# Information Flow Control with System Dependence Graphs
## Improving Modularity, Scalability and Precision for Object Oriented Languages

zur Erlangung des akademischen Grads eines

Doktors der Ingenieurswissenschaften

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

**genehmigte**
**Dissertation**

von

**Jürgen Graf**

aus Bad Kötzting

| | |
|---|---|
| Tag der mündlichen Prüfung: | 14.11.2016 |
| Erster Gutachter: | Prof. Dr.-Ing. Gregor Snelting |
| Zweiter Gutachter: | Prof. Dr. Walter F. Tichy |

# Contents

# Acknowledgements

First of all, I wish to thank my advisor Gregor Snelting for giving me to opportunity to do this work.

I thank my parents for their support during all that time from my early education to enabling my studies in computer science.

A special thanks to my partner Nicole that went with me through all ups and downs during the work on this thesis.

I thank all my current and former coworkers Martin Mohr, Martin Hecker, Christian Hammer and Dennis Giffhorn for the interesting discussions and their ideas and work that resulted in many improvements.

My other coworkers working in a different field also helped through numerous dicussions and provided new insights with their different point of view. A "thank you" goes to: Matthias Braun, Joachim Breitner, Sebastian Buchwald, Andreas Lochbihler, Denis Lohner, Manuel Mohr, Daniel Wasserrab and Andreas Zwinkau.

I also thank Brigitte Sehan for providing support in the background and the numerous students that worked on parts of Joana.

Finally I want to thank my former teacher Werner Steidl for sparking my interest in computer science with the voluntary computer science classes he taught in his spare time.

# Abstract

This work is concerned with the field of static program analysis —in particular with analyses aimed to guarantee certain security properties of programs, like *confidentiality* and *integrity*. Our approach uses so-called *dependence graphs* to capture the program behavior as well as the information flow between the individual program points. Using this technique, we can guarantee for example that a program does not reveal any information about a secret password.

In particular we focus on techniques that improve the dependence graph computation —the basis for many advanced security analyses. We incorporated the presented algorithms and improvements into our analysis tool Joana and published its source code as open source. Several collaborations with other researchers and publications using Joana demonstrate the relevance of these improvements for practical research.

This work consists essentially of three parts. Part 1 deals with improvements in the computation of the dependence graph, Part 2 introduces a new approach to the analysis of incomplete programs and Part 3 shows current use cases of Joana on concrete examples.

In the first part we describe the algorithms used to compute a dependence graph, with special attention to the problems and challenges that arise when analyzing object-oriented languages such as Java. For example we present an analysis that improves the precision of detected control flow by incorporating the effects of exceptions. The main improvement concerns the way side effects —caused by communication over methods boundaries— are modelled. Dependence graphs capture side effects —memory locations read or changed by a method— in the form of additional nodes called parameter nodes. We show that the structure and computation of these nodes have a huge impact on both the precision and scalability of the entire analysis. The so-called *parameter model* describes the algorithms used to compute these nodes. We explain the weakness of the old parameter model based on *object-trees* and present our improvements in form of a new model using *object-graphs*. The new graph structure merges redundant information of multiple nodes into a single node and thus reduces the number of overall parameter nodes significantly — which in turn speeds up the analysis without sacrificing the precision of the resulting dependence graph. Theses changes are already visible when analyzing smaller programs with a few thousand lines of code: We achieve on average a 8 times faster runtime while the precision of the result remains intact and is usually even enhanced. The differences are even more pronounced for larger programs. Some of our test cases and all tested programs larger then 20,000 lines of code could only be analyzed with the object-graph parameter model. Due to these enhancements Joana is now able to analyze much larger programs and also profits from enhanced precision with smaller programs.

In the second part we tackle the problem that security analyses based on dependence graphs previously required a whole program in order to compute. For example it was impossible to preprocess or analyze program parts like library code without knowledge of the application code using it. We discovered a *monotonicity property* in the current analysis that allows us to reuse analysis results from a program part at a given usage point to conservatively approximate the expected results at another point without the need to reanalyze the program part. Due to monotonicity we are able to make valid statements about the security properties of a program part in general, without knowledge of its usage points. It also allows us to preprocess program parts in form of a *modular dependence graph* in a way that can be adjusted later on to a concrete usage point. We define the monotonicity property in detail and outline a proof of its correctness. Based on this observation, we develop an approach to preprocess program parts with modular dependence graphs. As the precise computation of modular dependence graphs can become very costly, we developed an algorithm based on so-called *access paths* to improve scalability. Finally, we sketch a proof showing the presented algorithm in fact always computes a conservative approximation of the modular graph and therefore any security analysis performed on a modular graph remains sound.

The third part contains successful applications of Joana. We present the results of a cooperation with Ralf Küsters from the University of Trier. We explain how our security tool Joana is used in general and also how —in combination with other security tools and techniques— we were able to proof cryptographic security properties for Java programs — a task previously not possible for tools based purely on information flow control. These applications show how the usability improvements of Joana —developed in the context of this work— considerably reduce the effort required to setup the analysis. We also explain how our precision enhancements were crucial for the successful analysis of these applications.

# Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Gebiet der statischen Programmanalyse — insbesondere betrachten wir Analysen, deren Ziel es ist, bestimmte Sicherheitseigenschaften, wie etwa *Integrität* und *Vertraulichkeit*, für Programme zu garantieren. Hierfür verwenden wir sogenannte *Abhängigkeitsgraphen*, welche das potentielle Verhalten des Programms sowie den Informationsfluss zwischen einzelnen Programmpunkten abbilden. Mit Hilfe dieser Technik können wir sicherstellen, dass z.B. ein Programm keinerlei Information über ein geheimes Passwort preisgibt.

Im Speziellen liegt der Fokus dieser Arbeit auf Techniken, die das Erstellen des Abhängigkeitsgraphen verbessern, da dieser die Grundlage für viele weiterführende Sicherheitsanalysen bildet. Die vorgestellten Algorithmen und Verbesserungen wurden in unser Analysetool Joana integriert und als Open-Source öffentlich verfügbar gemacht. Zahlreiche Kooperationen und Veröffentlichungen belegen, dass die Verbesserungen an Joana auch in der Forschungspraxis relevant sind.

Diese Arbeit besteht im Wesentlichen aus drei Teilen. Teil 1 befasst sich mit Verbesserungen bei der Berechnung des Abhängigkeitsgraphen, Teil 2 stellt einen neuen Ansatz zur Analyse von unvollständigen Programmen vor und Teil 3 zeigt aktuelle Verwendungsmöglichkeiten von Joana an konkreten Beispielen.

Im ersten Teil gehen wir detailliert auf die Algorithmen zum Erstellen eines Abhängigkeitsgraphen ein, dabei legen wir besonderes Augenmerk auf die Probleme und Herausforderung bei der Analyse von Objektorientierten Sprachen wie Java. So stellen wir z.B. eine Analyse vor, die den durch Exceptions ausgelösten Kontrollfluss präzise behandeln kann. Hauptsächlich befassen wir uns mit der Modellierung von Seiteneffekten, die bei der Kommunikation über Methodengrenzen hinweg entstehen können. Bei Abhängigkeitsgraphen werden Seiteneffekte, also Speicherstellen, die von einer Methode gelesen oder verändert werden, in Form von zusätzlichen Knoten dargestellt. Dabei zeigen wir, dass die Art und Weise der Darstellung, das sogenannte *Parametermodel*, enormen Einfluss sowohl auf die Präzision als auch auf die Laufzeit der gesamten Analyse hat. Wir erklären die Schwächen des alten Parametermodels, das auf *Objektbäumen* basiert, und präsentieren unsere Verbesserungen in Form eines neuen Modells mit *Objektgraphen*. Durch das gezielte Zusammenfassen von redundanten Informationen können wir die Anzahl der berechneten Parameterknoten deutlich reduzieren und zudem beschleunigen, ohne dabei die Präzision des resultierenden Abhängigkeitsgraphen zu verschlechtern. Bereits bei kleineren Programmen im Bereich von wenigen tausend Codezeilen erreichen wir eine im Schnitt 8-fach bessere Laufzeit — während die Präzision des Ergebnisses in der Regel verbessert wird. Bei größeren Programmen ist der Unterschied sogar noch deutlicher, was dazu führt, dass einige unserer Testfälle und alle von uns getesteten Programme ab einer Größe von 20000 Codezeilen nur noch mit Objektgraphen berechenbar sind. Dank dieser Verbesserungen kann Joana mit erhöhter Präzision und bei wesentlich größeren Programmen eingesetzt werden.

Im zweiten Teil befassen wir uns mit dem Problem, dass bisherige, auf Abhängigkeitsgraphen basierende Sicherheitsanalysen nur vollständige Programme analysieren konnten. So war es z.B. unmöglich, Bibliothekscode ohne Kenntnis aller Verwendungsstellen zu betrachten oder vorzuverarbeiten. Wir entdeckten bei der bestehenden Analyse eine *Monotonie-Eigenschaft*, welche es uns erlaubt, Analyseergebnisse von Programmteilen auf beliebige Verwendungsstellen zu übertragen. So lassen sich zum einen Programmteile vorverarbeiten und zum anderen auch generelle Aussagen über die Sicherheitseigenschaften von Programmteilen treffen, ohne deren konkrete Verwendungsstellen zu kennen. Wir definieren die Monotonie-Eigenschaft im Detail und skizzieren einen Beweis für deren Korrektheit. Darauf aufbauend entwickeln wir eine Methode zur Vorverarbeitung von Programmteilen, die es uns ermöglicht, *modulare Abhängigkeitsgraphen* zu erstellen. Diese Graphen können zu einem späteren Zeitpunkt der jeweiligen Verwendungsstelle angepasst werden. Da die präzise Erstellung eines modularen Abhängigkeitsgraphen sehr aufwendig werden kann, entwickeln wir einen Algorithmus basierend auf sogenannten Zugriffspfaden, der die Skalierbarkeit verbessert. Zuletzt skizzieren wir einen Beweis, der zeigt, dass dieser Algorithmus tatsächlich immer eine konservative Approximation des modularen Graphen berechnet und deshalb die Ergebnisse darauf aufbauender Sicherheitsanalysen weiterhin gültig sind.

Im dritten Teil präsentieren wir einige erfolgreiche Anwendungen von Joana, die im Rahmen einer Kooperation mit Ralf Küsters von der

Universität Trier entstanden sind. Hier erklären wir zum einen, wie man unser Sicherheitswerkzeug Joana generell verwenden kann. Zum anderen zeigen wir, wie in Kombination mit weiteren Werkzeugen und Techniken kryptographische Sicherheit für ein Programm garantiert werden kann — eine Aufgabe, die bisher für auf Informationsfluss basierende Analysen nicht möglich war. In diesen Anwendungen wird insbesondere deutlich, wie die im Rahmen dieser Arbeit vereinfachte Bedienung die Verwendung von Joana erleichtert und unsere Verbesserungen der Präzision des Ergebnisses die erfolgreiche Analyse erst ermöglichen.

*In the one and only true way. The object-oriented version of 'Spaghetti code' is, of course, 'Lasagna code'. (Too many layers).*

Roberto Waltman

# 1 Introduction

This thesis presents improvements for algorithms mainly used in static security analyses. Most of this work is also relevant for other program analyses that are concerned with a conservative approximation of program semantics and method side-effects. However we are going to focus on the benefits our work provides for software security analyses — more specifically *information flow security*.

*Software security* is concerned about three main program properties: *Availability*, *confidentiality* and *integrity*.

**Availability** describes the property that a program should be able to process user requests in a reasonable amount of time, no matter how the requests are structured. A well known example of attacks to web-based applications are denial-of-service attacks, which aim to put so much load on the application, that subsequent requests time out and cannot be answered by the application.

**Confidentiality** is concerned with the treatment of secret information. Confidentiality is assured when an attacker cannot gain any meaningful information about secret values, such as passwords, by observing publicly visible channels, such as network communication.

**Integrity** is a concept dual to confidentiality. It concerns the treatment of publicly accessible information and input. An Attacker should not be able to alter sensitive parts of program behavior or data, through unauthorized inputs. Common attack techniques to integrity are buffer-overflow and string-format attacks, where insufficient input validation is exploited to execute arbitrary code.

## 1.1 Information flow control

*Information flow control* (IFC) tackles the latter two security properties that both address the treatment of information inside a program: Confidentiality and integrity.

IFC has been intensely studied for more than 25 years. Many advances were made in formalizing and proving the foundations of information flow that gave rise to specific attacker models and precise definitions of security properties. These foundations help to understand and reason about the security of a program and the inner semantics of information flow. Early work on IFC has been done by Denning and Denning [25, 26] and was later extended with an intuitive definition of standard *noninterference* by Goguen and Meseguer [32, 33].

### 1.1.1 Example: A program with illegal information flow

```
1  public class InformationFlowLeaks {
2
3    static int l, h;
4
5    public static void main(String[] argv) {
6      h = inputPIN();    // secret (HIGH)
7      if (h < 1234)
8        print(0);       // indirect leak
9      l = h;
10     print(l);         // direct leak
11   }
12
13 }
```

**Figure 1.1:** A small program with illegal information flow.

Figure 1.1 shows a small example of a program that violates confidentiality. We assume that method `inputPIN` returns secret information and method `print` produces publicly visible output. In this case the example contains two information leaks. A *direct leak* in l. 10 where the value of the secret pin is directly printed to public output and an *indirect leak* in l. 8 where the secret value decides if a certain output occurs.

