \in PV$ of Java types $T_1, \ldots, T_2$ such that extending *Prg* with an additional class

```
1 class C{
2    static void m(T1 a1, ...,  Tn an){ p }
3 }
```

yields again a legal program.[1]

We can now present the syntax of JavaDL.

Definition 2.7 (JavaDL syntax)**.** For a JavaDL signature $\Sigma = (\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$, the sets $Term^A_\Sigma$ of terms of type A, $Formula_\Sigma$ of formulas, and $Update_\Sigma$ of updates are defined by the following grammar:

$$
\begin{aligned}
Term^A_\Sigma \quad &::= \quad x \mid a \mid f(Term^{B'_1}_\Sigma, \ldots, Term^{B'_n}_\Sigma) \mid \{Update_\Sigma\} Term^A_\Sigma \mid \\
&\qquad\quad \mathit{if}\ (Formula_\Sigma)\ \mathit{then}\ (Term^A_\Sigma)\ \mathit{else}\ (Term^A_\Sigma) \\
Formula_\Sigma \quad &::= \quad \mathit{true} \mid \mathit{false} \mid r(Term^{B'_1}_\Sigma, \ldots, Term^{B'_n}_\Sigma) \mid \neg Formula_\Sigma \mid \\
&\qquad\quad Formula_\Sigma \wedge Formula_\Sigma \mid Formula_\Sigma \vee Formula_\Sigma \mid \\
&\qquad\quad Formula_\Sigma \rightarrow Formula_\Sigma \mid Formula_\Sigma \leftrightarrow Formula_\Sigma \mid \\
&\qquad\quad \forall\, A\ x\ Formula_\Sigma \mid \exists\, A\ x\ Formula_\Sigma \mid [\mathtt{p}] Formula_\Sigma \mid \\
&\qquad\quad \langle \mathtt{p} \rangle Formula_\Sigma \mid \{Update_\Sigma\} Formula_\Sigma \\
Update_\Sigma \quad &::= \quad a := Term^{A'}_\Sigma \mid Update_\Sigma \parallel Update_\Sigma \mid \{Update_\Sigma\} Update_\Sigma
\end{aligned}
$$

for any variable $x \in V$ with $\alpha(x) = A$, any program variable $a \in PV$ with $\alpha(a) = A$, any function symbol $f \in F$ with $\alpha(f) = (B_1, B_2, \ldots, B_n, A)$, any predicate symbol $r \in P$ and $\alpha(r) = (B_1, B_2, \ldots, B_n)$, where $B'_1 \preceq B_1, \ldots, B'_n \preceq B_n$, any legal program fragment $\mathtt{p}$ in the context of Prg, and any type $A' \in \tau$ with $A' \preceq A$. The set $Term_\Sigma$ is defined as $\cup_{A \in \tau} Term^A_\Sigma$.

JavaDL semantics. JavaDL formulas are interpreted in Kripke structures, which we define in the following.

Definition 2.8 (Kripke structure)**.** For a JavaDL signature $\Sigma = (\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$, we define a Kripke structure as a tuple (D, δ, M, S, ρ) consisting of:

- a set D of semantic values,
- a dynamic typing function $\delta : D \rightarrow \tau$ which gives rise to the subdomains $D^A = \{x \in D | \delta(x) \preceq A\}$ for all types $A \in \tau$,
- an interpretation function I mapping every function symbol $f : A_1, \ldots, A_n \rightarrow A \in F$ to a function $I(f) : D^{A_1}, \ldots, D^{A_n} \rightarrow D^A$ and every predicate symbol $r : A_1, \ldots, A_n \in P$ to a relation $I(r) \subseteq D^{A_1} \times \ldots \times D^{A_n}$,
- a set S of states, which are functions $s \in S$ mapping every program variable $a \in PV$ of type A to a value $s(a) \in D^A$, and

[1] Some exceptions apply, see Definition 5.2 in [Weiß, 2011, Chapter 5] for more details.

- a function ρ that associates with every program fragment $\mathtt{p}$ a transition relation $\rho(\mathtt{p}) \subseteq S \times S$ such that $(s_1, s_2) \in \rho(\mathtt{p})$ if and only if $\mathtt{p}$, when started in s_1, terminates normally (i.e., by not throwing an exception) in s_2. We consider Java programs to be deterministic, so for all program fragments $\mathtt{p}$ and all $s_1 \in S$ there is at most one s_2 such that $(s_1, s_2) \in \rho(\mathtt{p})$.

Note that the special symbols used to formalize certain aspects of Java have a fixed interpretation which is given in [Weiß, 2011, Chapter 5]. Also as done in [Weiß, 2011, Chapter 5] we do not give the semantics of Java here, but instead consider the transition function ρ to be a black box that captures the behavior of every program fragment $\mathtt{p}$. The behavior of program fragments is explicitly formalized in the JavaDL calculus that is presented at the end of this section.

Definition 2.9 (JavaDL semantics)**.** Given a JavaDL signature $\Sigma = (\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$, a Kripke structure $K = (D, \delta, M, S, \rho)$, a state $s \in S$, and a variable assignment $\beta : V \to D$ which maps every variable x of type A to a value in D^A, we evaluate every term $t \in Term^A_\Sigma$ to a value $val_{K,s,\beta}(t) \in D^A$, every formula $\varphi \in Formula_\Sigma$ to a truth value $val_{K,s,\beta}(\varphi) \in \{\mathbf{0}, \mathbf{1}\}$, and every update $u \in Update_\Sigma$ to a state transformer function $val_{K,s,\beta}(u) : S \to S$ such that[2]:

- $val_{K,s,\beta}([\mathtt{p}]\varphi) = \mathbf{1}$, iff $\mathbf{0} \notin \{val_{K,s',\beta}(\varphi) | (s, s') \in \rho(\mathtt{p})\}$,
- $val_{K,s,\beta}(\langle\mathtt{p}\rangle\varphi) = \mathbf{1}$, iff $\mathbf{1} \in \{val_{K,s',\beta}(\varphi) | (s, s') \in \rho(\mathtt{p})\}$,
- $val_{K,s,\beta}(\{u\}\varphi) = val_{K,s',\beta}(\varphi)$, where $s' = val_{K,s,\beta}(u)(s)$,
- $val_{K,s,\beta}(a := t)(s')(b) = \begin{cases} val_{K,s,\beta}(t), & \text{if } b = a \\ s'(b), & \text{otherwise} \end{cases}$
  for all $s' \in S, b \in PV$,
- $val_{K,s,\beta}(u_1 \parallel u_2)(s') = val_{K,s,\beta}(u_2)(val_{K,s,\beta}(u_1)(s'))$ for all $s' \in S$, and
- $val_{K,s,\beta}(\{u_1\}u_2) = val_{K,s',\beta}(u_2)$ where $s' = val_{K,s,\beta}(u_1)(s)$.

JavaDL provides two modal operators: the box operator $[\mathtt{p}]$ and the diamond operator $\langle\mathtt{p}\rangle$, where $\mathtt{p}$ is a program fragment. The JavaDL formula $[\mathtt{p}]\varphi$ is true in a state s if the formula φ is true in all states s' in which $\mathtt{p}$ may terminate when started in s. Because the Java programs we consider are deterministic, they can have at most one state in which they terminate. Thus the formula formula $[\mathtt{p}]\varphi$ states that the program fragment $\mathtt{p}$, when executed in state s, either (1) terminates in a state s' in which φ holds, or (2) does not terminate. The JavaDL formula $\langle\mathtt{p}\rangle\varphi$, on the other hand, is true in a state s if the program fragment $\mathtt{p}$, when started in state s, does terminate in at least one state s' in which φ holds, and—since we only consider deterministic programs—if the state s' exists, it is uniquely determined by s and $\mathtt{p}$.

[2] Purely first order terms and formulas are evaluated as in first order logic, refer to Definition 5.5 in [Weiß, 2011, Chapter 5] for the complete semantics of JavaDL.

JavaDL also provides an *update operator*. Updates describe changes that are to be applied to a state. They are similar to substitutions in purpose, but are part of the logic. They are used to describe the changes produced by a program statement. They are accumulated when processing the program statement-by-statement, using the calculus rules in a process called *symbolic execution*. At the end of the symbolic execution, accumulated updates can be simplified using special calculus rules before being applied to the final state, thus simplifying the applied substitutions.

JavaDL calculus. We present a *sequent calculus* (introduced by Gentzen [1935]) that allows us to prove the validity of JavaDL formulas. In the following we present the definition of a sequent.

Definition 2.10 (Sequent)**.** A sequent is a pair $(\Gamma, \Delta) \in 2^{Formula_\Sigma} \times 2^{Formula_\Sigma}$ where Γ (called the *antecedent*) and Δ (called the *succedent*) are finite sets of formulas. We denote a sequent (Γ, Δ) as $\Gamma \Rightarrow \Delta$ and also use the notation $\psi_1, \ldots, \psi_n \Rightarrow \varphi_1, \ldots, \varphi_m$ where $\Gamma = \{\psi_1, \ldots, \psi_n\}$ and $\Delta = \{\varphi_1, \ldots, \varphi_m\}$. The set of all sequents is denoted as Seq_Σ. The semantics of a sequent is $val_{K,s,\beta}(\Gamma \Rightarrow \Delta) = val_{K,s,\beta}(\bigwedge \Gamma \rightarrow \bigvee \Delta)$.

The calculus is composed of rules, which we define in the following.

Definition 2.11 (Rule)**.** A rule is a binary relation $r \subseteq Seq^*_\Sigma \times Seq_\Sigma$. If $((p_1, \ldots, p_n), c) \in r$, we say that the conclusion c is derivable from the premisses $(p_1, \ldots, p_n)$ using r.
A rule r is *sound* if the following holds for all $((p_1, \ldots, p_n), c) \in r$: if all premises $p_1, \ldots, p_n$ are logically valid, then c is also logically valid.

Starting with a single root sequent, also called *proof obligation*, a *proof tree* is constructed by applying rules on the leaves of the proof tree. We can apply a rule on a leaf, if the leaf matches the conclusion of the rule, by adding the premises of the rule as children of that leaf. There are special rules called *axioms* or *closing rules*, which have no premises. Applying a closing rule on a leaf of the proof tree closes the proof tree branch of that leaf. Thus, the sequent calculus proves the validity of a proof obligation by showing that it can be derived—using sound calculus rules—from a set of axioms. In the following we present examples of (simplified) calculus rules, and we refer to Schmitt [2016] and Beckert et al. [2016b] for the rest.

$$\frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta}\ \text{andLeft} \qquad \frac{}{\Gamma \Rightarrow \mathit{true}, \Delta}\ \text{trueRight}$$

We present two first-order rules. The rule *andLeft* removes a conjunction $\varphi \wedge \psi$ from the antecedent and adds the two formulas—φ and ψ—to the antecedent. Note that Γ and Δ do not represent a single formula, but all

other formulas in the antecedent and, respectively, the succedent. The rule *trueRight* is an example of a closing rule. It states that if *true* is present among the formulas of the succedent, the sequent is valid.

Because JavaDL formulas can contain program fragments, special calculus rules are used to symbolically execute them. During the symbolic execution process, the program in a JavaDL formula is processed statement-by-statement, and the statements are replaced by case distinctions and updates. In the following we present some symbolic execution rules.

$$\frac{\Gamma \Rightarrow \{u\}\varphi, \Delta}{\Gamma \Rightarrow \{u\}[\ ]\varphi, \Delta}\ \text{emptyBox} \qquad \frac{\Gamma \Rightarrow \{u\}\{v{:=}t\}[\omega]\varphi, \Delta}{\Gamma \Rightarrow \{u\}[\texttt{v=t;}\omega]\varphi, \Delta}\ \text{assignment}$$

The *emptyBox* (or the similar *emptyDiamond*) rule can be used at the end of the symbolic execution, when the JavaDL formula contains an empty box (or diamond) modal operator, which can be discarded. Assignments to a local variable `v` of a side-effect-free expression `t` are handled by the *assignment* rule. This rule transforms an assignment in the Java program into an update and then removes the assignment from the program in the JavaDL formula, thus simplifying it.

$$\frac{\Gamma, \texttt{c} \Rightarrow \{u\}[\texttt{p1;}\omega]\varphi, \Delta \qquad \Gamma \Rightarrow \texttt{c}, \{u\}[\texttt{p2;}\omega]\varphi, \Delta}{\Gamma \Rightarrow \{u\}[\texttt{if(c)\{p1\}else\{p2\};}\ \omega]\varphi, \Delta}\ \text{ifElseSplit}$$

Another example of a rule for symbolic execution, that is used to handle branching, is the *ifElseSplit* rule. Note that the condition `c` of the `if` statement is assumed to be a boolean program variable, otherwise the condition is unfolded and symbolically executed before the *ifElseSplit* rule is applied. The rule splits the proof tree for the two cases of the `if` statement. In the first case, it is assumed that the condition `c` is true, and the symbolic execution continues with the program fragment `p1`. In the second case, it is assumed that the condition `c` is false, and the symbolic execution continues with the program fragment `p2`.

$$\frac{\Gamma \Rightarrow \{u\}[\texttt{if(c) \{b; while(c) \{b\}\};}\omega]\varphi, \Delta}{\Gamma \Rightarrow \{u\}[\texttt{while(c) \{b\};}\ \omega]\varphi, \Delta}\ \text{loopUnwind}$$

The *loopUnwind* rule is an example of calculus rule that is used to symbolically execute loops. This rule—as shown here—works on loops without any `break` or `continue` statements and unwinds a `while` loop once. The rule is useful only for loops for which the number of iterations is bounded.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathit{inv}, \Delta \\ \Gamma \Rightarrow \{u\}\{a\}(\mathit{inv} \wedge \texttt{c}) \rightarrow [\texttt{b}]\mathit{inv}, \Delta \\ \Gamma \Rightarrow \{u\}\{a\}(\mathit{inv} \wedge \neg\texttt{c}) \rightarrow [\omega]\mathit{inv}, \Delta\end{array}}{\Gamma \Rightarrow \{u\}[\texttt{while(c) \{b\};}\ \omega]\varphi, \Delta}\ \text{loopInvariant}$$

For unbounded loops, the *loopInvariant* rule must be used instead to apply a loop invariant *inv*, which is provided by the user. This rule adds three new sequents to the branch of the proof tree respectively stating that—from top to bottom—(1) the loop invariant is true before entering the loop, (2) the loop invariant is true after each loop iteration, and (3) the loop invariant can be used to describe the state after the loop. The update $\{a\}$ is an *anonymizing* update that assigns undefined values to all variables that may be changed by the loop. The only information about those variables, that can be used in the rest of the proof, must be contained in the loop invariant *inv*.

2.2.2 The Theorem Prover KeY

KeY (see Ahrendt et al. [2016a]) is a theorem prover that works on JavaDL and uses the sequent calculus presented in Section 2.2.1.

For functional properties (i.e., properties that consider a single program execution) a *total correctness* proof obligation has the form $\Rightarrow \mathit{Pre} \rightarrow \langle \mathtt{p} \rangle \mathit{Post}$, meaning that the program `p`, when executed in a prestate in which the precondition *Pre* holds, terminates in a poststate in which the postcondition *Post* will hold. On the other hand, *partial correctness*, as expressed by the proof obligation $\Longrightarrow \mathit{Pre} \rightarrow [\mathtt{p}] \mathit{Post}$, means that either (1) the program terminates, and the postcondition *Post* holds in the poststate, or (2) the program never terminates.

During proof construction, the program is symbolically executed using the appropriate rules for Java programs of the JavaDL calculus (see Section 2.2.1). Those rules are applied automatically according to a strategy provided by KeY. After the entire program has been symbolically executed, KeY attempts to show the validity of the remaining sequents, which contain only first order formulas. The proof of those sequents can often be done automatically with the automatic strategy of KeY. However, in some cases user interaction is required (e.g., to instantiate a quantifier or perform induction). For such cases, KeY provides a graphical user interface to assist the user.

2.2.3 JML

KeY does not take a JavaDL formula directly as an input, but instead transforms a specified program into a JavaDL proof obligation. The properties of the Java program that is to be verified are specified in the form of method contracts and auxiliary specifications (e.g., loop invariants) using an extension of the Java Modeling Language (JML), introduced by Leavens et al. [2007]. The JML specification language used by KeY is described in Huisman et al. [2016].

We present the main features of JML using an example, shown in Listing 2.1. The method `arrCopy` copies the contents of the integer array `a` into

the array `b`. The method is specified with a JML method contract which contains a precondition and a postcondition. The method contract states that if the precondition holds when the method is called, the postcondition holds after the execution of the method.

The JML clause `normal_behaviour` states that the program terminates normally (i.e., without throwing an exception). The `requires` clause introduces the precondition of the method contract which requires `a` and `b` to not be aliases of each other and to have the same length. Java expressions without side effects (which are valid in the context of the program) can be used in the JML specification.

The `ensures` clause introduces the postcondition of the method contract, which requires the two arrays—`a` and `b`—to have the same length and the same values at each index i after the execution of the method `arrCopy`. In this example the postcondition uses the JML quantifier `\forall`. Besides the universal and existential ones, quantifiers for expressing the minimum, maximum, sum, product, and number of elements of a sequence are supported. For non-void methods the keyword `\result` can be used to refer to the value returned by the method.

The method contract also contains an `assignable` clause which specifies the heap locations that may be changed by the method. For the example in Listing 2.1 the method may change the elements of array `b`.

```
1  public class ArrayUtils {
2  /*@ public normal_behavior
3    @ requires a.length == b.length && a != b;
4    @ ensures (\forall int i; 0<=i && i<a.length; a[i]==b[i
         ]) && a.length==b.length;
5    @ assignable b[*];
6    @*/
7  public void arrCopy(int[] a, int[] b) {
8   /*@ loop_invariant 0 <= i && i <= a.length;
9     @ loop_invariant a != b && a.length == b.length;
10    @ loop_invariant (\forall int j; 0<=j && j<i;
11                          a[j]==b[j]);
12    @ decreases a.length - i;
13    @ assignable b[*];
14    @*/
15    for(int i=0; i<a.length; i++) {
16       b[i]=a[i];
17    }
18 }
19 }
```

Listing 2.1: Example of a specified Java program

The loop inside the method `arrCopy` is specified with a loop invariant. The formulas that constitute the loop invariant are introduced with the `loop_invariant` clause. The `decreases` clause introduces the *loop variant*, which is needed to prove the termination of the loop (if the user wants to prove total correctness). The loop variant is an integer expression whose

value is strictly decreased after each loop iteration, but remains nevertheless positive. Assignable clauses can also be used in loop invariants.

2.3 Proving Noninterference Properties with KeY

KeY was extended by Beckert et al. [2013] to support the verification of noninterference properties for sequential programs. In this section we give an overview of that extension, which is needed to understand Chapters 4, 6, and 7. Parts of this section are based on Herda et al. [2019c]. A complete reference to the extensions of KeY for the specification and verification of noninterference properties can be found in Scheben [2014] and Scheben and Greiner [2016].

2.3.1 Proof Obligations

The formalization of noninterference in KeY uses an adaptation of a technique called *self-composition*, which was introduced by Barthe et al. [2004]. Using self-composition, the noninterference property of a program `p` is translated to a functional property of a new program `p;p'` which consists of `p` composed with a renaming of itself. When proving a specified noninterference property for a given program, KeY generates two proof obligations. Besides the noninterference proof obligation, some functional properties, such as the validity of class invariants, are discharged into a separate, functional proof obligation, which we will not discuss in this section. In the following we present the noninterference proof obligations that KeY generates for the properties defined in Section 2.1.

Classical noninterference. The noninterference proof obligation for Definition 2.2 expresses that two program runs that start in low-equivalent states will also terminate in low-equivalent states, and it has the following simplified form:

$$
\begin{aligned}
\Rightarrow \Big(& \underbrace{(\{in_l := in_l^A\}[\mathtt{p}]\, out_l = out_l^A)}_{\text{Execution A}} \\
\wedge\ & \underbrace{(\{in_l := in_l^B\}[\mathtt{p}]\, out_l = out_l^B)}_{\text{Execution B}} \Big) \\
\rightarrow \Big(& \underbrace{(in_l^A = in_l^B) \rightarrow (out_l^A = out_l^B)}_{\substack{\text{Low-equivalent prestates imply} \\ \text{low-equivalent poststates}}} \Big)
\end{aligned}
$$

In the above proof obligation in_l and out_l are placeholders that respectively represent the low input and the low output of program `p`. The low inputs and outputs of a program are specified by the user, as shown Section 2.3.2. The proof obligation contains in the left hand side of the main implication

two executions of p. These are execution A, which starts with the low input in_l^A, and returns the low output out_l^A and execution B, which starts with the low input in_l^B, and returns the low output out_l^B. On the right hand side of the main implication the proof obligation contains a second implication stating that if the two low inputs in_l^A and in_l^B are equal (i.e., they are low-equivalent), then the low outputs out_l^A and out_l^B are also equal (i.e., they are also low-equivalent). Note that for simplicity we used the placeholders in_l^A, in_l^B, out_l^A, and in_l^B to represent the (universally quantified) inputs and outputs of executions A and B.

The proof obligation can be handled with the JavaDL calculus presented in Section 2.2. Optimizations have been built in KeY (see Scheben [2014]) to avoid symbolically executing the same program twice by reusing the results of the first symbolic execution.

Noninterference with declassification. To support declassification as defined in Definition 2.3, the second implication of the proof obligation also requires that the declassified expressions are equal in the prestates of the two executions:

$$(in_l^A = in_l^B \land expr^A = expr^B) \rightarrow (out_l^A = out_l^B)$$

The proof obligation above shows that declassification expressions can be considered as additional low inputs of the analyzed program and they can be specified as such, as we will show in Section 2.3.2.

Low-equivalence with isomorphism. To support object isomorphism as defined in Definition 2.4, the second implication of the proof obligation is enhanced with the predicate *newObjIso*:

$$(in_l^A = in_l^B) \rightarrow (newObjIso(L, heap_{out}^A, heap_{out}^B) \rightarrow out_l^A = out_l^B)$$

The predicate *newObjIso* takes as arguments a list L of reference type expressions, and the two heaps $heap_{out}^A$ and $heap_{out}^B$ that respectively represent the Java heap memory in the two poststates of executions A and B. The predicate is true under the following conditions:

1. Every expression in L is newly created after executions A and B.
2. At every position i in the list L, the type of $L[i]$ is the same after executions A and B.
3. If the reference expressions at two positions i and j in L are equal after execution A then they must be equal after execution B as well.

These three requirements ensure that the reference expressions in L are isomorphic in the two poststates. Note that the list L is provided by the user as part of the noninterference specification.

2.3.2 Extensions to JML

The JML specification language (see Section 2.2.3) was extended in [Scheben, 2014, Chapter 4] to support the specification of noninterference properties. This is done by specifying the low program parts in the prestate and poststate of the program and requiring low program parts in the poststate to depend at most on the low program parts in the prestate. We present the JML extensions for noninterference using the example methods from Listing 2.2.

```
1  /*@ requires n >= 0;
2    @ determines \result \by low, n;
3    @*/
4  public int foo(int low, int n, int high) {
5     int l = low;
6   /*@ loop_invariant n >= 0;
7     @ determines l, n \by \itself;
8     @ decreases n;
9     @*/
10    while(n > 0) {
11       l += n*high-n*high;
12       n = n-1;
13    }
14    return l;
15 }
16
17 /*@ determines \result \by \nothing
18   @            \declassifies (h1+h2)/2;
19   @*/
20 public int avg(int h1, int h2) {
21    return (h1+h2)/2;
22 }
23
24 /*@ requires l == 0;
25   @ determines \exception,  \by l
26   @            \new_objects \exception;
27   @*/
28 public int div(int  h, int l){
29    return h/l;
30 }
```

Listing 2.2: Example programs specified with the JML extensions for nonintereference

The methods `foo`, `avg`, and `div` in Listing 2.2 are specified with noninterference method contracts, which refer to two program executions. This is different than the functional method contracts presented in Section 2.2.3, which refer to a single program execution. Similar to functional method contracts, a `requires` clause (e.g., lines 1 and 24) can be used to express a precondition. This precondition must hold at the beginning of both executions for the contract to be applicable. A `determines` clause (e.g., lines 2, 17, and 25) consists of two lists L_{out} and L_{in} of JML expressions which are separated by the keyword `\by`. The expressions L_{out}, which are written before `\by` and represent the low output, are allowed to depend at most on the expressions L_{in}, which are written after `\by` and represent the

low input. More precise, the specified method when executed in two prestates that agree[3] on the expressions in L_{in} terminates in a two poststates which agree on the expressions in L_{out}. For example, the `determines` clause of the method `foo` states that the return value of the method depends at most on the parameters `low` and `n`. The keyword `\nothing` (e.g. at line 17) denotes an empty list of expressions, and the keyword `\itself` (e.g. at line 7) can be used to express that L_{out} contains the same expressions as L_{in}.

As shown in line 7, `determines` clauses can also be used in loop invariants. Noninterference loop invariants must contain the same expressions in L_{out} and L_{in} (i.e., L_{in} must be `\itself`), and they require that (1) the states before entering the loop in each of the two executions agree on the expressions in the list and on the loop condition and (2) assuming the states of the two executions agree on the list expressions and loop condition before an iteration, they must do so after that iteration as well. If the two requirements hold, then the loop invariant can be used in the rest of the proof. The lists L_{out} and L_{in} must have the same elements in order for the induction argument that is made with the loop invariant to be valid. Note that such loop invariants can only be used when the loop condition is low, otherwise the user must show through functional loop invariants that all paths that depend on a high variable lead to the same low result. See [Scheben, 2014, Chapter 7] for more on noninterference loop invariants.

A `determines` clause can be extended with a list of declassification expressions, introduced with the keyword `\declassifies`, as was done in line 18. The method `avg` computes the average of the two high parameters `h1` and `h2`, and the noninterference contract states that the result declassifies the average of `h1` and `h2`, but otherwise provides no further information on the two parameters. Note that the `\declassifies` keyword is syntactic sugar and the declassification expressions can be written as elements of L_{in}.

A `determines` clause can be extended with a list of expressions of reference type, introduced with the keyword `\new_objects` which are required to be newly created and isomorphic in the two end states (instead of being required to be equal). The method `div` divides the high parameter `h` by the low parameter `l` which may result in an exception if `l` is 0. The noninterference contract states that for the case in which l is 0, the thrown exception—denoted with the keyword `\exception`—depends at most on `l`. Because the thrown exception is an object that is created during the execution of the method, it cannot be compared using equality in the poststates of the two executions, as the generated reference addresses may different. Using the keyword `\new_objects`, we can express that the thrown exception does not leak any information about the parameter `h`.

[3] This means that for each index i in the list, the expression at index i evaluates to the same value in both states.

2.4 Automatic Test Generation with KeY

The KeY theorem prover was extended in Engel and Hähnle [2007] with the ability to automatically generate tests for functional properties. In this Section, which is needed to understand chapters 4 and 6, we give an overview of KeYTestGen, the current extension of KeY for automatic test generation. This section is based on Ahrendt et al. [2016b], where the test generation features of KeY are fully documented.

2.4.1 Tests and Coverage Criteria

We present what kind of tests and test suites can be generated by KeYTestGen along with the coverage criteria that the generated test suites achieve.

Tests and test suites. A *test* of a program `p` can formally be described as a tuple $\langle s, Or \rangle$ consisting of a state s and an *oracle* Or. The state s, also called *test data* or *test input*, serves as an input for `p`. The oracle is a function $Or(s') \mapsto \{pass, fail\}$, telling for each poststate s' obtained by executing the program with a test input s whether those are the expected results of respective test execution. A *test suite* $TS^{\mathtt{p}}$ for a program `p` consists of n test cases:

$$TS^{\mathtt{p}} = \langle \{s_1, \ldots, s_n\}, Or^{\mathtt{p}} \rangle$$

where $\{s_1, \ldots, s_n\}$ is the set of test data, and $Or^{\mathtt{p}}$ is the oracle for `p`.

The approach for automatic test generation presented in Section 2.4.2 automatically assembles a JUnit test suite from a given Java program specified with a functional JML method contract (see Section 2.2.3). The generated test suite is formally guaranteed to satisfy certain coverage criteria which are explained in the following.

Test coverage criteria. To measure the adequacy of a test suite, different types of test coverage criteria have been introduced; an overview thereof can be found in Zhu et al. [1997]. Some of the most prominent test coverage criteria[4] are based on the *control flow graph* (CFG)[5] of a program, which we define in the following, by adapting the definition from Ranganath et al. [2007].

Definition 2.12 (Control flow graph)**.** Given a program `p`, a control flow graph (CFG) $G = (N, E, n_{start}, n_{end})$ is a labeled directed graph in which

- N is a set of nodes that represent the instructions in the program,

[4] CFG-based coverage criteria were introduced in White [1981].
[5] The CFG was introduced in Allen [1970]

- The set N is partitioned in two subsets N^S and N^P where N^S are statement nodes with each $n_S \in N^S$ having at most one successor, and where N^P are predicate nodes with each $n_P \in N^P$ having two successors,
- E is a set of labeled edges that represent the control flow between graph nodes where each $n_p \in N^P$ has two outgoing edges labeled T and F respectively, and each node $N^S \setminus \{n_{end}\}$ has an outgoing edge labeled A (representing *always taken*),
- the end node $n_{end} \in N^S$ has no successors and is reachable from all other nodes in N, and
- the start node n_{start} has no incoming edges and all nodes in N are reachable from n_{start}.

A path in the CFG of a program `p` from the start node to the end node represents a possible execution of `p`. Note that not every CFG-path corresponds to a real program execution, the CFG represents an over-approximation of the possible program behavior. The reverse is, however, true, each program execution has a corresponding CFG-path. Also note that CFG-paths between predicate nodes or between n_{start} and n_{end} consisting only of statement CFG-nodes are referred to as *basic blocks*. We now define the statement, branch, and label coverage of a test suite.

Definition 2.13 (Statement, branch, path coverage)**.** For a given program `p`, its CFG and a test suite $TS^{\texttt{p}}$ we define the statement, branch and path coverages achieved by $TS^{\texttt{p}}$ as follows:

- Statement coverage: $\frac{\#(\text{covered nodes})}{\#(\text{nodes})}$
- Branch coverage: $\frac{\#(\text{covered edges})}{\#(\text{edges})}$
- Path coverage: $\frac{\#(\text{covered paths})}{\#(\text{paths})}$

A node, edge, or path in the CFG is considered *covered* if they are contained (in the case of paths equal to) by at least one CFG-path that corresponds to an execution of the program determined by $TS^{\texttt{p}}$. Note that $\#(nodes)$, $\#(edges)$, and $\#(paths)$ denote the number of CFG-nodes, CFG-edges, and CFG-paths respectively.

Because the CFG can contain cycles caused by loops in the program, the number of CFG-paths may be infinite. Since we cannot have a test suite consisting of an infinite number of tests, we define the *bounded path coverage criterion*. For this, we only consider b-paths, which are CFG-paths in which loops are iterated up to a maximum number of b times, where b is a bound provided by the user. Thus, the bounded path coverage criterion is defined as follows.

Definition 2.14 (Bounded path coverage)**.** For a given program p, its CFG, a test suite $TS^{\mathtt{p}}$, and a bound b we define the bounded path coverage criterion as:

$$\frac{\#(\text{covered b-paths})}{\#(\text{b-paths})}$$

Definition 2.15 (Path condition)**.** Given a program p and its CFG, a path condition of a CFG-path π is a first order formula F on the input s of p such that s satisfies F iff π is the CFG-path corresponding to the execution of p with input s.

Path conditions of CFG-paths can be constructed by analyzing the effects of the statement nodes and the conditions of the predicate node. If the program is in SSA form (see Cytron et al. [1991]), then the statements can be treated as equalities. Then, the path condition of a path is the conjunction of those equalities and of the conditions of the predicate nodes, which are negated if and only if the outgoing edge with label F is on the path. If a CFG-path cannot be executed because its path condition is unsatisfiable, then the path is called *infeasible*. Otherwise, the CFG-path is *feasible*. Note that it is an undecidable problem whether a CFG-path is feasible. Thus, if a test suite achieves a certain coverage, we do not know by how much that coverage can be improved, because the parts of the program that were not tested may lie on infeasible paths. We define the feasible bounded path coverage criterion in the following.

Definition 2.16 (Feasible bounded path coverage)**.** For a given program p, its CFG, a test suite $TS^{\mathtt{p}}$ and a bound b we define the bounded path coverage criterion as:

$$\frac{\#(\text{covered b-paths})}{\#(\text{feasible b-paths})}$$

Obviously, only a feasible CFG-path can be covered by the a test execution.

2.4.2 Generating Tests with KeYTestGen

Software testing involves the creation and execution of tests. While the execution of tests is nowadays largely automatized through frameworks such as JUnit (see Beck [2004]), approaches for the automatic test creation have not been widely adopted yet. Nevertheless, approaches to automatic test generation have been proposed, see Anand et al. [2013] for a survey of such approaches. In this section we introduce KeYTestGen, which is an approach for automatically generating a test suite for a given program. KeYTestGen uses symbolic execution and attempts to generate a test suite that achieves full feasible bounded path coverage.

As explained in Ahrendt et al. [2016b], approaches used for generating test data can be divided in two main categories:

- *White-box testing:* when the generation of test data is based on analyses of program code.
- *Black-box testing:* when the generation of test data is based on external descriptions of the software (e.g., specification, design documents, requirements, probability distributions).