In this example the program allows an attacker to infer information about the secret value through observation of its public visible behavior —the program output. In case of the indirect leak he knows that the value is < 1234 in case "0" is printed and >= 1234 otherwise. Hence he gained some information. In case of the direct leak the confidentiality violation is worse, as the actual secret is printed. However the severity of an information leak is not the main focus of information flow control. IFC aims to guarantee the absence of any kind of leak and hence to discover any leak, no matter how minuscule it may seem —a main difference to bug-finding tools. The decision on the severity of discovered leaks is left to the user. He may then choose to ignore or repair the leak or perform an additional analysis.

### 1.1.2 Attacker model

Our security analysis assumes an attacker model that is considered standard in the information flow community. This attacker has knowledge of the program source code. He can observe all input and output marked as public and can influence public input. He cannot observe or tamper with secret input. He is aware of history and can combine the observations of multiple runs to gain additional information. The attacker has no access to the actual hardware that executes the program, so he can't abuse physical side-channels. He is also not aware of time and timing related issues. Hence our analysis does not cover these type of information leaks based on timing of physical side-channels. It should be used in combination with other security tools specialized in detection of such side-channels to maximize security. As they are not part of our attacker model, we consider physical and timing side-channels out of scope for this work.

### 1.1.3 Noninterference and low-deterministic security

Noninterference is a central property for information flow control in the context of sequential programs. Intuitively, a program working on public and secret data is not allowed to change its publicly observable behavior when the secret data is changed. This way it is impossible for an attacker to infer information about the secret data by observing program behavior.

The standard noninterference of Goguen and Meseguer can be defined on the level of an abstract automaton that represents the state transitions of a program [111,44,115]. A program is considered noninterferent with respect to a security level $l$ if it obeys the following rules: Each statement is labeled with a security level of the information it may process or influence. The output of the program does not change if all statements with security level $l$ are deleted.

This definition has some drawbacks for a practical implementation of a static noninterference analysis: (1) It requires a security label for each statement and (2) removing arbitrary statements from a program may render it inexecutable, even if the removed statements do not influence the output. The noninterference criterion, called *low-deterministic security*, by Volpano and Smith [126] however is less general but more practical than Goguen's and Meseguer's. They assign a security level to each variable of a program. For the sake of simplicity we consider only the security labels *high* and *low*. For every two program states $s$, $s'$ and every program statement $c$ low-deterministic security holds, iff

$$s \cong_{\text{low}} s' \implies [\![c]\!]s \cong_{\text{low}} [\![c]\!]s'$$

Two program states are considered low-equivalent $\cong_{\text{low}}$ iff all variables marked as *low* are mapped to the same value in both states. So $s$ and $s'$ are only allowed to differ on the values of *high* variables. Thus, whenever two states are low-equivalent, a program statement $c$ is considered noninterferent, if the value of all *low* variables after its execution $[\![c]\!]$ is the same, no matter what the values of any *high* variables are. The approach to information flow control we present in this work has been formalized and proven to enforce the Volpano-Smith noninterference with the help of the theorem prover Isabelle [129, 128, 127].

### 1.1.4 Observational determinism for multithreaded programs

Low-deterministic security is a simple and intuitive definition, but it is not without flaws. It assumes input and output only happens at the beginning and end of a program run and fails to capture interactive systems. It also does not respect specific additional problems of security for multithreaded programs, such as execution order and timing. Therefore extensions such as *low-security observational determinism* (LSOD) [89, 109] have been proposed. LSOD requires a deterministic execution order for all statements that contribute to the program output, which is a very restrictive demand for a multithreaded program. Giffhorn [29,31] provides an excellent overview on the inherent challenges of security for multithreaded programs and evaluates different noninterference properties. He also developed and integrated a less restrictive version of LSOD —called *relaxed-LSOD* (RLSOD)— based on slicing into our information flow control tool Joana [30].

### 1.1.5 Declassification

In general, noninterference and low-deterministic security are too restrictive for many applications, as they forbid any kind of information leak. Sometimes we want to allow certain leakages: For example a password login system needs to act differently depending if a valid or invalid password has been entered. Encryption algorithms should be allowed to return the encrypted secret, but not the plain text —albeit from an information flow perspective, both contain the same information. So methods to declassify certain parts of secret information have been researched: General work on *declassification* [112,87] aims towards semantically justifiable critera for allowing and handling information leaks. It categorizes them into the four dimensions: what, who, where and when. What information is declassified, who is allowed to declassify, where is the information released in the system, and when is the information released? *Quantitative information flow* [23,21,88] measures the size of leaked information in bits which helps to decide on the severity of a leak and to further specify what parts need to be declassified.

### 1.1.6 Ideal functionality

An alternative approach to declassification is *ideal functionality* [75]. With the help of ideal functionality it suffices to verify noninterference in order to proof a program cryptographically secure. The absence of information flow in a variant of the program that contains idealized versions of encryption implies that the variant with real encryption can only leak encrypted information. Thus the program is *cryptographically noninterferent* even if the real implementation contains leaks. In Chapter 4 we show how to use our analysis tool Joana to successfully proof computational indistinguishablility with the help of ideal functionality.

## 1.2 Program slicing and information flow control

We focus on security analysis through *program slicing*. Slicing is a concept first discovered by Weiser [130,131]. It describes the way a programmer searches for the cause of a bug. Starting from a statement that did not yield the desired result —called the *slicing criterion*— he traces back though the program looking at statements that may have influenced the criterion, while ignoring irrelevant program parts. This notion of influence is closely related to information flow: Information may only flow to the slicing criterion through statements in the slice. Hence if the slicing criterion covers all publicly observable operations, the program is guaranteed to be noninterferent if no statement touching secret information is in the slice.

### 1.2.1 Program slicing with dependence graphs

Program slicing can be statically approximated through so-called dependency graphs. Ottenstein and Ottenstein were the first to introduce graph-based slicing [98]. Since then many improvements have been developed [27, 105, 108, 70, 44] and graph-based slicing is used for a wide array of applications such as: Debugging [61,120] and testing [13], measuring code complexity [130], model checking [51], or specialized analyses such as duplicate detection [68]. In this work we focus on dependence graphs used for information flow control.

### 1.2.2 Dependence graphs and information flow control

Dependence graphs capture the semantics of a program. Each statement is represented by a node and edges between statements occur if one statement may influence the other. Therefore they are well suited for information flow analysis [117, 1, 10, 133]. Wasserrab and Lohner even provided a machine-checked proof that noninterference —for sequential programs— can in fact be guaranteed by graph-based slicing [129, 128, 127].

Our group researched dependency graphs for information flow control extensively: Snelting invented path conditions for additional precision [117] which were first improved by Robschink [119] and later on by Lochbihler and Katz [65], Krinke [70] implemented VALSOFT —the predecessor of Joana— a program slicing analysis for C programs, Hammer [47,44] implemented the first prototype of Joana—our program slicing framework for Java programs— which included several optimizations for object-oriented languages and Giffhorn [29,31] later extended Joana for information flow control of multi-threaded programs.

We use an analysis approach inspired by Denning style security lattices [25]. Instead of only allowing two security levels *high* and *low*, an arbitrary lattice of security levels can be used. This enables a more fine-grained specification of program security properties and works well in combination with dependence graphs: Hammer et al. [44,47] proposed an approach for IFC with security lattices in dependence graphs, where only nodes corresponding to input and output statements need to be labeled. All other statements are labeled automatically through a sophisticated data flow analysis on the dependence graph. We label input statements with a *provided security level P* and output statements with a *required security label R*. Then a *monotone data flow analysis* [60,66] propagates the provided security levels along the dependence graph. Later on we only need to check if any provided security label at an output statement violates the expected required security label. The data flow transfer function $f_n(l)$ of the security levels at node $n$ is a standard *"kill gen"* transfer function:

$$f_n(l) = (l \setminus kill(n)) \sqcup gen(n)$$

with

$$gen(n) = \begin{cases} P(n) & \text{if } n \text{ is an input statement} \\ \bot & \text{else} \end{cases}$$

and

$$kill(n) = \top$$

The simplicity of the *gen* and *kill* functions shows how elegant graph based IFC analysis can be defined. The *gen* function returns the provided security level for annotated input statements and otherwise returns the neutral element ($\bot$) of the $\sqcup$ operator. The *kill* function only returns the neutral element ($\top$) of the $\sqcap$ operator —hence it can be ignored.

In our data flow framework every node $n$ has an incoming security level $in(n)$ and an outgoing level $out(n)$. The incoming level is computed by the outgoing levels of the predecessors.

$$in(n) = \bigsqcup_{n' \in preds(n)} out(n')$$

The outgoing level is computed from the incoming level and the node's transfer function.

$$out(n) = f_n(in(n)) = (in(n) \setminus kill(n)) \sqcup gen(n) = in(n) \sqcup gen(n)$$

The fixed-point of the outgoing level of $n$ is also called the *actual security level* $S(n)$ of $n$.

The program is considered secure iff for any output statement $n$ marked with an required security level $R(n)$ it holds that

$$S(n) \sqsubseteq R(n)$$

In the remainder of this work we are going to restrict our examples to the simple standard security lattice $low(public) \leq high(secret)$. However this is not a restriction of the presented algorithms, all of them operate independently of the chosen lattice.

### 1.2.3 Slicing object-oriented languages

This work heavily focuses on the analysis of statically typed object-oriented languages. These languages provide some benefits —such as static types— as well as several drawbacks —such as dynamic dispatch and side-effects— for a static analysis. Several publications [135,108] already tackled the basics of dependency graph computation for object-oriented languages. We base our work on the results of Hammer [47,44,46]: He proposed a special *parameter model* to handle method side-effects, which we are going to improve and extend [35,36].

Points-to analysis is also an important topic in the object-oriented setting. We are going to provide a overview of pointer-analysis options and their influence on dependence graph computation in section §2.5.2 — upcoming section §2.1 starts with a detailed discussion of the typical problems that occur when analyzing object-oriented languages.

### 1.2.4 Slicing multithreaded programs

Slicing of dependence graphs can also be used to analyze the security of multi-threaded programs. Krinke [69] was the first to introduce the concept of *interference dependence* between threads, which was later on used and extended by Nanda [95] and Giffhorn [29]. While our work provides the basis for multi-threaded dependence graphs, we do not include the details of the required *may-happen-in-parallel (MHP)* analysis [97,5,69,29] and special time-sensitive slicing techniques [95–97,31,29] in this thesis. However, we were able to further improve upon existing algorithms and incorporate additional *lock-sensitivity* into our analysis as a result of the cooperation with the research group of Prof. Müller-Olm [38].

We do not go into implementation details regarding information flow control for multi-threaded programs with dependence graphs, however sections §2.1.9 and §2.2.5 contain a brief summary of the typical problems and our proposed solutions.

## 1.3 Contributions

The main contribution of this work is the Joana information flow control framework [40,41,37,39,118] available for download under `joana.ipd.kit.edu`. Joana is open-source and comes with several GUIs and an API for developers to include Joana into their own analyses. The framework has been developed over several years by the program analysis group of Prof. Gregor Snelting. Previously and currently contributing members are Christian Hammer, Dennis Giffhorn, Martin Mohr, Martin Hecker and Jürgen Graf. Joana builds upon the general program analysis framework WALA (`wala.sf.net`) —developed by IBM— and consists of a total of 450kLoC: 220kLoC of specialized dependence graph and information flow algorithms and 230kLoC of program analysis code from WALA. During this work we developed several extensions and improvements to the WALA framework, such as support for Android Dalvik bytecode, exception analysis and many performance and bug fixes.

Currently Joana allows a wide range of analysis options and supports information flow control for Java and Android programs. It can handle full Java and Dalvik bytecode —with the exception of some reflection constructs. Joana models the effects of multi-threaded execution and exception handling. Our approach aims at a conservative analysis result that can guarantee the absence of illegal information flow. Compared to common widely used bug detection tools like HP Fortify[1], IBM AppScan[2] or FindBugs[3] —that do not provide a guarantee— our analysis algorithms are in general more heavy-weight and include many precision enhancements like object-, flow-, field- and context-sensitivity in order to minimize false alarms. Still Joana scales for programs up to 100kLoC — previous to this work the early prototype has only been able to analyze programs up to 20kLoC. We suggest using Joana for the security critical kernels of a software system and to apply standard bug detection tools additionally to the whole system.

So far Joana has been and is successfully used in several collaborations:

- *Implementation-level analysis of e-voting systems* (Ralf Küsters, University of Trier): Analysis of e-voting systems and IFC treatment of encrypted messages [75,76,73,72,71,74].

- *Program-level specification and deductive verification of security properties - DeduSec* (Bernhard Beckert, KIT): Integration of KeY and Joana [74].

- *System-wide data-driven runtime usage control across layers of abstraction - SADAN* (Alexander Pretschner, TU Munich): Improvement of run-time usage control with static Joana information [85].

- *Static code analysis for securing Cordova applications* (Achim Brucker, SAP).

- *Secure type systems and deduction - SecDed* (Tobias Nipkow, TU Munich) provides Isabelle support for the machine-checked verification of Joana's IFC machinery, for proofs such as [128].

- *Cyber Security Lab* (Christian Hammer, Saarland University / CISPA) general cooperation on the use of the Android front-end for program analysis using WALA.

[1] `http://www8.hp.com/us/en/software-solutions/application-security/`
[2] `http://www.ibm.com/software/products/en/appscan`
[3] `http://findbugs.sourceforge.net/`

- *Developing systems with secure information flow - IFlow* (Wolfgang Reif, University of Augsburg) uses Joana as IFC tool [64, 63, 123].

In this thesis we will focus on the following technical contributions to Joana. In Chapter 2 we discuss the general improvements achieved in the field of static analysis with dependence graphs, including

- A *null-pointer detection analysis* for a more precise static approximation of control flow in presence of exception handling (§2.4.1).
- An interprocedural extension to *termination-sensitive control dependencies* enabling termination-sensitive IFC analysis (§2.4.2).
- A fine-grained model of field access operations in dependence graphs providing further precision improvements (§2.4.2).
- A new model for interprocedural side-effects in dependence graphs [36] (§2.6). The new parameter model includes:
  - *Object-graphs* —an extension of object-trees from Hammer— that improve memory footprint and runtime of dependence graph computation. They also tackle the scalability problem of object-trees for less precise points-to information by merging duplicate information in subtrees into a single representation (§2.6.3).
  - An optimization of the interprocedural propagation for object-graphs and -trees that replaces the mutually recursive algorithm of object-trees with 3 non-alternating phases (§2.6.4).
  - An evaluation of the influence of parameter model and points-to analysis on runtime and precision of dependence graph computation (§2.6.5).

In Chapter 3 we present our new approach to modular information flow and dependence graph computation for components in unknown context.

- We define an partial order on the context configurations of a component that implies a *monotonicity property* for its analysis results. This property allows a safe prediction of the components information flow properties in a given context without the need to run a context specific analysis (§3.2.2).

- An inference algorithm for *relevant context conditions* that describe which modifications to the context can influence the analysis result (§3.2.3).
- A new language *FlowLess* to specify required context conditions and expected information flow for a component (§3.3).
- We introduce the concept of *conditional data dependencies* that allow precomputation of dependence graphs for components in any context (§3.4.2).
- An algorithm based on *access paths* to approximate conditional data dependencies (§3.4.3).
- An extension to the summary edge computation algorithm that copes well with conditional dependencies (§3.4.4).

Finally Chapter 4 contains several case studies that present applications of Joana.

## 1.4 Related work

This work is about static program analysis with dependence graphs and its applications to information flow control. It is —to our best knowledge— unique in its approach to provide a fully automated conservative information flow analysis for Java that can guarantee the absence of any illegal flow. However there are several tools and publications that can be considered closely related to Joana. These tools differ in the amount of supported language features, required user interaction and soundness.

Similarly to Joana, the Indus [103] tool supports concurrent Java and utilizes several auxiliary analyses to compute dependence graphs. These are used for slicing in order to reduce the state space in model checking applications. Unlike Joana, no explicit support for IFC is provided.

Tools like Jif [93,94] extend Java with *security types*. The user has to annotate variables, fields and method signatures with additional labels that restrict allowed information flow. Then Jif checks the validity of these annotations in order to detect illegal flow. Since Jif supports security type inference only for local variables, the user is usually required to annotate the whole program with security types. This leads to a huge number of required annotations as it does not suffice —in contrast to our approach— to only mark program points where information is read in or written out. Additionally Jif does not support Java features such as concurrency, therefore we consider Jif or approaches based on Jif [22] to be less practical, especially when existing code bases need to be analyzed. Better type inference algorithms for these approaches have been proposed [116], but as of yet, we are not aware of any practical implementation for full Java.

Erik Bodden and his group developed multiple tools for dependence and information flow analysis based on the SOOT program analysis toolkit[4]. In general his work is less focused on a strictly conservative result and aims more towards practical and scalable solutions. In [15] they present an IFC tool tailored for software production lines that applies an IFDS/IDE analysis. It is able to cope with conditionally compiled code efficiently and includes a nice GUI. However this tool can only detect a very specific kind of information flow, namely direct leaks through data flow in variables. At this time it cannot detect indirect flow through branches, data flow through heap allocated objects or any kind of concurrency related leaks. Another work by Bodden et. al. is Tamiflex [16] — a tool that helps to analyze the behavior of Java programs that use reflection. Reflection is a huge threat to any static analysis, as it allows dynamic loading of classes and almost arbitrary execution of code. Tamiflex observes the program behavior during multiple executions and keeps track of any reflection calls. Then it transforms the program into a variant better suited for static analyses by replacing reflection with the observed behavior. This approach is obviously not sound, however it is definitely useful for other analyses like bug detection tools.

Mostly commercial security scanners like AppScan[5] fall into the unsound yet practical bug detection category. Their goal is to analyze large applications in a short amount of time whilst detecting most of the security vulnerabilities and pointing out error prone coding practices. Tripp et al. [124] extend AppScan with a taint analysis for JavaScript. Their approach is based on the WALA framework and applies *hybrid thin-slicing*. Their tool scales well and can detect many security leaks

[4]`http://www.sable.mcgill.ca/soot/`

[5]`http://www.ibm.com/software/awdtools/appscan/developer`

in almost arbitrarily large programs with only few false alarms. Due to thin-slicing they miss indirect information leaks and thus cannot guarantee noninterference.

Another JavaScript analysis is from Guarnieri et al. [43]. They apply a demand-driven taint analysis based on access paths to detect direct and indirect information leaks. Their approach scales well and seems useful for many real world applications, but it cannot detect any probabilistic leaks and fails to handle the effects of the `eval` function — a JavaScript speciality that allows the execution of dynamically constructed code.

Seth et al. [59] suggest a solution for the sound approximation of the `eval` function through a combination of static and dynamic analysis.

The approach by Genaim and Spoto [28] uses abstract interpretation of Java bytecode to detect illegal information flow. It captures direct as well as indirect flow, but lacks support for concurrency and precision features such as object sensitivity.

The KeY [2] tool allows the user to manually specify and verify arbitrary semantic properties of sequential Java programs and use them for information flow control [7]. This generally requires a considerable amount of manually provided JML [19] annotations in the program's source code. The benefit of this approach is a very precise analysis result. In §4.3 and §4.4 we propose to use Key in combination with Joana in order to get the best of both worlds when analyzing a program for illegal information flow: A fast and automatic analysis for large parts of the program with Joana and a very precise manual verification of the remaining parts with Key.

Related work specific to the modular approach on dependence graph computation is presented at the start of Chapter 3.

*People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.*

Donald Knuth

# 2 Information Flow Control with System Dependence Graphs for Object-oriented Languages

## 2.1 Analyzing object-oriented languages: Challenges and opportunities

The step from static analysis that can guarantee noninterference for a toy-like language to an analysis for a real world object-oriented language like Java raises many different problems in terms of precision and scalability. In order to retrieve practical usable results we [36, 35], together with previous work in this area [70, 44, 105], developed and integrated many optimizations for an object-oriented setting.

Our analysis and optimizations are specifically tailored to the Java language, but most parts also apply to any other object-oriented setting. We are going to explain the most challenging parts in implementing a sound yet precise and scalable analysis for Java in the following subsections. We use short source code examples to illustrate the common problems in an information flow scenario. Variables named `<SECRET>` are considered to hold secret information that should not be leaked via a `print` statement. `<PUBLIC>` is a placeholder for any data that is public and may be leaked. Finally `<INPUT>` represents public user input, that can be provided by the attacker. So it should not be possible to deduce information about secret information through altering the input.

```java
class A {
  public void foo() {}

  public int bar() {
    return S.pub;
  }

  public final void print() {
    foo();
    println(bar());
  }
}

class S {
  public static int pub = <PUBLIC>;
  public static int sec = <SECRET>;
}
class B extends A {
  public void foo() {
    S.pub = S.sec;
  }
}

class Main {
  public static void main() {
    A a = new A();
    B b = new B();
    // ok
    a.print();
    // illegal
    b.print();
    a.print();
  }
}
```

**Figure 2.1:** A program with a security leak due a side-effect through static fields.

### 2.1.1 Static variables

The main problem of analyzing Java with static variables is their unlimited scope. They may be referenced from anywhere in the program and can result in potentially arbitrary side-effects between otherwise separated program parts. The example in Figure 2.1 shows how such a side-effect may lead to illegal information flow.

The example contains two classes `A` and `B`, where `B` extends `A` and introduces a side-effect through static fields in method `foo`. Calling method `print` on an instance of class `A` is safe only as long as the same method has not been called on an instance of class `B`. The source code of class `A` alone does not suffice to detect this illegal flow. So a change in another class leads to illegal flow in `A`. It is hard to predict these kind of side-effects only through looking at parts of a program. There is for example no way to detect the side-effect between the multiple calls to `print` in the `main` method. Without knowing about the static variables they appear to be completely independent.

Almost any Java program uses static variables, if not directly then at least through code from the runtime library. This library contains a huge number of classes and side-effects through static fields on itself. Just keep in mind that e.g. `System.out` is a static field. An analysis needs to keep track of all static fields in order to guarantee the detection of all possible illegal flow. In combination with the huge runtime library this leads to serious scalability issues, as we are going to show in the next subsection.

### 2.1.2 Runtime libraries

We have shown in the previous Section §2.1.1 that —among other reasons— side-effects through static fields make it unfeasible to decide on the noninterference of a single method or program part, without looking at the whole program. In Java this includes also the quite large runtime library. E.g. the runtime library of Java 7 contains about 26970 class and interface declarations[6] with approximately 980kLoC. It is bigger than most programs using it. Without any optimizations[7] in place a static analysis of the very basic "Hello World" program finds 22885 classes and 123154 methods potentially involved. To reduce these large numbers we are going to focus our analysis on the much smaller runtime libraries of Java 1.4 or Java ME[8].

Even with these smaller libraries a static analysis of a medium sized program with about 10kLoC has to analyze in average a total of 60kLoC, including the library code used. However the amount of library code varies greatly from program to program. So some 10kLoC program may blow up to a total number of 100kLoC lines, while others stay at 15kLoC. This is also the reason why lines of code as a measurement for program size can be deceiving.

Another problem of runtime libraries is that they contain calls to native code. So for some parts there is no actual Java code that can be analyzed. We use manually crafted stubs in these cases to approximate the expected behavior instead of just ignoring those calls. However there is no guarantee that these stubs capture every effect on the program. This is a limitation of our approach, but to our knowledge no other static analysis for Java includes native code. Most of them ignore native calls or even build stubs on the level of library calls, so they don't even include any library code.

[6]Counting actual class files in the runtime jar archives of Java 1.7.0_04-b21 for Mac OSX

[7]We used a simple rapid type analysis, as more sophisticated approaches did not finish within a time frame of 15 minutes.

[8]Java environment for mobile devices

### 2.1.3 Static initialization

Static initialization poses two main problems: Firstly due to the large number of classes in the runtime library, many of them exist and may be potentially executed and secondly there is a way to exploit the order in which initializations take place in order to achieve illegal information flow. The first issue is mainly a scalability problem, which we tackle more in general with our performance improvements in §2.5. In the following example we are going to focus on the subtle difficulties of the order of static initialization in Java.

In order to improve performance, not all static initializers of a Java program are called at program start. They are called in a lazy fashion the first time they are needed. The Java language specification [34][9] defines the following rules for the invocation of static initializers: As long as class *C* has not already been initialized, all initializers of *C* are called, if

- an instance of *C* is created.
- a static method of *C* is invoked.
- a static field of *C* is accessed.
- an `assert` statement lexically nested within *C* is executed.

So basically the execution of a static initializer can be triggered by the runtime behavior of the program, which may depend on secret information and lead to an information leak.

In the example in Figure 2.2 the values of the static variables `B.i1` and `C.i2` depend on the order of initialization. If class `B` is initialized before `C` variable `B.i1` contains 1 and 2 otherwise. The same holds for variable `C.i2`. The main method creates either an instance of class `B` or `C` depending on the user input and the secret value. Even tough those instances are never used, their creation triggers the corresponding static initializers. Through observing the output of both `println` statements an attacker can decide if the user input was equal to the secret value. Thus an information leak occurred.

In order to detect these kind of order dependent information leaks, the process of static initialization needs to be carefully modelled. Currently our analysis does not contain an adequate model —we assume that all

[9]Section 12.4.1, page 316ff

```
class A{
  public static int val = 1;
}

class B {
  public static int i1 = A.val++;
}

class C {
  public static int i2 = A.val++;
}
```

```
class Main {
  public static void main() {
    if (<INPUT> == <SECRET>) {
      // trigger initializer of A and C
      new C();
    } else {
      // trigger initializer of B and C
      new B();
    }

    // illegal
    println(B.i1); // 1 iff input != sec
    println(C.i2); // 1 iff input == sec
  }
}
```

**Figure 2.2:** A program with a security leak due to the order of static initialization.

initializers are called directly upon program start— and cannot detect these leaks. However this is not a general problem, as such a model can be integrated straightforwardly by treating each potential initialization point as a call to the matching initialization methods. To our best knowledge no other static information flow analysis can detect these kind of leaks, which leaves this issue open for future work.

## 2.1.4 Reflection

Reflection is a mechanism in Java that can be used to circumvent almost any access restrictions and scope limitations. So in general it is a very hard problem for a static analysis to decide on the possible effects of an reflection statement. Some analyses [83, 113] try to combine type information with advanced constant propagation and string analysis for the text arguments given to reflection calls. In some special cases they are able to detect the effects of a reflection statement, but this approach does not work reliably in general, as the arguments may not be constant at all.