KeYTestGen is a hybrid of these two categories. Its generation of the test data is mainly white-box, with elements of black-box. It is based on a thorough analysis of the source code, but also on the preconditions from the specification. KeYTestGen treats the generation of oracles entirely in black-box fashion, so that the oracles do not to inherit errors from the implementation.

In the following we present the steps taken by KeYTestGen to automatically generate tests.

Step 1: constraints generation. The input to KeYTestGen is a Java method under test (MUT), with a specified JML method contract. In the first step, KeYTestGen loads the proof obligation for the specified MUT (as is done when proving the validity of the contract, see Section 2.2.2). KeYTestGen then symbolically executes the program one statement after the other (again, as in the case of proving the validity of the contract). Java code is turned into updates, which are a compact representation of the effect of the statements. The branches of the proof tree mimic the execution of the program with symbolic values (i.e., expressions over variables). Case distinctions (including implicit distinctions like, 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 bound b provided by the user. Loops in the program are handled with the loopUnwind calculus rule (see Section 2.2.1), which can be applied a maximum of b times for each loop. The proof tree that is obtained this way[6] has as many branches as there are b-paths in the CFG of the program, and each branch in the proof tree represents a b-path in the CFG. At the end of symbolic execution a model (i.e., first-order logic interpretations satisfying a formula) of a leaf of the proof tree is also a model of the precondition and of the path condition of the b-path determined by that proof tree branch.

Step 2: test data generation. A *path condition*, together with the *precondition* from the specification, constitute a *test data constraint*, which has to be satisfied by the test data of a test for this path. To create a test, a concrete test input s must be generated which satisfies the test data constraint obtained from the first step. This task is handled by the *model*

[6] Assuming only symbolic execution calculus rules were used.

generator. The challenge of model generation in the context of KeYTestGen is to generate models for quantified formulas which may stem from the requirement specification or from the logical modeling of the heap in JavaDL. Currently the third party SMT solver z3 (see de Moura and Bjørner [2008]) is used to find models for test data constraints. Constraints are translated from KeY's Java first-order logic (i.e., first order logic with interpreted symbols for formalizing aspects of Java programs, see Schmitt [2016]) to the SMT-LIB 2 language (see Barrett et al. [2010]), which is supported by most SMT solvers.

The translation from Java first order logic to SMT-LIB poses two challenges. First, it must be ensured that the model found by the SMT solver is also a model for the original Java first order logic formula. Second, it must be ensured that the SMT solver is able to find a model within a reasonable amount of time. Unfortunately, the current state of the art does not allow KeYTestGen to fully address both objectives. For this reason bounded data types (i.e., each data type can have only a bounded number of instances) are used. As a consequence, the SMT solver can find models a lot faster, but at the same time some models may be missed because the bounds might be too small. For formulas which are valid on infinite domains (such as the mathematical integers) some spurious models can be found because the bounds are finite. This issue is discussed in [El Ghazi, 2015, Chapter 6].

Step 3: code generation In the third and final step, the tests are generated. Each test consists of a *test preamble*, a call of the MUT, and a call to the test oracle. The test preamble prepares the inputs for the MUT call. The inputs are taken from the model which was found in the previous step. The MUT call in the test is the same as in the modal operator of the KeY proof obligation.

The generated test oracle is a boolean method that returns true if the test satisfies the JML specification of the MUT and false otherwise. It checks whether the postcondition holds after the MUT was executed. The oracle is not generated directly from the JML specification, but rather from the proof obligation in KeY . This is because KeY may include some implicit class invariants and termination conditions as part of the postcondition. The precondition does not need to be checked, because it is part of the test data constraint and is always satisfied by the process of test input data generation. Each test suite contains only one oracle method which is used in all tests. In each test, after running the MUT, it is asserted that the test oracle method returns true by using the JUnit method `assertTrue`.

2.5 Program Slicing

In this section we introduce *program slicing* and give an informal definition thereof. This section is based on Beckert et al. [2017b] and is needed to understand Chapter 8 and also Section 2.6.

Program slicing, a notion coined by Weiser [1981], is a technique to reduce the size of a program by removing statements from it such that a part of its behavior remains the same. Different kinds of slicing approaches have been developed since then (see e.g., the survey in De Lucia [2001]). A *static slice* preserves the behavior for all inputs, while a *dynamic slice* preserves it only for a single input. A *backward slice* keeps only those parts of the program that influence certain variables at a certain location in the program, while a *forward slice* keeps those parts which are influenced by those variables at that location. The form of slicing introduced by Weiser is now known as *static backward slicing*. This is the form of slicing which is pursued by the framework for program slicing presented in Part III of this thesis. Slicing techniques can be used to optimize the results of compilers (see e.g., Ferrante et al. [1987]). Slicing is also a powerful tool for challenges in software engineering such as code comprehension, debugging, refactoring, and fault localization (see e.g., Binkley and Harman [2004]), as well as in information flow security (see e.g., Hammer and Snelting [2009] and Section 2.6).

Slice candidates and slices. A *slice candidate* is a variant of the original program where zero or more statements have been replaced with the side-effect-free `skip` statement. A slice candidate is considered a *valid slice* if, given the same input to the slice candidate and original program, the following two conditions hold:

1. During execution of the slice candidate and the original program, respectively, the location specified in the slicing criterion is reached for the same number of times.

2. When the location is reached for the ith time in the original program and for the ith time in the slice ($i \geq 1$), each variable specified in the slicing criterion has the same value in the original program's state and in the slice's state.

A formal semantics of slicing is presented in Section 8.2, a syntactic approach for program slicing which is used to prove a specified noninterference property of a given program is shown in Section 2.6.

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        x = h;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

(a)

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        x = 42;
9      i++;
10   }
11   return x;
12 }
```

(b)

```
1  int f(int h, int N){
2    int i = 0;
3    int x = 0;
4    while(i < N) {
5      if(i < N - 1)
6        skip;
7      else
8        skip;
9      i++;
10   }
11   return x;
12 }
```

(c)

Figure 2.3: (a) Original program, (b) Slice with respect to variable x at line 8, (c) Incorrect slice candidate

Example. Figure 2.3 shows an example of static backward slicing. The goal is to slice the C routine in Figure 2.3a with respect to a slicing criterion, which requires the value of x at the statement in line 8 to be preserved. A valid slice for this criterion is shown in Figure 2.3b: The assignment in line 6 of the program has been replaced with a side-effect-free `skip` statement to keep the program similar structure to the input program. This line has no effect on the value of x, as it is always set to 42 in the last loop iteration.

2.6 Using Dependence Graphs for Proving Noninterference

In this section we present the program dependence graph (PDG), the system dependence graph (SDG), and we show how they can be used for slicing and proving noninterference. This section is based on [Hammer, 2009, Chapter 2] and is needed to understand Chapters 3, 6, 7, and 8.

2.6.1 Dependence Graphs

In the following we present the PDG (introduced by Ferrante et al. [1987]), which is used to model possible dependencies between program statements in intraprocedural programs and the SDG (introduced by Horwitz et al. [1990]), which is the extension of the PDG for interprocedural programs.

The program dependence graph (PDG). The PDG is defined using the CFG (see Definition 2.12). All CFG-nodes are also PDG-nodes, and the PDG can contain additional nodes that represent input parameters. Edges in the PDG represent possible dependencies between the nodes (i.e., an edge between nodes exists if the value or execution of a node may depend on the outcome of the other node). There are roughly two types of edges in a PDG: data dependency edges, which represent possible direct dependencies and control dependencies, which represent possible indirect dependencies. Whether an edge exists between two nodes in the PDG is determined syntactically by analyzing the CFG of the given program. An overview of formal definitions of the two types of dependencies can be found in Wasserrab and Lohner [2012]. We present the standard ones here.

To define data dependencies we define for each CFG-node n the set $Def(n)$ that contains all variables which are defined (assigned to) in n and the set $Ref(n)$ that contains all variables which are referenced in n.

Definition 2.17 (Data dependency)**.** A PDG-node y is data dependent on a PDG-node x if

- there exists a variable v with $v \in Def(x)$ and $v \in Ref(y)$, and
- there exists a path π in the CFG from x to y on which the definition of v in y is not definitely killed (i.e., v is not redefined).

To define control dependencies we need to define the post-domination relation between two CFG-nodes.

Definition 2.18 (Post-domination)**.** Given the CFG of a program, a CFG-node x is *post-dominated* by a CFG-node y if all paths from x to n_{end} pass through y.

We can now define control dependencies.

Definition 2.19 (Control dependency)**.** Given the PDG of a program, a PDG-node y is control dependent on a PDG-node x if

- there exists a path π from x to y in the CFG such that y post-dominates every node in π (except for x), and
- x is not post dominated by y.

Intuitively, a node y is control-dependent on a node x if the choice of the outgoing edge from x in the CFG determines whether node y is reached.

Note that it is undecidable whether a CFG-path is realizable during the execution of the program (i.e., some paths in the CFG represent executions that cannot actually take place). Thus, the CFG is an over-approximation of the program behavior. Since the dependencies are defined using CFG-paths, they too are an over-approximation of the actual dependencies in the program. Because we distinguish between actual and possible program dependencies, we define in the following what we mean by actual (or real) program dependencies.

Definition 2.20 (Actual dependency)**.** Given the PDG of a program, a PDG-node y depends on[7] a PDG-node x (respectively the program part represented by y depends on the program part represented by x) if

- the values of the variables $v \in Def(x) \cup Ref(x)$ determine the values of the variables $v \in Def(y) \cup Ref(y)$ (i.e., different values of the variables in x lead to different values of the variables in y), or
- the values of the variables $v \in Def(x) \cup Ref(x)$ determine whether the program statement corresponding to the PDG-node y is executed.

The system dependence graph (SDG). The SDG is a graph that consists of interconnected PDGs, where each PDG represents the possible dependencies in a single procedure of the program. A detailed discussion on SDGs can be found in Horwitz et al. [1990].

In addition to control and data dependencies the SDG contains interprocedural dependencies, which represent dependencies between nodes of different PDGs (other dependencies, which we do not discuss here, have been introduced to support object orientation and multithreading, see Hammer [2009]).

For each method special formal-in and formal-out SDG nodes are used. Formal-in nodes represent 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 program parts that are influenced by the method. In many 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. For example, the function `f` in Listing 2.4 has a formal-in node for `x` and a formal-out node for the return value of `f`.

```
1 int f(int x){
2     ...
3     return y;
4 }
```

[7] We also say that node x influences node y

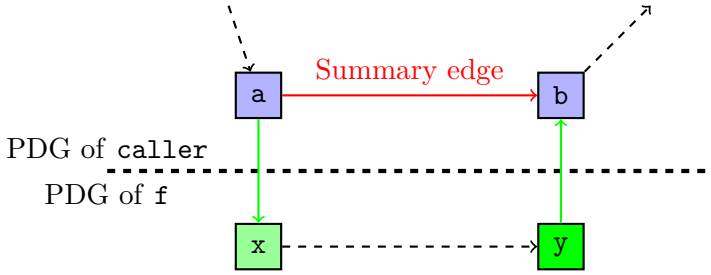

Figure 2.5: Summary edge for the example in Listing 2.4

```
5
6  void caller(){
7      ...
8      b = f(a);
9      ...
10 }
```

Listing 2.4: Example of a method call

At each method call site, there are actual-in nodes representing the parameters (or other inputs) of the method and actual-out nodes representing the returned values (or other program parts influenced by the method). For a given method call site, each actual-in node corresponds to a formal-in node of the callee and vice versa. The same holds for actual-out and formal-out nodes. Interprocedural dependencies connect actual-in nodes to the corresponding formal-in nodes and formal-out nodes to the corresponding actual-out nodes, respectively. As shown in Figure 2.5, for the call in Listing 2.4 there is the actual-in node for `a` that corresponds to the formal-in node `x` of `f`. There is also an actual-out node representing the returned value of `f` which corresponds to the formal-out node of `f`. For every method call there are so called *summary edges* in the SDG from an actual-in node to an actual-out node of the method whenever there is a path in the PDG between the corresponding formal-in to the formal-out node of the called method. In Listing 2.4 there is a PDG-path in `f` from `x` to the returned value, so a summary edge is inserted at the call site, namely from the actual-in node representing `a` to the actual-out node that represents the value returned by `f`.

2.6.2 Using the SDG to Prove Noninterference

While the PDG and the SDG have been developed during the eighties, their usefulness in the context of information flow security has been first noticed by Snelting [1996] in the nineties. Decades of research in this area have resulted in JOANA, a tool that statically analyzes Java programs of up to 100k lines of code for integrity and confidentiality (see Graf et al.

[2016]; Hammer and Snelting [2009]). We have used JOANA for a prototype implementation of the approaches presented in Chapters 6 and 7.

We explain how SDG-based approaches to proving noninterference work using—without loss of generality—the JOANA tool. We begin by showing how program slicing is done using the SDG, explain how a points-to analysis can be used to increase the precision, and then show how these techniques can be used for proving a noninterference property for a program.

Slicing and chopping. SDG-based approaches to proving noninterference use a special form of conditional reachability analysis that applies slicing and chopping techniques. Program slicing, as explained in Section 2.5, removes statements from a program in order to reduce its size and complexity while retaining some specified aspects of the behavior of the program. Slices are defined with respect to a slicing criterion, which usually consists of a program location and a set of variables. In the case of SDG-based slicing, the criterion usually consists of an SDG-node n, which represents a location in the program and the set of variables consists of $Def(n) \cup Ref(n)$.

Using the SDG, we define forward slicing, which is used to compute the program statements that are influenced by the criterion statement and backward slicing, which finds the programs statements that influence the criterion statement.

Definition 2.21 (Forward and backward slice)**.** Given an SDG and a criterion statement represented by a node n in the SDG, we define the forward slice $S_{\text{fw}}(n)$ and the backward slice $S_{\text{bw}}(n)$ as the following sets:

- $S_{\text{fw}}(n) = \{n_s \mid n_s \text{ is reachable from } n \text{ in the SDG}\}$
- $S_{\text{bw}}(n) = \{n_s \mid n \text{ is reachable from } n_s \text{ in the SDG}\}$

As shown by Wasserrab and Lohner [2012], a node that is not contained in the backwards slice of some node n cannot influence n (see Definition 2.20). Similarly, a node that is not contained in the forward slice of n cannot be influenced by n. The slice nodes also determine a sliced program, which is constructed from the original program by removing those program statements which are not in the slice. An important property of the backward slice is that, for every input, the criterion statement has the same behavior in the sliced program as in the original program. In particular, the variables in the criterion statement have the same values as in the original program.

Definition 2.22 (Chop)**.** Given two nodes in an SDG, n_h and n_l, we define the chop of these nodes as the set $C(n_h, n_l) = S_{\text{fw}}(n_h) \cap S_{\text{bw}}(n_l)$.

Program chopping can be thought of as a filtered slice. In the case of proving noninterference, it provides information on the program statements that are on an SDG-path from a high input to a low output. Hence, chopping provides a way of obtaining information about effect transmission through a program, which is more condensed than a slice, see Reps and Rosay [1995].

Points-to analysis. For object-oriented programs, a *points-to* analysis answers the question of which objects a given pointer or reference may point to during the execution of the program. Using this analysis, relations between two references (e.g., *may-alias* or *must-alias*) can be derived. The points-to analysis is usually solved by generating constraints—based on the program semantics—form the CFG of the given program. An over-approximating solution is then found for the generated constraints. Approaches to points-to analyses usually represent the constraints as a *points-to graph* in which

- there is a node for each pointer variable in the program,
- there is a node for each object creation site, and
- an edge between two nodes p and q is inserted, if p may point to q during program execution.

An overview of different algorithms for the points-to analysis can be found in [Hammer, 2009, Chapter 2]. The results of the points-to analysis can show that two references will never alias during the program execution, and this information can be used to remove some possible dependencies from the SDG.

Proving noninterference. The first step for proving noninterference with JOANA is the construction of the SDG for the given program. The noninterference property which needs to hold is then specified by annotating the SDG-nodes that correspond to the high inputs and low outputs. SDG-based approaches, such as the one implemented by JOANA, detect possible dependencies between high inputs and low outputs through graph analysis, using slicing and chopping at the SDG level. JOANA reports a possible noninterference 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 those two nodes is not empty).

Wasserrab and Lohner [2012] proved that no potential dependency is missed (i.e., that JOANA is sound) and that any influence by the high input on the low output in the program can occur only along an SDG-path from the high SDG-node to the low SDG-node that correspond to the high input and, respectively, low output. The noninterference property that is proved by JOANA (for sequential, deterministic, and terminating programs) corresponds to the one from Definition 2.2. Because the dependencies in the SDG are an over-approximation of the actual dependencies in program, if no SDG-path corresponding to a potential dependency between the high input and the low output is found, the program is guaranteed to be noninterferent. However, when there exists an SDG-path between a high input and a low output, the program may still be noninterferent.

2.7 Relational Verification

This section, which is based on Kiefer et al. [2018], presents the relational verification approach on which the framework for program slicing, presented in Chapter 8, is based on.

Relational verification is an approach for establishing a formal proof that if a relational precondition holds on two prestates of two programs—P and Q—then the respective poststates of P and Q will fulfill a relational postcondition. For two complex programs that yet are similar to each other, much less effort is required to prove that their outputs fulfill a relational property than to prove that they both satisfy a complex functional specification. The effort for proving equivalence mainly depends on the difference between the programs and not on their overall size and complexity.

A predicate π is a transition predicate for a program P if for any two states, s and s', $\pi(s, s')$ holds if and only if program P when started in state s terminates in state s'. Thus, the goal of relational verification is to prove the following property for two programs, P and Q:

$$Pre(s_P, s_Q) \wedge \pi(s_P, s'_P) \wedge \rho(s_Q, s'_Q) \rightarrow Post(s'_P, s'_Q)$$

where π and ρ are transition predicates for P and Q respectively, and *Pre* and *Post* are the relational precondition and postcondition respectively.

The slicing framework presented in Chapter 8 is based on the LLRêve (see Felsing et al. [2014] and Kiefer et al. [2018]) relational verification approach and tool. LLRêve works on programs written in the LLVM intermediate representation (IR); see Lattner and Adve [2004]. It analyzes the CFGs (see Definition 2.12) of the programs and reduces the program equivalence problem to that of the satisfiability of a set of Horn-constraints over uninterpreted predicates. The satisfiability of Horn-constraints can be solved with state of the art SMT solvers such as Z3 (see de Moura and Bjørner [2008]) and Eldarica (see Rümmer et al. [2013]). In Section 8.3 we present the generated constrains together with the adaptation that is necessary for program slicing.

If the analyzed programs contain loops, their CFGs contain cycles, which constitute a challenge for verification because their number of iterations is unknown. LLRêve handles cycles using *synchronization points*, at which the program state is abstracted by means of predicates. The paths between synchronization points are cycle free and can be handled easily. Synchronization points are defined by labeling basic blocks of the CFG with unique numbers. The entry and the exit of a function are considered special synchronization points B and, respectively, E. Additionally, the user can also define synchronization points at any location of the analyzed programs. The user must ensure that there is at least a synchronization point for each basic block of the CFG of the two programs and match them appropriately.

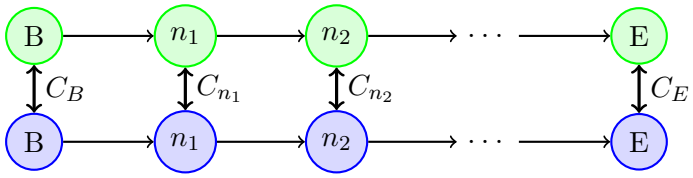

Figure 2.6: Coupled control flow of two fully synchronized programs, as shown in Kiefer et al. [2018]

For two programs with a similar structure, it is expected that there exist coupling predicates that describe the relation between the program states at two corresponding synchronization points. For two programs P and Q we introduce a coupling predicate $C_n(s_p, s_q)$ at each synchronization point index n, as shown in Figure 2.6. The coupling predicates C_B and C_E form the relational specification for the equivalence between P and Q. If the SMT solvers are able to find a interpretation of the coupling predicates that satisfy the Horn-constrains generated by LLRêve, then this constitutes a proof of the relational property specified by C_B and C_E.

Part II

Information Flow Security

3

A Framework for Checking Noninterference

Part II of the thesis presents contribution **C1.1**, which consists of a framework (illustrated in Figure 3.1) for checking noninterference properties. Note that by *checking* we mean both proving that a given program fulfills a specified noninterference property (as defined in Section 2.1) and—for the case in which the program violates the noninterference property—searching for a counterexample. The noninterference framework provides multiple contributions (described in Chapters 4, 5, 6, and 7) to the area of information flow security. In this chapter we give a short overview of the framework and show how its individual contributions are integrated and work with each other. The noninterference framework employs the following four approaches:

- *A1*: Dependency analysis
- *A2*: Deductive theorem proving
- *A3*: Automatic test generation
- *A4*: Noninterference debugging

Each of the four approaches in the framework has its their strengths and weaknesses. By integrating the approaches, the framework uses the strengths of the approaches to mitigate their weaknesses. The approaches contained in the framework handle two cases. In the first, the analyzed program does not fulfill the specified noninterference property, and the user is searching for a counterexample that showcases the noninterference violation and attempts to understand that counterexample. For this case, the framework provides an approach to automatic test generation and one to debug a program with respect to a noninterference property, with the goal of analyzing and understanding noninterference violations. In the second case, the analyzed program fulfills the specified noninterference property, and

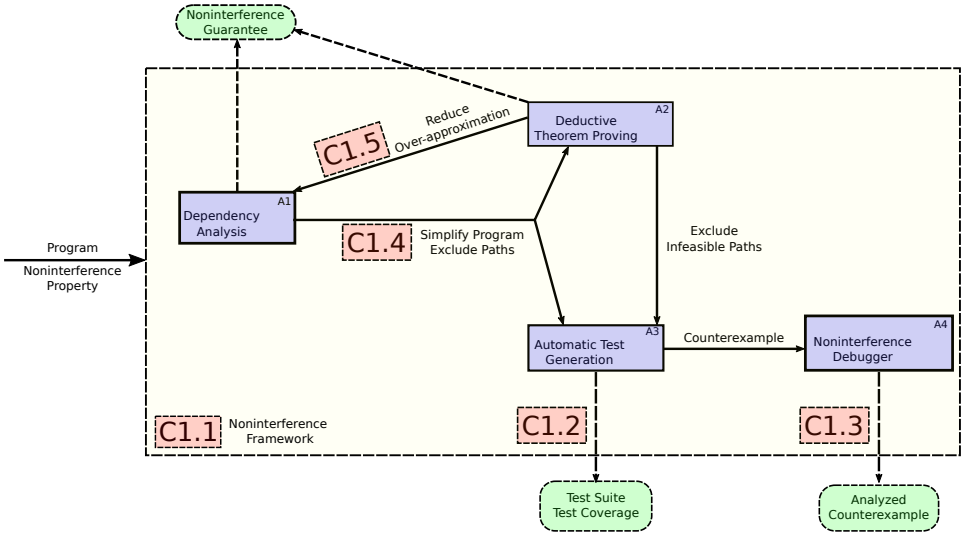

Figure 3.1: The noninterference framework

the user is attempting to prove this. For this case, the framework combines deductive theorem proving and dependency analysis. In the following we give an overview of the contributions in the context of the two cases.

Finding and analyzing noninterference counterexamples. For the case in which the user tries to find and understand a noninterference violation, the framework provides an approach for automatic test generation for noninterference properties and an approach for analyzing noninterference counterexamples. The approach for automatically generating noninterference tests constitutes contribution **C1.2** and is presented in Chapter 4. It serves two purposes. First, it can be employed to search for counterexamples of the analyzed noninterference property. If such a counterexample is found, the user gets two inputs that lead to a noninterference violation. The second purpose of the test generation is to generate a noninterference test suite that achieves a certain test coverage. This is useful when the program was neither proved correct nor was a counterexample found. Then, the user is provided with a test coverage value which can increase his confidence in the program's correctness (or—on the contrary—decrease it). The test generation approach benefits from deductive verification (on which it is based) by using it to remove infeasible program execution paths from the test generation and from the computation of the achieved coverage. As shown in Chapter 6, this approach also uses the results provided by the SDG-based analysis to remove parts of the program which are not relevant to the analyzed noninterference property. The framework provides an approach to analyze noninterference counterexamples that were found by the test generation approach. This assists the user to better understand the problem with the program and/or

specification. The approach for analyzing noninterference counterexamples provides contribution **C1.3** and is described in Chapter 5. It extends well-known concepts from program debugging and supports relational properties such as noninterference. The user can use this approach to search for the place in the program where a variable which should be assigned a low value is assigned a value that depends on the high input.

Proving noninterference. For the case in which the user is trying to prove that a given program fulfills a specified noninterference property, the framework combines two existing approaches: (1) dependency analysis (see Section 2.6) and (2) deductive theorem proving (see Section 2.3). Dependency analysis approaches are based on SDGs which syntactically compute the dependencies between the various program parts and check whether the low output depends on the high input. Whereas they scale very well, such approaches over-approximate the actual dependencies in the program, which results in false alerts. *Logic-based* approaches that use deductive theorem provers have a higher precision (i.e., they produce less false alerts) as they also consider the semantics of the program statements. However, they have a lower scalability. To gain the advantages of both SDG-based and logic-based approaches the framework combines these approaches in two ways. The first combination constitutes contribution **C1.4** and is presented in Chapter 6. In this combination the SDG-based approach simplifies the noninterference proof obligations for deductive verification by removing those parts of the program which are not relevant to the analyzed property and by excluding program execution paths which are guaranteed to have no effect on the validity of the noninterference property from the analysis with the deductive theorem prover. The second combination constitutes contribution **C1.5** and is presented in Chapter 7. In this combination the logic-based approach increases the precision of dependency analysis by showing that certain SDG dependencies are over-approximated. By soundly removing those over-approximated dependencies, the SDG-based approach is able to prove the noninterference property for more programs than before. The program simplification presented in Chapter 6 can also be used to simplify the proof obligations that are used to show that a program dependency is over-approximated.

4 Automatic Generation of Noninterference Tests

4.1 Introduction

This chapter describes the part of the noninterference framework that handles the case in which the user cannot prove that a given program fulfills a specified noninterference property. For the case that the program indeed violates the property, the noninterference framework must provide the means to search for counterexamples that showcase the violations. Testing is well suited for this purpose, and the approach presented in this chapter can be used to automatically generate tests that cover a large extent of the analyzed program (a full coverage is not always theoretically achievable). However, checking whether a program satisfies a given noninterference property is undecidable, and there can be situations in which the verification of the noninterference property has failed (because either the program indeed violates the property or the verification process failed), and yet no counterexample was found. For this, we provide a coverage criterion that allows the user to see to what extent the program was tested and allows him to gain confidence in the analyzed program. Thus, in this chapter we extend the approach to automatic test generation based on KeY (see Section 2.4) such that it can be applied to noninterference properties. This chapter is based on previous work by the author published in Herda et al. [2019c], which is an extension of the work of the author that was published in Ahrendt et al. [2016b]. Parts of the results of this chapter are based on a bachelor's thesis (see Müssig [2018]) which was supervised by the author.

Motivation. Testing is useful for checking noninterference properties, since it complements existing approaches in this area. There exist various approaches to formally prove that a program fulfills a specified noninterference property (see, e.g., the survey of Sabelfeld and Myers [2003]). However, in

practice they may not be enough to ensure that a program is secure. Existing approaches check only the source code and not the underlying operating system or the compiled program, which can both contain bugs that may lead to noninterference violations. Testing, on the other hand, takes these aspects into consideration. Furthermore, those approaches do not provide good support to distinguish between cases were no proof is found for a correct program and cases where the program indeed violates the noninterference property (as there may be false alarms).

Testing the noninterference property can help overcome these difficulties since it both can show the existence of a noninterference violation for some concrete test input, and it considers the concrete program executions thus including aspects such as the operating system. Approaches to automatically find counterexamples and for automatic test generation for noninterference properties have been developed (see e.g., Do et al. [2016]; Milushev et al. [2012]). However, those approaches do not support heap data structures and also provide no solution for the case in which counterexamples could not be found (i.e., they assume that the user tests the program until he finds a noninterference counterexample).

Contribution C1.2 The first contribution of this chapter is the design of tests for checking several variations of the noninterference property. These tests improve on those generated with existing automatic test generation approaches for noninterference properties by supporting heap-based programs and the declassification of user provided expressions. The second contribution consists of an approach that employs symbolic execution for the automatic generation of such tests. We have implemented this approach on top of the KeY theorem prover[1]. The third contribution is the extension of existing test coverage criteria such that they now are appropriate also for noninterference test suites. We have evaluated the suitability of both the coverage criteria and the automatic test generation approach.

Structure of the Chapter. In Section 4.2 we show how the noninterference properties defined in Section 2.1 can be tested. Then we provide in Section 4.3 a step-by-step description of the test generation approach. Section 4.4 defines extended coverage criteria for noninterference test suites, and in Section 4.5 we evaluate our approach on a collection of examples. We then conclude in Section 4.6.

[1]The implementation is available in Herda et al. [2019e].

4.2 Noninterference Tests and Test Suites

In this section we show how the noninterference properties defined in Section 2.1 can be tested. For this, we adapt the definitions of functional tests and test suites from Section 2.4.1.

A *noninterference test* can formally be described as a tuple $\langle s_1, s_2, Or \rangle$ consisting of two prestates—s_1 and s_2—representing two *test inputs* and an *oracle function* Or. The two states s_1 and s_2 are required to be low-equivalent ($s_1 \sim_L s_2$) according to Definition 2.1 of $\sim_L$. The oracle function $Or(s_1', s_2') \mapsto \{pass, fail\}$ checks the two poststates obtained when twice running the program—once starting in s_1 and once in s_2—and checks whether the test is successful or not. In the case of noninterference testing, the oracle checks whether the two poststates are low-equivalent.

Thus, to perform a noninterference test defined by such a tuple, we execute the method under test (MUT) twice—with inputs `in1` and `in2`, respectively, as shown in Listing 4.1. For this to be a valid noninterference test, the second input (`in2`) must be generated by the `generateLowEqInput` function such that `in1` and `in2` are low-equivalent. The oracle function then checks whether the two outputs, `out1` and `out2`, are also low-equivalent.

```
1  testMUT() {
2     //Execution 1 preamble
3     in1 = generateInput();
4     // MUT call 1
5     out1 = executeMUT(in1);
6     //Execution 2 preamble
7     in2 = generateLowEqInput(in1);
8     //MUT call 2
9     out2 = executeMUT(in2);
10    //Oracle call
11    oracle(out1,out2);
12 }
```

Listing 4.1: Structure of a noninterference test

To test the noninterference properties defined in Section 2.1, we have to define the semantics of the `generateLowEqInput` and `oracle` functions (shown in Listing 4.1) for each of those properties. For the classical noninterference property of Definition 2.2, `generateLowEqInput` generates a second input (`in2`) that is low-equivalent to `in1` according to the relation $\sim_L$ from Definition 2.1, and the `oracle` checks whether the two outputs (`out1` and `out2`) are low-equivalent according to the same relation $\sim_L$.

For the noninterference with declassification property provided in Definition 2.3, `generateLowEqInput` generates a second input (`in2`) such that the two inputs (`in1` and `in2`) are low-equivalent according to Definition 2.1, and the results of evaluating the given expression in the two prestates are identical. The oracle function still checks whether the two outputs fulfill the low-equivalence relation $\sim_L$.

For the noninterference property with isomorphism given in Definition 2.4, `generateLowEqInput` generates a second input (`in2`) that is low-equivalent to the first input (`in1`) according to $\sim_L$ while `oracle` is the low-equivalence relation with isomorphism $\sim_L^\pi$. In practice, however, it must be ensured that the second execution of the MUT does not interfere with the results of the first execution. For this reason, in the implementation of the approach we generate two isomorphic inputs which are low-equivalent according to $\sim_L^\pi$.

To test noninterference in a given method m, multiple tests are used, and together they form a *test suite*. A test suite TS^m for a method m is defined as:

$$TS^m = \langle \{(s_{11}, s_{12}), \ldots, (s_{n1}, s_{n2})\}, Or^m \rangle$$

Note that for a given noninterference property all tests in a test suite have one common oracle function.

4.3 Automatic Noninterference Test Generation

In this section we describe step-by-step our approach for generating noninterference test suites for a given method under test (MUT) and a specified noninterference property as described in Section 2.1. The steps of the approach are illustrated in Figure 4.2.

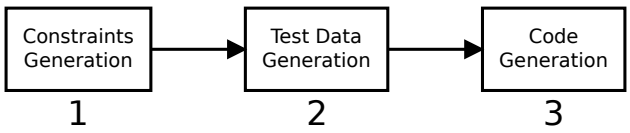

Figure 4.2: Steps of the test generation approach

In the first step, the MUT is loaded into the KeY theorem prover. A noninterference proof obligation is obtained that constraints the prestates and the poststates of two executions of the MUT, as explained in Section 2.3. Still in the first step, the MUT is symbolically executed twice, resulting in a proof tree in which each leaf contains a pair of path conditions resulting from the two executions. In the second step, the pairs of path conditions are extracted from the proof tree, along with the condition that the prestates are low-equivalent. If there are declassification expressions, the condition that requires them to evaluate to the same values is also extracted from the proof tree. Then, the model generator—in our case the SMT solver z3 (see de Moura and Bjørner [2008])—attempts to generate two program inputs for the MUT. The generated inputs have to fulfill the conditions extracted from the proof tree. Finally, in the third and last step, a noninterference test is generated using the model for the test inputs and the noninterference specification for the test oracle.

The automatic test generation approach offers two options that lead to the following two scenarios:

1. The user searches for noninterference violations. In this case, the constraints that are passed on to the model generator also require the two poststates to be non-low-equivalent (i.e., to demonstrate a nonintereference violation).

2. The user generates a noninterference suite with a high test coverage. In this case the constraints do not restrict the poststates in any way.

The first option is useful when the user has failed to prove (for example with the approaches presented in Chapters 6 and 7 or others presented in Section 9.1) that the program is noninterferent and suspects that the program violates the noninterference property. The second option is useful in the case that both the verification and the counterexample generation have failed. This may happen, as checking noninterference is an undecidable problem. The coverage of the generated noninterference test suite can increase the user's confidence in the correctness of the program. Moreover, while approaches for proving noninterference usually only check the program itself, generating a high-coverage test suite is useful for finding noninterference violations introduced by the operating system or by the compiler. Thus, the second option is useful even in the case when the verification succeeded.

In the rest of this section we describe each step of our approach and also show an example of a generated noninterference test.

4.3.1 Constraints Generation

The first step of the automatic test generation approach for a given MUT specified with a noninterference property consists in loading the specified MUT into KeY such that a proof obligation is created, as described in Section 2.3.

The proof obligation for noninterference contains two modal operators (see Section 2.2.1), each referring to one of the two program executions. The programs inside the two modal operators are symbolically executed by automatically applying[2] the JavaDL calculus rules (see Section 2.2.1), and test input constrains are generated similarly to how this is done for functional properties (see Section 2.4.2). During symbolic execution we use finite loop unwinding (up to a number provided by the user) and method inlining. Thus, the user does not have to provide method contracts or loop invariants. This technique is known as bounded symbolic execution. At the end of this step, we obtain a proof tree in which each leaf is a sequent that contains a pair of path conditions (see Definition 2.15)—one for each

[2]For optimizations that allow the reuse of the first symbolic execution, see Scheben and Greiner [2016].

of the two program executions. After the bounded symbolic execution is finished, the leaves of the proof tree contain no modal operators, but only first order formulas. Thus, a model that satisfies the formulas in a leaf is also a model for the two path conditions represented by the leaf, for the requirement that the prestates are low-equivalent, and the poststates are not low-equivalent. With the second option, the constraint requiring that the poststates violate the low-equivalence requirement is ignored when searching for a model. If the second option is active, the low-equivalence requirement for the poststates is removed from the initial proof obligation before the beginning of the symbolic execution. The next step of our approach consists of finding such a model.

4.3.2 Test Data Generation

In terms of noninterference testing, a model is an assignment of concrete values to object fields and parameters that constitute the initial states s_1 and s_2 of the noninterference test. After extracting the test data constraints from the proof tree, we need to find a model that satisfies them. From this model we can then extract the test data that is used as input for the MUT in the test suite. If a model is found for the path conditions of execution paths p_1 and p_2, the noninterference test using the input resulting from this model takes the two paths p_1 and p_2 during execution. The model is found by translating the test data constraints to the SMT-LIB language (see Barrett et al. [2010]) and handing them to an SMT solver as described in Section 2.4.2. Note that the translation to SMT used in our approach is bounded (i.e., the maximal number of instances of each type is bounded; the bound for each type is provided by the user). Our approach cannot find models that contain more instances of a certain type than the bound set for that type. We, however, rely on the *small scope hypothesis*, as stated in Jackson [2002], which claims that most problems in computer programs can be detected by testing the program using inputs within a small scope. This step of the approach is also responsible for generating models that fulfill any specified preconditions as well as the declassification expressions needed to support noninterference with declassification (see Definition 2.3).

4.3.3 Code Generation

As shown in Listing 4.1, a noninterference test contains 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 check whether the test was successful or not. We begin this section by describing the preambles of the two method calls, and then we explain how the test oracles are generated.

Test preambles. The test preamble constructs the prestate for the MUT call. The prestate consists of the MUT parameters, the heap locations that are reachable from the MUT parameters, and the (implicit) `this` pointer. The prestate values are obtained from the model returned by the SMT solver. All constants (representing the values of the MUT parameters) and heap locations of the model are declared and initialized in the first part of the test preamble. For the two preambles, we create two input states by duplicating the objects and values from the model. This is done to avoid the second MUT execution affecting the results of the first execution. Note that the current implementation does not support static fields. This limitation is only implementation related, as support for static fields can be achieved by storing the results (including the values of static fields) of the first MUT execution before continuing with the second MUT execution. In the second part of the test preamble, the fields of the created objects and the elements of the created arrays are initialized with the values that they have in the model. The preamble also contains some Java containers consisting of all values of all Java types that were created in the test preamble. These containers are used for functional test oracles that check quantified formulas. For noninterference tests the containers are used by the oracle to check whether a reference is newly created during the MUT execution, as required by the *newObjIso* predicate (see Section 2.3.1). The test preamble can use the Objenesis library (see Objenesis [2018]) to create objects of Java classes that do not have a default constructor and to initialize the values of private object fields.

MUT Calls. For both calls, the MUT and its surrounding code is taken from the JavaDL modal operator in the root node of the proof tree. Using the surrounding code, rather than just the invocation of the MUT, is important to ensure that the actual execution of the code has the same semantics as the symbolic execution of the code. The surrounding code consists of a *try/catch* block that catches any exception thrown during the execution of the MUT. This allows the test oracle check whether an exception was thrown, what type this exception is, and—based on this information—whether the test was successful.

Test Oracle. Each test suite contains 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 2.3) are checked, while for references created in one of the preambles only the second and third requirements (i.e., that the references are of the same type and are isomorphic) of *newObjIso* are checked.

4.3.4 Example

We use the example in Listing 4.3 to show how a noninterference test generated by our approach looks like. The example consists of a Java method, `foo`, which has the input parameter `h`. The method creates an array of length `h` and returns its fifth element. The parameter `h` is considered high whereas the return value of the method as well as exceptions that may be thrown are considered low outputs. The `determines` clause in the specification states that the method's result must not depend on any part of the prestate. The `new_objects` clause states that the exceptions that might be thrown must be isomorphic in the two executions of `foo`. See Section 2.3.2 for the JML extensions for specifying noninterference properties.

```
1 public class program {
2     /*@determines \result \by \nothing
3          \new_objects \exception;@*/
4     public static int foo(int h) {
5         int [] a = new int [h];
6         return a[5];}}
```

Listing 4.3: Array example

The program violates the specified noninterference property, as it depends on the value of `h`, whether an `ArrayIndexOutOfBoundsException`, a `NegativeArraySizeException`, or no exception at all is thrown. Thus, the exceptions implicitly reveal information about `h`. With our approach, when using the noninterference property based on object isomorphism (see Definition 2.4), we are able to generate tests that showcase these violations.

Thrown exceptions are newly created during the program execution, and their reference addresses are unlikely to be identical in different executions. Therefore, existing approaches that support only classical noninterference consider a case in which the two program executions throw the same exception as a noninterference counterexample. This, despite the fact that an attacker would still be unable to distinguish between the two thrown exceptions. With our approach we avoid such tests that in fact are false alarms.

The program in Listing 4.4 is an abridged version of a test generated using our approach for the program of Listing 4.3. The full implementation of the test can be seen in Appendix A.1. The generated test consists of two executions—A and B—that start in low-equivalent states (for this example a trivial requirement, as there are no low inputs), but end in two states which are not low-equivalent. Executions A and B lead respectively to an `ArrayIndexOutOfBoundsException` and to a `NegativeArraySizeException`. The test preambles for execution A (lines 5–12) and execution B (lines 19–26) are generated from the model that is provided by the SMT solver, as described in Section 4.3.2. The model contains values for constants along with the content of all heaps that appear in the test data constraint. The goal of the test preamble is to reproduce in

the executable environment the initial state from the model. Therefore, we consider only the contents of the initial heap from the model.

The two calls of the MUT `foo` and their surrounding code are in lines 14–17 for execution A and in lines 28–31 for execution B. The code for calling the MUT is taken from the JavaDL proof obligation. Using the surrounding code, and not just the invocation of the MUT, is important to ensure that the actual execution of the code has the same semantics as the symbolic execution of the code. The surrounding code consists of a *try/catch* block that catches the thrown exception—if any—and gives it to the test oracle. The code may contain variables that do not appear in the generated model. Therefore, we have to declare them. Those variables are declared and initialized in lines 2–3. In lines 32–34 some Java containers are created; they are needed by the test oracle to check whether a reference was created during an execution of the MUT. Finally, in line 36, the test oracle function is called.

```
1  public void  testcode(){
2    /*@ nullable */ java.lang.Throwable exc_2_A = null;
3    /*@ nullable */ java.lang.Throwable exc_2_B = null;
4    //Test preamble for execution A
5    program _o1_A = new program();
6    java.lang.ArrayIndexOutOfBoundsException _o2_A =
7       new java.lang.ArrayIndexOutOfBoundsException();
8    int[] _o4_A = new int[0];
9    java.lang.NegativeArraySizeException _o3_A =
10      new java.lang.NegativeArraySizeException();
11   int h_2_A = (int)0;
12   int result_2_A = (int)4;
13   //Calling the method under test
14   int _h_2_A = h_2_A;
15   {exc_2_A=null;
16   try {result_2_A=program.foo(_h_2_A);}
17   catch (java.lang.Throwable e) {exc_2_A=e;}}
18   //Test preamble for execution B:
19   java.lang.ArrayIndexOutOfBoundsException _o2_B =
20      new java.lang.ArrayIndexOutOfBoundsException();
21   java.lang.NegativeArraySizeException _o3_B =
22         new java.lang.NegativeArraySizeException();
23   program _o1_B = new program();
24   int[] _o4_B = new int[0];
25   int h_2_B = (int)-16;
26   int result_2_B = (int)0;
27   //Calling the method under test
28   int _h_2_B = h_2_B;
29   {exc_2_B=null;
30   try {result_2_B=program.foo(_h_2_B);}
31   catch (java.lang.Throwable e) { exc_2_B=e; }}
32   Set<Boolean> allBools= new HashSet<Boolean>();//...
33   Set<Integer> allInts= new HashSet<Integer>();//...
34   Set<Object> allObjects= new HashSet<Object>();//...
35   //calling the test oracle
36   assertTrue(testOracle( exc_2_B, result_2_B, exc_2_A,
37           result_2_A, allBools, allInts, allObjects));}
```

Listing 4.4: Noninterference test for the array example in Listing 4.3

The test oracle for the test shown in Listing 4.4 is given in Listing 4.5, with the full implementation being available in Appendix A.1. The arguments of the oracle function can be seen in line 39 of Listing 4.4. To see whether the two poststates fulfill the specified requirements (Listing 4.3), the test oracle checks two things. First, it checks (line 2) whether the values returned by the two executions of the MUT are equal. Second, it checks whether the two objects corresponding to the exceptions thrown by the two execution are isomorphic (according to Definition 2.4). This is done by the automatically generated method `sub2` (line 3), which checks whether the two post states fulfill the *newObjIso* predicate (see Section 2.3.1). For this, the procedure creates the two lists of objects[3] that need to be isomorphic and checks whether these objects are newly created (using the `newObjects` procedure), whether they have the same types (using the `sameTypes` procedure) and whether they are isomorphic (using the `objectsAreIsomorphic` function). In our example the two thrown exceptions have different types, thus causing the `sameTypes` procedure to return `false`. The implementation of the procedures that are called by `sub2` is common for all tests generated by our approach.

```
1 public boolean testOracle(...){
2    return (result_2_A == result_2_B) && sub2(...);}
3 public boolean sub2(...){
4    Object[] l1 = {exc_2_A};
5    Object[] l2 = {exc_2_B};
6    return newObjects( l1, allObjects)
7           && newObjects( l2, allObjects)
8           && sameTypes( l1, l2)
9           && objectsAreIsomoprhic( l1, l2);}
```

Listing 4.5: Oracle function for the noninterference test in Listing 4.4

4.4 Coverage Criteria

In this section we extend existing test coverage definitions to make them appropriate for measuring the coverage of noninterference test suites, and then we discuss the coverage provided by the test suites generated with our approach. The coverage of a test suite can be used as an indicator of how thoroughly the program was tested in the case in which neither could it be proved that the specified noninterference property holds, nor was a counterexample found that shows that the program does not fulfill the specified noninterference property.

Well established coverage criteria such as statement, branch, and path coverage are defined on the CFG (see Definition 2.12). As explained in Section 2.4.1, full statement coverage of a test suite requires each node in the CFG to be traversed during the execution of the test suite, full branch

[3] The references contained by these lists are provided by the noninterference specification with the `new_objects` clause.

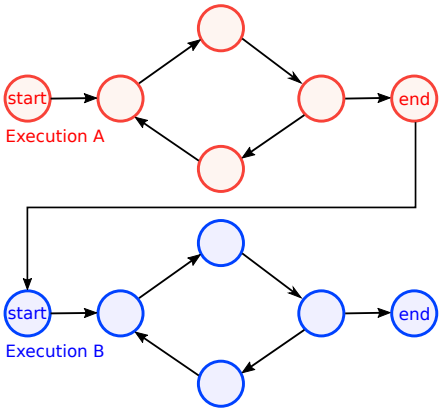

Figure 4.6: Self-composed CFG

coverage requires each edge to be traversed, and for full path coverage each path has to be traversed.

To extend these coverage criteria also for the noninterference test suites defined in this chapter, we no longer use the CFG when determining the coverage but rather a self-composed CFG. The self-composed CFG consists of two copies of the MUT's CFG with renamed variables (in the second copy) and with a directed edge from the end node of the first copy to the start node of the second copy. We illustrate the structure of a self-composed CFG in Figure 4.6.

We define the following relational[4] coverage criteria on the self-composed CFG:

- Relational statement coverage: $\frac{\#(\text{covered nodes})}{\#(\text{nodes})}$
- Relational branch coverage: $\frac{\#(\text{covered edges})}{\#(\text{edges})}$
- Relational path coverage: $\frac{\#(\text{covered paths})}{\#(\text{paths})}$

Because a CFG may contain cycles, the number of paths may be infinite. To handle this, we unwind loops and inline called methods up to a maximum number of b-times (the bound b is provided by the user), and consider only b-paths (see Section 2.4.1) on the self-composed CFG. We can then define the *relational bounded path coverage criterion* as

$$\frac{\#(\text{covered } b\text{-paths})}{\#(b\text{-paths})}$$

[4]We name them relational as they can be used not only for noninterference but also for relational properties (i.e., properties referring to two executions of the program).

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.

A problem of the relational bounded path coverage criterion is that some program execution paths may be infeasible (i.e., the path conditions for those execution paths are unsatisfiable). It is an undecidable problem 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 our approach would be able to generate tests for those paths, given more time. In addition, the low-equivalence requirement for the two inputs may cause certain paths in the self-composed CFG to be infeasible. A more useful criterion would therefore be the *relational bounded feasible path coverage*, defined as

$$\frac{\#(\text{covered } b\text{-paths})}{\#(\text{feasible } b\text{-paths})}$$

on the self-composed CFG. We cannot compute the number of feasible paths. However, since our approach is based on using a theorem prover, we can use it to show that certain paths are infeasible, and the remaining paths that could not be proved infeasible constitute an over-approximation of the feasible paths. This over-approximated number of feasible paths is used to compute an under-approximation of the *relational bounded feasible path coverage* for a generated test suite.

The achieved coverage can be easily measured when using our test generation approach. The b-paths in the self-composed CFG correspond to the paths in the proof tree obtained after the symbolic execution of the programs inside the two modal operators is finished. We can compute the achieved relational bounded path coverage from the number of generated tests and the number of leaves in the KeY proof tree. Closed branches of the proof tree correspond to infeasible paths of the self-composed CFG. Thus, the relational bounded feasible path is approximated by using the number of open leaves instead of the total number of leaves.

There are two indicators for how thoroughly a program was tested with respect to a given noninterference property. On the one hand, there are the bounds for the maximum number of instances for each type and for the maximum number of loop unwindings and method inlinings which are provided by the user. On the other hand, there is the achieved relational bounded feasible path coverage (for the bound set by the user). Since we assume the small scope hypothesis, a high relational feasible path coverage for lower bounds is a stronger indicator of the thoroughness than a low relational feasible path coverage achieved for higher bounds.

In the next section we evaluate our approach with respect to the relational bounded path and relational bounded feasible path coverage criteria.

4.5 Evaluation

In this section we evaluate our test generation approach for noninterference properties (as defined in Section 2.1) on various Java examples. We use a subset of the IFSpec benchmark collection (see Hamann et al. [2018]) for noninterference properties of Java programs. The purpose of this benchmark collection is to enable the evaluation of approaches for checking noninterference. For this evaluation we limit ourselves to programs that are supported by KeY (i.e., single threaded programs that do not use Java standard libraries). For each benchmark it is specified what are the high inputs and the low outputs, as well as whether the benchmark violates the specified noninterference property (i.e., the *ground truth*). The subset of IFSpec that we have analyzed contains 21 secure and 16 insecure benchmarks and is published in Herda et al. [2019d]. The largest benchmark program has 44 lines of code. The benchmark programs contain different Java features such as loops, arrays, object field access, interfaces, overloading, exceptions, try-catch blocks, inheritance, typecasts, and arithmetic operations.

We evaluated the two options mentioned in Section 4.3 (searching for counterexamples and generating a high-coverage test suite) on both secure and insecure benchmarks. When applying our approach for finding counterexamples on the secure benchmarks shown in Table 4.7, no test was generated—which is the expected outcome. This serves as a good sanity check for our approach. For the insecure benchmarks shown in Table 4.8, the column *CE* provides the number of counterexamples that our approach has found. Our approach is able to find counterexamples for every benchmark and present them to the user as a noninterference test.

With the second option, aiming to obtain a good test coverage, we no longer generate input pairs that are guaranteed to lead to a violation of the noninterference property. Instead, we attempt to generate, for every b-path in the self-composed CFG, input pairs that are low-equivalent. We use the same benchmarks to evaluate the second option, and the results of the evaluation are shown in Tables 4.7 and 4.8. The column *Paths* presents the number of b-paths in the self-composed CFG, the column *TC* shows the number of generated tests, the column *RBPC* contains the obtained relational bounded path coverage, the column *RBFPC* contains the obtained relational bounded feasible path coverage, and—for the insecure benchmarks—the column *Failed TC* contains the number of failing tests generated with the second option. The *CE* column is relevant only for the first option. As expected, for the secure benchmarks no test failed.

The relational bounded path coverage obtained in the evaluation is quite low for most benchmarks. However, using the theorem prover, we can easily prove that many paths are infeasible. Thus, for almost all benchmarks we actually obtain full relational bounded feasible path coverage. That is, for all

Table 4.7: Evaluation results for secure benchmarks

Benchmark	Paths	TC	RBPC (%)	RBFPC (%)
Aliasing-secure	9	1	11.1	100.0
Array1	13	1	7.7	100.0
BooleanOperations	3	1	33.3	100.0
CallContext	1	1	100.0	100.0
ExControlFlow1	20	4	20.0	100.0
ExControlFlow2	36	4	11.1	100.0
FieldsOfParams	7	1	14.2	100.0
IFLoop	21	1	4.7	100.0
IFLoop2	15	1	6.6	100.0
IfMethodContract	17	4	23.5	100.0
IFMethodContract2	4	4	100.0	100.0
interface-noleak	11	1	9.1	100.0
lostInCast	1	1	100.0	100.0
ObjectSensleak	7	1	14.3	100.0
CondAssignment	4	4	100.0	100.0
CondChecks	12	4	33.3	100.0
Webstore	17	1	5.9	100.0
Webstore2	70	3	4.0	100.0
Webstore3	9	1	11.1	100.0
Webstore4	182	14	7.7	87.5
Declass1	4	2	50.0	100.0

Table 4.8: Evaluation results for insecure benchmarks

Benchmark	Paths	TC	RBPC (%)	RBFPC (%)	Failed TC	CE
Aliasing	9	1	11.1	100.0	0	1
Array2	55	9	16.3	100.0	6	8
ArrayLength	24	4	16.6	100.0	2	4
DivisionByZero	20	4	20.0	100.0	1	3
Eg1	1	1	100.0	100.0	0	1
Eg2	36	25	69.4	69.4	22	25
Eg4	624	36	5.7	73.4	28	31
Exception1	20	4	20.0	100.0	1	2
Exception2	20	4	20.0	100.0	2	4
Exception3	36	4	11.1	100.0	2	2
TryCatch	20	4	20.0	100.0	2	2
FieldsOfParams	20	9	45.0	100.0	4	4
simpleTypes	12	4	33.3	100.0	2	2
typesCastingError	36	4	11.1	100.0	2	2
StaticDispatching	7	2	28.5	100.0	1	1
Declass2	4	4	100.0	100.0	2	2

b-paths in the self-composed CFG we either show that they are infeasible[5] or we generate a noninterference test for it. Moreover, when we used the option of generating a high-coverage test suite, the generated tests for the insecure benchmarks still were able to uncover noninterference violations for all but one benchmark, without even requiring the poststates to be low-equivalent when generating the test inputs. A possible reason for this is the fact that the analyzed insecure benchmarks are specifically designed to demonstrate certain noninterference violations and provide no functionality other than that. All in all, the evaluation shows that the relational bounded feasible path coverage is an appropriate coverage criterion for noninterference properties: for high coverage values we either find no violations for secure benchmarks or find at least one violation for most insecure benchmarks.

4.6 Conclusion

In this chapter we have shown how the noninterference properties defined in Section 2.1 can be tested for a given program and explained the notions of noninterference tests and test suites. We presented an approach that is based on symbolic execution for automatic test generation for the three noninterference properties that we consider. We extended existing definitions of test coverage criteria such that they now are usable for noninterference test suites. We evaluated our test generation technique on a set of noninterference benchmarks and showed that the approach is capable of both finding noninterference violations in insecure benchmarks as well as achieving a high coverage.

Related work in the area of testing and automatic test generation for noninterference properties is described in Section 9.1.1. In Chapter 5 we show how a counterexamples found with our test generation approach can be analyzed using a noninterference debugger. In Chapter 6 we show how programs can be simplified for noninterference testing.

[5] This is the case when one of the pairs is infeasible or when two low-equivalent inputs cannot satisfy both path conditions.

5 Analysis of Noninterference Counterexamples

5.1 Introduction

This chapter presents the part of the noninterference framework that deals with the situation in which the automatic test generation approach presented in Chapter 4 has found a noninterference violation. In such a case, the user is presented with a counterexample that consists of two low-equivalent inputs (see Definition 2.1) that result in two outputs that are not low-equivalent. While such a counterexample demonstrates that the program violates the specified noninterference property, the user may find it difficult to understand the specific implementation details that lead to this violation. In this chapter we present an approach that assists the user in understanding a noninterference counterexample, in order to help him fix the cause of the noninterference violation. This approach was implemented as *DIbugger*, a relational debugging tool, and it is available in Herda et al. [2019b]. This chapter is based on work by the author previously published in Herda et al. [2019a]. DIbugger was implemented during a "software development laboratory" (*Praxis der Softwareentwicklung*) by the students Etienne Brunner, Joana Plewnia, Ulla Scheler, Chiara Staudenmaier, Benedikt Wagner, and Pascal Zwick under the supervision of the author.

Motivation. Software verification is a tedious process that involves the analysis of multiple failed verification attempts and corrections of the program or specification. Oftentimes, this is an incremental process where at first neither the formal specification captures the informally-given requirements nor the program adheres to its specification.

For the cases in which the verification of a given program and specified property fails, existing verification tools such as KeY can provide program

inputs that constitute counterexamples. However, understanding *why* the provided inputs are a counterexample is not a trivial task. Whereas this task is already difficult for functional properties, it becomes even more challenging for relational properties (i.e., properties that consider two program executions). This is because the user needs to concomitantly check the values of program variables across multiple (two in the case of noninterference) program runs.

Understanding counterexamples is, nonetheless, a very important step that the user needs to do in order to improve the analyzed specification and/or code. The process of verifying software is an iterative one, as stated in Beckert et al. [2017c]: "Until the verification succeeds, (a) failed attempts have to be inspected in order to understand the cause of failure and (b) the next step in the proof process has to be chosen".

Thus, an approach that assists the user in analyzing and understanding noninterference counterexamples is necessary.

Contribution C1.3 The contribution of this chapter consists of a novel approach for the analysis of counterexamples to noninterference properties. This approach extends established concepts from program debugging (e.g., stepping and breakpoints) and makes them suitable for relational properties. Thus, we assist the user in performing step (a) in the iterative software verification process described in Beckert et al. [2017c]. To the best of our knowledge the state-of-the-art approach for analyzing noninterference counterexamples consists of running two normal debuggers side-by-side using the two inputs provided by the counterexample. In addition to the state-of-the-art approach, we assist the user by allowing him to control and analyze the two executions at the same time.

Structure of the chapter. In Section 5.2 we explain the required functionalities of an approach for analyzing noninterference counterexamples and present the assumptions that motivate those functionalities. In Section 5.3 we describe the tool DIbugger, which is the implementation of our approach. In Section 5.4 we show how our approach can be used and how it is useful by means of examples, and we discuss which relational properties can be handled by our approach. We then conclude in Section 5.5.

5.2 Requirements of the Approach

We begin this section by stating the *assumptions* which motivate the requirements of the approach for analyzing noninterference counterexamples. The assumptions, which we present in the following, state what the user knows and what he wants to find out.

1. The user knows which program variables must contain only low data.

2. For those variables which must contain only low data, the user does not know and wants to find out whether they are assigned different values during each of the two executions determined by the inputs provided by the counterexample.

3. The user is provided with a counterexample that consists of two low-equivalent inputs for a program specified with a noninterference property.

The goal of the approach is established by the assumptions above. The approach is meant to assist the user in finding out which low variables are assigned different values during the executions determined by the two inputs provided by the counterexample. Because the two inputs are low-equivalent, it is expected that the low variables will contain the same values during the two executions of the program the two inputs. Once the user identifies a low variable which is assigned different values during the two executions, he can investigate the cause of this. Moreover, it would be helpful for the user if the approach extends familiar functionalities of program debuggers to support noninterference properties.

Requirements of the Approach. Based on the assumptions, we identified the following requirements for which we designed the approach for analyzing noninterference counterexamples. In the following we explain these requirements.

Requirement 1: Side-by-side view of the two executions. The user must view the two executions side-by-side in order to see which point both execution have reached.

Requirement 2: Marking of low variables. The user must be able to mark those variables that he considers to be low, and the two executions must halt when they reach the marked locations. This allows the user to analyze and compare the two states. Since we extend well-known functionalities of program debuggers, this requirement should be realized using the familiar concept of breakpoints.

Requirement 3: Debugging operations. The user must be able to have complete control over the two executions. He must be able to perform all classic debugging operations (i.e., step in, step out, and step over) as well as step back in any of the two executions to a previous state.

Requirement 4: Side-by-side view of the two states. The user must be able to see the current states of the two executions side-by-side in order to compare the values of the variables in them. He must be able to hide variables which are of no interest to him. This is the case for variables for which the user does not know whether they are low.

Requirement 5: Assisted variable comparisons. The user must be assisted in comparing variables of his interest (i.e., it must be shown whether a

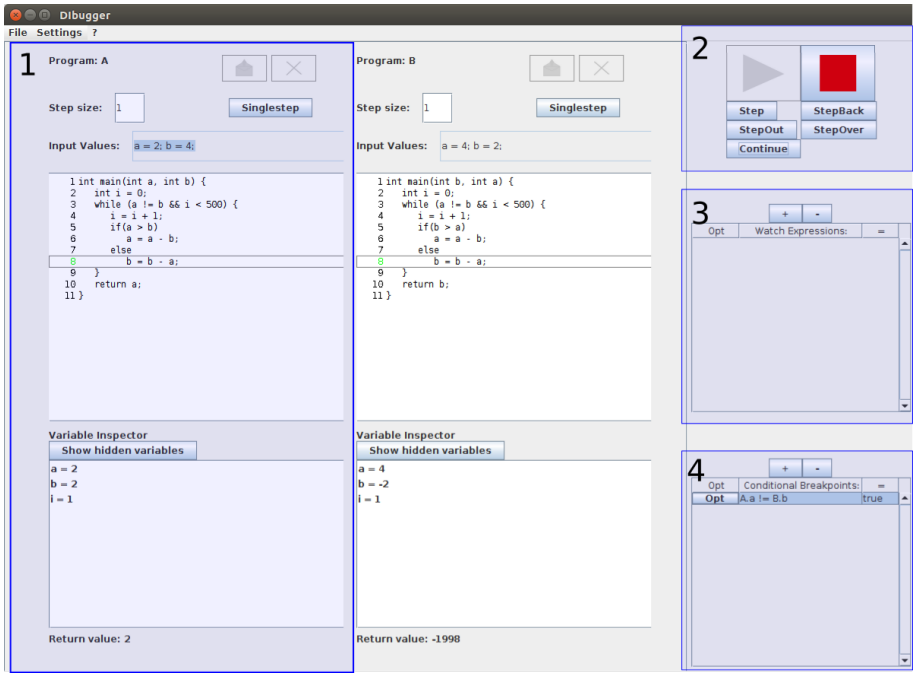

Figure 5.1: The user interface of DIbugger

variable has the same value in the two states or not). Depending on this comparison, the two program executions must stop automatically if the values differ.

In the next section we present the implementation of the approach, which fulfills these requirements.

5.3 Implementation

To implement our approach, we have built DIbugger, a relational debugger for a subset of the C programming language. It supports sequential, inter-procedural programs. Dynamic memory allocation programming features are not yet supported. DIbugger consists of four components responsible for the user interface, control, file handling, and debugging. The debugging functionality is built on top of a C interpreter. The interpreter generates the trace (i.e., the sequence of values of the program variables at each point of execution) of each analyzed program with its given input. The debugger works on those traces and executes the debugging operations. The graphical user interface (GUI) of DIbugger is shown in Figure 5.1. We explain the available features in the following.

5.3.1 Debugging Operations

The buttons for debugging operations are located at the top right part of the GUI (see Figure 5.1, in the highlighted area **2**). They realize *Requirement 3 (Debugging operations)*. The large *Play* (the grey button) and *Stop* (the red button) buttons are used for switching between the debug and the edit modes. The analyzed programs can be edited only in the edit mode. Once the user switches into debug mode, the programs can no longer be edited but only debugged. When switching to debug mode, the tool generates the execution trace for each program and the inputs provided via the program panels (program panels are explained in Section 5.3.2). To keep the generated trace valid, editing the program is not allowed in debug mode. While debugging, the user can switch back to the edit mode at any time. However, he needs to start debugging the programs from the beginning with newly generated traces, when switching back into debug mode. Below those two buttons (*Play* and *Stop*) are the following buttons that provide the stepping functionality, which was extended to support two program executions:

- *Step*: The execution of each analyzed program advances by the number of points of execution given by the user as the step size in the program panel (see Section 5.3.2). The user can use different step sizes for the analyzed programs that allow him to keep the executions of the analyzed programs synchronized and to examine loops with different numbers of instructions. Depending on the analyzed property and program, mutable step sizes allow the user to keep the programs in lockstep even when some programs execute more instructions than others.

- *StepBack*: The execution of each analyzed program is taken one step back.

- *StepOut*: The execution of each analyzed program jumps out of current method scope, or steps to the end of the main method.

- *StepOver*: The execution of each analyzed program jumps over a method call—if any—otherwise it performs a normal step.

- *Continue*: The execution of each analyzed program advances in steps of the size (i.e., the number of instructions executed in a step) provided by the user in the program panel (see Section 5.3.2). The executions stop when (1) each program has reached a break point set by the user or the end of the program, or (2) a conditional breakpoint evaluates to *true* (conditional breakpoints are explained in Section 5.3.3).

5.3.2 Program Panels

The central part of the GUI consists of a *program panel* for each analyzed program (e.g., in Figure 5.1, in the highlighted area **1**). The two buttons at the top of the program panel allow the user to add a new program panel or to remove the existing panel. For the noninterference properties considered by us, only two program panels are used. Below these buttons, the user can set the step size which will be used when debugging. By using the single-step button, the user can also perform a single step (only in the program of the panel where the button was clicked). Further down, the user provides the inputs of the program, which will be used when debugging. The analyzed program is in the center of the program panel, and the program statement that is about to be executed is marked. Using two program panels side-by-side, *Requirement 1 (Side-by-side view of the two executions)* is fulfilled. The user can set breakpoints at any code line, thus realizing *Requirement 2 (Marking of low variables)*. When clicking the *Continue* button, all analyzed programs advance to the next breakpoint, unless a conditional breakpoint is activated before. Below the analyzed program, the variable inspector shows the values of the program variables at the current point of execution and the return value of the main method. The user can choose to hide certain variables. The variable inspector realizes *Requirement 4 (Side-by-side view of the two states)*. Note that each program panel has a unique identifier (e.g., the highlighted program panel in Figure 5.1 has identifier *A*) that is used to refer to the program a variable belongs to when adding watch expressions and conditional breakpoints (see Section 5.3.3).

5.3.3 Watch Expressions and Conditional Breakpoints

In order to analyze noninterference properties while debugging the program, the user needs to constantly compare the values of the variables in the two executions. In the context of checking noninterference, the user needs to check whether the low variables are equal during the two executions. Doing this in each step of the debugging process would be a tiresome task. To help with this, *watch expressions* and *conditional breakpoints* can be inserted by the user in the highlighted areas **3** and **4** of Figure 5.1 respectively. Watch expressions are C expressions which can contain variable identifiers from any of the analyzed programs. They help the user with the comparison of values at the two points of execution. At each point of the debugging process, the value of the expression is computed, and the result is displayed.

Conditional breakpoints are boolean C expressions which are evaluated at every point of execution that is reached when using the step sizes set by the user. They help the user to find the execution points of interest. If the expression of the conditional breakpoint evaluates to true, then the execution of the analyzed programs halts at the execution points in which

this evaluation occurred. Conditional breakpoints allow the user to search for execution points in which relational invariants (e.g., a low variable is assigned different values during the two executions) are violated. Using the *OPT* button (see Figure 5.1) for a given expression or conditional breakpoint allows for the setup of a program scope, which consists of a starting line number and an end line number. Then, the given watch expression or conditional breakpoint is only evaluated when the program execution is between those two line numbers and is otherwise ignored. Thus, if the program execution is outside the specified scope, the value of a watch expression will be unknown, and—in the case of conditional breakpoints—the execution will not halt outside the scope. Watch expressions and conditional breakpoints realize *Requirement 5 (Assisted variable comparisons)*.

5.4 Discussion

In this section we show how the approach can be employed to analyze counterexamples of the noninterference property (as defined in Definition 2.2), and we also discuss the properties that are supported by DIbugger.

Examples. We show how the user can be assisted in understanding the counterexample for two example programs, shown in Listings 5.2 and 5.3, that respectively contain a data and a control dependency (see Section 2.6.1) between the high input (parameter `h`) and the low output (the returned value).