The example in Figure 2.3 demonstrates how a program can use reflection to read a private member variable and also to alter the value of a final member. It reads val without regard to the private access restrictions through a generic field object av. These generic fields can change the access restrictions of the member they represent, e.g. make a private field public or even otherwise, and they have full access to the

```
1  class A{
2    private int val;
3    public final int pub;
4
5    public A(int val) {
6      this.val = val;
7      this.pub = <PUBLIC>;
8    }
9  }
```

```
10 class Main {
11   public static void main() {
12     A a = new A(<SECRET>);
13     Class c = A.class;
14     Field av = c.getDeclaredField("val");
15     av.setAccessible(true);
16     println(av.getInt(a)); // illegal
17     println(a.pub);        // ok
18     Field ap = c.getDeclaredField("pub");
19     ap.setAccessible(true);
20     ap.setInt(a, av.getInt(a));
21     println(a.pub);        // illegal
22   }
23 }
```

**Figure 2.3:** A program with a security leak due to reflection.

value of the member. The program creates another generic field at l. 18 and subsequently uses it to set the value of the final member `pub` to the value of the private member `val`.

This short example demonstrates how reflection can be used to overcome certain access restrictions on class members. In addition it may also be used to load classes at runtime and call arbitrary methods. We, as any other IFC analysis known to us, do not support analysis of programs with reflection. We argue that a program that needs to ensure its security should not rely on mechanisms that render the inherent access and type restrictions of a language useless.

### 2.1.5 Types and object-fields

Variables cannot only hold values of different types, they can also refer to objects. Objects themselves may contain an arbitrary number of member variables, that again may hold values or refer to other objects. So from a single variable that refers to an object a whole set of transitively reachable member variables emerges. When we have to decide whether a variable may contain secret information, it does not suffice to look at its value, we also need to check the values of all reachable members. The following example illustrates this problem.

The program in Figure 2.4 contains two node classes `NodeA` and `NodeB` that both have a member `val` to store a number value and a reference `next` pointing to another node object. The member `next` of class `NodeA`

```
 1 class NodeA {
 2   public NodeB next;
 3   public int val;
 4
 5   public NodeA(int val) {
 6     this.val = val;
 7   }
 8 }
 9 class NodeB {
10   public NodeA next;
11   public int val;
12
13   public NodeB(int val) {
14     this.val = val;
15   }
16 }
17 class Main {
18   public static void main() {
19     NodeA a = new NodeA(<SECRET>);
20     NodeB b = new NodeB(<PUBLIC>);
21     b.next = a;
22     a.next = b;
23
24     // ok
25     println(b.val);
26     // illegal
27     println(a.val);
28     println(b.next.val);
29     // ok
30     println(a.next.val);
31   }
32 }
```

**Figure 2.4:** A program with two security leaks due to secret information stored in object fields.

refers to a node of type NodeB and vice versa. The main method creates an instance of NodeA to store secret information and an instance of NodeB to store public information. At the end of l. 22 the next members of both nodes point respectively to the other node. So the secret value in a.val can also be accessed through variable b. Even as node b never holds secret information itself.

A simple IFC analysis would consider that all four println statement may leak secret information, as they print values from variables a and b that both can potentially reach the secret value stored in a.val. Our analysis distinguishes the different fields a variable can access and is able to show that only the second and the third println statement may leak secret information. We are going to present this feature called *field-sensitivity* in more detail in §2.2.3 and we discuss our approach that leads to a field-sensitive analysis in §2.6.

### 2.1.6 Aliasing

*Aliasing* describes the situation that two different variables or fields may refer to the same object. A foul effect of aliasing is that any modifications to a variable does also change all aliased variables. So statements can have effects on variables they do not directly refer to. These effects are a common source for programming failures as they are easily overlooked

```
1  class A {
2    public B b;
3
4    public A(B b) {
5      this.b = b;
6    }
7  }
8
9  class B {
10   public int val;
11
12   public B(int val) {
13     this.val = val;
14   }
15 }
16 class Main {
17   public static void main() {
18     A a1 = new A(new B(<PUBLIC>));
19     A a2 = new A(new B(<PUBLIC>));
20     println(a1.b.val); // ok
21     println(a2.b.val); // ok
22
23     a2.b.val = <SECRET>;
24     println(a1.b.val); // ok
25     println(a2.b.val); // illegal
26
27     a2.b = a1.b;
28     println(a1.b.val); // ok
29     println(a2.b.val); // ok
30
31     a2.b.val = <SECRET>;
32     println(a1.b.val); // illegal
33     println(a2.b.val); // illegal
34   }
35 }
```

**Figure 2.5:** A program with a security leak due to aliasing.

and often unwanted. The following example shows how aliasing can influence the semantics of seemingly and lexically similar statements.

The `main` method in Figure 2.5 initially creates two instances of class `A` in the variables `a1` and `a2`. At the first two `println` statements in ll. 20f both variables are not aliasing and only contain public information. So these statements do not leak secret information. In the third and fourth `println` statements in l. 24 a member reachable from `a2` contains secret information, so the fourth `println` leaks information. After l. 27 `a1` and `a2` are aliasing, the member `b` of both variables now refers to the instance created for `a1`. Thus the following two `println` statements do not leak information. At last in l. 31 a member of `a2` is set to a secret value. But now also the value of the member of `a1` changes and the following last two `println` statements both leak secret information. So even as the statements in l. 23-l. 25 are syntactically equal to the statements in l. 31-l. 33 their effect differs and additional leaks occur due to aliasing.

Aliasing is one of the hardest problems to deal with in a static analysis, as it is in general undecidable [102]. *Points-to* and *alias analysis* try to tackle this problem. They preserve conservative results through overapproximation. But too coarse-grained approximation can render the result practically useless, while more fine-grained approaches suffer from huge impact on runtime and memory consumption. Much research

```
class A {
  public boolean foo() {
    return false;
  }
}

class B extends A {
  public boolean foo() {
    return true;
  }
}
```

```
class Main {
  public static void main() {
    A a = new A();
    if (<INPUT> == <SECRET>) {
      a = new B();
    }
    if (a.foo()) {
      // illegal
      println("input␣==␣secret");
    } else {
      // illegal
      println("input␣!=␣secret");
    }
  }
}
```

**Figure 2.6:** A program with a security leak due to dynamic dispatch.

has been done in this area. We provide an overview of the most important results in §2.5.2 and show how these results integrate into our IFC analysis in §2.6.

### 2.1.7 Dynamic dispatch

*Dynamic dispatch* is a key feature in every object-oriented language. It allows objects to act depending on their dynamic type rather than their static type, by resolving method calls at runtime. Together with inheritance this feature enables classes to reuse properties and code from a so-called superclass. Often only a small portion of code needs to be exchanged while the rest of the functionality can be inherited and does not need to be rewritten. For example a sorting algorithm that sorts elements in alphabetical order only differs from an algorithm that sorts numerically in the way elements are compared. So both sorters can inherit the main functionality from a sorting algorithm superclass and only need to replace the compare method.

In the context of information flow, dynamic dispatch may be exploited to reveal information about the program state. The example in Figure 2.6 shows how an illegal flow of secret information may occur due to dynamic dispatch.

The program contains two classes `A` and `B`, where `B` is a subclass of `A` that overrides its method `foo`. The main method holds a variable `a` of static type `A` that has —depending on the user input— either the dynamic type `A` or `B`. So when method `foo` is called in l. 18 the dynamic type of `a` decides which implementation is executed and thus what the return value of the call is. If the return value is true, we can infer that the implementation of class `B` has been called. Hence the dynamic type is `B` and therefore the user input was equal to the secret value.

We have shown how dynamic dispatch can lead to subtle illegal information flow. While it greatly enhances the flexibility and usefulness of a language, it comes with a steep price for static analyses. The actual code executed by a dynamically dispatched call depends on the state of the program at runtime. A static analysis in general can not decide what is going to be executed and again needs to use conservative approximations of the effects of the call, by assuming that every potentially valid method may have been called. With the help of type and points-to information it is possible to narrow the set of potentially valid methods down to a more reasonable size, but it remains a big obstacle. Yet it does not suffice to detect which methods may be called. As the previous example has shown it is also important to keep track of the reasons why a certain method is called. Our analysis can detect illegal flow through dynamic dispatch. In §2.5.3 we explain how our model for method calls respects these implicit dependencies through dynamic calls.

### 2.1.8 Exceptions

Java and many other modern object-oriented programming languages support the use of *exceptions*. They streamline and simplify error handling by separating error handling code from the core program code. Whenever an error occurs during a program run, normal execution flow is interrupted and an exception is raised. Exceptions regularly occur if a null pointer is dereferenced or an array index is out of bounds. The exception may be caught by a so called *exception handler* or lead to program termination. The programmer can declare an exception handler through a **`try`**-**`catch`** block. He can specify which types of exceptions should be caught and then declare appropriate countermeasures for the current error in the **`catch`** block. Every exception raised from a statement inside a **`try`** block can be caught —including one from a transitively called method. This mechanism helps to separate the error handling from the rest of the code and put it in a single place, but it may also be misused as an inter-procedural goto instruction and can lead to unexpected behavior.

```
 1 class A {
 2   public static A create(int i) {
 3     if (i != <SECRET>)
 4       return new A();
 5
 6     return null;
 7   }
 8
 9   public void foo() {}
10 }

11 class Main {
12   public static void main() {
13     A a = A.create(<INPUT>);
14
15     try {
16       a.foo();
17       // illegal
18       println("input␣!=␣secret");
19     } catch (NullPointerException exc) {
20       // illegal
21       println("input␣==␣secret");
22     }
23   }
24 }
```

**Figure 2.7:** A program with a security leak due to exceptions.

Exceptions pose a big problem in the scope of information flow, because they may occur at almost any statement and potentially influence any statement that is executed afterwards. Every exception throwing statement can be seen as a conditional goto instruction, that is executed whenever the conditions for an exception are met. Thus it decides if execution flows to the next statement or to an exception handler somewhere else in the program. So the execution of a statement depends on the exception conditions of all its predecessors. These conditions are easily overlooked by the programmer as they are only implicit and not part of the source code. The example in Figure 2.7 shows how an unexpected leak of secret information may occur due to an exception.

The program contains a method `create` that returns either a new instance of class `A` or **`null`** depending on the secret value and input parameter `i`. The `main` method calls `create` with the user input as parameter. Whether variable `a` contains a **`null`** reference depends on the user input and the secret value. This leads to an information leak in l. 16. Even as the value of `a` is never read or printed, it decides if a `NullPointerException` is thrown upon the call to `foo` and either l. 18 or l. 21 executes.

Our analysis fully supports the detection of these kind of leaks. We discuss exception analysis in §2.4.1 and show that a naive approach leads to many false alarms. Therefore we introduce a way to reduce these false alarms by detecting impossible exceptions: For example we analyze if a referenced value may never be null.

```
1  class A extends Thread {
2
3    public int f1;
4    public int f2;
5
6    public A(int f1) {
7      this.f1 = f1;
8    }
9
10   public void run() {
11     print(f1);     // possibilistic leak
12     f2 = 23;
13   }
14 }
```

```
15 class Main {
16   public static void main() {
17     A a = new A(<PUBLIC>);
18     a.start();
19
20     a.f1 = <SECRET>;
21     for (int i = 0; i < <SECRET>; i++) {
22       // skip
23     }
24     a.f2 = 42;
25
26     print(a.f2);  // probabiblistic leak
27   }
28 }
```

**Figure 2.8:** A multithreaded program with a possibilistic and a probabilistic security leak.

### 2.1.9 Threads

In the age of multi-core processors, concurrent programs become more and more common. So does the support for concurrent programming in modern languages. There are multiple approaches to support concurrent programming, like threads communicating through shared memory or distributed systems with message passing. We are going to focus on the concurrency model of most commonly used languages like Java or C#: Threads with shared memory. As most of the advanced analysis algorithms for concurrent programs —especially in the context of information flow— are out of scope for this work, we would like to point to the excellent dissertation of Dennis Giffhorn [29] for further details.

We use this subsection to provide a brief overview of the additional challenges in information flow security for concurrent programs. Concurrency further complicates the detection of information leaks. In addition to normal sequential leaks the effects of synchronization, timing and interference may also lead to illegal flow. So called *possibilistic* and *probabilistic* channels can occur. We explain the difference between these two channels in a short example.

The program in Figure 2.8 contains two threads: The main thread starting a the `main` method and one instance of the thread from class `A` started form the main thread at l. 18. Both threads communicate through the publicly visible fields `f1` and `f2` of class `A`. This program contains two security leaks — a possibilistic leak in l. 11 and a probabilistic leak in l. 26.

We inspect the possibilistic leak first. A possiblistic leak is a statement that may or may not leak secret information, depending on the interleaving between threads. In this case l. 11 prints the value of `f1`. This field is initialized with public information in l. 17 and later in l. 20 replaced with secret information. Thus the `run` method of thread `A` prints secret information if the main thread has already executed l. 20 before thread `A` reaches l. 11.

The probabilistic leak is even more delicate, as it is easily overlooked. These leaks occur when the probability of a certain output depends on a secret value. An attacker can run the program multiple times, observe the distribution of outputs and use these data to infer information about the secret value. l. 11 contains such a leak. At first glance it prints the value of `f2`, which never contains secret information. However whether the value of `f2` is `23` or `42` depends on the order in which the statements in l. 12 and l. 24 have been executed. In this example the probability of the execution order is influenced by the secret value, through the loop in l. 21. The secret value determines the number of loop iterations. The larger the secret value is, the more statements are executed before l. 24 and thus the more likely it is, that an interleaving occurs where l. 24 is executed after l. 12. This situation can be exploited by an attacker to gain some information about the secret value. Even if he cannot infer the actual value, he may compute the probability that the secret value is in a certain range.