```
1  int main(int h, int l){
2     int low = 0;
3     if(l==0){
4        low = l;
5     }
6     else{
7        low = l + h;
8     }
9     return low;
10 }
```

Listing 5.2: Data dependency example

For the program in Listing 5.2 the approach for generating noninterference tests presented in Chapter 4 found a counterexample containing the inputs $h = 1, l = 1$ for execution A and $h = 2, l = 1$ for execution B. The user loads the program in two program panels with the two respective inputs. When entering debug mode, the user sees that execution A finished with 2 as the return value, while execution B returns 3. Thus, the noninterference property is violated. The user knows that the variable `low` must have the same value in each of the two analyzed executions. He uses a conditional breakpoint with "`A.low != B.low`" as a condition to automatically find the line in which the variable `low` is assigned a different value in the two executions. For the

example in Listing 5.2 this conditional breakpoint automatically halts the two executions at line 7. As an alternative, the user can set breakpoints at every assignment to the variable `low` (lines 2, 4, and 7). With the *Continue* button, each of the two executions advances to the next breakpoint, and the user compares the values of `low` at each of those lines in the two executions. At line 7 he sees that the variable has different values and notices the dependency between the high input and the low output. The user can use the watch expression "`A.low == B.low`" to more easily compare the value of `low` in the two executions.

Now we consider the program in Listing 5.3, which contains a conditional dependency between the high input and the low output. Depending on the value of the high input `h`, the `then` or the `else` branch is taken. The test generation approach found a counterexample containing the inputs $h = 0$, $l = 1$ for execution A and $h = 1$, $l = 1$ for execution B. In this example the user can proceed as in the previous example and manually set breakpoints at every assignment to `low` (lines 2, 4, and 7). When running the two executions using the *Continue* button, execution A will reach line 4, and execution B will reach line 7 at the same time. The user thus notices the control dependency. The same two execution points can be reached also by using the conditional breakpoint with "`A.low != B.low`" as a condition instead of the manual breakpoints.

```
1  int main(int h, int l){
2     int low = 0;
3     if(h==0){
4        low = l;
5     }
6     else{
7        low = l + 1;
8     }
9     return low;
10 }
```

Listing 5.3: Control dependency example

Already from those two simple examples we can see that using conventional debuggers, which only allow the inspection of a single program execution, would be more difficult. This is because in a conventional debugger the user must guide the debugging process for each program separately. He cannot use watch expressions and conditional breakpoints to find pairs of execution points which violate the noninterference property.

Supported properties. The approach for analyzing relational counterexamples presented in this chapter supports the inspection of counterexamples of any *k-safety property* (i.e., those properties that can be refuted by at most k traces, according to Clarkson and Schneider [2010]). By far the most researched are 2-safety properties. Besides noninterference, another such property is program equivalence. Verification approaches that check program

equivalence for C programs (e.g., Kiefer et al. [2018]) or for PLC software (e.g., Beckert et al. [2015]) are available and can provide counterexamples. In the context of computational social choice, relational properties such as monotonicity are also being verified using formal methods (e.g., in Beckert et al. [2016a]), and counterexamples can be obtained.

5.5 Conclusion

We presented an approach for analyzing counterexamples of the noninterference property and its implementation. As part of the noninterference framework, the approach is applied on the counterexamples that are generated with the automatic test generation approach presented in Chapter 4 but can also be adapted for other k-safety properties. Related work in the area of combining formal methods with program debugging is discussed in Section 9.1.2.

6 Using SDGs to Assist Deductive Verification and Testing

6.1 Introduction

In this chapter we describe an approach proving noninterference that combines an SDG-based approach (see Section 2.6) with deductive verification (see Section 2.3) and automatic test generation (see Chapter 4). Thus, the approach presented in this chapter is contained in both the part of the noninterference framework that searches for noninterference violations and the part of the framework that attempts to prove that a given noninterference property holds for a given program. This chapter is based on work by the author previously published in Herda et al. [2018]. Under the supervision of the author, the student Joachim Müssig has contributed to the implementation of the approach described in this chapter.

Motivation Various approaches and tools for checking noninterference exist. Some have a high degree of automation, yet produce many false alarms, as they over-approximate the dependencies in the program. Others are more precise, but require more effort and user interaction. Approaches that are based on SDGs (e.g., the one presented in Section 2.6) syntactically compute the possible dependencies between the program statements and check whether the low output depends on the high input. Whereas they scale very well, such approaches over-approximate the actual dependencies (see Definition 2.20) in the program, which results in false alerts. Logic-based approaches (e.g., the one presented in Section 2.3) have a higher precision (i.e., they produce less false alarms), as they also consider the semantics of the program statements. However, they have a lower scalability. In logic-based approaches the proof obligation is to show that the terminating states of two program executions are low-equivalent, assuming that the two initial

states are low-equivalent[1]. False alarms only occur when the system fails to find a proof in the allotted time even though the proof obligation is valid. Compared to proving a functional property, proving noninterference using this approach requires a quadratic number of program execution paths to be checked. Approaches for *test generation* are also affected by this: a quadratic number of tests is necessary to achieve the same coverage as for a functional property.

Thus, SDG-based and logic-based approaches are complementary and trade precision for scalability. It is therefore reasonable to assume that these approaches can be combined in a way that program parts for which the scalable SDG-based approach has shown that they do not affect the noninterference property can be excluded from the more precise but less scalable logic-based approach, thus allowing the advantages of both kinds of approaches to be gained.

Contribution C1.4 In this chapter we describe an approach that uses an SDG-based analysis to simplify a program for a given noninterference property. The simplified program is then handled with deductive verification and test generation. Note, however, that our simplification enables other approaches (e.g., type-system based analyses) to be used together with the SDG-based approach. These simplified programs are noninterference equivalent (with respect to the analyzed noninterference property) to the original program. This means that every noninterference violation in the simplified program has a corresponding noninterference violation in the original program and vice versa. The necessary effort for the second approach (i.e., deductive verification and test generation) is decreased by the our approach by: (1) excluding pairs of high inputs and low outputs and the possible noninterference violation they represent; (2) excluding execution paths in the program; and (3) excluding programs statements.

We have implemented our approach using the JOANA tool (see Section 2.6) for the SDG-based analysis and the KeY system for proving noninterference (see Sections 2.3) and automatic test generation (see Chapter 4). Applying our approach on some example programs has shown that it increases the scalability of the deductive verification and the test generation techniques, allowing them to focus just on those program parts that may cause violations of a specified noninterference property.

Structure of the chapter. In Section 6.2 we present a running example that is used to demonstrate our ideas. Section 6.3 describes how we generate a simplified program. In Section 6.4 we show how the simplified program can help with verification, and in Section 6.5 we explain how the simplified program helps with noninterference test generation. In Section 6.6 we discuss

[1]In this thesis we define low-equivalence using Definitions 2.1 or 2.4.

theoretical and technical details of our approach, and in Section 6.7 we conclude.

6.2 Running Example

In this chapter we use the program shown in Listing 6.1 to explain the concepts that we introduce. It contains the method `secure` that has a secret input—`high`—and a public output—the return value of the method `secure`.

```
1  public int secure(int high, int low) {
2     if(low == 5){
3        low = identity2(low, high);
4     }
5     else{
6        if(low == 2){
7           low = identity1(low, high);
8        }
9        else{
10          low = identity2(low, high);
11       }
12    }
13    return low;
14 }
15
16 public int identity1(int low, int high) {
17    low = low + high;
18    low = low - high;
19    return low;
20 }
21
22 public int identity2(int low, int high) {
23    return low;
24 }
```

Listing 6.1: Running example

The method is noninterferent, because the returned value does not depend on the value of the parameter `high`. Noninterference for this program can be proved using deductive verification. The theorem prover checks nine symbolic execution paths: it analyzes two program executions as required by Definition 2.2, and for each program execution it considers three cases, namely that the input `low` (a) is 5, (b) is 2, or (c) has any other value. SDG-based approaches (see Section 2.6) for checking noninterference report a possible violation, because the called method `identity1` contains a possible dependency between its return value (that gets afterwards assigned to `low`) and the parameter `high`. This dependency, however, only affects the path in which the initial value of `low` is 2. Hence, for the other two execution paths noninterference is guaranteed by the SDG-based approach. Thus, noninterference is guaranteed under the condition that the program input respects the path condition ($\texttt{low} = 5$ and $\texttt{low} \neq 5 \wedge \texttt{low} \neq 2$ respectively) of those two execution paths. In Section 6.3, we explain how we can simplify the

running example program, and we show the advantages of this simplification in proving the noninterference of the program using a logic-based approach from Section 2.3 or testing it using the approach from Chapter 4.

6.3 Generation of the Simplified Program

In this section we explain on the running example shown in Listing 6.1 how we simplify a program for a given noninterference property. We use the tool JOANA (see Section 2.6) to explain the concepts related to the SDG-based approach. However, the simplification can be implemented with other SDG-based tools as well. Throughout this chapter we refer to a (possible) program execution described by a CFG-path as *execution path*. If an SDG-path is a sub-path of a CFG-path (i.e. the nodes of the SDG-path appear in the same order in the CFG-path) we say that the execution path described by the CFG-path corresponds to the SDG-path. An SDG-path has one or more corresponding execution paths. For the approach in this chapter we consider sequential, deterministic, and terminating programs for which there is a CFG available. For the definitions and properties of the simplified programs we assume that the SDG-based and logic-based approaches work on the same language. This is not the case, however, for the implementation of the approach, as JOANA works on Java byte code, and KeY works on Java source code. We discuss this in Section 6.6.

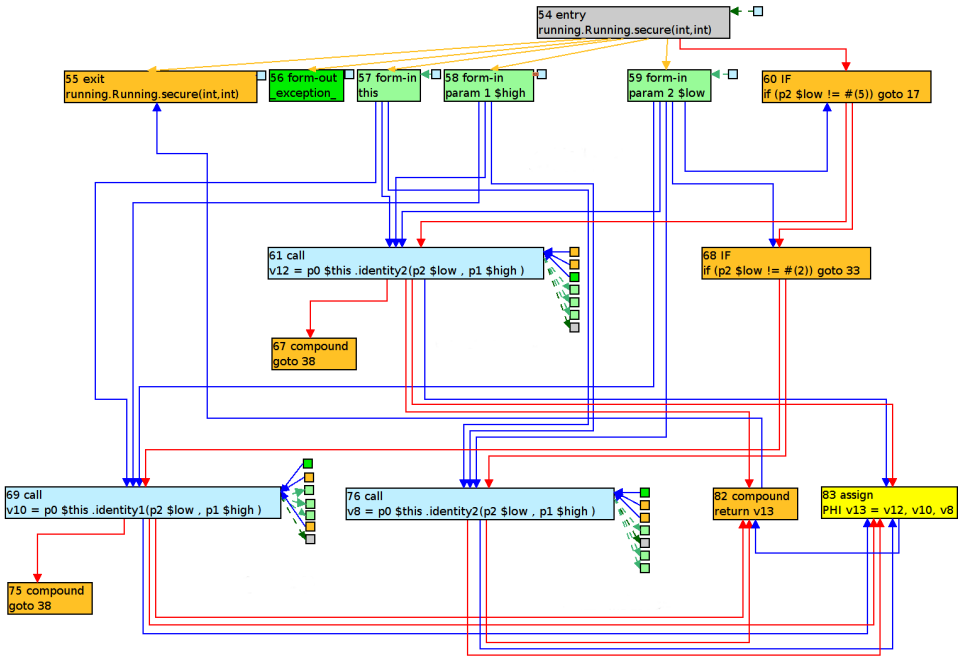

Figure 6.2: The SDG of the running example

Figure 6.2 presents the SDG generated by JOANA for the `secure` method in the running example (Listing 6.1). Note that in order to keep the graph simple, the dependencies inside the three calls of the methods `identity1` and `identity2` have been hidden, and only the method call nodes are shown. We will use this simplification also later in the chapter, when we describe paths in the SDG. Of particular interest are node 58, which represents the parameter `high` of the method `secure` and node 55—the exit node of the method. These two nodes are annotated with `high` and `low`, respectively.

As shown in Section 2.6, SDG-based program analysis approaches, such as the one implemented by JOANA, detect possible noninterference violations through graph analysis by using a special form of conditional reachability analysis—slicing and chopping—at the SDG level. JOANA reports a possible noninterference violation whenever there exists a path from a node in the SDG that is annotated as `high` to a node annotated as `low`.

Because the dependencies in the SDG are an over-approximation of the actual dependencies (see Definition 2.20) in the program, if no SDG-path corresponding to a possible dependency between a high input and a low output is found, the program is guaranteed to be noninterferent. However, when there exists an SDG-path between a high input and a low output, the program may still be noninterferent. To check whether the violations that are reported by the SDG-based approach are real violations of the noninterference property, we try to either disprove them as false positives or to generate concrete program inputs that showcase them.

Since the program can have multiple annotated high inputs and low outputs, JOANA may show at this point that there is no dependency between certain pairs of high inputs and low outputs. Since the SDG-based approach is sound, the noninterference property for those pairs does not need to be checked any further. In order to distinguish between real and false alarms in the other SDG-paths that JOANA reports as leading to possible noninterference violations, we use a second technique (verification or test generation).

For the running example shown in Listing 6.1, JOANA reports a possible violation that contains an SDG-path from the parameter `high` to the return value of the method `secure`. JOANA successfully determines that there is no dependency between the parameter `high` and the return value of the two calls of to the method `identity2`. Since the edges in the SDG represent dependencies that are defined using purely syntactical criteria, JOANA cannot reach the same conclusion for the call of to `identity1`. Thus, the SDG-path $58 \rightarrow 69 \rightarrow 82 \rightarrow 55$ is reported as a possible violation (the numbers correspond to the node ids in Figure 6.2). Note that this path is shortened by an operation similar to the collapse of the call nodes in Figure 6.2. The path reported by JOANA contains in fact an *actual in* node—representing an input of the method call and an *actual out* node—representing an output of the method call. These two nodes are connected

by a *summary edge* that encodes the dependence of the actual output node on the actual input node (see Section 2.6.1).

Despite this false alert, JOANA was still able to determine that there is no dependency between the high input and the low output along the two execution paths in the program where the method `identity2` is called. When we use a second, more precise approach, it makes sense to skip the analysis of those two execution paths.

A noninterference violation reported by the SDG-based approach has a corresponding chop (see Definition 2.22) for a high input and low output pair represented by nodes in the SDG. This chop represents an over-approximation of those nodes that are influenced by the high input and that at the same time influence the low output.

The chop that was created by JOANA for the running example contains only one path: $58 \rightarrow 69 \rightarrow 82 \rightarrow 55$. Based on this, we know that only the statements of the nodes in this path can be relevant for the potential dependency between the high input and the low output. Using a chop and the analyzed program we can generate a simplified program, which we call *chop-based program* that is defined as follows:

Definition 6.1 (Chop-based program)**.** For a given program P and a chop $C(n_h, n_l)$, the chop-based program P_C is constructed by removing all instructions from P that have no corresponding node in $C(n_h, n_l)$.

Listing 6.3 shows the chop-based program generated from the chop of our running example in Listing 6.1. Note that SDG-based program slices are not executable in general. We discuss this aspect in Section 6.6.

```
1 public int secure(int high, int low) {
2     low = identity1(low, high);
3     return low;
4 }
```

Listing 6.3: Chop-based program for the running example

Theorem 6.1. *Given a program P and a chop $C(n_h, n_l)$, if the noninterference property (according to Definition 2.2) with respect to the high input corresponding to n_h and the low output corresponding to n_l holds for the chop-based program P_C, then it holds for the original program P as well.*

Proof. The soundness of the SDG-based approach guarantees that the high input can influence the low output only along the SDG-paths from the high input to the low output and only by the program instructions whose corresponding SDG-nodes are on those paths. All those SDG-paths make up the chop that constitutes the reported violation. Since the chop-based program contains—by construction—all program instructions corresponding to the nodes in the chop, it contains the sequences of program instructions that could allow the low output to be influenced by the high input in the

original program. Thus, if the noninterference property holds for the chop-based program, then the high input does not influence the low output along SDG-path of the chop, and the noninterference property holds for the original program as well. ◁ ◁

The chop-based program is greatly simpler than the original. This fact can help to considerably reduce the necessary effort for deductive verification. However, some statements that affect path conditions (see Definition 2.15) under which the high input may affect the low output in the original program may be lost in the chopping process. Therefore, the chop-based program may allow executions that are not possible in the original program. For this reason, a noninterference violation of the chop-based program does not necessarily have a corresponding violation in the original program, since the path conditions that need to hold for the violation to occur may be unsatisfiable in the original program. Suppose, for example, that the method `identity1` in the running example is noninterferent only if the condition `low == 2` holds when the method is called. This condition holds in the original program, but not in the chop-based program. In this case the original program would be noninterferent, but the chop-based program would violate the noninterference property. Thus, whereas the verification effort can be significantly reduced for some programs, the missing path conditions may cause the noninterference property to become impossible to verify for other programs.

Furthermore, the chop-based program is of little use for automatic test generation, since the test data for this program may take another execution path when used to test the original program. For example, the input parameter `low` in the original program must be 2 for the `identity1` method to be called, but this is not true for the chop-based program. Thus, the coverage (see Section 4.4) achieved when testing the simplified program does not translate to a coverage in the original program. Moreover, it may be that a noninterference violation detected when testing the chop-based program is not a violation in the original program. For example, to find inputs that showcase the possible noninterference property in the running example, when the method `identity1` is called, the value of `low` has to be 2 in order for the noninterference violation to be observable in the original program. Since the path condition is no longer present in the chop-based program, the test generation approach can generate two inputs that lead to a potential noninterference violation in the chop-based program but not in the original program.

To overcome the problems of the chop-based programs, we introduce a new kind of program which we call *simplified program*. The simplified program is based on the backward slice (see Definition 2.21) of the low output and has some execution paths excluded. We decide which paths to exclude by analyzing both the SDG of the entire program and the chop, to determine

whether a predicate node (e.g., a node representing an if-statement, see Definition 2.12) has to be true or false for noninterference violation to occur.

Definition 6.2 (Analysis of predicate nodes)**.** Let $n_b \in S_{\text{bw}}$ be a predicate node in the backward slice of a node n_l (the node corresponding to the low output). Let N_{true} be the set of successor nodes following the edge labeled T of n_b in the CFG, and let N_{false} be the successor nodes following the edge labeled F 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, we define n_b to be a condition that *must be false* if the chop contains nodes from N_{false} and no nodes from N_{true}.

Now we can show which program paths may be removed from a second program analysis.

Theorem 6.2. *Given a high input and a low output corresponding, respectively, to the SDG nodes n_h and n_l and a predicate node n_b that must be true (respectively false), any execution path of the original program along the false (true) branch of n_b will not lead a violation of the noninterference property.*

Proof. This property results from the soundness of the SDG-based approach: a dependency between n_h and n_l can occur only along an SDG-path in the chop $C(n_h, n_l)$. However, because the node n_b must be true (respectively false), we know that no false (true) successor of n_b is in the chop, thus the program executions determined by the chop do not include any execution along the false (true) branch of n_b. ◁ ◁

Theorem 6.2 allows us to exclude the execution paths that are guaranteed to not lead to a noninterference violation from the analysis in the second step, thus reducing the testing and verification effort. Because both deductive verification and automatic test generation approaches that we use are based on symbolic execution, 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 the purpose of verification and reaches a disruptive statement, the proof closes automatically for that branch. Test generation also immediately halts for that path once the symbolic execution reaches a disruptive statement.

We can now define simplified programs.

Definition 6.3 (Simplified program)**.** Let n_h and n_l be two nodes in the SDG for a program P (corresponding to a high input and a low output). We construct the simplified program P_S from P by:

1. Removing all nodes that are *not* in the backward slice $S_{\text{bw}}(n_l)$.

2. Analyzing the remaining predicate nodes and adding disruptive statements on their true (respectively false) branches that cannot lead to a noninterference violation according to Theorem 6.2.

We can now show that the simplified program is noninterference equivalent to the original program.

Theorem 6.3. *Given a high input and a low output in a program P with corresponding nodes n_h and n_l in the SDG for P, the simplified program constructed according to Definition 6.3 is noninterferent with respect to n_h and n_l if and only if the original program is noninterferent with respect to n_h and n_l.*

Proof. If the simplified program is noninterferent, then along none of its SDG-paths does the high input influence the low output. The simplified program contains all SDG-paths from the chop $C(n_h, n_l)$, therefore along none of the chop paths does the high input influence the low output. Due to the soundness of the SDG-based approach, a noninterference violation can occur only along an SDG-path in the chop. Therefore also the original program must be noninterferent.

If the original program is noninterferent, then the backward slice $S_{\text{bw}}(n_l)$ of the low output is noninterferent as well, since for every input the low output of the backward slice is identical to that of the original program. As shown in Theorem 6.2, adding disruptive statements excludes only execution paths on which it is guaranteed that the high input does not affect the low output. Hence the simplified program must be noninterferent as well. ◁ ◁

Next we will show that the simplified program is also useful for testing noninterference: a counterexample for the simplified program is also a counterexample for the original program.

Theorem 6.4. *Given a high input and a low output in a program P with corresponding nodes n_h and n_l in the SDG for P, two concrete high inputs h_1 and h_2 for n_h (with the same low input) lead to two different low outputs, l_1 and l_2, in n_l in the simplified program P_S if and only if h_1 and h_2 lead to the same two different low outputs l_1 and l_2 in the original program P.*

Proof. The simplified program P_S is a backward slice with respect to the low output, in which some paths that are guaranteed not to lead to a noninterference violation are excluded. The two high inputs that lead to two different low outputs in the simplified program cannot have taken one of the excluded paths—otherwise these paths would not have been excluded (Theorem 6.2). Since the remaining, not excluded, execution paths of the simplified program are those of the backward slice with respect to the low output, the inputs that take those execution paths will lead to the same low outputs in the original program.

If the two high inputs lead to two different low outputs in the original program, then the simplified program will lead to the same two different low outputs, because the simplified program is a backward slice of the original program and because the execution paths cannot contain an excluded branch, as on those paths the high input does influence the low output.

◁ ◁

The chop reported by JOANA for the running example contains nodes 58, 69, 82, and 55 (see Figure 6.2). In the backward slice of the low output (i.e., of node 55) there are two predicate nodes, 60 and 68, corresponding to the two if-statements in the example program. Analyzing the two predicate nodes, our approach automatically determines that the first if-statement has to take the *false* branch and the second if-statement has to take the *true* branch for a noninterference violation to occur.

```
1
2  public int secure(int high, int low) {
3     if(low == 5){
4        disruptExecution();
5        low = identity2(low, high);
6     }
7     else{
8        if(low == 2){
9           low = identity1(low, high);
10       }
11       else{
12          disruptExecution();
13          low = identity2(low, high);
14       }
15    }
16    return low;
17 }
```

Listing 6.4: Simplified program for the running example

The program shown in Listing 6.4 is the simplified program for the running example. While in general the backward slice of the return statement can be much smaller that the original program, for the running example it contains the entire program. Nevertheless, our approach is able to determine that the paths leading to the call of the `identity2` method cannot lead to a noninterference violation, and it adds two `disruptExecution()` statements, which stop symbolic execution when verifying the running example or generating tests.

6.4 Verification of the Simplified Program

In this section we show how using the simplified program can assist in reducing the verification effort compared to the effort needed to verify the original program. We also discuss the noninterference properties that can be handled by a prototypical implementation of our approach, where we use JOANA (see Section 2.6) as the SDG-based analysis tool and KeY (see Section 2.3) as the deductive theorem prover.

Using KeY to verify noninterference for the original version of our running example (Listing 6.1) requires 771 rule applications. After the symbolic execution of the programs in the two modal operators is finished, nine proof tree branches remain to be closed. This is to be expected, as there is one such branch for each combination of paths in the two program executions, as explained in Section 6.2.

For the chop-based program shown in Listing 6.3, verification needs 298 rule applications, and only one proof goal remains to be closed after symbolic execution, as there is only one possible path combination. As explained in Section 6.3, however, noninterference of the chop-based program is a sufficient but not a necessary condition for noninterference of the original program.

To obtain a necessary condition ensuring that noninterference is preserved when the program is simplified, path conditions need to be preserved, which is the case in the simplified program (according to Definition 6.3). The verification of the simplified program with preserved path conditions, displayed in Listing 6.4, requires 511 rule applications. In this case, the symbolic execution halts when one of the two program runs reaches a path that has already been deemed secure by JOANA, and the corresponding proof branch is closed. Thus, of the nine proof goals remaining after symbolic execution, eight are trivially closed.

The running example showcases how the SDG-based approach can assist verification by excluding statements and execution paths from the program. The exclusion of execution paths is especially useful when dealing with the noninterference property (according Definition 2.2). If the original program has n execution paths, the verification process must prove that the noninterference property holds for n^2 execution paths. By adding a single disruptive statement, the number of execution 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 at 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 required for the original program.

The statements that are removed from the original program can also lead to a dramatic decrease in the effort for the theorem prover to prove the noninterference property. Consider the example in Listing 6.5. The

method `secure` contains a call to the method `sort` that has no influence[2] whatsoever on any potential dependency between the parameter `high` and the return value of `secure`. The verification of the method `secure` would normally be done using a verified method contract for `sort`. Using the method contract in the noninterference proof of `secure` increases the size of the proof. Moreover, the method `sort` itself needs to be specified and verified before it can be soundly used in the proof of `secure`, thus increasing the workload of the verification engineer. In an example as simple as the one in Listing 6.5 a trivial contract for the method `sort` is sufficient. However, the verification engineer still is required to look into the code, notice that the call of the `sort` method has nothing to do with a potential dependency between the high input and the low output and then to specify and verify it. Our approach can automatically detect such statements and soundly remove them.

```
1 public int[] a;
2 public int secure(int high, int low) {
3    low = high * 0;
4    sort(a);
5    return low;
6 }
```

Listing 6.5: Example containing a complex method call

```
1 public int[] a;
2 public int secure(int high, int low) {
3    low = high * 0;
4    a = new int[5];
5    for(int i = 0; i < a.length; i++){
6       a[i] = low;
7    }
8    return low;
9 }
```

Listing 6.6: Example for multiple low outputs

Another way in which the generation of simplified programs can help the verification process is by analyzing each potential dependency between a pair of high input and low output individually. Consider the program in Listing 6.6. We regard the parameters `high` and `low` to be the high and respectively low inputs, and both the return value of the `secure` method and the potentially thrown exception of this method are regarded as low outputs. In this case, two simplified programs will be generated—one for the potential dependency between the parameter `high` and the return value and one for the potential dependency between the parameter `high` and the thrown exception. For the first pair, the array generation with `new` and the loop initializing the array are removed (the chop-based program and the simplified program are identical), thus making the absence of this dependency trivial to verify. For the second pair, no statements are removed; however,

[2]We assume here that the method does not throw any exception.

the verification engineer can verify the absence of a thrown exception by specifying the normal termination of the method as a functional property and doing a functional proof. Note that an SDG-based approach has false alarms for this program because its purely syntactical analysis fails to "see" that at line 3 there is no dependency between `high` and `low`.

Most tools support the classical noninterference property (see Definition 2.2), in which the low-equivalence relation is equality. KeY in addition allows the noninterference property to be defined by requiring object structures in the two low-equivalent program states to be (only) isomorphic (see Definition 2.4). This is useful for showing noninterference for methods that create new objects, because two independent program runs will generate different references but isomorphic structures. KeY also allows an expression to be declassified (i.e., the attacker is allowed to know the value of the declassified expression, but no more than that; see Definition 2.3). It is important to note that both these extensions of the noninterference property are relaxations. Thus, a program that fulfills the noninterference property as defined in Definition 2.2 will automatically fulfill the extended properties as well. Thus, we can use our approach to generate the chop-based or the simplified program even when we are attempting to prove the extended noninterference property.

6.5 Testing the Simplified Program

We now show how the simplified program can be used to reduce the effort required for generating noninterference tests. Furthermore, we show that the simplified program is more appropriate than the original when measuring the test coverage of an noninterference test suite.

The program in the running example contains three execution paths. Thus, the automatic test generator will attempt to generate 3^2 input pairs. However, only three of these pairs are low-equivalent because—for this program—the branch that is taken is determined by the low input. Thus, our automatic test generator generates a test suite with three noninterference tests. When running the automatic test generator on the simplified program, only one test will be generated for the case in which both `low` inputs are 2. This is possible due to the inserted statements in the simplified program that disrupt symbolic execution. When the symbolic execution reaches a `disruptSymbolicExecution` statement, no test is generated for that path. For the simplified program, the test generator attempts (and succeeds) to generate a noninterference test only once, compared to the nine attempts for the original program. Hence, in this case eight test generation attempts (calls to the SMT solver) that cannot lead to a noninterference violation are soundly skipped.

Testing the simplified program can reduce the number of generated tests also by removing program statements that are not relevant for the computation of the low output. When testing (using tests generated with the approach from Chapter 4) the program shown in Listing 6.5, the method `sort` would be inlined during the symbolic execution phase, thus requiring a large number of execution paths to be tested. By removing the call to the `sort` method, the simplified program contains only one execution path. Removing statements also leads to a reduction in the computational resources that are needed to run the noninterference tests.

An additional benefit of using our two-step approach is the fact that we treat each high input and low output pair separately, by generating a simplified program for each pair for which the SDG-based approach reports that a noninterference violation may occur. If a noninterference violation is found during testing, then the user can easily identify the high input and low output of the noninterference violation, by noticing for which simplified program the test fails.

It is difficult to define an appropriate coverage criterion for testing noninterference properties that provides a measure on how good a test suite is. Ideally, we would like to have full path coverage. This, however, is not always achievable (even ignoring the fact that loops may cause the program to contain an infinite number of paths). In Section 4.4 we defined a relational bounded path coverage criterion that is suitable for noninterference test suites. However, some program paths may have unsatisfiable path conditions (i.e., a test input taking such a path cannot be generated). Because the inputs of the two program executions are required to be low-equivalent, many pairs of execution paths in the two executions are incompatible. Therefore, it is often not possible to generate two low-equivalent inputs satisfying the path conditions of a pair of execution paths. This results in a low path coverage even though it is not an indication for a badly designed test suite. A higher coverage is not achievable in such cases. Considering our running example, only for three out of the nine path pairs, a noninterference test can be generated; this results in a relational bounded path coverage of only 33%. The execution paths excluded from the original program are paths where the output of the program does not depend on the high input. Therefore, those paths are determined only by the low input and are likely to form incompatible pairs with other paths, thus lowering the achieved path coverage. By excluding them, the achieved path coverage becomes a more useful indicator for the thoroughness of testing. For the simplified program constructed from our example program, three tests achieve full relational bounded path coverage.

6.6 Discussion

The novelty of our approach is that we soundly bridge the gap between two kinds of approaches: the scalable over-approximating SDG-based approach and the more precise but less scalable logic-based one. This is achieved by automatically transforming the output of the SDG-based approach into an input of a more precise approach, thus simplifying the analyzed program. Furthermore, the simplified program that we generate is not a mere slice of the original program; by taking advantage of the noninterference property, we also exclude entire program branches from the analysis with the precise approach. We have shown that a single branch exclusion can lead the more precise approach to handle only one quarter of the execution paths that would otherwise need to be handled. This is a crucial advantage, as existing precise approaches for checking noninterference suffer from an exacerbated path explosion problem.

We defined two types of simplified programs: the chop-based program (see Definition 6.1) and the simplified program (see Definition 6.3); both can be useful for verification, but sometimes the chop-based program makes verification impossible. To help the user chose between the two versions, we devise a criterion based on the following theorem:

Theorem 6.5. *Given the SDG of a program and a chop representing a reported noninterference violation, then if every predicate node in the SDG is also present in the chop, the chop-based program is noninterferent if the original program is noninterferent.*

Proof. All branching conditions depend on the high input, otherwise the corresponding predicate nodes would not be in the chop. The consequence for such a program is that all execution paths must lead to the *same* low output, otherwise the low output would be conditionally dependent on a high output. Removing statements that do not depend on the high input would change the value of this output but the noninterference property remains unaffected. ◁ ◁

For such programs that fulfill the requirement from Theorem 6.5 the chop-based program is better suited to assist verification, since no path can be excluded when generating the simplified program, as the predicate nodes have at least one true and one false successor in the chop. The simplified program in this case is the backward slice based on the low output of the original program, and it may contain more statements than the chop-based program. The inputs that lead to a noninterference violation in the chop-based program will as well lead to one in the original program. Thus, the chop-based program can even be used for testing. We check whether the chop fulfills the condition described in Theorem 6.5 and respectively use the chop-based or the simplified program.

SDG-based forward and backward slicing—as done in JOANA—can result in a program that is not executable or may incorrectly handle jump statements, such as `goto`, `break`, or `continue`. This is not a problem for JOANA, since it does not need to generate any code for its analysis. For the second step in our approach, however, having an executable program is of great importance. This is a problem when implementing our approach using the slicers provided by JOANA, but our approach is nevertheless feasible, since (as stated in [Hammer, 2009, Chapter 2]) various solutions (e.g., by Ball and Horwitz [1993]; Choi and Ferrante [1994]; Agrawal [1994]; Harman and Danicic [1998]; Harman et al. [2006]; Abadi et al. [2012]) have been proposed that enhance SDG-based slicing and enable the generation of executable program slices. For our prototype we deal with this problem by generating an over-approximation of the chop-based program and restrict ourselves to programs without jump statements. Thus, we obtain executable slices by not removing lines containing certain types of statements such as constructors or static initializers and by supporting only programs without jump statements.

While JOANA supports full Java (minus reflection), KeY handles sequential Java programs only. KeY also requires that the source code or method contracts of library methods to be available. The implementation of our approach using these two tools thus supports the same Java subset that KeY does.

The scalability of our approach is bounded by the scalability of the two tools it uses. JOANA is able to handle programs of up to 100k LOC, whereas KeY can handle programs of up to 1000 LOC. The generation of the backward slice and the chop that are necessary for constructing the simplified program are anyway the operations that JOANA performs in its analysis, and the analysis of the predicate nodes is a graph reachability problem similar to how slicing is done in the SDG-based approach. The main bottleneck of our approach is therefore the analysis using KeY rather than the generation of the simplified program. Thus, the most favorable case for using our approach is when an original program that is too large for KeY is simplified and reduced to a size that KeY can handle. This is achievable either by removing program parts as done for the program in Listing 6.5 or by excluding execution paths as done for the running example.

At this point we would like to discuss some challenges related to the implementation of our prototype using JOANA and KeY. First, the two tools do not work on the same programming language (respective level). While JOANA works on Java bytecode that is brought into a single static assignment (SSA) form, KeY works on Java source code. For the soundness of our prototype (and not for the soundness of our approach), we must assume 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 into source code statements. We are able to determine the source code line (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, some source code statements may not even have a corresponding byte code statement. The chop-based and the simplified program as defined in the previous section are thus impossible to generate using these tools. Instead we generate an over-approximation of the chop-based and the simplified programs 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.

2. No such node is in the chop (for the chop-based program) or in the backward slice of the low output (for the simplified program).

To avoid multiple source code statements on the same line, we preprocess the source code program and bring it to a form that only has one statement per line.

A second issue of combining the two tools is the fact that the analysis performed by KeY is done modularly at a 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 a `main` method. If the goal is to verify the whole program, then the approach described in this chapter does not need any adaptation. 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 a main method as an entry point for JOANA's analysis that only calls the method for which noninterference is to be proven. However, the SDG-based analysis can be implemented such that any other method can serve as an entry.

6.7 Conclusion

We have explored an approach for proving or testing the noninterference property of a program that uses SDG-based analysis to remove irrelevant program parts and to exclude execution paths that do not lead to a noninterference violation. For each pair of high input and low output we generate a simplified program. We have shown that the simplified program is noninterference equivalent to the original program. Thus, a noninterference proof for the simplified program is also one for the original program. The same holds for the counterexample. The examples in the chapter show how the simplified program assists in the verification and testing of the noninterference property.

We have discussed implementation details of our approach using JOANA as an SDG-based analysis tool and KeY as both a theorem prover and

a test case generator. Because the two tools work respectively on Java bytecode and Java source code, our prototypical implementation generates an over-approximation of the simplified program. This is only an engineering challenge, our approach can be implemented using tools that work on the same programming language.

The approach presented in this chapter can be used to simplify a program with respect to a specified noninterference property. The simplified program can be used in combination with the approaches presented in Chapters 4, 5, and 7. Related work with respect to combining SDG-based approaches with logic-based approaches and other approaches in the area of information flow security can be found in Section 9.1.

7 Increasing the Precision of SDG-based Approaches

7.1 Introduction

Like Chapter 6, this chapter presents a combination of an SDG-based approach (see Section 2.6) and a logic-based approach (see Section 2.3) with the goal of proving a given noninterference property (according to Definition 2.2) for a given program. Thus, the approach presented in this chapter is contained in the part of the noninterference framework that is responsible for proving noninterference. Unlike the approach presented in Chapter 6, however, the approach presented in this chapter uses the SDG-based approach as the main approach to prove the fulfillment of the specified noninterference property by the program. The logic-based approach is used to increase the precision of the SDG-based approach by showing that some dependencies in the SDG are in fact over-approximations of the actual dependencies (see Definition 2.20) in the program. This chapter is based on work by the author that was previously published in Beckert et al. [2017a] and Beckert et al. [2018a]. Under the supervision of the author, the students Marko Kleine Büning, Holger Klein and Joachim Müssig contributed to the implementation of the *Combined Approach* presented in this chapter. Parts of the results of this chapter were obtained in a "research laboratory" (*Praxis der Forschung*) project in which the student Marko Kleine Büning was supervised by the author. The idea of showing with a theorem prover that a summary edge is an over-approximation and of using the results of a points-to analysis to generate preconditions was first discussed informally in the Master's Thesis of Simon Bischof (see Bischof [2016]), which was not supervised by the author.

Motivation. As already mentioned in the motivation of Chapter 6, approaches that are based on SDGs syntactically compute the dependencies

between the program statements and check whether the low output depends on the high input (see Section 2.6). Whereas they scale very well, such approaches over-approximate the actual dependencies (see Definition 2.20) in the program, which results in false alerts. Logic-based approaches (see Section 2.3), have a higher precision (i.e., they produce less false alarms), as they also consider the semantics of the program statements. However, logic-based approaches can scale to programs of a much smaller size than those that can be handled with SDG-based approaches. Because logic-based approaches check whether certain outputs of a program method depend on certain inputs of that method, we can use a logic-based approach to check dependencies at method call sites in the program. Thus, we can use a more precise approach to check whether the possible dependencies of function calls that are modeled in the SDG are real or over-approximated. Also, because only some of the called functions are analyzed, the logic-based approach needs to check a smaller part of the analyzed program.

Contribution C1.5 In this chapter we present the *Combined Approach*, a novel approach for proving noninterference that combines an SDG-based approach with a logic-based approach, and—in consequence—achieves a higher precision than the solely SDG-based approach. The Combined Approach analyzes the dependencies from possible noninterference violations reported by the SDG-based approach and disproves the dependencies using a theorem prover. While the theorem prover might require user interaction, we automatically generate its proof obligations from the reported dependencies.

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

Structure of the chapter. Section 7.2 presents the Combined Approach. A prototypical implementation of the Combined Approach is presented in Section 7.3. The implementation is evaluated in Section 7.4, and some theoretical and implementation aspects of the Combined Approach are discussed in Section 7.5. Finally, Section 7.6 concludes.

7.2 The Combined Approach

In this section we describe the Combined Approach for proving that a given program P fulfills a specified noninterference property (according to Definition 2.2). The first step of the Combined Approach consists of running the SDG-based analysis to check the noninterference property for P. If there is no possible dependency between the high input and the low output for P, we

need no further action, as noninterference is guaranteed to hold. If, however, the SDG-based approach detects a possible noninterference violation, we apply the second step of the Combined Approach in order to check whether reported violation is a false positive or a genuine noninterference violation. Since the SDG-based analysis is performed as the first step, the results provided by our approach are at least as good (with respect to precision and scalability) as those of the SDG-based analysis.

The SDG-based analysis creates an SDG that models the possible dependencies between the program parts of P. However, as explained in Section 2.6.1, these dependencies represent an over-approximation of the actual program dependencies (see Definition 2.20). The goal of the Combined Approach is to use a logic-based approach (see Section 2.3) to prove that certain dependencies modeled as edges in the SDG do not represent actual dependencies. If all modeled dependencies between the high inputs and the low outputs reported by the SDG-based analysis are proved, using the logic-based approach, to not exist semantically, then the specified noninterference property is proved to hold for P. We assume that the SDG-nodes corresponding to high inputs and low outputs are annotated as high and low respectively. Let N_h denote the set of all nodes annotated as high and N_ℓ the set of all nodes annotated as low.

The SDG-based approach then returns a set of *violations*. A violation is a pair (n_h, n_ℓ) of a high node $n_h \in N_h$ and a low node $n_\ell \in N_\ell$ such that there is a path from n_h to n_l in the SDG of P. We then call the set $c(n_h, n_l)$ of all nodes lying on a path from n_h to n_l the *violation chop* (see also Definition 2.22). To keep the notation simple, we will also use $c(n_h, n_l)$ for the subgraph induced by those nodes. If the set of all violation chops, denoted by C_V, is empty, the SDG-based approach guarantees noninterference. If, however, there is a false positive, C_V contains at least one chop. The idea of the Combined Approach is then to validate each violation chop $c(n_h, n_l) \in C_V$ and attempt to prove that the possible dependencies in the chop do not exist on the semantic level in program P. We prove this by showing that each violation chop is interrupted (see Definition 7.1) with the help of a logic-based approach. We interrupt a violation chop by showing that certain summary edges (see Section 2.6.1) in it are unnecessary.

Definition 7.1 (Unnecessary summary edge, Interrupted violation chop)**.** A summary edge $e = (a_i, a_o)$ is called *unnecessary* if the formal-out node f_o corresponding to a_o does not depend on to the formal-out node f_i corresponding to a_i.

A violation chop $c(n_h, n_l)$ is *interrupted*, if we find a non-empty set S of unnecessary summary edges in this chop, such that after deleting the edges in S from the SDG, no path exists between the source n_h and the sink n_l 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 dependency between the formal-in node f_i to the formal-out node f_o corresponding to the summary edge e (Definition 7.2 provides a more precise description of the generated specified program that corresponds to this proof obligation). The proof is done for all possible contexts of the called method. If the proof is successful, we have shown that the summary edge was only inserted as a result of the over-approximation, and we can soundly delete this edge.

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

Algorithm 1: The Combined Approach

The Combined Approach, shown in Algorithm 1, attempts to interrupt each violation chop in C_V. For each violation chop a summary edge is taken, the appropriate noninterference proof obligation (see Section 2.3.1) is generated for the method corresponding to the summary edge, and a proof attempt is made using the theorem prover. If the proof is successful, the summary edge can then be deleted from the SDG, based on Definition 7.1. The order in which the summary edges are checked is established by a heuristic which is explained in Section 7.5. Note that we only need to consider summary edges that belong to a *chop* between high and low. Thus, it is sufficient to regard only a smaller subset of all summary edges. We then check whether this violation chop is interrupted. In this case we can proceed to analyze the remaining violation chops until all of them are interrupted. In case the violation chop is still not interrupted, or the proof attempt is not successful, another summary edge from the violation chop is chosen. If we are able to interrupt every violation chop by deleting unnecessary edges, our approach guarantees noninterference.

Theorem 7.1 (Noninterference Combined Approach)**.** *The Combined Approach guarantees noninterference.*

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

Note that each violation chop is guaranteed to contain at least one summary edge (e.g., the one corresponding to the `main` method). Generating a proof obligation for the `main` method, however, is equivalent to verifying the entire program with the theorem prover. In practice, however, programs are inter-procedural. Thus, there are plenty of summary edges for our approach to check. Nevertheless, the verification of the `main` method with the theorem prover is still the worst case of our approach and can occur in case not enough summary edges of inner method calls can be proved to be unnecessary.

```
1 public int test(int high, int low) {
2    int result = identity(high, low);
3    return result;
4 }
5 public int identity(int h, int l) {
6    l = l + h;
7    l = l - h;
8    return l;
9 }
```

Listing 7.1: Example program

For the example in Listing 7.1, when trying to show that there is no dependency between the parameter `high` to the return value of the method `test`, the SDG-based approach reports a possible noninterference violation, because the return value of the method `identity` depends on the parameter `h` of the same method. This is, however, a mere syntactic dependency and the reported violation is a false alarm.

The reported violation chop can be observed in Figure 7.2 which shows the SDG of the method `test` from Listing 7.1. The SDG-node with the id 50 represents the first parameter (`high`) of the method `test`, which is annotated as *high* while the SDG-node with the id 47 represents the exit node of the method `test`, which is annotated as *low*. The violation chop contains only one path from the high node to the low node that goes through

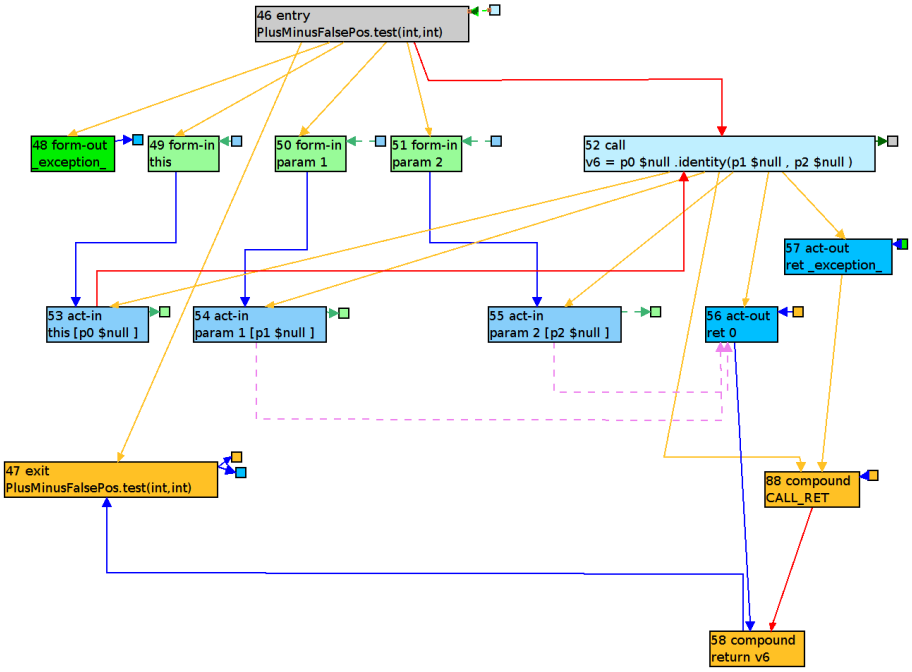

Figure 7.2: SDG of the method `test` in Listing 7.1

the nodes 50 → 54 → 56 → 58 → 47 (the numbers represent the node ids from Figure 7.2). The violation contains the actual-in SDG-node with the id 54 representing parameter h and the actual-out SDG-node with the id 56 representing the return value of `identity`, connected by a summary edge (see Section 2.6). The Combined Approach automatically generates a proof obligation for the logic-based approach which states that the return value of `identity` does not depend on parameter `h`. By proving this, we also prove that the return value of the method `test` does not depend on the parameter `high` of method `test`. Thus, we prove the noninterference of the method. This simple example showcases a major advantage of our approach: the logic-based approach does not need to analyze the entire program, but only those parts that cannot be handled with the SDG-based approach, in this case only the method `identity`.

7.3 Implementation

We implemented the Combined Approach using JOANA (see Section 2.6.2) as the SDG-based tool and KeY (see Section 2.3) as the theorem prover. The implementation is available in Beckert et al. [2019a]. In this section, we show how we generate the proof obligations for KeY in the form of specified Java code.

For the called method corresponding to the summary edge selected by the heuristics, we generate a noninterference method contract such that a successful proof that the method fulfills this contract shows that there is in fact no actual dependency from the formal-in to the formal-out node corresponding to the summary edge. Thus, in order to show that a summary edge $se(a_i, a_o)$ is unnecessary, we prove that there is no actual dependency between the corresponding formal-in node f_i and formal-out node f_o. 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 7.2. The JML specification elements for noninterference contracts that we use in the following definitions are explained in Section 2.3.2.

Definition 7.2 (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 the actual-in parameter a_i. Since there is no dependency between a_i and a_o, the summary edge can be soundly deleted from the violation chop.

As explained in Section 2.6.2, to increase its precision, JOANA uses a points-to analysis which keeps track of the objects a reference o may point to (i.e., the points-to set of o) during runtime. This information is useful, since it may show that two references cannot be aliased. We use the results of the points-to analysis to generate preconditions for the method contracts, as shown in Definition 7.3. Thus, information about the context is transferred from JOANA to KeY, thus increasing the likelihood of a successful proof.

Definition 7.3 (Generation of preconditions)**.** Let $o1$ and $o2$ be references that are also method inputs and P_{o1} and P_{o2} their respective points-to sets. If $P_{o1} \cap P_{o2} = \Phi$ we generate the precondition `requires o1 != o2;`.

Such preconditions can help the theorem prover to show that an output of a method does not depend on an input. However, this proof is only valid for such contexts in which the generated preconditions hold. Because the

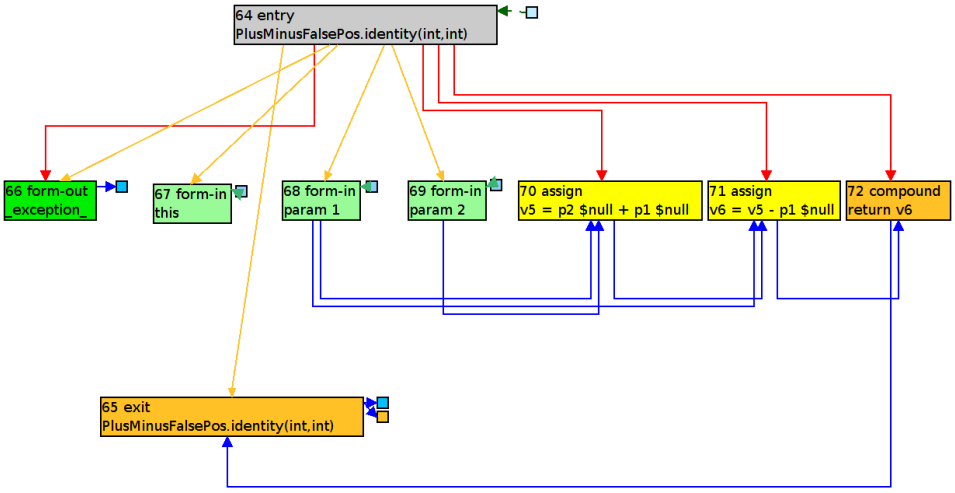

Figure 7.4: SDG of the method `identity` in Listing 7.1

analysis done by JOANA is correct, we can use the preconditions to disprove a dependency at a specific call site of a method. For other call sites, in which the preconditions do not hold, the proof cannot be reused.

```
1 /*@ requires true;
2   @ determines \result \by this, l;
3  */
4 public int identity ( int h , int l ){
5    l = l + h ;
6    l = l - h ;
7    return l ;
8 }
```

Listing 7.3: Specified program generated with the Combined Approach for the program in Listing 7.1

The program in Listing 7.3 presents the specified program which was generated when running the Combined Approach on the program in Listing 7.1. Because JOANA finds no context information that can be useful to KeY, the precondition is set to *true*. The `determines` clause requires that the return value of the method `identity` depends at most on the parameter `l` and on the implicit parameter `this`. These two parameters were chosen because, as shown in Figure 7.4, the method `identity` has two formal-in nodes (with the ids 67 and 69, corresponding to `this` and `l`) other than the formal-in node with id 68 representing parameter `h`. If the return value of `identity` depends at most on `l` and `this`, then it does not depend on h. Therefore, the summary edge between the nodes 54 and 56 is an over-approximation of the actual dependencies in the program.

Note that the Combined Approach removes only summary edges from the SDG. The SDG still contains paths from the high inputs to the low outputs that pass through the called methods which correspond to the summary edges. However, when checking whether there is a path, the SDG-based approach only considers paths that pass through summary edges[1]. Thus, it is sufficient for our analysis to only remove summary edges.

The method contracts generated this way are necessary for proving that a summary edge that is unnecessary. However, in the general case they are not sufficient for a successful proof. If the method contains loops, the theorem prover needs loop invariants. The automatic generation of loop invariants is an active research field, see for example Kapur [2006]; Rodríguez-Carbonell and Kapur [2007]. These approaches focus on functional loop invariants and do not consider noninterference loop invariants.

Determines clauses similar to the ones generated for method contracts, can be used to specify the allowed dependencies inside a loop (see Section 2.3.2). 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 by the user may be necessary. Definition 7.4 shows what determines clauses are generated for loop invariants.

Definition 7.4 (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. The following clause is added to all invariants inside the analyzed method: `determines` L_i `\by \itself;`.

Note that the loop invariants that we generate are not guaranteed to be correct by construction—their validity must be shown with the theorem prover. The loop invariants—if valid—show that after the execution of each loop, the low variables (which are specified as such in the method contract) of the state do not depend on any high variable. Thus, if the computations done after the last loop only depend on the low variables of the state, the loop invariant is strong enough. Still, inside the loops high and low data may be written to other variables which can influence the output of the method. In this case, the loop invariant will not be strong enough to prove the specified noninterference property. We expect that in most cases it is necessary that the user extends the generated loop invariants (especially with functional properties) in order to be able to prove the specified noninterference property.

[1] This way the SDG-based approach does not consider paths that enter a method at a call site and exit that method at a different call site. The SDG-based approach is thus *context sensitive*.

Table 7.5: Evaluation of the Combined Approach

Example	KeY Calls	Time(s)	CA RuleApps	KeY RuleApps
Identity	2	8	309	424
Precondition	2	8	339	454
Excluding	2	8	723	594
Loop Override	2	9	1926	2008
Array Access	2	10	2095	2197
KeY Example	2	9	1039	1363
Single Flow	2	8	510	625
Branching	2	9	576	691
Nested	2	8	513	628
Mixture	4	17	648	823
MixtureLoops	5	24	4847	7475

7.4 Evaluation

In this section we present the results of running the Combined Approach on a collection of examples, shown in Table 7.5. We considered eleven examples, in the range of 5 to 30 lines of code, which cover different program structures and reasons for false positives. The examples are available in Beckert et al. [2019b]. Each of these examples is not solvable by SDG-based approaches like JOANA.

The eleven examples are divided into two groups. First, there are individual methods that cause false positives. The example `Identity` is the program shown in Listing 7.1. In the method `Precondition` there is an if-condition that can never be true, and the method `Excluding` contains two if-statements with conditions that can not both be true in the same program execution. The example `Loop Override` contains a bounded loop which overrides the low value in the last loop execution. For this example, in the unbounded case, the noninterference loop invariant is not enough for an automated proof, and further functional information has to be given by the user. The example `Array Access` contains array handling code, and `KeY Example` contains an if-statement that computes the same value on both of its branches. The second group consists of programs that include these problems in different program structures. Based on the possible SDG, we regard simple calls, branching, nested summary edges and a combination of all.

The columns *KeY Calls* and *Time* in Table 7.5 respectively show how many times KeY was called and how long these calls lasted in total (wall time). The experiments were conducted on a machine with a Core i7-4600MQ CPU and 16GB RAM. The next two columns respectively show how many JavaDL calculus rule applications were performed by KeY when proving the

noninterference property for the examples as part of the Combined Approach (column *CA RuleApps*) and when proving the noninterference property on its own (column *KeY RuleApps*). Note that even for these small examples, which can be proved with KeY alone, the Combined Approach requires less rule applications than KeY on its own.

As explained in Section 2.3, KeY actually generates two proof obligations for a Java method with a noninterference property specified in JML. Besides the noninterference proof obligation, some functional properties are discharged in a functional proof obligation. For this reason, KeY may be called twice when required to prove that a summary edge represents an over-approximation. If KeY fails to prove the noninterference proof obligation for a summary edge, then the Combined Approach does not call KeY to prove the functional proof of that edge, as a successful proof would be useless. All the evaluated examples could be automatically proved with the Combined Approach. Note, however, that the loops contained in the examples were bounded. Thus, loop unwinding could be used, and loop invariants were not needed. When using unbounded loops, the user needs to enhance the loop invariants in order for the proof to succeed.

Although the examples could be proved in a matter of seconds the time needed by KeY can be improved by not starting it anew for each proof obligation. Instead KeY should run continuously and receive the proof obligations, because the initialization of KeY takes a significant amount of time during each call.

7.5 Discussion

Supported properties. In the context of the Combined Approach, the SDG-based approach is the one that delivers the noninterference guarantee, while the logic-based approach improves the precision of the SDG-based approach. Therefore, the noninterference guarantees that can be offered by the Combined Approach can only be a subset of the guarantees offered by the SDG-based approach. Furthermore, in order for the Combined Approach to be sound, the noninterference property that is considered by the logic-based approach when checking whether a summary edge is unnecessary must be at least as strict as the noninterference property checked by the SDG-based approach. Since we limit ourselves in this thesis to sequential and deterministic programs, the noninterference property checked by both SDG-based and logic-based approaches is the classic noninterference property as presented in Definition 2.2. The SDG-based allows the user to designate one or more SDG-nodes as *declassification nodes* in which high information is allowed to influence the low variables. For this kind of declassification (also called *where declassification*) the Combined Approach can still be used, as the property checked by the logic-based approach is stricter.

Analysis order. Proofs with the theorem prover are often reached fully automatically, but may sometimes need auxiliary specification and user interaction. Therefore, we want to minimize the theorem prover usage as much as possible. The order in which the summary edges of the violation chops are checked has a major impact on the performance of the Combined Approach. Ideally, we want to avoid proof attempts of methods for which the SDG-based approach is precise enough (i.e., the program dependencies are not over-approximated) or of very large methods that would overwhelm the theorem prover. In order to achieve these goals, we developed several heuristics for establishing the order in which we check the summary edges with the logic-based approach. A first category of heuristics searches the code for patterns that are likely to cause false positives by the SDG-based approach. Such patterns include code that contains array handling, arithmetic operations, or code that can throw runtime exceptions. SDG-based approaches are particularly prone to report false positives for such code, because they neither distinguish between the different array fields, nor do they take the values of variables and semantics of operators into account. The second category of heuristics attempts to identify the methods that are likely to run through the theorem prover automatically. Earlier, we mentioned that it is difficult to create precise loop invariants. Thus, methods without loops are assigned a higher priority. Additionally, depending on the tools used, we can exclude methods that contain programming language features that are not supported by the logic-based approach, or library methods from the analysis. A third category of heuristics tries to identify the methods that, if proved noninterferent, would bring the greatest benefit to the goal of proving the entire program noninterferent. We assign a high priority to summary edges which are *bridges* in the SDG (i.e., an edge whose removal from the SDG would result in two unconnected graphs). In case no bridge exists within the SDG, we prefer the method with the highest number of connections (i.e., the most often called method).

Modularity. Due to its low scalability, the logic-based approach is more likely to handle methods that are deeper in the call graph (i.e., that call few other methods) than methods which are high in the call graph. However, the parts of the program that can disprove a reported possible noninterference violation may be present in a high level method. In order to still be able to handle such cases, we can automatically generate over-approximated noninterference contracts for the method calls occurring inside the analyzed method based on the results of the SDG-based analysis. The generated noninterference contract for a called method states that the outputs of that method depend at most on the inputs whose corresponding formal-in nodes are reachable from the formal-out nodes corresponding to the outputs in the PDG of the method. Furthermore, if the output that corresponds to

the exception that may be thrown by the method does not depend on any statement in the method we add the `normal_behavior` clause to the contract, which states that the method will not throw any exception.

Due to the soundness of the SDG-based analysis, this noninterference contract is correct-by-construction for the analyzed program. However, the over-approximation done by the SDG-based analysis is also present in the contracts generated this way. Moreover, as in the case of generated loop invariants, the generated method contracts may need additional functional specification (e.g., `assignable` clauses) in order to be strong enough to be used in the proof, and the user must prove the added functional specification. The results of the points-to analysis can be used to generate preconditions (as explained in Definition 7.3). However, the contracts generated with such preconditions are valid only for the call context for which they were generated. If a method is called at multiple locations we can either (1) not generate any preconditions and use one contract for all calls, or (2) we can generate for each call a contract with preconditions. In the second case only the corresponding contract may be used for a call. Thus, using such contracts does not guarantee that the logic-based approach will successfully disprove the reported possible noninterference violation, because the over-approximations done by the SDG-based approach are encoded in the generated contracts. Nevertheless, these contracts allow in some cases for an analysis of higher-level methods without having to inline the called methods.

```
1  /*@ requires true;
2    @ determines \result \by this, low;
3    @*/
4  public int identity(int high , int low ){
5     high = prod(low, high) ;
6     low = low + high;
7     low = low - high;
8     return low ;
9  }
10 /*@ normal_behaviour
11   @ requires true;
12   @ determines \result, \exception \by this, low, high;
13   @*/
14 private int prod(int low, int high){
15    low = low * high ;
16    return low ;
17 }
```

Listing 7.6: Example with generated contracts for called methods

The example program shown in Listing 7.6 was generated with the Combined Approach to show that the return value of the method `identity` does not depend on the parameter `high`. The method `identity` calls the method `prod` for which the Combined Approach has automatically generated a noninterference contract. In the PDG of the method `prod` the formal-out nodes corresponding to the return value and to the exception that may be thrown are reachable from the formal-in nodes that correspond the

parameters `this`, `high`, and `low`. Thus the clause `determines \result, \exception \by this, low, high` is generated.

The formal-out node that corresponds to the exception does not depend, however, on any statement in the method (it is control dependent only on the entry node). Thus, since none of the method's statements may raise an exception the `normal_behavior` clause is added to the contract, which states that the method `prod` does not throw any exceptions. The contract for `prod` is strong enough to prove the specified property of the method `identity` and is also correct-by-construction (as a result of the soundness of the SDG-based approach).

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

The Combined Approach and the approach shown in Chapter 6 are orthogonal. In fact, the program fragment analyzed with the Combined Approach can be further reduced by applying the approach presented in Chapter 6. By simplifying the program (according to Definition 6.3), we can remove statements and execution paths that are not relevant with respect to the possible dependency represented by the analyzed summary edge. Because the program simplified this way is noninterference equivalent to the original program fragment (see Theorem 6.3), the soundness of the Combined Approach is not affected by the simplification.

The program in Listing 7.7 shows an example of a simplified program that was generated by the Combined Approach. For this example JOANA reports a possible noninterference violation, which is in fact a false alarm. The reported noninterference violation contains a summary edge between the parameter `high` and the return value of the method `keYExample`. The reason for this reported violation is that JOANA cannot observe that the value assigned to `low` is the same in both branches of the `if`-statement, and considers the variable `low` to be control dependent on `high`. However, JOANA can determine that in this example there is no data dependency between `low` and `high` because the method `n5` returns the value 15 no matter the values of its arguments. Therefore, by generating the simplified program as explained in Definition 6.3, the lines 8 and 15 can be removed from the program, without any effect on the possible occurrence of the reported noninterference violation.