The problem of a probabilistic leak may seem more like a theoretical problem than a security issue that is exploited in a real world scenario. But this is far from true. These kinds of leaks have already been successfully used to break encryption algorithms [67] previously considered to be secure. Also, depending on the scheduler, they may even leak actual values: If the scheduler uses a deterministic round-robin approach, probabilistic leaks act similar to indirect information flow in conditional branching [29].

Joana can deal with concurrency and is able to detect possibilistic as well as probabilistic leaks. Giffhorn [29] has shown that it can guarantee a security property called *low-security observational determinism* (LSOD) [134] — a well known security property for concurrent programs. His work also contains a substantial evaluation of the underlying algorithms

```
class A {
  public static void doPrint(int i) {
    println(i);
  }
}
```

```
class Main {
  public static void main() {
    A.doPrint(<PUBLIC>);  // ok
    A.doPrint(<SECRET>);  // illegal
  }
}
```

**Figure 2.9:** A program with a spurious security leak for context-insensitive analyses.

on real world applications and provides an extension called relaxed-LSOD (RLSOD) that is less strict yet can still guarantee security of a program. We were able to improve the precision of the analysis by Giffhorn with the help of *dynamic pushdown networks* (DPN) [38] and also to further improve the RLSOD criterion [18].

## 2.2 Precision in static analyses

In the previous section we introduced the challenges of analyzing a real world object-oriented language like Java in the context of information flow control. In this section we focus on the different methods used to tackle these problems in a static analysis. The main problem of a static information flow analysis is to maintain sound results —detect all possible security leaks— while minimizing the amount of false alarms, i.e. by improving analysis precision.

In this section we discuss important properties for static analyses that help to improve precision and are implemented in our tool.

### 2.2.1 Context-sensitive

A *context-sensitive* analysis is able to distinguish between different contexts in which certain parts of the program are executed. In most cases context-sensitivity is used as a synonym for distinguishing the effects of the execution of a method by the site (*call site*) it has been called from. The following example shows that methods may have different effects depending on the context they are called from.

In the example in Figure 2.9 the method `doPrint` is called twice. The first call from the `main` method prints public information and the second

```
 1 class A {
 2   private int i;
 3
 4   public A(int i) {
 5     this.i = i;
 6   }
 7
 8   public void doPrint() {
 9     println(this.i);
10   }
11 }
```

```
12 class Main {
13   public static void main() {
14     A a1 = new A(<PUBLIC>);
15     A a2 = new A(<SECRET>);
16
17     // ok
18     a1.doPrint();
19     // illegal
20     a2.doPrint();
21   }
22 }
```

**Figure 2.10:** A program with a spurious security leak for object-insensitive analyses.

call prints secret information. A context-insensitive analysis would not distinguish between these two different calling contexts and thus assume that doPrint always leaks secret information. A context-sensitive analysis however is able to distinguish the two call sites and detect that only the second call can leak secret information.

These kind of situations —where methods are reused to operate on different data and in different contexts— naturally appear very often. Hence it is crucial for an analysis to support context-sensitivity, as otherwise many false alarms would occur.

## 2.2.2 Object-sensitive

*Object-sensitivity* in general describes to ability to distinguish actions on different object instances of the same type. It also acts as a special form of context-sensitivity for dynamic method calls in an object-oriented setting: An object-sensitive analysis distinguishes dynamic calls by the receiver object the method is called upon. Often times member methods access the attributes of their respective object instance and do not interfere with other instances of the same type. Hence this distinction proves to be quite effective. E.g. it works well with standard implementations of getter and setter methods as these only modify or retrieve the attribute values of the instance they are called upon.

The example in Figure 2.10 shows how object-sensitivity can help to reduce false alarms. The main method creates two instances of the same class A in l. 14 and l. 15. The first instance is initialized with public

```
1  class A {
2    public int i1;
3    public int i2;
4
5    public A(int i1, int i2) {
6      this.i1 = i1;
7      this.i2 = i2;
8    }
9  }
10 class Main {
11   public static void main() {
12     A a = new A(<PUBLIC>, <SECRET>);
13     // ok
14     println(a.i1);
15     // illegal
16     println(a.i2);
17   }
18 }
```

**Figure 2.11:** A program with a spurious security leak for field-insensitive analyses.

information, while the second stores a secret value. Class `A` contains a method that prints its attribute `i` to the console. So it depends on the value stored in `i` whether `doPrint` prints a secret value. The `main` method calls `doPrint` for both instances in l. 18 and l. 20, but only the call on instance `a2` in l. 20 leaks secret information, because only this instance of `A` contains secret information. An object-sensitive analysis is able to detect this, while an object-insensitive analysis does not distinguish between the two calls and assumes a possible security violation in both cases.

We observed that object-sensitivity is a very important feature when analyzing object-oriented code, but it comes at the cost of scalability. Large programs use too many different objects to distinguish all instances in reasonable time. Hence special adjustments have to be made that limit object-sensitivity to a subset of all instances. We discuss our implementation of object-sensitivity in §2.5.2 and show how it enables us to prove noninterference for an example program in §4.2.

## 2.2.3 Field-sensitive

An analysis is called *field-sensitive* if it can distinguish between different fields of an object instance. A field-insensitive analysis treats every access to an object field as an access to the same field. So whenever secret information is stored in a single field, all other fields of the same object are also considered to hold secret information.

The example in Figure 2.11 illustrates this problem. Class `A` has two fields `i1` and `i2`, where `i1` is initialized with public and `i2` secret information. Therefore only the print statement in l. 16 leaks secret

```
1  class A {
2
3    public int i;
4
5    public A(int i) {
6      this. i = i;
7    }
8  }

9  class Main {
10   public static void main() {
11     A a = new A(<PUBLIC>);
12     println(a.i);    // ok
13     a.i = <SECRET>;
14     println(a.i);    // illegal
15   }
16 }
```

**Figure 2.12:** A program with a spurious security leak for flow-insensitive analyses.

information. A field-insensitive analysis however would also report a security violation for the first print statement in l. 14.

Our tool supports field-sensitivity in different variants. It allows partial field-sensitivity where fields are grouped together in equivalence classes. This helps to reduce the overhead of tracking all possible fields and to achieve precise results at the same time. The parameterizable field-sensitivity is one of the main contributions of this work and is discussed in detail in §2.6.

### 2.2.4 Flow-sensitive

A *flow-sensitive* analysis respects the order in which statements are executed. It is able to detect that a statement may never influence another statement simply because it can only be executed afterwards. In general all analysis that operate on so-called *flow-graphs* (see §2.3.3) are flow-sensitive per default, but other approaches e.g. ones based on type-systems have to introduce an extra layer of complexity to achieve it [57].

The example in Figure 2.12 shows how flow-sensitivity helps to identify impossible leaks. It contains two print statements of the same object field `a.i` in l. 12 and l. 14. Only the second print statement can leak secret information, as `a.i` initially contains public information and is only set to a secret value after the first print statement. A flow-insensitive analysis would consider both print statements as leaks, because `a.i` may contain a secret value.

```
1 class T extends Thread {
2
3   public int x = <PUBLIC>;
4   public int y = <PUBLIC>;
5
6   public void run() {
7     x = y;
8   }
9 }
```

```
10 class Main {
11   public static void main() {
12     T t = new T();
13     t.start();
14     int p = t.x;
15     t.y = <SECRET>;
16     println(p);    // ok
17   }
18 }
```

**Figure 2.13:** A multithreaded program that can be proven noninterferent by a time-sensitive analysis.

## 2.2.5 Precision options for multithreaded programs

In addition to the previously presented precision properties for the analysis of sequential programs, multithreaded programs introduce a new dimension for precision: The detection of impossible execution orders and interferences between statements in different threads. This section contains various properties that are also implemented in Joana. Note that we assume *sequential consistency*, which means that statements are executed in the order they are defined in the source code. Our analysis does not cover the additional effects allowed by the Java memory model (JMM). However as long as programs are synchronized correctly, sequential consistency can be assumed [84].

### Time-sensitive

*Time-sensitivity* describes the ability of an analysis to detect impossible execution orders of statements in different threads. More specifically a time-sensitive analysis discards execution orders that include a so-called time-travel and thus break sequential consistency. The example in Figure 2.13 illustrates this problem.

The example contains two threads, the main thread and an instance of class `T` that is started from the main thread in l. 13. Both threads may communicate through the public fields `x` and `y` of class `T`. Initially both attributes contain public information. But they both may end up containing secret information, if l. 15 is executed before l. 7. As local variable `p` reads the value of `x` in l. 14, a time-insensitive analysis may assume that `p` may also contain secret information. However this is not

```
 1 class T extends Thread {
 2   public static Object l = new Object();
 3   public int y;
 4
 5   public void run() {
 6     synchronized (T.l) {
 7       y = <SECRET>
 8     }
 9   }
10 }

11 class Main {
12   public static void main() {
13     T t = new T();
14     synchronized (T.l) {
15       t.start();
16       println(t.y);   // ok
17     }
18   }
19 }
```

**Figure 2.14:** A program that uses synchronization to prevent a security leak.

possible, because `x` can only contain secret information if l. 15 and l. 7 have already been executed before l. 14 copies the value of `x` to `p`. Thus l. 15 would have been executed before l. 14.

We are able to detect these impossible situations with a technique called time-sensitive *slicing* and *chopping*. This technique has first been proposed by Krinke [70] and has been further improved and implemented by Giffhorn [29].

### Lock-sensitive

Threads can use locks to synchronize the concurrent execution of statements. Whenever a thread *t* holds a lock *l* no other thread that tries to acquire *l* is allowed to proceed until *t* releases *l*. This mechanism prevents concurrent execution of any statements that are synchronized by the same lock object. Synchronization is often used to prevent unwanted interferences between threads, that otherwise would lead to program bugs. An analysis that can detect threads that are synchronized through the same lock is called *lock-sensitive*.

Figure 2.14 shows an example of a program that uses synchronization through a shared lock `T.l` to prevent an information leak. It contains two threads, the main thread and an instance of class `T`. This thread is started in l. 15 after the main thread acquired the lock `T.l`. This is crucial for the statement that modifies the value of `y` in l. 7. This statement is also synchronized with lock `T.l`, so it can only be executed after the main thread releases the lock in l. 17. At this time the output of the print statement in l. 16 already occurred. So the attribute `y` can not contain secret information at the time the print statement executes. A lock-sensitive analysis is able to detect this.

We incorporate a technique called *dynamic pushdown networks* (DPN), that models concurrent programs in a very precise way, into our analysis tool. They enable us to identify synchronization via well-nested locks [38].

## 2.3 Intraprocedural analysis

In the previous sections, we presented the challenges of analyzing an object-oriented language like Java and introduced various precision properties for static analyses. Now we are going to describe how we achieve properties like flow- or field-sensitivity. In this section we focus on the intraprocedural part of our analysis, then we extend the presented algorithms to the interprocedural case in the upcoming section §2.5.

### 2.3.1 Overview

Our information flow analysis for Java programs is based on a graph structure called *procedure dependence graph* (PDG). A PDG represents the possible information flow between statements of a single method in form of a graph. It is a conservative static approximation of any information flow that may occur during runtime. Thus a path between two statements in the PDG means that there may be information flow, the absence of a path however guarantees that no information flow is possible. So the more precise the underlying PDG computation algorithm performs, the less edges occur in the resulting PDG.

For the intraprocedural computation of a PDG we start at the level of Java bytecode and subsequently take 5 main steps: Conversion to SSA-bytecode (§2.3.2), control flow graph computation (§2.3.3), control dependence computation (§2.3.4), data dependence computation (§2.3.5) and finally combining control and data dependence into a procedure dependence graph (§2.3.6). Figure 2.15 shows an overview of the involved steps. Each node represents a single step and incoming edges show which results of previous steps are needed to compute the current step. Our implementation is based on the WALA framework for static analysis. We use WALA to compute the intermediate representation in form of SSA-bytecode and to extract information about the control flow.

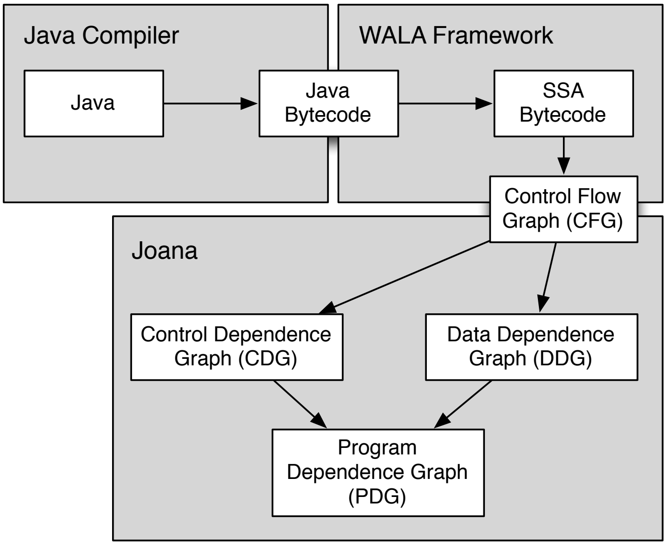

Java Compiler
Java
Java Bytecode
WALA Framework
SSA Bytecode
Control Flow Graph (CFG)
Joana
Control Dependence Graph (CDG)
Data Dependence Graph (DDG)
Program Dependence Graph (PDG)

**Figure 2.15:** Overview of the intraprocedural computation steps of the Joana IFC analyis.

Therefore we start by explaining some technical details of Java bytecode and the transformation to SSA-bytecode.