```
1  /*@ requires true;
2    @ determines \result \by this, low;
3    @*/
4  public int keYExample(int high, int low){
5     if ( high > 0 ) {
6        low = n5(high, high);
7     } else {
8  //    high = -high;
9        low = n5(high + low , high);
10    }
11    return low;
12 }
13
14 public int n5(int x , int high){
15 // high = 2 * x;
16    return 15;
17 }
```

Listing 7.7: Simplified program generated with the Combined Approach

7.6 Conclusion

In this chapter, we introduced a novel Combined Approach to prove noninterference with less user interaction while keeping the same precision. Our approach combines an automated SDG-based analysis with a deductive theorem prover. We demonstrated that the noninterference properties guaranteed by the two approaches are compatible. Thus, our approach is sound. The Combined Approach has been developed tool-independently, but implemented using KeY and JOANA and evaluated on a selection of examples. Although the programs covered in our evaluation do not exceed 100 lines of code and could—as such—also be proved without the help of SDG-based approach, they could, however, also be embedded in much larger programs, which—as such—may be clearly too large for the analysis with a theorem prover. Related work with respect to combining SDG-based approaches with logic-based approaches and other approaches in the area of information flow security can be found in Section 9.1.

Part III

Program Slicing

8 A Framework for Automatic and Precise Program Slicing

8.1 Introduction

Part III of this thesis deals with the question of whether a program is a valid slice (see Section 2.5) of another program. Its main contribution (**C2.1**) consists of a framework for automatic and precise program slicing which is presented in this chapter. The slicing framework combines static (relational verification, see Section 2.7) and dynamic (dynamic slicing, see Section 2.5) program analysis techniques to automatically search for a slice of a given program. The framework is implemented as the tool SemSlice.

This chapter is based on previous work by the author published in Beckert et al. [2017b], Beckert et al. [2019d], and in Beckert et al. [2019c]. Parts of the results of this chapter (including the implementation of the tool SemSlice, see Beckert et al. [2019f]) were obtained in a "research laboratory" (*Praxis der Forschung*) project in which the students Stephan Gocht and Daniel Lentzsch were supervised by the author.

Motivation. All applications of slicing—from optimization to comprehension of programs—can benefit from small and precise slices. Most existing approaches, however, are only syntactical (i.e., they do not take the semantics of the various program instructions into account). On the other hand, many of the existing approaches that do take the semantics into account are not fully automatic. They require auxiliary specifications from the user, for example precomputed or user-provided functional loop invariants are used in Barros et al. [2012] and Jaffar et al. [2012a].

The research of relational verification approaches has progressed significantly in the last couple of years, and approaches that take the program semantics into consideration and can automatically reason about loops have now become available (see e.g., Kiefer et al. [2018], De Angelis et al. [2016],

and Verdoolaege et al. [2012]). These approaches can show the equivalence of two programs in an efficient and automatic fashion—provided that the two analyzed programs have a similar structure. Since slices are constructed by removing program instructions, they have a similar structure to the original program. Thus, they constitute an ideal use case for relational verification.

Contributions (C2.1, C2.2, and C2.3) In this chapter we make the following contributions:

1. We present an easily extensible framework for precise and automatic program slicing (**contribution C2.1**) as well as the semantics of the programs and slice properties that is supports. The slicing approaches that use this framework need no (auxiliary) specification other than the slicing criterion.

2. We adapt relational verification to check whether a slice candidate obtained by removing instructions from a program is a valid slice (**contribution C2.1**).

3. We adapt a dynamic slicing algorithm and use it to generate slice candidates as part of the framework (**contribution C2.3**).

Structure of the chapter. In Section 8.2 we formally describe the programs which we handle and define for them what a valid slice is. We explain how relational verification can be adapted to check slice candidates in Section 8.3. The framework for slicing and three slicing strategies based on it are described in Section 8.4. Section 8.5 consists of a discussion of the evaluation results and other implementation aspects the slicing framework. We then conclude in Section 8.6.

8.2 Slicing Semantics

Static backward slicing, introduced by Weiser [1981], reduces a program by removing instructions from it such that a specified subset of its behavior is preserved. The *slicing criterion*—the specification of the behavioral aspects that must be retained—comprises a set of program variables and a location within the program. Instructions which have no effect (a) on the value of the specified program variables at the specified point and (b) on how often the point is reached may be removed.

High level programming languages are feature rich and require an increased effort to perform a program analysis. A solution for dealing with the language complexity is to perform the analysis on a simpler, intermediate representation (IR). In this section we formalize the notions of *slice candidate*, *slicing criterion*, and *valid slice* using a model of computation based on a

$$\frac{P[pc] = skip}{(s, pc) \rightsquigarrow (s, pc + 1)}$$

$$\frac{pc > len(P)}{(s, pc) \rightsquigarrow (end, pc)}$$

$$\frac{P[pc] = jnz\ v\ target \qquad s(v) = 0}{(s, pc) \rightsquigarrow (s, pc + 1)}$$

$$\frac{P[pc] = jnz\ v\ target \qquad s(v) \neq 0}{(s, pc) \rightsquigarrow (s, target)}$$

$$\frac{P[pc] = halt}{(s, pc) \rightsquigarrow (end, pc)}$$

$$\frac{}{(end, pc) \rightsquigarrow (end, pc)}$$

$$\frac{P[pc] = assign\ v\ exp \qquad x = s(exp)}{(s, pc) \rightsquigarrow (s[v \backslash x], pc + 1)}$$

Figure 8.1: The semantics of our IR for a fixed program P

register machine with an unbounded number of registers. We do not have high-level constructs such as `if` or `while` statements but instead branching and looping are done using conditional jump instructions. The advantage of using such a simple language is the fact that the control flow is reduced to jumps. Furthermore, in the context of slicing, a program will always remain executable after we remove instructions from it. Slicing on an intermediate representation can be used to optimize the code. However, if the goal is to debug a program, then the results of slicing must be transferred to the high-level programming language. We discuss this in Section 8.5.

The implementation of our slicing approach (see Beckert et al. [2019f]) works on LLVM IR programs. To keep the definitions simple, we define a similar low level language containing four instructions: *skip*, *halt* , *assign*, and *jnz*. To obtain precise slices, we restrict expressions on the right hand side of an assignment to only one operator.

Now we define the semantics of our IR language. Let Var be the set of program variables, S the set of states, where a state is a function $s : Var \to \mathbb{N}$, and $pc \in \mathbb{N}$ the program counter. An instruction I is an atomic operation that can be executed by the machine. Let $\mathcal{I}$ be the set of all four instructions provided by our IR language. When an instruction is executed, the system

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  assign x h
8  jnz 1 10
9  assign x 42
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

(a)

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  skip
8  jnz 1 10
9  assign x 42
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

(b)

```
0  assign i 0
1  assign x 0
2  assign c1 (i >= N)
3  jnz c1 12
4  assign t1 (N - 1)
5  assign c2 (i >= t1)
6  jnz c2 9
7  skip
8  jnz 1 10
9  skip
10 assign i (i + 1)
11 jnz 1 2
12 halt
```

(c)

Figure 8.2: Examples from Figure 2.3 in our IR.

changes its state and program counter as determined by the transition function $\rho : S \times \mathbb{N} \times \mathcal{I} \to S \times \mathbb{N}$. A program P is a finite sequence of instructions: $\langle I_0, I_1, \ldots I_n \rangle$. We denote a location i of program P as $P[i]$ with $P[i] = I_i$ for any $i \in \{0, 1, \ldots n\}$ with $0 \leq i \leq len(P) - 1$, where $len(P)$ is the length of the program.

The semantics of the four instructions in our language is shown in Figure 8.1. The instruction *skip* increments the program counter and has no other effects. For slicing we replace instructions at some locations in the original program with *skip*. To model the termination of programs we introduce a special state—*end*—such that once the system reaches this state, it will remain in this state forever. The instruction *halt* is used to bring the system in the *end* state. The assignment instruction—*assign*—takes a variable v and an integer expression *exp* as arguments. After the execution of this instruction, the value of the variable v in the new state is updated with the result x of the expression *exp*, and the program counter is incremented. The conditional jump instruction—*jnz*—allows the register machine to support branching and looping. The instruction gets a variable

v and an integer expression *target*. If the variable v evaluates to zero in the state in which *jnz* is executed, the program counter is incremented, otherwise the program counter is set to the value of *target*. Figure 8.2 shows the examples from Figure 2.3 written in our IR. We can now define program traces.

Definition 8.1 (Program trace)**.** A trace T of a program P is a possibly infinite sequence of state and program counter pairs $\langle (s_0, pc_0), (s_1, pc_1), \ldots \rangle$ such that:

1. $pc_0 = 0$
2. For each trace index i but the last, $(s_i, pc_i) \rightsquigarrow (s_{i+1}, pc_{i+1})$

We use $T^s[i]$ and $T^{pc}[i]$ to denote the ith state and the ith program counter of a trace respectively. Also we use $len(T) \in \mathbb{N} \cup \{\omega\}$ to denote the length of trace $\mathcal{T}$—note that it can be infinite. If the trace is finite, then the last program counter of the trace points to the *halt* instruction. We define F^l_T as a sequence of states, obtained by filtering the trace T with respect to the program location l and projecting the resulting elements on the state. The trace F^l_T contains the states of the elements from those indexes i of T (in the same order as they appear in T) in which the instruction at location l was executed (i.e., $T^{pc}[i] = l$). We now provide formal definitions of a slicing criterion, slice candidate and valid slice.

Definition 8.2 (Slicing Criterion)**.** A slicing criterion C for a program P is a pair (i_C, Var_C) where i_C is a location in P and $Var_C \subseteq Var$.

Definition 8.3 (Slice Candidate)**.** A slice candidate for a program P_o is a program P_L that is constructed by replacing the instructions at some locations in P_o with the *skip* instruction. That is, given a set L of locations of program P_o, for every location i of P_o:

$$P_L[i] = \begin{cases} skip, & i \in L \\ P_o[i], & i \notin L \end{cases}$$

Definition 8.4 (Valid Slice)**.** Given a slicing criterion (i_C, Var_C), a slice candidate P_s for a program P_o is a valid slice if for any two traces T_s and T_o of the respective two programs with $T_s[0] = T_o[0]$ the following two requirements hold:

1. $len(F^{i_C}_{T_o}) = len(F^{i_C}_{T_s})$
2. $F^{i_C}_{T_o}[i](v) = F^{i_C}_{T_s}[i](v)$, for every $v \in Var_C$ and
   for every i with $0 \leq i < len(F^{i_C}_{T_o})$

The first requirement ensures that the criterion instruction is reached in both original program and slice candidate the same number of times. The second requirement ensures that the criterion variables have the same values

every time the criterion instruction is reached in the original program and slice candidate.

Weiser [1981] deals with the feature-richness of programming languages by working on flow graphs, and slices are constructed by removing nodes from the flow-graph. In his approach, however, only nodes with a single successor can be removed while we can remove conditional jumps. Definition 8.4 is similar to the concept of observation windows in Weiser [1981]; however, we do not require the original program to terminate. Thus, we extend the definition of Weiser [1981] to nonterminating programs, as opposed to many other slicing approaches (as stated in Ranganath et al. [2007]) that are not termination sensitive. Compared to other extensions of the definition of Weiser [1981] (e.g., the one in Barraclough et al. [2010]), Definition 8.4 allows for slices which are not *quotients* of the original program (i.e., it allows the removal of conditional jumps while preserving the instructions which are in the program locations between the conditional jump and the jump target). The program

```
0 assign x 42
1 halt
```

is thus a valid slice of the program shown in Figure 2.3, according to Definition 8.4. Not requiring the slice to be a quotient, allows the removal of additional instructions. However, the structure of a slice may differ significantly from that of the original program. When using slicing with the goal of program optimization this is a clear advantage. If the goal is program comprehension, however, then the slice not being a quotient of the original program presents both advantages and disadvantages. One the one hand, a significantly different slice structure compared to the original program may cause the user to have difficulties to understand the behavior of the original program. On the other hand, the fact that some conditional jump statements are not in the slice may indicate to the user that certain program branches are irrelevant with respect to the given slicing criterion, thus helping him better understand the program behavior.

8.3 Verification of Slice Candidates

In this section we show how the relational verifier LLRêve (see Section 2.7) is extended such that it can check whether a slice candidate is a valid slice according to Definition 8.4. LLRêve generates a set M of Horn-constrains, which we explain in the following along with the properties that result from the satisfiability of M. As explained in Section 2.7, LLRêve automatically assigns a synchronization point to each basic block in the CFG. Figure 8.3 shows the CFG (see Definition 2.12) for the program in Figure 8.2a, and each basic block is labeled with the number of a synchronization point.

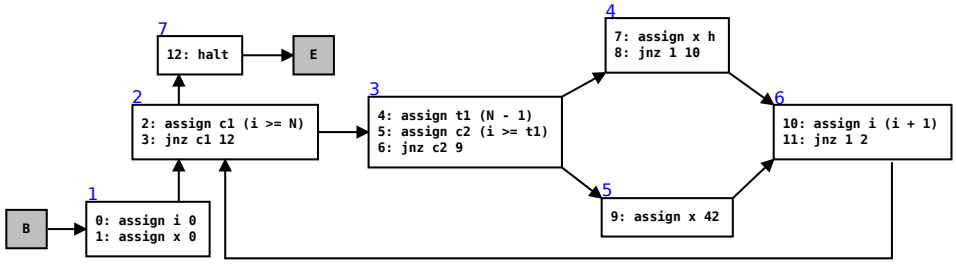

Figure 8.3: The CFG for the program in Figure 8.2a

Finding matching synchronization points for the two analyzed programs is difficult in the general case. For the relational verification of slice candidates, however, we first consider the verification of the program with itself. We construct the CFG and automatically label its basic blocks with unique numbers, as shown in Figure 8.3. The CFG of the slice candidate is obtained by replacing instructions in the CFG of the original program with *skip*. For the example in Figure 8.2a we replace the instruction in block 5 with *skip*. If the replaced instruction is a conditional jump, the CFG edge to the target of that jump is removed, and we do not merge any basic blocks of the CFG. Thus, we can always automatically find synchronization points for the relational verification of slice candidates.

Because there is at least one synchronization point per basic block, the CFG can be viewed as a set of linear paths $\langle n, \pi, m \rangle$, where n and m denote the starting and end synchronization points of the path, and $\pi(s, s')$ is the two state transition predicate between the two synchronization points, with s and s' being the states before and, respectively, after the transition. Because LLRêve transforms the programs in SSA form before the analysis, the transition predicate is the concatenation of all assignments on the linear path.

For two programs with a similar structure, it is expected that there exist coupling predicates that describe the relation between the program states at two corresponding synchronization points. For two programs P and Q we introduce an uninterpreted coupling predicate $C_n(s_p, s_q)$ for each synchronization point n. The relational precondition *Pre* and postcondition *Post* are the coupling predicates for the special synchronization points B and E, respectively. The set M consists of Horn-constraints over these coupling predicates. For two linear paths between synchronization points n and m in programs P and Q characterized by the two transition predicates π and ρ, respectively, the following constraint is added to M:

$$C_n(s_p, s_q) \wedge \pi(s_p, s'_p) \wedge \rho(s_q, s'_q) \rightarrow C_m(s'_p, s'_q) \tag{8.1}$$

To ensure that there is no divergence from lockstep, for every two paths $\langle n, \pi, m \rangle$ and $\langle n, \rho, k \rangle$ in programs P and Q, respectively, with $m \neq k$, $m \neq n$,

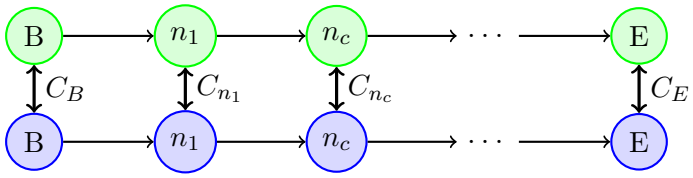

Figure 8.4: Verification of slice candidates

and $n \neq k$ the following constraint is added to M:

$$C_n(s_p, s_q) \wedge \pi(s_p, s_p') \wedge \rho(s_q, s_q') \rightarrow \mathit{false} \tag{8.2}$$

Note that even though the synchronization points m and k do not appear in Constraint 8.2, they respectively determine the transition predicates π and ρ. Now we show that the satisfiability of the constraints generated according to Constraints 8.1 and 8.2 implies that the two analyzed programs fulfill the relational specification.

Theorem 8.1. *Let P and Q be programs specified with the relational precondition Pre and postcondition Post, for which matching synchronization points have been found. Let M be the set of constraints generated according to 8.1 and 8.2. If M is satisfiable, then for every pair of prestates satisfying Pre:*

1. *The synchronization points are reached in the same order in P and Q,*
2. *If P terminates, then so does Q and Post holds for the two poststates.*

Proof. For the distinct synchronization points n, m, k, the fact that Constraint 8.2 has a model implies that *(case 1)* π or ρ is *false*, meaning that the execution of P or Q cannot reach respectively m or k from n, or *(case 2)* C_n is false meaning that n is not reachable in P or Q, or per (chaining of) Constraint 8.1 the prestates do not satisfy the precondition. Thus, P and Q reach the synchronization points (including E, thus implying mutual termination) in the same order. For two synchronization points n, m, the fact that Constraint 8.1 has a model implies that *(case 1)* m cannot be reached from n in P or Q, or *(case 2)* C_n is false and n is not reachable, or the prestates do not satisfy the precondition, or *(case 3)* starting in n with C_n holding, both programs reach m and C_m holds there. The constraints generated according to 8.1 are thus interpolants that show the validity of the relational specification. ◁ ◁

Thus, for the case in which the location of the criterion location is at the end of the program (in the example from Figure 8.2 that corresponds to location 12) relational verification can be used to check whether a slice candidate is a valid slice. To that end, the precondition is set to require equal prestates, while the postcondition requires all criterion variables two have equal values in the two poststates.

To check the validity of a slice candidate for the cases in which the criterion location is in the middle of the program, we adapt the constraints generated by the relational verifier. The relational precondition C_B remains unchanged while the relational postcondition C_E is no longer used (i.e., it will be set to *true*). As shown in Figure 8.4, a synchronization point n_C is added to the program and slice candidate at the location of the criterion instruction. For the example in Figure 8.2a the coupling predicate for the slicing criterion corresponds to basic block 5 in Figure 8.3. If the criterion location is part of a basic block with more than one instruction, we split that basic block up. For a program P with a slice candidate Q and a given slicing criterion $\langle i_C, V_C \rangle$ with a corresponding synchronization point n_C we add the following constraint:

$$C_{n_C}(s_P, s_Q) \rightarrow \forall x \in V_C \;\; s_P(x) = s_Q(x) \tag{8.3}$$

Theorem 8.2. *Let P be a program and Q a slice candidate specified with the relational precondition Pre requires equal prestates and postcondition Post is true. Let M be the set of constraints generated according to 8.1, 8.2, and 8.3. If M is satisfiable, then for every pair of prestates that fulfill Pre:*

1. *The criterion location is reached equally often in P and Q,*
2. *At the i-th time (for $i \geq 1$) the criterion instruction is reached, the criterion variables are equal in P and Q,*
3. *If P terminates, then so does Q.*

Proof. From Theorem 8.1 it results that P and Q run in lockstep with respect to the synchronization points. The instruction at the criterion location has its own synchronization point. As a consequence of this, the criterion instruction is executed in both P and Q the same number of times, and the candidate terminates iff the original program terminates. Due to Constraint 8.3, the coupling predicate corresponding to the criterion locations ensures that each time the criterion location is reached, the criterion variables have the same values. ◁ ◁

Thus, for a program P with a slice candidate Q and a slicing criterion (i_C, V_C), if the set M containing the Constraints 8.1, 8.2 and 8.3 for every synchronization point is satisfiable, then Q is a valid slice according to Definition 8.4. Moreover, if the set M is unsatisfiable, then the SMT solver returns an unsatisfiability proof that contains a counterexample with two concrete inputs for which the slice property is violated—provided the SMT solver does not time out.

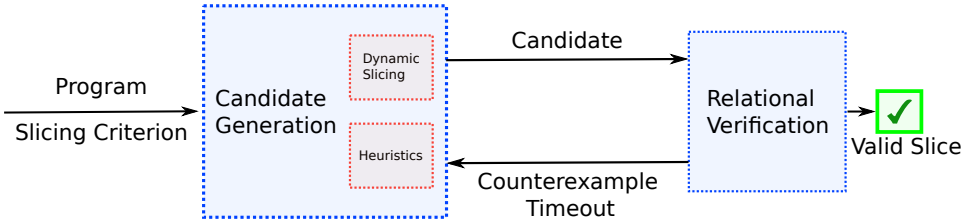

Figure 8.5: The slicing framework

8.4 A Framework for Automatic Slicing

Being able to use relational verification to check whether a slice candidate is valid or not, we can now construct a framework for automatic program slicing. The framework, illustrated in Figure 8.5, consists of two components which interact with each other. The first component—the candidate generation engine—generates the slice candidates and sends them to the second component—the relational verifier (represented in our case by LLRêve). The relational verifier receives the slice candidate and transmits one of three possible answers to the candidate generation engine: (1) the candidate is a valid slice, (2) the candidate is not valid along with an input that leads to a violation of the slice property (see Definition 8.4), or (3) a timeout. The candidate generation engine receives the answer and uses it to adapt its candidate generation strategy.

An advantage of the framework is that the candidate generation engine does not need to care about the correctness of the slice candidates it generates—that is taken care of by the relational verifier. Besides the candidate generation strategies that we present in this chapter, the framework can easily be extended with others. Thus, it provides a platform for slicing based on relational verification for the software slicing community.

We distinguish between two types of candidate generation strategies. On the one hand, there are strategies that generate candidates by replacing program instructions with *skip* according to some heuristics and do not use any information from the relational verifier other than the existence of a counterexample. Examples for such properties are described in Section 8.4.1. On the other hand, there are strategies that also take the values from the counterexample into consideration when generating the next slice candidates. We present one such strategy—*counterexample guided slicing*—in Section 8.4.2.

8.4.1 Removing Instructions Using Heuristics

The strategy *brute forcing* (BF) generates all possible slice candidates. As their number is exponential with respect to the number of instructions in the original program, it is clear that this strategy does not scale for large programs. Nevertheless, this strategy has the benefit of generating the smallest possible slice with our framework. Brute forcing can be used as part of a divide and conquer strategy to slice program parts which are small enough. As an improvement, this strategy can start with the generation of the candidate in which all instructions were replaced with *skip* and progressively add back instructions from the original program. Once a candidate is shown to be a valid slice, candidates that contain the instructions (other than *skip*) of this slice need no longer be checked.

The strategy *single statement elimination* (SSE) successively replaces a single instruction from the original program with *skip*, and checks whether the program thus obtained is a valid slice. If this is the case, the strategy attempts to remove all other instructions as well. The strategy requires quadratically many calls to the relational verifier in the worst case, in which in each iteration the last candidate is shown to be a valid slice. Although this approach scales better than brute forcing, it can only find slices in which program instructions can be removed individually. Groups of instructions such as

```
0 assign x (x + 50)
1 assign x (x - 50)
```

where the removal of a single instruction results in an invalid slice candidate, but removing the entire group would result in a valid slice cannot be removed. The SSE strategy can be generalized to support the removal of groups of up to a given number of instructions.

8.4.2 Counterexample Guided Slicing

The *counterexample guided slicing* (CGS) strategy uses *dynamic slicing* to generate slice candidates. Dynamic slicing was first introduced in Korel and Laski [1988], and a survey on dynamic slicing approaches can be found in Korel and Rilling [1998]. For the CGS strategy we adapted the dynamic slicing algorithm from Agrawal and Horgan [1990], which is a syntactic approach based on on the PDG (see Section 2.6.1). On the most basic level, the algorithm in Agrawal and Horgan [1990] receives the PDG and an execution trace of the original program as inputs, and it works by computing the subgraph of the PDG which contains only the nodes corresponding to those instructions which have been executed in the program trace. The dynamic slice is then computed using this subgraph (see Definition 2.21). Further optimizations are possible, as the resulting slice has to be valid only for a single input.

A PDG node can depend on multiple other nodes, but some of these dependencies are determined by the execution path of the program (e.g., a variable can be assigned on more than one branch, resulting in multiple dependencies for instructions that use that variable). Unlike static slicing, for dynamic slicing only one execution path is relevant—the one corresponding to the input for which the dynamic slice is computed. Thus, PDG edges representing dependencies that are relevant only for other inputs can be removed. A similar situation arises with loops: at different loop iterations, a node inside the loop body may have different dependencies. When performing dynamic slicing, the number of iterations done by a loop is known (assuming the program terminates for the input). The PDG can thus be extended with nodes representing the body instructions at different iterations, which also leads to an increased precision of the dynamic slice.

In Agrawal and Horgan [1990], the extended PDG is called a *dynamic dependence graph* (DDG). Based on the observation that the nodes inside the loop body can depend on only a finite number of other nodes, a new node is added to the PDG just for those iterations in which the corresponding instruction has different dependencies than in all previous iterations. These optimizations give rise to the *reduced dynamic dependence graph* (RDDG). Thus, by ignoring dependencies caused by other inputs than the one for which the dynamic slice is computed, more instructions can be removed than in the case of static slicing. To ensure compatibility with the slicing property from Definition 8.4, we adapt this algorithm to support criterion locations other than the end of the program. For this, when computing the dynamic slice with the RDDG we do not mark the `return` statement, as is done in Agrawal and Horgan [1990], but rather all nodes that correspond to the criterion location. If the criterion location is inside a loop, then multiple nodes are marked. The adapted RDDG dynamic slicing algorithm is purely syntactical and scales much better than a semantic approach. Thus, if we use it as part of the candidate generation strategy, the relational verification of slice candidates remains the bottleneck of our framework.

For the CGS strategy we wish to merge several dynamic slices $P_{d_1}, \ldots, P_{d_n}$ for the respective input states $s_1, \ldots, s_n$ into a single dynamic slice P_u that is a valid for all inputs $s_1, \ldots, s_n$. In general, the union slice of dynamic slices (which contains all program instructions that are in at least one dynamic slice) is not a correct dynamic slice for all respective inputs of the given dynamic slices. A solution to this was presented in Hall [1995] in the form of an iterative algorithm called *simultaneous dynamic slicing (SDS)*, which computes a single dynamic slice valid for each input in a given set.

We can now present the CGS strategy, shown in Algorithm 2. It starts with an initialization of the slice candidate P_s with a program Φ, in which all instructions have been replaced with *skip*, of an arbitrary initial state s (e.g., one in which all variables are set to 0), and of the variable b which is set to true when a valid slice is found. The strategy uses the initial state s

Data: Program P, Slicing criterion (i_C, V_C)
Result: Program Slice P_s
$P_s \leftarrow \Phi$; $s \leftarrow \bar{0}$; $b \leftarrow \mathit{false}$;
repeat
  $P_d \leftarrow \mathit{dynamicSlice}(P, s, (i_C, V_C))$;
  $P_s \leftarrow \mathit{SDS}(P_s, P_d)$;
  $(b, s) \leftarrow \mathit{relationalVerification}(P, P_s, (i_C, V_C))$;
until $b \vee \mathit{timeout}$;
if *timeout* **then**
  $P_s \leftarrow P$;
end
return P_s;

Algorithm 2: The CGS Strategy

with the criterion (i_C, V_C) to compute a dynamic slice P_d. The instructions from P_d are then added—using the SDS procedure—to the slice candidate P_s which is checked for validity by the relational verifier. If P_s is a valid slice, the variable b is set to *true* and the strategy returns P_s. Otherwise, the relational verifier delivers a counterexample, which is used as the initial state s in the next iteration. Both the dynamic slicer and relational verifier may timeout, in which case the strategy returns the original program P.

Theorem 8.3. *Let P be a program and P_d be a dynamic slice for all initial states $s \in S_d$, and s_{ce} be the counterexample obtained when checking whether P_d is a valid slice of P. Then the following holds:*

1. *$s_{ce} \notin S_d$.*
2. *The dynamic slice P_{ce} for the initial state s_{ce} contains at least one instruction which is not in P_d.*

Proof. Both properties follow from the correctness of the relational verifier and of the dynamic slicer and of the SDS algorithm. (1) If $s_{ce} \in S_d$ then the relational verifier delivered a spurious counterexample, the dynamic slicer delivered an invalid dynamic slice, or the SDS algorithm computed a wrong simultaneous dynamic slice. (2) If P_{ce} contains no additional instruction compared to P_d, then $\mathit{SDS}(P_d, P_{ce})$ which means that P_d is a dynamic slice for s_{ce}. This implies that the relational verifier delivered a spurious counterexample. ◁ ◁

Theorem 8.3 guarantees that the CGS strategy adds at least one instruction back after each iteration. Thus, the number of calls of the relational verifier is linear in the number of program instructions. The SDS algorithm is needed for this theorem to hold. The validity of the slice computed with CGS, however, is guaranteed by the relational verifier. Thus, if the CGS algorithm computes the simple union of dynamic slices, the relational verifier

Table 8.6: Evaluation

Original			BF			SSE			CGS		
Example	Source	#stmts	time (s)	#stmts	#calls	time (s)	#stmts	#calls	time (s)	#stmts	#calls
count_occurrence_error	self	50				20	42	11			
count_occurrence_result	self	50				23	44	13			
dead_code_after_ssa	Ward [2009]	4	<1	2	4	<1	2	4	<1	2	1
dead_code_unused_variable	self	3	<1	2	2	<1	2	3	<1	2	1
identity_not_modifying	Field et al. [1995]	8	<1	3	3	<1	7	5	<1	6	1
identity_plus_minus_50	Barros et al. [2012]	5	<1	2	4	<1	5	4	<1	5	1
iflow_cyclic	Ward [2009]	18	108	14	2197	<1	16	6	<1	17	1
iflow_dynfamic_override	self	15	39	8	1298	<1	11	8	<1	12	1
iflow_endofloop (8.2)	self	19	223	15	4065	<1	16	7	<1	18	2
intermediate	self	13	7	11	129	<1	12	5	<1	12	2
requires_path_sensitivity	Jaffar et al. [2012a]	20	1131	16	26894	<1	17	10	<1	18	3
single_pass_removal	self	13	<1	3	7	<1	6	11	<1	8	1
unchanged_over_itteration	self	20	45	9	932	1	15	14	<1	20	2
unreachable_code_nested	self	10	<1	2	1	<1	9	1	<1	4	1
whole_loop_removable	self	20	24	8	469	<1	17	5	<1	17	2

may return a counterexample that it already provided in a previous CGS iteration. In this case the CGS algorithm needs to terminate and return the original program. Given the fact that computing the union of dynamic slices is much easier than computing the simultaneous dynamic slice, the user of the framework must make a choice between performance and completeness. Our implementation of CGS computes the union of dynamic slices.

Although out of all strategies presented in this chapter, the CGS strategy has the least number of calls to the relational verifier, it comes with some disadvantages. First, the program needs to be executed at each iteration. Depending on the analyzed program, this can cause performance issues, and for some inputs the program may not even terminate. Second, the CGS strategy is vulnerable to timeouts of the relational verifier, which for certain analyzed programs are inevitable, as the program equivalence problem is undecidable. If a timeout occurs, then the strategy fails entirely and must return the original program as the slice candidate, while the BF and SSE strategies can continue their search for a valid slice candidate. Third, the precision of CGS depends on the precision of the dynamic slicing approach used in the candidate generation. Even though the used dynamic slicing approach can remove more instructions than static syntactic slicing approaches, the dynamic slices it computes are still over-approximations.

8.5 Discussion

We start the discussion by presenting the evaluation results (shown in Table 8.6; the evaluation data is available in Beckert et al. [2019e]) of the implementation of the framework, consisting of the tool SemSlice (which is available in Beckert et al. [2019f]). For the evaluation, a collection of small but intricate examples (e.g., the example of Figure 8.2 or a routine in which the same value is first added and then subtracted) that each focus on a particular challenge for semantics-aware slicing was used. Some examples are taken from slicing literature (from Barros et al. [2012]; Canfora et al. [1998];

Field et al. [1995]; Jaffar et al. [2012a]; Ward [2009]). The second column indicates the source of each example, the third the number of LLVM-IR instructions in the program. For each candidate generation method from Section 8.4, the table lists the number of instructions in the smallest slice found by SemSlice, the (wall) time needed by the tool, and the number of calls to the SMT solver. The experiments were conducted on a virtual machine with an Intel Core I7-4600MQ CPU (two cores) and 2GB RAM. The exponential BF strategy works satisfactorily fast on functions with up to 20 instructions. Although it requires more time than the other strategies, it computes more precise slices. For examples with less than 10 instructions the BF strategy takes less than one second. The other two strategies (SSE and CGS) achieved slices of similar precision (to each other) and required less than one second for most examples. The evaluation shows that the framework can handle programs that require a large number of calls to the relational verifier (e.g., the program *requires_paths_sensitivity* with the BF strategy called the relational verifier almost 27000 times and needed about 18 minutes to find the slice). The BF strategy serves as a *worst-case* scenario when using the slicing framework to automatically slice programs, and other strategies need less calls. For this example the other strategies were still able to remove some instructions with much less calls to the relational verifier, and they can scale to larger programs. Thus, the scalability of our slicing approach can be increased by using candidate generation strategies that do not call relational verifier often. Another way to ensure that our approach to slicing scales to large programs is to apply it to individual program functions (as opposed to applying it to the entire program). Our current prototypical implementation supports only a subset of the LLVM IR instruction set, which is the main reason we did not evaluate it on real-life programs.

Our slicing approach works on an intermediate representation language. This is beneficial for the implementation of the approach, as it does not need to handle all features of a modern high level programming language. However, one of the uses for program slicing is to help the user debug and comprehend a program written in a high level language. It is possible to perform relational verification of such programs—the early version of LLRêve was in fact working on a simple *while* language (see Felsing et al. [2014]). LLVM-IR was later chosen (see Kiefer et al. [2018]) to increase the practicability of LLRêve. The current framework can be adapted for slicing high level languages by either (1) attempting to translate back the IR slice to the high level language, or (2) by defining the slicing candidate in the high level language and then translating both the original program and the slice candidate into the IR, and then using the extended relational verifier. For the first option we expect that (similar to the simplification approach in Chapter 6) only an over-approximation of the IR slice can be obtained by translating it back into the high level language. As for the second solution, the CFGs of the original program and slice candidate in

the IR may be so different that our approach would no longer be able to automatically find matching synchronization points. A solution to this can be to automatically annotate the original program and its slice candidate in the high level language, thus marking the synchronization points and using these marks in the IR translation. A further solution for supporting a high-level language would be to extend the work of Felsing et al. [2014] with the ideas of this chapter. For this, Definition 8.4 of a valid slice needs to be adapted for high-level programming languages, and the weakest liberal precondition calculus from Felsing et al. [2014] needs to be extended to support slicing in the case in which the criterion location is in the middle of the program. By working on the high-level programming language we would lose the advantages of an intermediate representation (i.e., relative language independence and existing support for various code optimizations) but our approach to slicing would become more suitable for program debugging and comprehension.

The IR language that we used to present our approach is not inter-procedural. While we could consider all programs as having been inlined beforehand, recursive procedures would not be supported. The relational verifier supports dealing with function calls using mutual function summaries (see Kiefer et al. [2018]). A mutual function summary abstracts two matching function calls using coupling predicates. In general it is difficult to find matching function calls, but for checking the validity of slice candidates this can be done automatically, similarly to finding matching synchronization points. Thus, our approach can be extended to support recursive functions. However, the function calls themselves may not be removed, otherwise the mutual function summaries cannot be used.

In the semantics that we provided in Section 2.5 we assume that an error (e.g., a division by zero) causes the system to transition to the end state. An interesting question in the context of program slicing is whether instructions which may cause errors can be removed from the program. While some approaches (e.g., in Podgurski and Clarke [1990]; Ranganath et al. [2007]) keep the error prone instructions in the slice, others (e.g. Léchenet et al. [2016]) allow the removal of such instructions—at the cost of a weaker soundness property (i.e., what constitutes a valid slice). This is nonetheless still useful in certain application scenarios such as software verification. With our slicing approach, we keep error prone instructions in the slice. However, because we take the semantics of the program instructions into account, we can remove error prone instructions which can never cause an error (e.g., a division where the divisor can never be zero).

The completeness of our approach (i.e., whether a valid slice according to Definition 8.4 is deemed as such) is limited in practice by two factors. First, the relational verifier is required to automatically infer the coupling predicates needed to verify the validity of a slice candidate. The relational verifier works well when the needed coupling predicates are limited to linear arithmetics (as

shown in Klebanov et al. [2018]). The second factor limiting completeness is the requirement that the original program and the slice candidate must run in lockstep. This is needed to ensure the mutual termination and that the criterion location is executed the same number of times. Thus, whereas we can remove instructions from inside a loop, we are not able to remove the loop itself (in our case the conditional jump instruction), even if it is empty (i.e., it loops over *skip* instructions). A possible solution to this is to check termination through other means and then remove empty loops that are guaranteed to terminate.

8.6 Conclusion

In this chapter we extended a relational verification approach such that it can check whether a slice candidate is indeed a valid slice. Based on this, we proposed a framework for precise and automatic static slicing which consists of a candidate generation engine and the extended relational verifier. We presented three strategies to compute slice candidates, among them CGS is more sophisticated. It uses the counterexample provided by the relational verifier to refine the slice candidate with the help of a dynamic slicer. The performance of relational verification depends on the similarity of the compared programs. As we compare programs and their slice candidates, close similarity is given and relational verification performs well, as shown in our evaluation.

Related work in the area of program slicing is presented in Section 9.2.

Part IV

Related Work and Conclusion

9 Related Work

In this chapter we present work that is related to the approaches used in the framework for checking noninterference (presented in Part II) and the framework for program slicing (presented in Part III). The chapter is based on the related work sections of the publications that are relevant to this thesis (i.e., the ones that are enumerated in Section 1.4). We present related work for the noninterference framework in Section 9.1 and for the program slicing framework in Section 9.2.

9.1 Noninterference

A lot of research has been done in the area of information flow security, dating back to the works of Denning [1976]; Denning and Denning [1977] and later Goguen and Meseguer [1982]. As already mentioned, approaches for checking noninterference range from fully automatic with much over-approximation to more precise interactive approaches.

9.1.1 Noninterference Testing

In the following we present work that is related to the automatic test generation approach for noninterference properties that we presented in Chapter 4.

Kinder [2015] proposed in a position paper the extension of automatic test generation approaches to support k-safety properties (i.e., properties for which at least k program executions are needed to demonstrate a counterexample). Noninterference is given as the main example for a 2-safety property.

Milushev et al. [2012] present a tool that uses symbolic execution in combination with self-composition in order to automate noninterference testing of C programs. Their tool uses dynamic symbolic execution to search for noninterference violations. The approach presented in Chapter 4 provides the following improvements:

- It can declassify quantified first order expressions.
- It supports a noninterference property based on isomorphism, and can be used for heap-based programs.
- It does not have to run until a violation is found, but only for a bounded number of loop iterations and method calls, and provides an achieved test coverage when it finishes.

Do et al. [2016] present a logic-based approach to find noninterference violations. It also generates constraints that are then given to an SMT solver. However, their translation is not bounded. Thus, the constrains given to the SMT solver are undecidable. Violations can be presented to the user as a test case. This approach, however, does not support a low-equivalence relation based on isomorphism, and there is also no option to generate a test suite for the case in which no violation was found, as we do with the second option (see Section 4.3).

Le Guernic [2007] proposes a sound noninterference testing mechanism based on standard testing techniques and on a combination of dynamic and static analyses. The idea is to use existing test generation techniques (that are used for functional properties) to generate test inputs that achieve a high coverage. The generated inputs are then relayed to a dynamic technique that checks at runtime (using software monitors) whether a program violates the noninterference property. It is claimed that if full branch coverage is achieved and no violation was found, then the program must be noninterferent. However, sound dynamic techniques for checking noninterference are limited, as shown in Russo and Sabelfeld [2010], and may report false alarms. This is not the case with our technique—if a test fails, then there is a real violation of the noninterference property.

Gruska [2013] and Hriţcu et al. [2016] also handle the problem of testing noninterference. They, however, do not work on the programming language level as we do, but on the level of process algebra and stack machines, respectively.

Noninterference testing can be considered a special case of metamorphic testing, see Chen et al. [2018]. Metamorphic testing checks whether for a set of test inputs, which are in a given relation (called *metamorphic relation*) with each other, the corresponding program outputs are also in a given relation with each other. For more related work on automatic test generation techniques for functional properties we refer to the surveys of Galler and Aichernig [2014] and Anand et al. [2013].

9.1.2 Noninterference Debugging

In the following we present work that is related to the approach for analyzing noninterference violations that was presented in Chapter 5. We are unaware of any other debugging tools for relational properties. However, the intuitiveness and familiarity of the program debugging operations and user interface has been recognized and applied in the area of formal methods before. Using a debugger interface to find out why a proof attempt has failed is also done in Beckert et al. [2017c], but there the proof attempt itself is debugged and not the program, as done in Chapter 5. In Hentschel et al. [2014] and Hentschel [2016] an approach is presented that also integrates the typical program debugging user interface with a theorem prover, in this case KeY. Using symbolic execution, the user can debug all program execution paths at the same time.

9.1.3 Combinations of Logic- and SDG-based Approaches

In the following we present work that also combines SDG-based and logic-based approaches for proving noninterference and is thus related to the approaches presented in Chapters 6 and 7.

The Hybrid Approach. The work by Küsters et al. [2015] also aims to obtain the best of both worlds—automatic analysis and interactive techniques for proving noninterference—by combining an automatic SDG-based analysis and a theorem prover. This approach (called *Hybrid Approach*) first attempts to prove noninterference using an SDG-based approach like JOANA. If this fails—and the user suspects that this is a false alarm—he must identify the possible cause of the 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. 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 returns the same low output).

The Hybrid Approach approach improves the precision provided by JOANA. However, the communication between tools is done manually by the user; there is no assistance when searching for the causes of the false alarms. This approach does not use the results provided by the SDG analysis tool to discard the program parts that are irrelevant with respect to the analyzed noninterference property and simplify the program that needs to be verified. In fact, by extending it, the program that is verified in the second step becomes even more complex.

Checking the Satisfiability of Path Conditions. Another way of combining SDG-based approaches with logic-based is by checking the satisfiability of the path conditions for the execution paths determined by the reported security violation (as done e.g., in Snelting et al. [2006], Taghdiri et al. [2011], and Hammer [2009]). If a path condition is unsatisfiable, then on the respective execution path the high input cannot influence the low output. However, not all false alarms reported by the SDG-based approach are on infeasible paths. For example, the program in Listing 7.1 has only one path that is feasible and yet the SDG-based approach reports a possible noninterference violation. With the approaches presented in Chapters 6 and 7 such cases can be handled.

A program input that satisfies the path condition generated by those approaches can serve as a *witness* for a noninterference violation. The engineer is then required to analyze a single program execution with the input data generated this way and to decide whether this is indeed noninterference violation. This is not an easy task, especially for indirect dependencies. Using the approach from Chapter 4, we can generate noninterference tests that contain two low-equivalent program inputs that lead to outputs which are not low-equivalent. The user can then use the relational debugger presented in Chapter 5 to analyze the reported noninterference violation.

9.1.4 Other Approaches for Checking Noninterference

In the following we present some approaches for checking noninterference which are not part of the noninterference framework.

A major category of approaches for checking noninterference which have not been covered in this thesis are approaches based on type systems. Those approaches assign a security type (e.g., high or low) to each program variable and expression. The rules of the security type system then enforce noninterference by not allowing high expressions to be assigned to low variables and by not allowing any assignments to low variables in a high context (e.g., in the body of a loop with a high guard). Volpano et al. [1996] have shown a type-based algorithm that guarantees noninterference (according to Definition 2.2). Other approaches based on type systems have been developed (e.g., in Myers and Myers [1999]; Hunt and Sands [2006]; Lortz et al. [2014]). Approaches based on type systems can have the same scalability and precision as SDG-based approaches, as shown by Mantel and Sudbrock [2013].

Dynamic approaches based on *runtime monitoring* (e.g., Guernic [2007]; Shroff et al. [2007]) have also been developed. A monitor checks every program instruction before its execution, and—if it determines that the instruction may reveal some high information—the execution of that instruction is either prevented, or the instruction is replaced with a safe instruction which is then executed. Dynamic approaches that use runtime monitors can prevent

instructions that assign high data to low variables, and they can also keep track of the security context in which the instruction is executed (e.g., whether the instruction is inside the body of a loop with a high guard). Dynamic flow-insensitive approaches (i.e., approaches in which each variable is assigned a security level which must be respected throughout the entire execution) offer the same guarantees as static flow-insensitive approaches based on type systems, as shown in Sabelfeld and Russo [2010]. However, for flow-sensitive analyses (i.e., some variable are allowed to hold both high and low data during the execution), the precision of sound dynamic techniques is limited, and they may report more false alarms than precise static techniques, as shown in Russo and Sabelfeld [2010]. *Secure multi-execution* (see e.g., Devriese and Piessens [2010]) is another dynamic approach for enforcing noninterference. This approach does not use software monitors but instead the program is executed once for each security level (e.g., if there are two security levels, high and low, then the program is executed twice). The program inputs which are of a higher security level than that of the execution (e.g., the high inputs for the low execution) are replaced with default values. The output of each security level are taken form the execution of the respective security level (e.g., the low outputs are taken from the low execution and the high outputs from the high execution).

A survey on approaches for checking noninterference has been done by Sabelfeld and Myers [2003].

9.2 Program Slicing

Static slicing is an active area of research and many approaches have been developed. In the following we present other approaches that—like the approach in Chapter 8—use the semantics of the program instructions when computing the slice.

Assertion based slicing (see Barros et al. [2012]) is a program slicing approach that takes the semantics of the program into consideration. Programs need to be specified with a method contract containing a precondition and postcondition. The method contract represents the slicing criterion (i.e., statements are removed from the program such that the reduced program still fulfills the contract). This approach works by combining precondition and postcondition-based slicing. Postcondition-based slicing works by computing the weakest precondition before every program location. If the weakest precondition at a location i implies the weakest precondition at a location j (with $i < j$), then the instructions between the locations i and j may be removed, as they do not contribute to the truth value of the postcondition. Precondition-based slicing works in a similar fashion by employing a strongest postcondition calculus. Unlike in our approach, loop invariants are required

for every loop, and only groups of instructions that are at consecutive program locations can be removed. This approach extends and improves older, similar approaches (e.g., Comuzzi and Hart [1996]; Chung et al. [2001]), and a tool implementing it also exists (see da Cruz et al. [2010]).

The approach in Liu et al. [2016] also uses a method's contract as the slicing criterion. Using a proof for the validity of the contract for a bounded number of loop iterations and type instances, it finds the parts of the program which were not needed for the proof in the bounded case, and uses this information to construct a slice candidate. However, the program parts that are deemed irrelevant are not removed, but replaced with an abstraction. Thus, the slice candidate over-approximates the behavior of the original program. If the contract is shown to be valid for the slice candidate, then it is also valid for the original program. A counterexample to for the validity of the contract for the slice candidate can be used to generate a more concrete candidate, making this approach similar to the CGS strategy. However, for proving the contract for the slice candidate, loop invariants are needed.

Path sensitive backward slicing (see Jaffar et al. [2012a]) is another slicing approach that takes the program's semantics into consideration. The main idea is to symbolically execute the program and compute the path condition of every execution path. The paths are then checked for satisfiability, and only the satisfiable paths are used for computing the slice. This approach needs to deal with the path explosion problem (i.e., the number of program paths is exponential with respect to the number of program branches). This problem is mitigated by reusing the results of already performed satisfiability checks. The approach handles loops by using abstract interpretation to generate loop invariants, which can lead to an over-approximated description of the loop behavior. Thus, while the approach offers an increased precision when compared to syntactic approaches, it is not able slice the program in Figure 2.3a. An implementation of this approach is available as part of the tool Tracer (see Jaffar et al. [2012b]). The idea of discarding dependencies that can only occur on infeasible program paths has also been explored in other works (e.g., in Snelting et al. [2006]; Canfora et al. [1998]). For these approaches, a compromise between the precision of path conditions and the scalability of the approach had to be found.

Abstract program slicing (see Halder and Cortesi [2013]) is an approach which makes use of the program's semantics. However, a different slicing criterion is used. Instead of preserving those instructions that affect the exact values of the criterion variables at the criterion location, as required by Definition 8.4, this approach preserves the statements that affect a property of the criterion variables. The properties pursued in this approach are whether the variables belong to a given abstract domain (e.g., the positive integers). Based on the same principle like abstract interpretation, for some operations the abstract domain of the output is known—provided the abstract domains

of the inputs are also known. Thus, some dependencies modeled in the PDG can be removed. Because of the different slicing criterion, this approach can generate slices which are not valid according to Definition 8.4.

The Frama-C framework (see Kirchner et al. [2015]) for software analysis provides components that support abstract interpretation and program slicing (based on PDGs). Abstract interpretation can be used to improve the precision of the slicing component by identifying some infeasible branches. Abstract interpretation can automatically handle loops, but it does this by over-approximating their effects.

In Podgurski and Clarke [1990] a different notion of semantic dependency between program statements is defined. In that work it is assumed that each node in the CFG of a program has an assigned function that represents the computation performed by that node. Thus, a statement s is semantically dependent on a statement s' if the logical interpretation of the function computed by s' affects the execution behavior of s. Consider a program that contains the statement `assign x (x + 0)` followed by the criterion location and x as a criterion variable. According to the definition from Podgurski and Clarke [1990] the assignment would be in the slice, because if the interpretation of the + symbol changes (e.g., to multiplication), then so would the value of x at the criterion location. In our approach, on the other hand, we consider the semantics of the program instructions to be fix. Thus, we can remove the statement from the program, as it leaves the value of x unchanged.

The approach in Riesco et al. [2013] uses the formal semantics definitions of a language to automatically generate a slicer for programs written in that language. The approach to slicing works in two steps: first it analyzes the formal semantics definitions and computes for every instruction the parameters on which that instruction has side effects (data or control dependencies). This information is then used in the second step in which for a set of program variables, which constitutes the criterion, an over-approximated set of instructions that have a direct or indirect side effect on them is computed. This set constitutes the slice. The focus of this work lies, however, on the automatic generation of program slicers and not on the precision of the slices generated by them. This approach is implemented in the tool Chisel (see Asăvoae et al. [2018]).

Other, syntactic, slicing approaches have been surveyed in Xu et al. [2005] and Tip [1994]. A survey of dynamic slicing techniques can be found in Korel and Rilling [1998].

10 Conclusion

10.1 Summary

The goal of this thesis is to advance the state-of-the-art in the analysis of relational properties of computer programs. It focuses on two specific relational properties: (1) noninterference and (2) whether a program is a correct slice of another. The goal is achieved by providing frameworks—one for each of the two properties—that combine new and existing approaches such that the advantages from all combined approaches are gained.

Noninterference. The framework for checking noninterference (described in Part II) provides approaches for automatic test generation (see Chapter 4) and analysis of noninterference violations (see Chapter 5) with respect to a given noninterference property. The approach provided for the automatic generation of noninterference tests allows the user to search for violations of the noninterference property, and also provides a coverage criterion for the generated test suites that is appropriate for relational properties. This approach is the first to support automatic test generation for noninterference properties of heap-based programs and which supports quantified first order declassification expressions. The approach for analyzing noninterference violations is realized through a debugger for relational properties—the first of its kind. It employs well known concepts from program debugging, and it extends them for relational program analysis.

For proving noninterference, the framework combines an SDG-based with a logic-based approach. The SDG-based approach can handle larger programs, but reports more false alarms than the logic-based approach. The SDG-based and logic-based approaches are combined in two ways, as follows:

1. The SDG-based approach simplifies the analyzed program, that is then checked by logic-based approach (see Chapter 6). The program is simplified by removing statements and certain branches in the program

that are irrelevant for the given noninterference property. The simplified program is easier to verify with the logic-based approach and also easier to test (e.g., with the approach from the framework described in Chapter 4).

2. The logic-based approach proves that certain dependencies computed by the SDG-based approach are over-approximations and can be discarded from the analysis (see Chapter 7). The logic-based approach analyzes possible dependencies between method inputs and outputs at a call site of that method. A proof obligation (in the form of a specified program) is automatically generated to show that (for the called method) a certain method output does not depend on a certain input. This proof obligation is enhanced with auxiliary specification that is generated by analyzing the SDG. The part of the program that needs to be handled with the logic-based approach can be a lot smaller than the entire program. The proof obligation can be further simplified using the SDG-based simplification approach presented in Chapter 6.

Program slicing. The framework for automatic program slicing (described in Part III) consists of an adapted relational verifier which can check whether a slice candidate is a valid slice and of a candidate generation engine which generates slice candidates according to a given strategy. The main challenge when checking the validity of a slice candidate is the case in which the criterion location is in the middle of the program. We adapted a relational verifier such that it can handle this case while still maintaining the advantages of relational verification for similar programs. Thus, the verification is performed automatically and no auxiliary specification (e.g., loop invariants) is needed. For the candidate generation engine we considered three strategies. The strategy BF generates all possible candidates, and the strategy SSE removes the instructions from the program one-by-one. A more complex strategy is CGS, which is based on refining dynamic slices (i.e., slices which are valid for a single input) by using the counterexamples provided by the adapted relational verifier. Whereas most state-of-the-art slicing approaches only perform a syntactical analysis of the program, this framework also considers the semantics of the program and can remove more statements from a program.

10.2 Future Work

In the following we present some ideas on how the two frameworks can be improved along with research questions that, in our opinion, should be pursued.

10.2.1 Improving the Noninterference Framework.

Combinations with type-system-based approaches. The framework for checking noninterference can be extended with an approach that checks noninterference using a type system. Type-system-based approaches (e.g., the one in Lortz et al. [2014]) handle interprocedural programs by using *method signatures*, which assign a security type (i.e., high or low) to each method parameter and to the return value of the method. A method signature constitutes a noninterference contract which states that if the parameters of the method have the respective security types specified in the method signature, then the return value will also have the type that is specified in the method signature. Such a type-system-based approach can easily be combined with the logic-based approach presented in Section 2.3 by considering the method signature to be a noninterference contract (and vice-versa) and proving their validity with either one of the approaches. There already exists work (e.g., Bauereiß et al. [2017] and [Scheben, 2014, Chapter 8]) which provide translations from the specification of noninterference properties between logic-based and type-system-based approaches.

The combination of logic-based and type-system-based approaches should be compared with the combinations of logic-based and SDG-based approaches (from Chapters 6 and 7) with respect to the reduction in effort needed to verify a noninterference property with the logic-based approach. It has been shown in Mantel and Sudbrock [2013] that approaches based on type systems can have the same scalability and precision as SDG-based approaches. However, in practice, implementations of type-system-based approaches are often less precise but more scalable than SDG-based approaches.

A type-system-based approach would also be useful in combination with the approach for analyzing noninterference violations form Chapter 5. An assumption in the context of that approach is that the user knows, for some program variables, whether they have a low or a high security level. A type-system-based approach can help the user find out the required security level of program variables.

Combinations with runtime monitors. Runtime software monitors may constitute an alternative to testing for the case in which for a given program the noninterference property was neither proved nor was a counterexample found for it. For the parts of the program which may affect the noninterference property we can generate runtime monitors which will then prevent noninterference violations at runtime. It should be investigated, however, how permissive the generated monitors are. Russo and Sabelfeld [2010] have shown that for flow-sensitive analyses the precision of sound dynamic techniques for checking noninterference is limited. Because both the SDG-based and the logic-based approaches that are used in the framework

are flow-sensitive, it could be that the generated software monitors will reject most executions.

Further development of the Combined Approach. The Combined Approach can be improved by using SDG-based analyses to generate additional auxiliary specifications—especially JML `assignable` clauses, which can be used in the generated method contracts and loop invariants. Such clauses increase the likelihood that the logic-based approach successfully disproves dependencies in the SDG. Furthermore, it should be investigated whether soundly generated JML `assignable` and `accessible` clauses (that respectively specify which memory locations may be written on or read from) can be useful for the verification of functional properties as well.

Another way in which the Combined Approach can be improved is by using the logic-based approach to show that a method does not throw any exception, which is a functional property. This can improve the precision of the SDG-based approach. For such a proof obligation, a points-to analysis can be especially useful, as it can show that the method inputs can never be null, when called in a given context. This can be then used by the logic-based approach to prove that a `NullPointerException` may never be raised.

The SDG-based approach can prove noninterference (according to an extended definition, see Giffhorn and Snelting [2015]) for multi-threaded programs. An interesting research direction to pursue is whether the Combined Approach can be used to disprove some over-approximated dependencies for multi-threaded programs as well, despite using a logic-based approach such as KeY that only supports sequential programs.

Further development of automatic test generation. For a better support of the case in which neither a proof nor a counterexample for the given program and noninterference property was found, the approach for automatic generation of noninterference tests from Chapter 4 can be further improved by researching additional coverage criteria. The new criteria should also take the parts of the program into consideration for which it was successfully shown with the SDG-based and logic-based approaches that they may not cause any violation of the noninterference property. Those program parts are represented by the SDG-nodes that are not in a reported violation chop and by the closed branches of a proof tree. Some first steps on defining and computing the coverage of a partial proof in KeY have been taken in Beckert et al. [2018c]. Combining the coverage achieved by tests and (partial) proofs was also done in [Omri, 2015, Chapter 5].

When generating test data, the approach from Chapter 4 searches models for every pair of path conditions from the two analyzed executions. It should be investigated whether an incremental SMT solver would be better suited for this task. Incremental SMT solvers reuse parts of the results obtained

when searching for a model for a similar formula. Their benefits for simple symbolic execution (where only single path conditions are solved) have been experimentally shown by Liu et al. [2014]. For the symbolic execution of two JavaDL modal operators, as done in Chapter 4, the performance benefits could be even greater.

Further development of the relational debugger. The relational debugger presented in Chapter 5 can be improved by allowing it to use the specified noninterference property to support the automatic suggestion of useful conditional break points or watch expressions. User studies should also be conducted in order to evaluate the usefulness of the approach for analyzing noninterference violations. Furthermore, it should be investigated for which other relational properties it can be used.

10.2.2 Improving the Slicing Framework

One major shortcoming of the slicing framework is that it must keep empty loops in the slice, otherwise it can offer no guarantees with respect to the mutual termination property of the original program and slice. The slicing approach can be improved by using an additional analysis on empty loops to check whether they terminate. If this is the case, then they can be removed without violating the slice property.

Another research direction that should be pursued is to improve the performance of the underlying relational verifier by using PDGs to simplify the programs that need to be checked for equivalence. This can be done by using the fact that two programs with isomorphic PDGs are equivalent, as shown by Horwitz et al. [1988].

Furthermore, it should be investigated how the results (e.g., coupling invariants) of the SMT solver employed by the relational verifier to check the validity of a slice candidate can be reused when checking another slice candidate that is constructed from the same original program. Incremental SMT solver may also be beneficial for checking the validity of similar slice candidates.

Part V

Appendix

Example of Noninterference Test

In Appendix A.1 we provide the full implementation of the example presented in Section 4.3.4.

A.1 Implementation

```
1    //Test Case for NodeNr: 1662
2  org.junit.Test
3   public void  testcode2(){
4     //Other variables
5     /*@ nullable */ java.lang.Throwable exc_2_A = null;
6     /*@ nullable */ java.lang.Throwable exc_2_B = null;
7
8     //Test preamble for execution A: creating objects and
          intializing test data
9     program _o1_A = new program();
10    java.lang.ArrayIndexOutOfBoundsException _o2_A = new
          java.lang.ArrayIndexOutOfBoundsException();
11    int[] _o4_A = new int[0];
12    java.lang.NegativeArraySizeException _o3_A = new java.
          lang.NegativeArraySizeException();
13    int h_2_A = (int)0;
14    int result_2_A = (int)4;
15
16    //Calling the method under test
17
18 int _h_2_A = h_2_A;
19 {
20   exc_2_A=null;try {
21     result_2_A=program.foo(_h_2_A);
22   }   catch (java.lang.Throwable e) {
23     exc_2_A=e;
24   }
25 }
26    //Test preamble for execution B: creating objects and
          intializing test data
27    java.lang.ArrayIndexOutOfBoundsException _o2_B = new
          java.lang.ArrayIndexOutOfBoundsException();
28    java.lang.NegativeArraySizeException _o3_B = new java.
          lang.NegativeArraySizeException();
29    program _o1_B = new program();
```

```
30      int[] _o4_B = new int[0];
31      int h_2_B = (int)-16;
32      int result_2_B = (int)0;
33
34      //Calling the method under test
35
36  int _h_2_B = h_2_B;
37  {
38    exc_2_B=null;try {
39      result_2_B=program.foo(_h_2_B);
40    }   catch (java.lang.Throwable e) {
41      exc_2_B=e;
42    }
43  }
44
45
46      Set<Boolean> allBools= new HashSet<Boolean>();
47      allBools.add(true);
48      allBools.add(false);
49
50      Set<Integer> allInts= new HashSet<Integer>();
51      allInts.add(-16);
52      allInts.add(-15);
53      allInts.add(-14);
54      allInts.add(-13);
55      allInts.add(-12);
56      allInts.add(-11);
57      allInts.add(-10);
58      allInts.add(-9);
59      allInts.add(-8);
60      allInts.add(-7);
61      allInts.add(-6);
62      allInts.add(-5);
63      allInts.add(-4);
64      allInts.add(-3);
65      allInts.add(-2);
66      allInts.add(-1);
67      allInts.add(0);
68      allInts.add(1);
69      allInts.add(2);
70      allInts.add(3);
71      allInts.add(4);
72      allInts.add(5);
73      allInts.add(6);
74      allInts.add(7);
75      allInts.add(8);
76      allInts.add(9);
77      allInts.add(10);
78      allInts.add(11);
79      allInts.add(12);
80      allInts.add(13);
81      allInts.add(14);
82      allInts.add(15);
83
84      Set<Object> allObjects= new HashSet<Object>();
85      allObjects.add(_o2_B);
86      allObjects.add(_o3_B);
87      allObjects.add(_o1_B);
88      allObjects.add(_o4_B);
```

```
89      allObjects.add(_o1_A);
90      allObjects.add(_o2_A);
91      allObjects.add(_o4_A);
92      allObjects.add(_o3_A);
93
94      //calling the test oracle
95      assertTrue(testOracle( exc_2_A, result_2_B, exc_2_B,
            result_2_A, allBools, allInts, allObjects));
96  }
97
98
99      public boolean testOracle(java.lang.Throwable exc_2_A,
            int result_2_B,java.lang.Throwable exc_2_B,int
            result_2_A,Set<Boolean> allBools,Set<Integer>
            allInts,Set<java.lang.Object> allObjects){
100         return (sub2( exc_2_A, result_2_B, exc_2_B,
                result_2_A, allBools, allInts, allObjects) && (
                result_2_A == result_2_B));
101     }
102
103     public boolean sameTypes(Object[] l1,Object[] l2){
104
105        for (int i = 0; i < l1.length; i++) {
106           if (l1[i] == null && l2[i] == null) return true;
107           if (l1[i] == null || l2[i] == null) return false
                  ;
108           if (!l1[i].getClass().equals(l2[i].getClass()))
                  return false;
109        }
110        return true;
111     }
112
113     public boolean objectsIsIsomorphic(Object[] l1,Object
           [] l2,Object o1,Object o2){
114
115        for (int i = 0; i < l1.length; i++) {
116           if ( (l1[i] == o1) != (l2[i] == o2)) return
                  false;
117        }
118        return true;
119     }
120
121     public boolean objectsAreIsomoprhic(Object[] l1,Object
           [] l2){
122
123        for (int i = 0; i < l1.length; i++) {
124           Object o1 = l1[i];
125           Object o2 = l2[i];
126           if (!objectsIsIsomorphic( l1, l2, o1, o2))
                  return false;
127        }
128        return true;
129     }
130
131     public boolean newObjects(Object[] l,Set<Object>
           allObjects){
132
133        for (Object o1 : l) {
134           for (Object o2 : allObjects) {
```