### 2.3.2 Intermediate representation

An intermediate representation (IR) has many benefits for program analysis. Typically it contains less complex instructions than the source language and is therefore easier to tackle. Also, once an analysis for the IR has been implemented, it can be extended to a different source language simply through creating a new front-end that translates the source language to IR code.

We use the SSA-bytecode IR of the WALA framework. WALA already contains front-ends for Java, Javascript and Java bytecode. In close collaboration with Julian Dolby —head of the WALA framework— we added support for Dalivk bytecode of Android based platforms [92]. Hence our analysis can be extended to support additional object-oriented

```java
static int calc(int a, int b) {
  int result;
  if (a > b) {


    result = a + b;

  } else {


    result =  a * b;
  }
  return result;
}
```

```
1  iload_0       // load param a
2  iload_1       // load param b
3  if_icmple 9   // if (b<=a) goto 9
4  iload_0       // load param a
5  iload_1       // load param b
6  iadd          // a + b
7  istore_2      // store result
8  goto 13
9  iload_0       // load param a
10 iload_1       // load param b
11 imul          // a * b
12 istore_2      // store result
13 iload_2       // load result
14 ireturn       // return result
```

**Figure 2.16:** A Java program fragment and its corresponding bytecode.

languages with relatively minor hassle. E.g. a front-end for C# as well as one for Apple's new programming language Swift will be added to WALA soon. Nevertheless we are going to focus on Java bytecode for the remainder of this work, as its representation is quite similar to the IR and the corresponding front-end is the most reliable.

### Bytecode

Every Java program is translated into bytecode before it can be executed in the Java virtual machine. Java bytecode is a stack-based and static typed language with an assembler-like instruction set. Every instruction pulls its operands off the stack and pushes the result back onto it. The omission of references to actual operands helps to keep the size of a single instruction very small (1 byte in general), but it is not easy to read and also complicates further analysis, as data dependencies between instructions are not clearly visible.

Figure 2.16 shows part of a Java program and its corresponding bytecode. Most noticeable is the huge amount of `iload` and `istore` instructions that load values onto or off the stack. Also most instructions are prefixed with an "`i`" because they refer to integer values. Similar instructions for other types like byte, char, etc. exist and use a different prefix specific to the type. This leads to a large amount of instructions and further complicates the analysis of bytecode. Therefore we translate bytecode into an intermediate representation that is easier to handle but still very close to actual bytecode.

```
1 A.calc(int v1, int v2)
2   if (v1 <= v2) goto 5
3   v5 = v1 + v2
4   goto 6
5   v4 = v1 * v2
6   v6 = phi v5, v4
7   return v6
```

**Figure 2.17:** The SSA intermediate representation of the program fragment in Figure 2.16.

## SSA-Form

The *static single assignment form* [110,3,24] (SSA-Form) has been designed as an intermediate representation for program analysis and compiler applications. Its design allows an easy and straight forward implementation of many standard analyses, such as constant propagation or reaching definitions. In SSA-Form the value of any variable can only be assigned once. Thus variables are never overwritten and every use of a variable refers to the same single statement where its value is defined.

Figure 2.17 shows the SSA-Form of the program from Figure 2.16. All variables have been renamed to `v<`*i*`>`. Parameter `a` is `v1`, parameter `b` is `v2` and the `result` variable corresponds to multiple SSA-variables `v5`, `v6` and `v7`. The value of some variables, like `result`, are defined more than once in the program. Therefore they are split into multiple variables, one for each definition: `v5` in l. 3 and `v4` in l. 5. Additional so-called *phi*-variables combine the values of multiple variables to a single new variable, like variable `v6` that combines the values of `v4` and `v5`.

We use the WALA framework to compute SSA-Form from the stack-based Java bytecode. The resulting intermediate representation is close to actual bytecode: Besides the additional definitions of phi-variables, every other instruction in the IR corresponds to a Java bytecode instruction. Only stack manipulating instructions that push, pop or duplicate values on the stack are omitted as their effects are explicitly captured through variables. The set of IR instructions is also significantly smaller than its bytecode counterparts, because all typed operations, like special `add` operations for integer, byte, etc. are subsumed with a single operation that can be applied to any valid type. These properties simplify further analysis algorithms as the instruction set is smaller, data dependencies are directly visible through variable access and the number of instructions per method is reduced, due to the omitted stack manipulation instructions.

### 2.3.3 Control flow graph

The *control flow graph* (CFG) is a graph that captures the potential execution order of all instructions in a method. It contains nodes for each instruction and edges between instructions iff one instruction may be executed directly after the other. The formal definition of a CFG also contains a special node $n_{entry}$ for method entry and single node $n_{exit}$ for method exit. We use a definition similar to Giffhorn [29].

**Definition 2.1** (Control Flow Graph (CFG)). *A control flow graph* $G = (N, E, n_{entry}, n_{exit})$ *of a method m is a directed graph, where*

- *N is the set of nodes where each instruction in m is represented by a node* $n \in N$.
- *E is the set of edges representing the control flow between the nodes. We write* $n_1 \xrightarrow{cf} n_2 \in E$ *iff control flow from* $n_1$ *to* $n_2$ *is possible.*
- $n_{entry}$ *is the start node. It has no incoming edges and reaches all nodes in N.*
- $n_{exit}$ *is the exit node. It has no outgoing edges and is reachable from every node in N.*

We use additional edge labels for conditional branches and exception related control flow: *true*, $false$ and *exc*. *true* marks all edges from a conditional branch (if-statement) to the instruction that executes iff the condition holds, $false$ marks the edges to the instruction that executes if the condition does not hold. *exc* marks all edges that correspond to control flow that occurs iff the instruction of the source node throws an exception. All other edges are unlabeled.

Figure 2.18 shows two CFGs for the previous example program. The upper graph shows the CFG for the source code version and the lower graph is the CFG for the program in its intermediate representation. The general structure of the graph remains the same, but the IR version contains additional nodes for `phi` and **`goto`** statements.

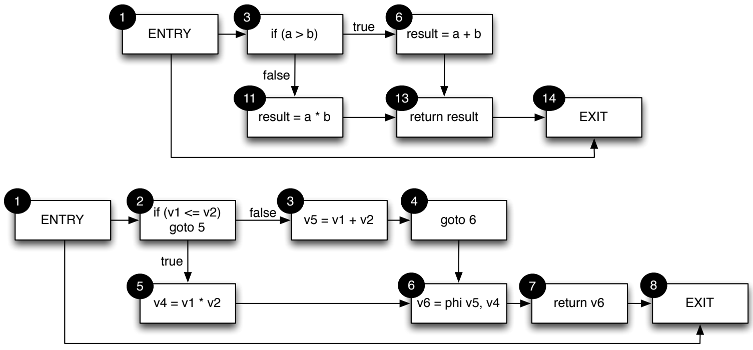


**Figure 2.18:** Control flow graphs from the example in Figure 2.16 and Figure 2.17.

## 2.3.4 Control dependence graph

The *control dependence graph* (CDG) is directly computed from the CFG. It captures the so-called *indirect* or *control dependencies* between statements that occur when the outcome of one statement decides if another statement is executed, e.g. the evaluation of the condition of an **`if`**-statement decides which branch is taken. Ferrante et.at. [27] define control dependence through the *post-dominance* relation.

**Definition 2.2** (Post-Dominance). *Given $n_1, n_2, n_{exit} \in CFG$. Node $n_1$ post-dominates $n_2$ iff all paths from $n_2$ to $n_{exit}$ contain $n_1$.*

**Definition 2.3** (Control Dependence). *Given $n_1, n_2 \in CFG$. $n_2$ is control dependent on $n_1$ ($n_1 \xrightarrow{cd} n_2$) if*

- *$\exists$ path $P = n_1 \xrightarrow{cf} \cdots \xrightarrow{cf} n_2 \in CFG$ with $n \in P$ and $n_2$ post-dominates $n$.*
- *$n_2$ does not post-dominate $n_1$.*

Figure 2.19 shows the control dependence graph for the example program in IR form. For example node 6 is control dependent on the entry node, because for all paths from entry to node 6, node 6 post-dominates each node in these paths except the entry. Node 3 is control dependent on the **`if`**-clause in node 2, because it does not post-dominate node 2.

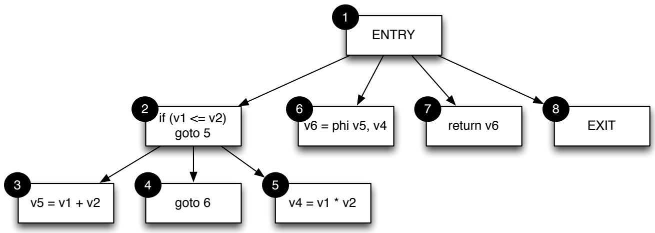


**Figure 2.19:** Control dependence graph of the example in Figure 2.17.

Note that the artificial control flow edge from entry to exit is essential for the control dependence definition, as it prevents post-domination of the entry. An efficient computation of control dependencies applies the Lengauer-Tarjan algorithm [78] to compute the post-dominator tree.

### 2.3.5 Data dependencies

Data dependencies are dependencies between two statements $s_1 \xrightarrow{dd} s_2$, where one statement $s_1$ produces a value the other statement $s_2$ uses. Typically these dependencies occur between definition and usage of a local variable and between modifications and references of heap values. In Java heap dependencies are introduced through references on object- and array-fields. The *data dependence graph* (DDG) captures all data dependencies that occur between statements of a single method. It contains nodes for each statement and edges between them if one is data dependent on the other.

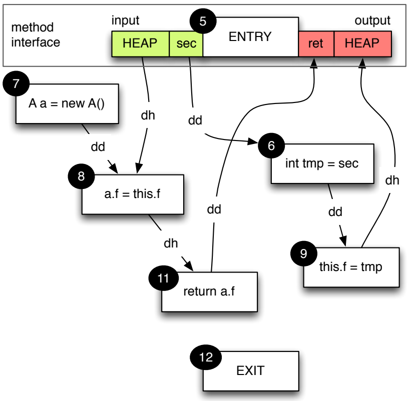
**Definition 2.4** (Data Dependence)**.** *Two statements $s_1, s_2$ are data dependent, iff they are* direct data dependent *or* heap data dependent.

**direct data dependent** *($s_1 \xrightarrow{dd} s_2$): $s_1$ defines a local variable $v$ that $s_2$ uses and $\exists$ path $P \in CFG$ where $P = s_1 \xrightarrow{cf} \ldots \xrightarrow{cf} s_2$ and $\nexists s' \in P : s' \neq s_1 \wedge s'$ redefines $v$.*

**heap data dependent** *($s_1 \xrightarrow{dh} s_2$): $s_1$ modifies a heap location $l$ that $s_2$ may refer to and $\exists$ path $P \in CFG$ where $P = s_1 \xrightarrow{cf} \ldots \xrightarrow{cf} s_2$ and $\nexists s' \in P : s' \neq s_1 \wedge s'$ redefines $l$.*

```
class A {

  int f;

  int datadeps(int sec) {
    int tmp = sec;
    A a = new A();
    a.f = this.f;
    this.f = tmp;

    return a.f;
  }

}
```

**(a)** program fragment



**(b)** data dependence graph

**Figure 2.20:** A program fragment (2.20a) and its DDG (2.20b).

We distinguish between direct data dependencies and heap data dependencies, because they differ in the way they can be approximated and have different properties concerning the modular analysis we present in Chapter 3. Basically direct data dependencies are easier to compute and are independent of the context a method is executed in, while heap data dependencies depend on the state of the heap before method execution.

In addition to statement nodes data dependence graphs also contain special nodes —called *parameter nodes*— that represent values passed into or returned by the corresponding method.

**Definition 2.5** (Parameter Nodes). Parameter nodes *are artificial nodes in the* data dependence graph *of a method m that represent values in method parameters or heap locations. The values of parameters passed into m or heap values read by m are called* input parameter nodes, *while the return value and modified heap values are called* output parameter nodes.

**Definition 2.6** (Method Interface). *The* method interface *of method m is the combination of all input and output parameter nodes summarizing all values potentially read or modified through execution of m.*

**Definition 2.7** (Data Dependence Graph)**.** *A data dependence graph (DDG) of a method $m$ is a graph $G = (N, E)$. The nodes $N$ correspond to program statements $S$ and artificial* parameter nodes *$P$ of the method interface of $m$. Edges $E$ between nodes represent data dependencies, they are either local or heap data dependencies.*

Figure 2.20 shows an example program (2.20a) that contains direct as well as heap data dependencies and its corresponding data dependence graph (2.20b). The method interface of `datadeps` contains two input parameter nodes and two output parameter nodes. The input parameter nodes consist of the method parameter `sec` and a generic node `HEAP` that represents values read from the heap — in this case the object field `f` of the **`this`**-object in statement 8. The output parameter nodes consist of a node `ret` for the return value of the method and another `HEAP` node that represents heap values modified by the method. The dependencies in the DDG show that the value of variable `sec` does not flow to the return value of `datadeps`. However its value is stored in an object field **`this`**`.f` on the heap. So subsequent access to **`this`**`.f` after the execution of `datadeps` can reveal the value of `sec`. This modification of the global program (heap) state is called a method *side-effect* and discussed in more detail in §2.6.

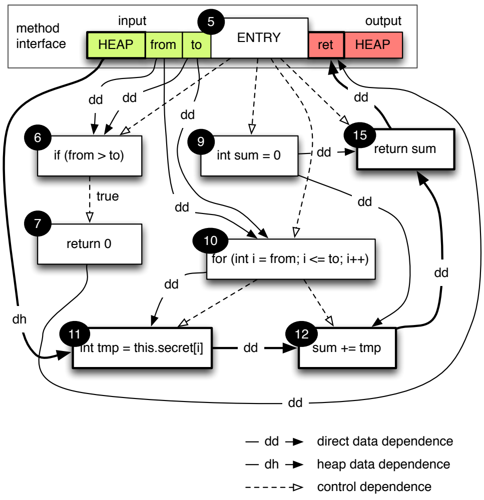
We use DDGs to detect direct information flow of secret values. Joana supports multiple variants of sophisticated computation algorithms for data dependencies that differ in precision and scalability. We are going to present them in detail in the upcoming sections §2.5.2 and §2.6.

### 2.3.6 Procedure dependence graph

A *procedure dependence graph* (PDG) is the combination of a control dependence and a data dependence graph. It captures all direct and indirect information flow inside a single method. PDGs have been researched for years [98] and are applied to many different areas, like debugging [61], testing [13], code duplicate detection [68] and also for information flow control [47]. We use a highly optimized version of PDGs and their interprocedural counterpart the *system dependence graph* (SDG), tailored to object-oriented languages and IFC. The upcoming sections §2.6 and §2.5.3 contain more of these specific optimizations. For now we focus on the principle structure of a standard PDG.

```
 1  class A {
 2
 3    int[] secret;
 4
 5    int arraysum(int from, int to) {
 6      if (from > to)
 7        return 0;
 8
 9      int sum = 0;
10      for(int i = from; i <= to; i++) {
11        int tmp = this.secret[i];
12        sum += tmp;
13      }
14
15      return sum;
16    }
17
18  }
```

**(a)** Program fragment



**(b)** Procedure Dependence Graph

**Figure 2.21:** A program fragment (2.21a) and its PDG (2.21b). Bold edges and nodes denote the flow of information from the attribute `secret` to the return value. The exit node is missing for brevity.

**Definition 2.8** (Procedure Dependence Graph). *A procedure dependence graph (PDG) of a method m is a graph $G = (N, E)$. The nodes N correspond to program statements S and artificial* parameter nodes *P of the method interface of m (see Definition 2.5 and Definition 2.6). Edges E between nodes represent either data or control dependencies.*

Figure 2.21 shows an example program (2.21a) that computes a sum of array entries for a given range and its corresponding PDG (2.21b). The program contains a class `A` with an array attribute `secret` and a single method `arraysum`. The PDG shows all direct and indirect information flow inside `arraysum`. A path in the PDG from the input parameter *HEAP* to the return value *ret* (*HEAP* → 11 → 12 → 15 → *ret*) captures flow of the values stored in the array `secret` to the return value of the method.

Our IFC analysis detects these paths and thus can not only show if there is an information flow possible, but also which statements in the program are involved. This additional information enables the user to better understand the nature of the flow and helps to locate the potential problem in the code or to decide if the flow is in fact intended.

The presented technique helps to detect information flow inside a single method. Depending on the size of the method and its structure it can be a difficult task to do manually, hence an automated analysis that guarantees to detect all possible flow is useful even for a single method. In practice information flow is often not confined to a single method. It occurs between multiple methods that pass information back and forth through parameters and heap locations. In the following Section §2.5 we discuss how to extend this intraprocedural technique to detect information flow in the whole program.

## 2.4 Enhancing the intraprocedural analysis

### 2.4.1 Control-flow optimizations for exceptions

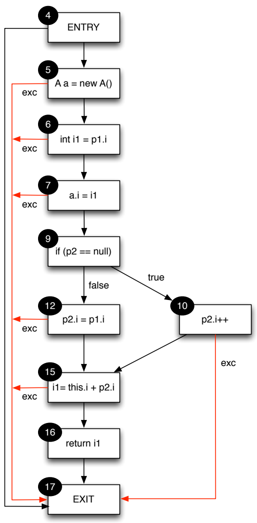
In a language like Java almost any instruction may potentially throw an exception, e.g. any field access operation or dynamic method call. Thus many instructions create a branch in the control flow graph leading to spurious control dependencies. A security analysis cannot ignore these effects, because it needs to compute sound results. But on the other hand a naive approach introduces many control dependencies that are not possible in practice. We measured the number of control dependencies that are introduced by exceptional control flow on a large set of example programs[10]. 87% of all control dependencies are induced by conservative approximated exceptions. Of all instructions that potentially may throw an exception[11], 83% of them are *NullPointerExceptions*, followed by 8% *ArrayIndexOutOfBoundsExceptions*, 7% *OutOfMemoryErrors* and 5% *ExceptionInInitializerErrors*. The remaining exception types are ≤ 1%. Therefore we propose additional analysis to detect and remove impossible paths in the CFG with a special focus on the detection of **`null`**-pointers. In this section we describe the intraprocedural **`null`**-pointer analysis that detects impossible exceptions and is able to reduce the number of control

[10]We used the JavaGrande benchmark suite, HSQLDB and jEdit with a total of 490000 instructions.

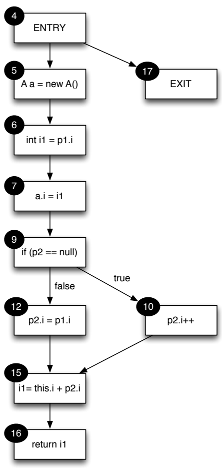
[11]A single instruction may throw more then a single type of exception, so % add up above 100%.

```
1  class A {
2    private int i = 0;
3
4    public int foo(A p1, A p2) {
5      A a = new A();
6      int i1 = p1.i;
7      a.i = i1;
8
9      if (p2 == null) {
10       p2.i++;
11     } else {
12       p2.i = p1.i;
13     }
14
15     i1 = this.i + p2.i;
16     return i1;
17   }
18 }
```

**(a)** program fragment



**(b)** unoptimized CFG



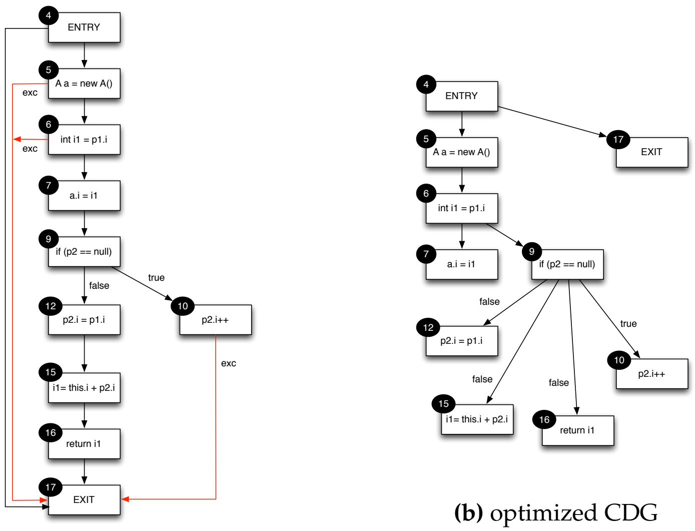
**(c)** unoptimized CDG

**Figure 2.22:** An example where 5 impossible and 1 always occurring NullPointerExceptions can be detected and its matching CFG (2.22b) and CDG (2.22c) without optimizations.

dependencies induced by exceptions to 53% of all control dependencies.

The code in Figure 2.22 illustrates the problem of potential **null**-pointers. Method foo contains many statements that may potentially throw an exception. The control flow graph (2.22b) shows the result of a naive control flow approximation. Red edges labeled "exc" represent potential exception control flow. The corresponding unoptimized control dependence graph (2.22c) is on the right side of the figure. Almost any statement is control dependent on its control flow predecessor. For example statement 9 is control dependent on statement 7, albeit in closer inspection statement 7 can never throw an exception, as variable a cannot be **null**.

We can detect and remove these spurious exceptions with an analysis that detects potential **null**-pointers. All exceptions except those in statement 5, 6 and 10 can never occur. The optimized CFG and CDG in



**(b)** optimized CDG

**(a)** optimized CFG

**Figure 2.23:** The optimized CFG (2.23a) of the example in Figure 2.22 and its resulting CDG (2.23b).

Figure 2.23 show the result of our analysis. As statement 10 shows, we can also detect exceptions that will always occur and thus remove the normal control flow.

**Implementation as data flow analysis** We implemented **`null`**-pointer detection as data flow analysis. The unoptimized CFG is used as flow graph and each edge is annotated with a state and a transfer function. The state for each edge stores the state of each variable in the current method as either *unknown*, *null*, ¬*null* or *both*. As shown in Figure 2.24 these values form a lattice similar to standard constant propagation. Initially the state of all variables is set to *unknown*. We write $v \leftarrow s \in \{unknown, null, \neg null, both\}$ to denote that the state $s_v$ of variable $v$ is set to $s'_v = s_v \sqcap s$. The transfer functions then implement the following rules.

**field access** ($v.f = \ldots$ or $\ldots = v.f$): Every field access operation $o_{acc}$ has two potential successor instructions: One that is executed if the operation was successful $o'$ and the other one that is executed in case of an exception $o_{exc}$. Hence the matching CFG contains $o \xrightarrow{cf} o'$ as well as $o \xrightarrow{exc} o_{exc}$.

$o_{acc} \xrightarrow{cf} o'$ : Variable $v$ is not null, hence we update the state of $v$ accordingly: $v \leftarrow \neg null$

$o_{acc} \xrightarrow{exc} o_{exc}$ : Variable $v$ was null and its state is updated accordingly: $v \leftarrow null$

**object instantiation** ($v = new \ldots$): If the instantiation operation $o_{new}$ does not throw an exception, $v$ is $\neg null$, otherwise $v$ is $null$. Again with $o_{new} \xrightarrow{cf} o'$ and $o_{new} \xrightarrow{exc} o_{exc}$ as the two successors in the CFG, we get

$o_{new} \xrightarrow{cf} o'$ : Variable $v$ is not null, hence $v \leftarrow \neg null$

$o_{new} \xrightarrow{exc} o_{exc}$ : Variable $v$ is null, hence $v \leftarrow null$

**conditional branching** ($if(v == null)\ \{B_1\}\ else\ \{B_2\}$): For all statements in block $B_1$ variable $v$ is $null$, for all statements in $B_2$ variable $v$ is $\neg null$. So with $o_{if}$ as the operation of the if-clause, $o_1$ as the first operation in $B_1$ and $o_2$ the first operation in $B_2$ we get

$o_{if} \xrightarrow{cf} o_1$ : Variable $v$ is null, hence: $v \leftarrow null$

$o_{if} \xrightarrow{cf} o_2$ : Variable $v$ is not null, hence $v \leftarrow \neg null$

**this-pointer** The **`this`**-pointer is never null. All field accesses on the **`this`**-object do not throw an exception. All variables corresponding to a **`this`**-pointer are initialized with $\neg null$ instead of *unknown*.

We use the CFG edge labels to decide which edge corresponds to normal and which to exceptional control flow. This allows us to set the appropriate transfer functions. For example in Figure 2.22b the edge 5 $\xrightarrow{cf}$ 6 is normal control flow from an instantiation. The matching transfer function sets the state of variable `a` to $\neg null$. The transfer function of 5 $\xrightarrow{exc}$ 17 sets the state of `a` accordingly to *null*.

Table 2.1a shows how transfer functions for all edges of the example in Figure 2.22 modify the variable state. Figure 2.1b shows the final

| edge | transfer function |
|---|---|
| 5 $\xrightarrow{cf}$ 6 | a ← *¬null* |
| 5 $\xrightarrow{exc}$ 17 | a ← *null* |
| 6 $\xrightarrow{cf}$ 7 | p1 ← *¬null* |
| 6 $\xrightarrow{exc}$ 17 | p1 ← *null* |
| 7 $\xrightarrow{cf}$ 9 | a ← *¬null* |
| 7 $\xrightarrow{exc}$ 17 | a ← *null* |
| 9 $\xrightarrow{true}$ 10 | p2 ← *null* |
| 9 $\xrightarrow{false}$ 12 | p2 ← *¬null* |
| 10 $\xrightarrow{cf}$ 15 | p2 ← *¬null* |
| 10 $\xrightarrow{exc}$ 17 | p2 ← *null* |
| 12 $\xrightarrow{cf}$ 15 | p1 ← *¬null*, p2 ← *¬null* |
| 12 $\xrightarrow{exc}$ 17 | p1 ← *null*, p2 ← *null* |
| 15 $\xrightarrow{cf}$ 16 | p2 ← *¬null* |
| 15 $\xrightarrow{exc}$ 17 | p2 ← *null* |
| default | do not modify state |

**(a)** transfer functions of CFG edges

| node | p1 | p2 | a |
|---|---|---|---|
| 4 | ⊥ | ⊥ | ⊥ |
| 5 | ⊥ | ⊥ | ⊥ |
| 6 | ⊥ | ⊥ | *¬null* |
| 7 | *¬null* | ⊥ | *¬null* |
| 9 | *¬null* | ⊥ | *¬null* |
| 10 | *¬null* | *null* | *¬null* |
| 12 | *¬null* | *¬null* | *¬null* |
| 15 | *¬null* | *¬null* | *¬null* |
| 16 | *¬null* | *¬null* | *¬null* |
| 17 | ⊤ | ⊤ | ⊤ |

**(b)** final variable states

**Table 2.1:** Transfer functions (2.1a) and final variable states (2.1b) of the **null**-pointer analysis for the example in Figure 2.22.