```
135                 if (o1 == o2) return false;
136             }
137         }
138         return true;
139     }
140 
141     public boolean sub2(java.lang.Throwable exc_2_A,int
            result_2_B,java.lang.Throwable exc_2_B,int
            result_2_A,Set<Boolean> allBools,Set<Integer>
            allInts,Set<java.lang.Object> allObjects){
142 
143         Object[] l1 = {exc_2_A};
144         Object[] l2 = {exc_2_B};
145         return newObjects( l1, allObjects) && newObjects(
                l2, allObjects) && sameTypes( l1, l2) &&
                objectsAreIsomoprhic( l1, l2);
146     }
147 }
```

Bibliography

Aharon Abadi, Ran Ettinger, and Yishai A. Feldman. Fine slicing. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 471–485, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28872-2. doi: 10.1007/978-3-642-28872-2_32. (Cited on page 90.)

Hiralal Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 302–312, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178456. URL `http://doi.acm.org/10.1145/178243.178456`. (Cited on page 90.)

Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990. ISSN 0362-1340. doi: 10.1145/93548.93576. URL `http://doi.acm.org/10.1145/93548.93576`. (Cited on pages 121 and 122.)

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *6th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, volume 8471 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, October 2014. ISBN 978-3-642-54107-0. doi: 10.1007/978-3-319-12154-3_4. URL `http://link.springer.com/chapter/10.1007/978-3-319-12154-3_4`.

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verifica-*

tion - The KeY Book: From Theory to Practice, volume 10001 of *Lecture Notes in Computer Science*. Springer, December 2016a. doi: 10.1007/978-3-319-49812-6. URL http://dx.doi.org/10.1007/978-3-319-49812-6. (Cited on pages 10 and 23.)

Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. *Proof-based Test Case Generation*, pages 415–451. Springer International Publishing, Cham, 2016b. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_12. URL https://doi.org/10.1007/978-3-319-49812-6_12. (Cited on pages 8, 10, 29, 31, and 49.)

Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL http://doi.acm.org/10.1145/390013.808479. (Cited on page 29.)

Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2013.02.061. URL http://www.sciencedirect.com/science/article/pii/S0164121213000563. (Cited on pages 31 and 132.)

Irina Măriuca Asăvoae, Mihail Asăvoae, and Adrián Riesco. Slicing from formal semantics: Chisel—a tool for generic program slicing. *International Journal on Software Tools for Technology Transfer*, 20(6):739–769, Nov 2018. ISSN 1433-2787. doi: 10.1007/s10009-018-0500-y. URL https://doi.org/10.1007/s10009-018-0500-y. (Cited on page 137.)

Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, pages 206–222, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-48141-6. doi: 10.1007/BFb0019410. URL http://dx.doi.org/10.1007/BFb0019410. (Cited on page 90.)

Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Ákos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theoretical Computer Science*, 411(11):1372 – 1386, 2010. ISSN 0304-3975. doi: 10.1016/j.tcs.2009.10.025. URL http://www.sciencedirect.com/science/article/pii/S0304397509007944. (Cited on page 116.)

Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. URL http://smtlib.cs.uiowa.edu/papers/smt-

lib-reference-v2.0-r12.09.09.pdf. Available at www.SMT-LIB.org. (Cited on pages 33 and 54.)

José Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. *Formal Aspects of Computing*, 24(2):217–248, Mar 2012. ISSN 1433-299X. doi: 10.1007/s00165-011-0196-1. URL https://doi.org/10.1007/s00165-011-0196-1. (Cited on pages 6, 111, 124, and 135.)

G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 100–114, June 2004. doi: 10.1109/CSFW.2004.1310735. URL http://dx.doi.org/10.1109/CSFW.2004.1310735. (Cited on page 25.)

Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. Rifl 1.1: A common specification language for information-flow requirements. Technical Report TUD-CS-2017-0225, TU Darmstadt, August 2017. URL http://dx.doi.org/10.5445/IR/1000092713. (Cited on page 141.)

Kent Beck. *JUnit Pocket Guide: Quick Look-up and Advice.* " O'Reilly Media, Inc.", 2004. (Cited on page 31.)

Bernhard Beckert. A dynamic logic for the formal verification ofjava card programs. In Isabelle Attali and Thomas Jensen, editors, *Java on Smart Cards:Programming and Security*, pages 6–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45165-5. doi: 10.1007/3-540-45165-X_2. URL http://dx.doi.org/10.1007/3-540-45165-X_2. (Cited on page 18.)

Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. Regression verification for programmable logic controller software. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 234–251, Cham, 2015. Springer International Publishing. ISBN 978-3-319-25423-4. doi: 10.1007/978-3-319-25423-4_15. URL http://dx.doi.org/10.1007/978-3-319-25423-4_15. (Cited on page 73.)

Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich. Automated verification for functional and relational properties of voting rules. In Umberto Grandi and Jeffrey S. Rosenschein, editors, *Sixth International Workshop on Computational Social Choice (COMSOC 2016)*, June 2016a. URL https://www.irit.fr/COMSOC-2016/proceedings/BeckertEtAlCOMSOC2016.pdf. (Cited on page 73.)

Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. *Dynamic Logic for Java*, pages 49–106. Springer International Publishing, Cham, 2016b. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_3. URL http://dx.doi.org/10.1007/978-3-319-49812-6_3. (Cited on pages 18 and 21.)

Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning. Combining graph-based and deduction-based information-flow analysis. In Ralf Küsters, editor, *5th Workshop on Hot Issues in Security Principles and Trust (HotSpot 2017) affiliated with ETAPS 2017: European Joint Conferences on Theory and Practice of Software*, pages 6–25, April 2017a. URL https://sec.informatik.uni-stuttgart.de/_media/events/hotspot2017/proceedings.pdf. (Cited on pages 8, 10, 11, and 93.)

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Semslice: Exploiting relational verification for automatic program slicing. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 312–319, Cham, 2017b. Springer International Publishing. ISBN 978-3-319-66845-1. doi: 10.1007/978-3-319-66845-1_20. URL http://dx.doi.org/10.1007/978-3-319-66845-1_20. (Cited on pages 9, 10, 11, 34, and 111.)

Bernhard Beckert, Sarah Grebing, and Mattias Ulbrich. An interaction concept for program verification systems with explicit proof object. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 163–178, Cham, 2017c. Springer International Publishing. ISBN 978-3-319-70389-3. doi: 10.1007/978-3-319-70389-3_11. URL http://dx.doi.org/10.1007/978-3-319-70389-3_11. (Cited on pages 66 and 133.)

Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning. Using theorem provers to increase the precision of dependence analysis for information flow control. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering*, pages 284–300, Cham, 2018a. Springer International Publishing. ISBN 978-3-030-02450-5. doi: 10.1007/978-3-030-02450-5_17. URL http://dx.doi.org/10.1007/978-3-030-02450-5_17. (Cited on pages 8, 10, 11, and 93.)

Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. Formal specification and verification of hyperledger fabric chaincode. In Guangdong Bai and Kamanashis Biswas, editors, *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM 2018: the 20th International Conference on Formal Engineering Methods*, November 2018b. URL https://symposium-dlt.org/.

Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018c. Springer International Publishing. ISBN 978-3-030-03421-4. doi: 10.1007/978-3-030-03421-4_4. URL http://dx.doi.org/10.1007/978-3-030-03421-4_4. (Cited on page 142.)

Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, Marko Kleine Büning, Holger Klein, and Joachim Müssig. Implementation of the combined approach, August 2019a. URL https://doi.org/10.5281/zenodo.3359433. (Cited on page 99.)

Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, Holger Klein, Marko Kleine Büning, and Joachim Müssig. Evaluation data of the combined approach, August 2019b. URL https://doi.org/10.5281/zenodo.3359387. (Cited on page 102.)

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Using relational verification for program slicing. In Peter Ölveczky and Gwen Salaün, editors, *17th International Conference on Software Engineering and Formal Methods (SEFM 2019)*, Lecture Notes in Computer Science, September 2019c. doi: 10.1007/978-3-030-30446-1_19. URL http://dx.doi.org/10.1007/978-3-030-30446-1_19. to appear. (Cited on pages 9, 10, 11, and 111.)

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Using relational verification for program slicing. Technical Report 2019,5, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, April 2019d. URL http://dx.doi.org/10.5445/IR/1000093895. (Cited on pages 9, 10, 11, and 111.)

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Evaluation data of semslice, July 2019e. URL https://doi.org/10.5281/zenodo.3334571. (Cited on page 124.)

Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Implementation of the semslice tool, July 2019f. URL https://doi.org/10.5281/zenodo.3334553. (Cited on pages 111, 113, and 124.)

Bernhard Beckert et al. Information flow in object-oriented software. In *Logic-Based Program Synthesis and Transformation (LOPSTR)*, pages 19–37, 2013. (Cited on pages 5, 17, and 25.)

David Binkley and Mark Harman. A survey of empirical results on program slicing. volume 62 of *Advances in Computers*, pages 105

– 178. Elsevier, 2004. doi: https://doi.org/10.1016/S0065-2458(03)62003-6. URL http://www.sciencedirect.com/science/article/pii/S0065245803620036. (Cited on pages 4 and 34.)

Simon Bischof. Combining SDG analysis and theorem proving for information flow control. Master thesis, Karlsruhe Institute of Technology, February 2016. (Cited on page 93.)

Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. Poster: Security in e-voting. In Sophie Engle, editor, *36th IEEE Symposium on Security and Privacy (S & P 2015), Poster Session*, May 2015. URL https://www.ieee-security.org/TC/SP2015/posters/paper_10.pdf.

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11):595 – 607, 1998. ISSN 0950-5849. doi: https://doi.org/10.1016/S0950-5849(98)00086-X. URL http://www.sciencedirect.com/science/article/pii/S095058499800086X. (Cited on pages 124 and 136.)

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, January 2018. ISSN 0360-0300. doi: 10.1145/3143561. URL http://doi.acm.org/10.1145/3143561. (Cited on page 132.)

Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, July 1994. ISSN 0164-0925. doi: 10.1145/183432.183438. URL http://doi.acm.org/10.1145/183432.183438. (Cited on page 90.)

I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, SAC '01, pages 605–609, New York, NY, USA, 2001. ACM. ISBN 1-58113-287-5. doi: 10.1145/372202.372784. URL http://doi.acm.org/10.1145/372202.372784. (Cited on page 136.)

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://doi.org/10.3233/JCS-2009-0393. (Cited on page 72.)

Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages

557–575, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-49749-3. doi: 10.1007/3-540-60973-3_107. URL http://dx.doi.org/10.1007/3-540-60973-3_107. (Cited on page 136.)

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13 (4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320. (Cited on page 31.)

Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: An online laboratory for program verification and analysis. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0063-6. doi: 10.1145/1868281.1868284. URL http://doi.acm.org/10.1145/1868281.1868284. (Cited on page 136.)

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through Horn clause transformation. In Xavier Rival, editor, *Static Analysis*, pages 147–169, Berlin, Heidelberg, 2016. Springer. ISBN 978-3-662-53413-7. doi: 10.1007/978-3-662-53413-7_8. URL http://dx.doi.org/10.1007/978-3-662-53413-7_8. (Cited on page 111.)

A. De Lucia. Program slicing: methods and applications. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Nov 2001. doi: 10.1109/SCAM.2001.972675. URL http://dx.doi.org/10.1109/SCAM.2001.972675. (Cited on page 34.)

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24. (Cited on pages 33, 41, and 52.)

Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL http://doi.acm.org/10.1145/360051.360056. (Cited on page 131.)

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL http://doi.acm.org/10.1145/359636.359712. (Cited on page 131.)

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124, May 2010. doi: 10.1109/SP.2010.15. URL http://dx.doi.org/10.1109/SP.2010.15. (Cited on page 135.)

Quoc Huy Do, Eduard Kamburjan, and Nathan Wasser. Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust - (POST), Part of (ETAPS)*, volume 9635 of *LNCS*, pages 97–115. Springer, 2016. (Cited on pages 50 and 132.)

Aboubakr Achraf El Ghazi. *Relational Reasoning - Constraint Solving, Deduction, and Program Verification*. PhD thesis, 2015. URL http://dx.doi.org/10.5445/IR/1000051022. (Cited on page 33.)

Aboubakr Achraf El Ghazi, Mattias Ulbrich, Mana Taghdiri, and Mihai Herda. Reducing the complexity of quantified formulas via variable elimination. In *11th International Workshop on Satisfiability Modulo Theories (SMT 2013)*, pages 87–99, July 2013. URL http://arxiv.org/abs/1408.0700.

Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-order transitive closure axiomatization via iterative invariant injections. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 143–157, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17524-9. doi: 10.1007/978-3-319-17524-9_11. URL http://dx.doi.org/10.1007/978-3-319-17524-9_11.

Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs*, pages 169–188, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73770-4. doi: 10.1007/978-3-540-73770-4_10. URL http://dx.doi.org/10.1007/978-3-540-73770-4_10. (Cited on page 29.)

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 349–360, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642987. URL http://doi.acm.org/10.1145/2642937.2642987. (Cited on pages 41, 125, and 126.)

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL http://doi.acm.org/10.1145/24039.24041. (Cited on pages 34 and 36.)

John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 379–392, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199534. URL http://doi.acm.org/10.1145/199448.199534. (Cited on pages 124 and 125.)

Stefan J. Galler and Bernhard K. Aichernig. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer*, 16(6):727–751, Nov 2014. ISSN 1433-2787. doi: 10.1007/s10009-013-0272-3. URL https://doi.org/10.1007/s10009-013-0272-3. (Cited on page 132.)

Gerhard Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische Zeitschrift*, 39(1):405–431, Dec 1935. ISSN 1432-1823. doi: 10.1007/BF01201363. URL https://doi.org/10.1007/BF01201363. (Cited on page 21.)

Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, Jun 2015. ISSN 1615-5270. doi: 10.1007/s10207-014-0257-6. URL https://doi.org/10.1007/s10207-014-0257-6. (Cited on page 142.)

J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982. doi: 10.1109/SP.1982.10014. URL http://dx.doi.org/10.1109/SP.1982.10014. (Cited on pages 3, 15, and 131.)

Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Tool demonstration: Joana. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust*, pages 89–93, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49635-0. doi: 10.1007/978-3-662-49635-0_5. URL http://dx.doi.org/10.1007/978-3-662-49635-0_5. (Cited on pages 5 and 38.)

Simon Greiner and Mihai Herda. Cocome with security. Technical Report 2017,2, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, April 2017. URL http://dx.doi.org/10.5445/IR/1000065106.

Damas P Gruska. Information flow testing. *Fundamenta Informaticae*, 128 (1-2):81–95, 2013. doi: 10.3233/FI-2013-934. URL https://doi.org/10.3233/FI-2013-934. (Cited on page 132.)

G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium*

(CSF'07), pages 218–232, July 2007. doi: 10.1109/CSF.2007.10. URL http://dx.doi.org/10.1109/CSF.2007.10. (Cited on pages 5 and 134.)

Raju Halder and Agostino Cortesi. Abstract program slicing on dependence condition graphs. *Science of Computer Programming*, 78(9):1240 – 1263, 2013. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2012.05.007. URL http://www.sciencedirect.com/science/article/pii/S0167642312001098. (Cited on page 136.)

Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, Mar 1995. ISSN 1573-7535. doi: 10.1007/BF00873408. URL http://dx.doi.org/10.1007/BF00873408. (Cited on page 122.)

Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. A uniform information-flow security benchmark suite for source code and bytecode. In Nils Gruschka, editor, *Secure IT Systems*, pages 437–453, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03638-6. doi: 10.1007/978-3-030-03638-6_27. URL http://dx.doi.org/10.1007/978-3-030-03638-6_27. (Cited on page 61.)

Christian Hammer. *Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs*. PhD thesis, Karlsruhe Institute of Technology, 2009. URL http://dx.doi.org/10.5445/KSP/1000012049. (Cited on pages 35, 37, 40, 90, and 134.)

Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec 2009. ISSN 1615-5270. doi: 10.1007/s10207-009-0086-1. URL https://doi.org/10.1007/s10207-009-0086-1. (Cited on pages 4, 34, and 39.)

David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*, pages 99–217. Springer Netherlands, Dordrecht, 2002. ISBN 978-94-017-0456-4. doi: 10.1007/978-94-017-0456-4_2. URL https://doi.org/10.1007/978-94-017-0456-4_2. (Cited on page 18.)

Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance: Research and Practice*, 10(6):415–441, 1998. doi: 10.1002/(SICI)1096-908X(199811/12)10:6<415::AID-SMR180>3.0.CO;2-Z. URL http://dx.doi.org/10.1002/(SICI)1096-908X(199811/12)10:6<415::AID-SMR180>3.0.CO;2-Z. (Cited on page 90.)

Mark Harman, Arun Lakhotia, and David Binkley. Theory and algorithms for slicing unstructured programs. *Information and Software Technology*, 48 (7):549 – 565, 2006. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.

2005.06.001. URL http://www.sciencedirect.com/science/article/pii/S0950584905000881. (Cited on page 90.)

Martin Hentschel. *Integrating Symbolic Execution, Debugging and Verification*. PhD thesis, Technische Universität Darmstadt, January 2016. URL http://tuprints.ulb.tu-darmstadt.de/5399/. (Cited on page 133.)

Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic execution debugger (sed). In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 255–262, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11164-3. doi: 10.1007/978-3-319-11164-3_21. URL http://dx.doi.org/10.1007/978-3-319-11164-3_21. (Cited on page 133.)

Mihai Herda, Shmuel S. Tyszberowicz, and Bernhard Beckert. Using dependence graphs to assist verification and testing of information-flow properties. In Catherine Dubois and Burkhart Wolff, editors, *12th International Conference on Tests and Proofs (TAP 2018)*, volume 10889 of *Lecture Notes in Computer Science*, pages 83–102. Springer, June 2018. ISBN 978-3-319-92994-1. doi: 10.1007/978-3-319-92994-1_5. URL http://dx.doi.org/10.1007/978-3-319-92994-1_5. (Cited on pages 8, 10, and 75.)

Mihai Herda, Michael Kirsten, Etienne Brunner, Joana Plewnia, Ulla Scheler, Chiara Staudenmaier, Benedikt Wagner, Pascal Zwick, and Bernhard Beckert. Understanding counterexamples for relational properties with dibugger. In Emanuele De Angelis, Grigory Fedyukovich, Nikos Tzevelekos, and Mattias Ulbrich, editors, *Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning (HCVS/PERR 2019)*, volume 296 of *EPTCS*, pages 6–13. Open Publishing Association, July 2019a. doi: 10.4204/EPTCS.296.4. URL http://dx.doi.org/10.4204/EPTCS.296.4. (Cited on pages 8, 10, and 65.)

Mihai Herda, Michael Kirsten, Etienne Brunner, Joana Plewnia, Ulla Scheler, Chiara Staudenmaier, Benedikt Wagner, Pascal Zwick, and Bernhard Beckert. Implementation of dibugger, July 2019b. URL https://doi.org/10.5281/zenodo.3334650. (Cited on page 65.)

Mihai Herda, Shmuel Tyszberowicz, Joachim Müssig, and Bernhard Beckert. Verification-based test case generation for information-flow properties. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2231–2238, New York, NY, USA, 2019c. ACM. ISBN 978-1-4503-5933-7. doi: 10.1145/3297280.3297500. URL http://doi.acm.org/10.1145/3297280.3297500. (Cited on pages 8, 10, 15, 25, and 49.)

Mihai Herda, Shmuel Tyszberowicz, Joachim Müssig, and Bernhard Beckert. Evaluation Data of the Implementation of the Approach for Automatic Test Generation for Information-Flow Properties, July 2019d. URL https://doi.org/10.5281/zenodo.3334380. (Cited on page 61.)

Mihai Herda, Shmuel Tyszberowicz, Joachim Müssig, and Bernhard Beckert. Implementation of the Approach for Automatic Test Generation for Information-Flow Properties, July 2019e. URL https://doi.org/10.5281/zenodo.3334532. (Cited on page 50.)

S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 146–157, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73573. URL http://doi.acm.org/10.1145/73560.73573. (Cited on page 143.)

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990. ISSN 0164-0925. doi: 10.1145/77606.77608. URL http://doi.acm.org/10.1145/77606.77608. (Cited on pages 36 and 37.)

Catalin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Denes, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. Testing noninterference, quickly. *Journal of Functional Programming*, 26:e4, 2016. doi: 10.1017/S0956796816000058. URL http://dx.doi.org/10.1017/S0956796816000058. (Cited on page 132.)

Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. *Formal Specification with the Java Modeling Language*, pages 193–241. Springer International Publishing, Cham, 2016. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_7. URL https://doi.org/10.1007/978-3-319-49812-6_7. (Cited on page 23.)

Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111045. URL http://doi.acm.org/10.1145/1111037.1111045. (Cited on page 134.)

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002. ISSN 1049-331X. doi: 10.1145/505145.505149. URL http://doi.acm.org/10.1145/505145.505149. (Cited on page 54.)

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Path-sensitive backward slicing. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 231–247, Berlin, Heidelberg, 2012a. Springer Berlin Heidelberg. ISBN 978-3-642-33125-1. doi: 10.1007/978-3-642-33125-1_17. URL http://dx.doi.org/10.1007/978-3-642-33125-1_17. (Cited on pages 6, 111, 124, 125, and 136.)

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Tracer: A symbolic execution tool for verification. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 758–766, Berlin, Heidelberg, 2012b. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7. doi: 10.1007/978-3-642-31424-7_61. URL http://dx.doi.org/10.1007/978-3-642-31424-7_61. (Cited on page 136.)

Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In Franz Baader, Peter Baumgartner, Robert Nieuwenhuis, and Andrei Voronkov, editors, *Deduction and Applications*, number 05431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2006/511. (Cited on page 101.)

Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR. *Journal of Automated Reasoning*, 60(3): 337–363, Mar 2018. ISSN 1573-0670. doi: 10.1007/s10817-017-9433-5. URL https://doi.org/10.1007/s10817-017-9433-5. (Cited on pages 41, 42, 73, 111, 125, and 126.)

Johannes Kinder. Hypertesting: The case for automated testing of hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot 2015)*, 3 2015. URL https://pure.royalholloway.ac.uk/portal/files/24051361/hotspot15.pdf. (Cited on page 131.)

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. ISSN 1433-299X. doi: 10.1007/s00165-014-0326-7. URL http://dx.doi.org/10.1007/s00165-014-0326-7. (Cited on page 137.)

Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification of pointer programs by predicate abstraction. *Formal Methods in System Design*, 52(3):229–259, Jun 2018. ISSN 1572-8102. doi: 10.1007/s10703-017-0293-8. URL https://doi.org/10.1007/s10703-017-0293-8. (Cited on page 127.)

Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988. ISSN 0020-0190. doi: https://doi.

org/10.1016/0020-0190(88)90054-3. URL http://www.sciencedirect.com/science/article/pii/0020019088900543. (Cited on page 121.)

Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11):647 – 659, 1998. ISSN 0950-5849. doi: https://doi.org/10.1016/S0950-5849(98)00089-5. URL http://www.sciencedirect.com/science/article/pii/S0950584998000895. (Cited on pages 121 and 137.)

R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A hybrid approach for proving noninterference of java programs. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 305–319, July 2015. doi: 10.1109/CSF.2015.28. URL http://dx.doi.org/10.1109/CSF.2015.28. (Cited on page 133.)

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673. (Cited on page 41.)

Gurvan Le Guernic. Information flow testing. In Iliano Cervesato, editor, *Advances in Computer Science – ASIAN 2007. Computer and Network Security*, pages 33–47, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76929-3. doi: 10.1007/978-3-540-76929-3_4. URL http://dx.doi.org/10.1007/978-3-540-76929-3_4. (Cited on page 132.)

Gary T. Leavens, Joseph R. Kiniry, and Erik Poll. A JML tutorial: Modular specification and verification of functional behavior for Java. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, page 37. Springer, 2007. (Cited on page 23.)

Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs: Sound verification on relaxed slices. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, pages 179–196, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49665-7. doi: 10.1007/978-3-662-49665-7_11. URL http://dx.doi.org/10.1007/978-3-662-49665-7_11. (Cited on page 126.)

Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In Eran Yahav, editor, *Hardware and Software: Verification and Testing*, pages 284–299, Cham, 2014. Springer International Publishing. ISBN 978-3-319-13338-6. doi: 10.1007/978-3-319-13338-6_21. URL http://dx.doi.org/10.1007/978-3-319-13338-6_21. (Cited on page 143.)

Tianhai Liu, Shmuel Tyszberowicz, Mihai Herda, Bernhard Beckert, Daniel Grahl, and Mana Taghdiri. Computing specification-sensitive abstractions for program verification. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications*, pages 101–117, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47677-3. doi: 10.1007/978-3-319-47677-3_7. URL http://dx.doi.org/10.1007/978-3-319-47677-3_7. (Cited on page 136.)

Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 93–104, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3155-5. doi: 10.1145/2666620.2666631. URL http://doi.acm.org/10.1145/2666620.2666631. (Cited on pages 5, 134, and 141.)

Heiko Mantel and Henning Sudbrock. Types vs. pdgs in information flow analysis. In Elvira Albert, editor, *Logic-Based Program Synthesis and Transformation*, pages 106–121, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38197-3. doi: 10.1007/978-3-642-38197-3_8. URL http://dx.doi.org/10.1007/978-3-642-38197-3_8. (Cited on pages 5, 134, and 141.)

J. McCarthy. *Towards a Mathematical Science of Computation*, pages 35–56. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7. doi: 10.1007/978-94-011-1793-7_2. URL https://doi.org/10.1007/978-94-011-1793-7_2. (Cited on page 18.)

Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 152–168, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-30793-5. doi: 10.1007/978-3-642-30793-5_10. URL http://dx.doi.org/10.1007/978-3-642-30793-5_10. (Cited on pages 6, 50, and 131.)

Joachim Müssig. Erweiterung des Theorembeweisers KeY um automatische Testgenerierung für Informationsflusseigenschaften, March 2018. (Cited on page 49.)

Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292561. URL http://doi.acm.org/10.1145/292540.292561. (Cited on page 134.)

Objenesis. http://objenesis.org/, 2018. [Online; accessed 28-July-2018]. (Cited on page 55.)

Fouad ben Nasr Omri. *Weighted Statistical Testing based on Active Learning and Formal Verification Techniques for Software Reliability Assessment*. PhD thesis, 2015. URL http://dx.doi.org/10.5445/IR/1000050941. (Cited on page 142.)

A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sep. 1990. ISSN 0098-5589. doi: 10.1109/32.58784. URL http://dx.doi.org/10.1109/32.58784. (Cited on pages 126 and 137.)

V. R. Pratt. Semantical consideration on floyo-hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121, Oct 1976. doi: 10.1109/SFCS.1976.27. URL http://dx.doi.org/10.1109/SFCS.1976.27. (Cited on page 18.)

Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007. ISSN 0164-0925. doi: 10.1145/1275497.1275502. URL http://dx.doi.org/10.1145/1275497.1275502. (Cited on pages 29, 116, and 126.)

Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 41–52, New York, NY, USA, 1995. ACM. ISBN 0-89791-716-2. doi: 10.1145/222124.222138. URL http://doi.acm.org/10.1145/222124.222138. (Cited on page 39.)

Adrián Riesco, Irina Măriuca Asăvoae, and Mihail Asăvoae. A generic program slicing technique based on language definitions. In Narciso Martí-Oliet and Miguel Palomino, editors, *Recent Trends in Algebraic Development Techniques*, pages 248–264, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37635-1. doi: 10.1007/978-3-642-37635-1_15. URL http://dx.doi.org/10.1007/978-3-642-37635-1_15. (Cited on page 137.)

E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443 – 476, 2007. ISSN 0747-7171. doi: https://doi.org/10.1016/j.jsc.2007.01.002. URL http://www.sciencedirect.com/science/article/pii/S0747717107000107. (Cited on page 101.)

Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 347–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8. doi: 10.1007/978-3-642-39799-8_24. URL http://dx.doi.org/10.1007/978-3-642-39799-8_24. (Cited on page 41.)

A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 186–199, July 2010. doi: 10.1109/CSF.2010.20. URL http://dx.doi.org/10.1109/CSF.2010.20. (Cited on pages 6, 132, 135, and 141.)

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121. URL http://dx.doi.org/10.1109/JSAC.2002.806121. (Cited on pages 49 and 135.)

Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 352–365, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-11486-1. doi: 10.1007/978-3-642-11486-1_30. URL http://dx.doi.org/10.1007/978-3-642-11486-1_30. (Cited on page 135.)

Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046878. (Cited on pages 25, 26, 27, 28, and 141.)

Christoph Scheben and Simon Greiner. *Information Flow Analysis*, pages 453–471. Springer International Publishing, Cham, 2016. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_13. URL http://dx.doi.org/10.1007/978-3-319-49812-6_13. (Cited on pages 25 and 53.)

Peter H. Schmitt. *First-Order Logic*, pages 23–47. Springer International Publishing, Cham, 2016. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_2. URL http://dx.doi.org/10.1007/978-3-319-49812-6_2. (Cited on pages 21 and 33.)

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, pages 138–152, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18070-5. doi: 10.1007/978-3-642-18070-5_10. URL http://dx.doi.org/10.1007/978-3-642-18070-5_10. (Cited on page 18.)

P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 203–217, July 2007. doi: 10.1109/CSF.2007.20. URL http://dx.doi.org/10.1109/CSF.2007.20. (Cited on page 134.)

Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis*, pages 332–348, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70674-8. doi: 10.1007/3-540-61739-6_51. URL http://dx.doi.org/10.1007/3-540-61739-6_51. (Cited on page 38.)

Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, October 2006. ISSN 1049-331X. doi: 10.1145/1178625.1178628. URL http://doi.acm.org/10.1145/1178625.1178628. (Cited on pages 134 and 136.)

Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In Pierpaolo Degano, Sandro Etalle, and Joshua Guttman, editors, *Formal Aspects of Security and Trust*, pages 65–79, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19751-2. doi: 10.1007/978-3-642-19751-2_5. URL http://dx.doi.org/10.1007/978-3-642-19751-2_5. (Cited on page 134.)

Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994. (Cited on pages 6 and 137.)

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35, November 2012. ISSN 0164-0925. doi: 10.1145/2362389.2362390. URL http://doi.acm.org/10.1145/2362389.2362390. (Cited on page 112.)

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996. doi: 10.3233/JCS-1996-42-304. URL http://dx.doi.org/10.3233/JCS-1996-42-304. (Cited on page 134.)

M. Ward. Properties of slicing definitions. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 23–32, Sep. 2009. doi: 10.1109/SCAM.2009.12. URL http://dx.doi.org/10.1109/SCAM.2009.12. (Cited on pages 124 and 125.)

Daniel Wasserrab and Denis Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *VERIFY-2010.*

6th International Verification Workshop, volume 3 of *EPiC Series in Computing*, pages 141–155. EasyChair, 2012. doi: 10.29007/nnzj. URL https://easychair.org/publications/paper/FSk. (Cited on pages 36, 39, and 40.)

Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL http://dl.acm.org/citation.cfm?id=800078.802557. (Cited on pages 3, 34, 112, and 116.)

Benjamin Weiß. *Deductive verification of object-oriented software : dynamic frames, dynamic logic and predicate abstraction.* PhD thesis, 2011. URL https://dx.doi.org/10.5445/KSP/1000021694. (Cited on pages 18, 19, and 20.)

L. J. White. Basic mathematical definitions and results in testing. *Computer Program Testing*, pages 13–24, 1981. (Cited on page 29.)

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30 (2):1–36, March 2005. ISSN 0163-5948. doi: 10.1145/1050849.1050865. URL http://doi.acm.org/10.1145/1050849.1050865. (Cited on pages 6 and 137.)

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL http://doi.acm.org/10.1145/267580.267590. (Cited on page 29.)

Publication List

B.1 Peer-Reviewed Conference and Workshop Papers

1. Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Using relational verification for program slicing. In Peter Ölveczky and Gwen Salaün, editors, *17th International Conference on Software Engineering and Formal Methods (SEFM 2019)*, Lecture Notes in Computer Science, September 2019c. doi: 10.1007/978-3-030-30446-1_19. URL `http://dx.doi.org/10.1007/978-3-030-30446-1_19`. to appear

2. Mihai Herda, Shmuel Tyszberowicz, Joachim Müssig, and Bernhard Beckert. Verification-based test case generation for information-flow properties. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 2231–2238, New York, NY, USA, 2019c. ACM. ISBN 978-1-4503-5933-7. doi: 10.1145/3297280.3297500. URL `http://doi.acm.org/10.1145/3297280.3297500`

3. Mihai Herda, Michael Kirsten, Etienne Brunner, Joana Plewnia, Ulla Scheler, Chiara Staudenmaier, Benedikt Wagner, Pascal Zwick, and Bernhard Beckert. Understanding counterexamples for relational properties with dibugger. In Emanuele De Angelis, Grigory Fedyukovich, Nikos Tzevelekos, and Mattias Ulbrich, editors, *Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning (HCVS/PERR 2019)*, volume 296 of *EPTCS*, pages 6–13. Open Publishing Association, July 2019a. doi: 10.4204/EPTCS.296.4. URL `http://dx.doi.org/10.4204/EPTCS.296.4`

4. Tobias Hamann, Mihai Herda, Heiko Mantel, Martin Mohr, David Schneider, and Markus Tasch. A uniform information-flow security benchmark suite for source code and bytecode. In Nils Gruschka, editor,

Secure IT Systems, pages 437–453, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03638-6. doi: 10.1007/978-3-030-03638-6_27. URL http://dx.doi.org/10.1007/978-3-030-03638-6_27

5. Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. Formal specification and verification of hyperledger fabric chaincode. In Guangdong Bai and Kamanashis Biswas, editors, *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM 2018: the 20th International Conference on Formal Engineering Methods*, November 2018b. URL https://symposium-dlt.org/

6. Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning. Using theorem provers to increase the precision of dependence analysis for information flow control. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering*, pages 284–300, Cham, 2018a. Springer International Publishing. ISBN 978-3-030-02450-5. doi: 10.1007/978-3-030-02450-5_17. URL http://dx.doi.org/10.1007/978-3-030-02450-5_17

7. Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a notion of coverage for incomplete program-correctness proofs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018c. Springer International Publishing. ISBN 978-3-030-03421-4. doi: 10.1007/978-3-030-03421-4_4. URL http://dx.doi.org/10.1007/978-3-030-03421-4_4

8. Mihai Herda, Shmuel S. Tyszberowicz, and Bernhard Beckert. Using dependence graphs to assist verification and testing of information-flow properties. In Catherine Dubois and Burkhart Wolff, editors, *12th International Conference on Tests and Proofs (TAP 2018)*, volume 10889 of *Lecture Notes in Computer Science*, pages 83–102. Springer, June 2018. ISBN 978-3-319-92994-1. doi: 10.1007/978-3-319-92994-1_5. URL http://dx.doi.org/10.1007/978-3-319-92994-1_5

9. Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Semslice: Exploiting relational verification for automatic program slicing. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 312–319, Cham, 2017b. Springer International Publishing. ISBN 978-3-319-66845-1. doi: 10.1007/978-3-319-66845-1_20. URL http://dx.doi.org/10.1007/978-3-319-66845-1_20

10. Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning. Combining graph-based and deduction-based information-flow analysis. In Ralf Küsters, editor, *5th Workshop on*

Hot Issues in Security Principles and Trust (HotSpot 2017) affiliated with ETAPS 2017: European Joint Conferences on Theory and Practice of Software, pages 6–25, April 2017a. URL https://sec.informatik.uni-stuttgart.de/_media/events/hotspot2017/proceedings.pdf

11. Tianhai Liu, Shmuel Tyszberowicz, Mihai Herda, Bernhard Beckert, Daniel Grahl, and Mana Taghdiri. Computing specification-sensitive abstractions for program verification. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications*, pages 101–117, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47677-3. doi: 10.1007/978-3-319-47677-3_7. URL http://dx.doi.org/10.1007/978-3-319-47677-3_7

12. Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-order transitive closure axiomatization via iterative invariant injections. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 143–157, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17524-9. doi: 10.1007/978-3-319-17524-9_11. URL http://dx.doi.org/10.1007/978-3-319-17524-9_11

13. Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *6th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, volume 8471 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, October 2014. ISBN 978-3-642-54107-0. doi: 10.1007/978-3-319-12154-3_4. URL http://link.springer.com/chapter/10.1007/978-3-319-12154-3_4

14. Aboubakr Achraf El Ghazi, Mattias Ulbrich, Mana Taghdiri, and Mihai Herda. Reducing the complexity of quantified formulas via variable elimination. In *11th International Workshop on Satisfiability Modulo Theories (SMT 2013)*, pages 87–99, July 2013. URL http://arxiv.org/abs/1408.0700

B.2 Book Chapters

1. Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. *Proof-based Test Case Generation*, pages 415–451. Springer International Publishing, Cham, 2016b. ISBN 978-3-319-49812-6. doi: 10.1007/978-3-319-49812-6_12. URL https://doi.org/10.1007/978-3-319-49812-6_12

B.3 Peer-Reviewed Posters

1. Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. Poster: Security in e-voting. In Sophie Engle, editor, *36th IEEE Symposium on Security and Privacy (S & P 2015), Poster Session*, May 2015. URL https://www.ieee-security.org/TC/SP2015/posters/paper_10.pdf

B.4 Non-Peer-Reviewed Publications

1. Bernhard Beckert, Thorsten Bormer, Stephan Gocht, Mihai Herda, Daniel Lentzsch, and Mattias Ulbrich. Using relational verification for program slicing. Technical Report 2019,5, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, April 2019d. URL http://dx.doi.org/10.5445/IR/1000093895

2. Simon Greiner and Mihai Herda. Cocome with security. Technical Report 2017,2, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, April 2017. URL http://dx.doi.org/10.5445/IR/1000065106

3. Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. Rifl 1.1: A common specification language for information-flow requirements. Technical Report TUD-CS-2017-0225, TU Darmstadt, August 2017. URL http://dx.doi.org/10.5445/IR/1000092713