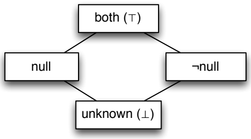


**Figure 2.24:** The lattice of the state values for the **null**-pointer detection.

variable states after the analysis finished. For example in statement 10 we know that p1 and a can never be **null**, while p2 will always be **null**.

We combine this information with the unoptimized CFG and check for each edge if the variable state of its source statement allows us to remove the edge from the graph. For example edge 6 $\xrightarrow{exc}$ 17 is only valid if p1 may be **null** at statement 6. The final variable state for p1 in statement 6 is *unknown* (⊥), therefore we cannot be sure that p1 is not **null** and thus cannot remove this edge. However this is possible for other edges, like 7 $\xrightarrow{exc}$ 17, where we know that a is never **null**. The resulting optimized CFG and CDG are shown in Figure 2.23.

We added several options for exception analysis to Joana. We include the option to intraproceduraly optimize the CFGs for each method with the presented analysis. Herhoffer extended this analysis in his Studienarbeit [54] with an interprocedural propagation of variable states for method parameters. His analysis almost doubles the number of edges that can be removed from the CFG. We also include an option to fully ignore influence of exceptions on the program execution. Albeit the results of the IFC analysis are no longer sound when exception are ignored, it helps to pin down the cause of illegal flow.

### 2.4.2 Termination-sensitive control dependencies

Besides standard control dependence (SCD) (see Definition 2.3), there are other possible definitions for control dependence with subtly different semantics [45]. In this section we introduce the *termination-sensitive* control dependence from Podgurski and Clarke [101] called *weak control dependence* (WCD) and an extension to track these dependencies interprocedurally.

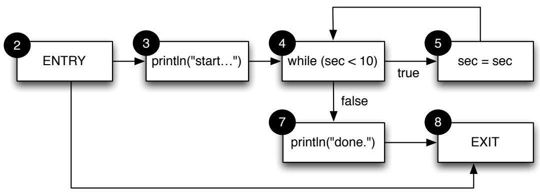
Informally, a control dependence $n_1 \xrightarrow{cd} n_2$ captures if the outcome of statement $n_1$ may influence if $n_2$ is executed. In case of SCD we search for alternative paths from $n_{entry}$ to $n_{exit}$ in the CFG that do not include $n_1$ but include $n_2$. Looking for the possibility that the program executes $n_2$ without ever executing $n_1$. This approach only works under the assumption that $n_{exit}$ is reached, hence the program terminates. Termination-sensitive control dependence like WCD also capture the possibility that a statement may force execution into an endless loop —preventing execution of statements after the loop.

Figure 2.25 illustrates this situation. Method `endless` contains a loop that may never terminate if the value of parameter `sec` is below 10. In an information flow setting this behavior can leak information about the value of parameter `sec`. Whenever an attacker observes the output of the last `print` statement in l. 7, he knows that `sec` $\geq$ 10.

Standard control dependence (Figure 2.26a) does not capture this effect: Statement 7 does not depend on the outcome of statement 4 and thus is considered independent of the value of `sec`. Weak control dependence (Figure 2.26b) however includes this dependency. It conservatively assumes each loop may execute infinitely and therefore all subsequent statements depend on the loop condition. The definition of weak control

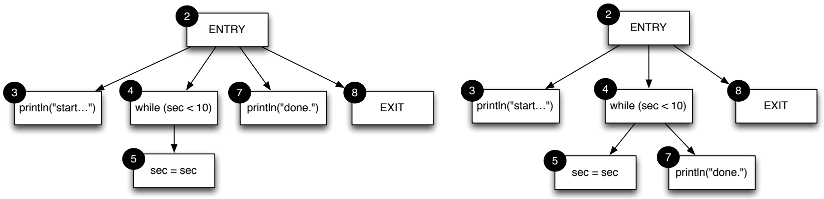
```
1 class A {
2   void endless(int sec) {
3     println("start...");
4     while (sec < 10) {
5       sec = sec;
6     }
7     println("done.");
8   }
9 }
```

**(a)** program fragment



**(b)** control flow graph

**Figure 2.25:** A program fragment (2.25a) that may not terminate and its CFG (2.25b).



**(a)** standard control dependence (SCD) **(b)** weak control dependence (WCD)

**Figure 2.26:** A termination-insensitive (2.26a) and a termination-sensitive (2.26b) CDG for the program fragment in Figure 2.25.

dependence is very similar to standard control dependence, it uses a slightly different version of post-domination, called *strong postdomination*.

**Definition 2.9** (Strong Post Domination). *Given $n_1, n_2 \in CFG$. $n_2$ strong post-dominates $n_1$ if*

- *$n_2$ post-dominates $n_1$.*
- *$\nexists$ path $P = n_1 \text{-}cf^{*}\!\!\rightarrow n_2 \in CFG$, where $P$ contains a loop.*

**Definition 2.10** (Weak Control Dependence). *Given $n_1, n_2 \in CFG$. $n_2$ is weak control dependent on $n_1$ ($n_1 \text{-}wcd\!\!\rightarrow n_2$) if*

- *$\exists$ path $P = n_1 \text{-}cf^{*}\!\!\rightarrow n_2 \in CFG$ with $n' \in P$ and $n_2$ strong post-dominates $n'$.*
- *$n_2$ does not strong post-dominate $n_1$.*

Most IFC analyses [124,93,94,43,20] argue that program termination is a prerequisite for a secure program. They assume each program terminates and ignore termination-sensitive leaks. E.g. the common noninterference criterion implicitly assumes program termination, as it is defined through the relation between values at program start and end states —hence implicitly assumes an end state exists.

**Interprocedural extension** We developed an interprocedural extension to a termination-sensitive whole program analysis. It detects calls to methods that may not terminate and treats these calls similar to a non-terminating loop during WCD computation. Our algorithm basically contains 4 major steps to analyze a given program $P$:

1. Detect each method that contains a potentially non-terminating loop in the intraprocedural control flow.

$$M_{loop} = \{m \in P \mid m \text{ contains non-terminating loop}\}$$

2. Detect methods that may be called recursively (direct as well as indirect).

$$M_{recurs} = \{m \in P \mid \exists m_1, \ldots, m_i \in P : m \xrightarrow{\mathit{call}} m_1, \ldots m_i \xrightarrow{\mathit{call}} m\}$$

3. Detect methods that may not terminate.

$$M_{non-term} = \{m \in P \mid \exists m' \in M_{loop} \cup M_{recurs} :$$
$$\exists m_1, \ldots, m_i \in P : m \xrightarrow{\mathit{call}} m_1, \ldots m_i \xrightarrow{\mathit{call}} m'\}$$

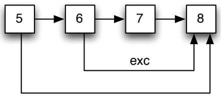
4. Treat calls to methods from $M_{non-term}$ similar to non-terminating loops. Use *extended strong postdomination* (Definition 2.11) during WCD computation.

**Definition 2.11** (Extended Strong Post Domination). *Given $n_1, n_2 \in CFG$. $n_2$ strong post-dominates $n_1$ if*

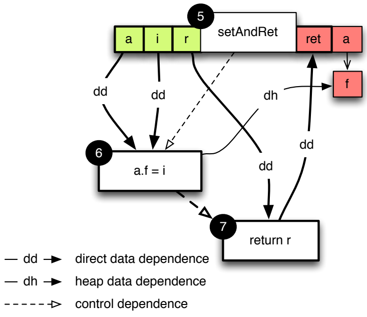
- *$n_2$ post-dominates $n_1$.*
- *$\nexists$ path $P = n_1 \xrightarrow{cf^*} n_2 \in CFG$, where $P$ contains a loop* **or a call to** *$m$* **with** *$m \in M_{non-term}$.*

```
1 class A {
2   int f;
3
4   static int
5   setAndRet(A a, int i, int r) {
6     a.f = i;
7     return r;
8   }
9 }
```

**(a)** example program



**(b)** control flow graph



**(c)** procedure dependence graph

**Figure 2.27:** Example program (2.27a) with its CFG (2.27b) and PDG (2.27c). It shows the effects of imprecise treatment of exceptions in field access operations: The return value is considered dependent on the value of parameter `i`.

We include an option for termination-sensitive weak control dependence in Joana, which is turned off by default. Evaluation has shown that by conservatively assuming non-termination for each loop in the control flow, the precision of the analysis takes a huge hit and many false alarms occur. Thus this option should only be used when termination-sensitivity is required for the current analysis.

## 2.4.3 Fine-grained field access

In this section we introduce a fine-grained model for field access operations in a PDG. Field accesses are special operations that read or modify values stored on the heap —the global state of the program. A field access can introduce new side-effects through modifications and its result depends on the context it is executed in. Besides method calls, they are the only points in the PDG that may change depending on the method context.

Previously a field access has been modeled as a single node in the PDG. This leads to some unwanted imprecision of the dependencies, as the example in Figure 2.27 shows. The example contains a single method `setAndRet` that takes three parameters and processes two seemingly independent operations. It sets the value of field `f` from the first parameter to the value of the second parameter in l. 6 and subsequently returns the value of the third parameter in l. 7. However as presented in §2.1.8 and §2.4.1 exceptions can introduce control dependencies between otherwise unrelated statements. This is the case in this example: When the field access throws a null-pointer exception, the return statement is never executed. The control flow graph in Figure 2.27b shows this potential behavior through an edge from node 6 to 8. This leads to a control dependency from node 6 to 7 in the corresponding PDG (Figure 2.27c). The bold nodes and edges in the PDG show all elements contained in the backward slice of the return value `ret`. The slice contains the field access including all three method parameters. Parameter `a` influences if the field access throws a null-pointer exception, while the value of parameter `r` is returned in the following statement. So both parameters can influence the existence and value of the return value. Parameter `i` has no influence on the return value, but is still included in the slice, because of its data dependency to the field access. However, the control flow from the field access is only influenced by the variable pointing to the object whose field is set, not the value that is written to the field. The current model of field accesses in PDGs does not capture this property.

We propose a new way to model field accesses as multiple nodes that correspond to 3 steps during a field write: (1) load base object, (2) load new value, (3) write value to field. Figure 2.28 shows a version of the PDG for the program in Figure 2.27a with a fine grained field access. In this version the return statement is only dependent on the load operation of base `a` (Node 6a) and not on the subsequent field write operation (Node 6, 6b). Therefore the backward slice from the return value no longer contains parameter `i`.

We differentiate 6 different forms of field access operations: static, object and array field write as well as static, object and array read operations. Figure 2.29 shows the control flow of all fine grained read operations together with their coarse grained counterparts. The control flow respects the order in which the different steps of a field access are executed. Field read operations may include up to 4 steps —depending on the type of the access— in the following order.

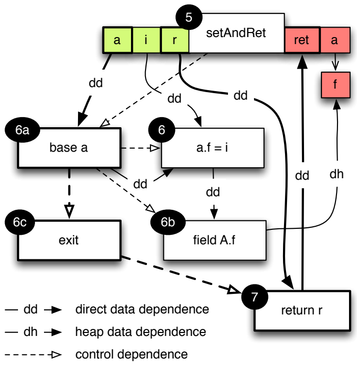

5
a
i
r
setAndRet
ret
a
f
dd
dd
dd
6a
base a
6
a.f = i
dd
dd
dd
dd
dh
6c
exit
6b
field A.f
7
return r
— dd → direct data dependence
— dh → heap data dependence
- - - - -▷ control dependence

**Figure 2.28:** The PDG with a fine-grained field access operation for the example program from Figure 2.27. It shows that parameter `i` no longer influences the return value.

1. Load base pointer (array and object field access only)
2. Load array index (array access only)
3. Read field
4. Store field value in local variable or stack

Steps 1 and 2 potentially can throw an exception in case the referenced base pointer is null, or the array index is negative or out of bounds. We model this behavior with additional exception control flow edges from the respective base and index nodes to an artificial exit node. We include a node for each step as well as an artificial exit node for the instruction in case it may throw an exception. The exit node provides a single point of exit, where the control flow of the intra-instruction steps leave the instruction.

Figure 2.30 contains a similar overview for all field write operations. Write operations include the similar steps as read operations, but in a different order. They read a value from a local variable or the stack before it is written to the field.

1. Load base pointer (array and object field access only)
2. Load array index (array access only)
3. Read new field value in local variable or stack
4. Write new value to field

Figure 2.31 and Figure 2.32 show how we model control and data dependencies for field read and write accesses. The new fine grained structure enhances precision of the modeled dependencies. Each value reading and writing step has its own node and the control dependencies due to potential exceptions are captured by a separate exit node that is not dependent on the field value. For example the field-set operation in Figure 2.32 "`v2.f = v1`" contains no dependency edge from the field node to the exit node. So a backward slice from the exit node will only lead to the source of the value `v2` and is independent of `v1`.

Aside from increased precision, fine-grained field accesses are also used in the computation of so-called *access paths* presented in the upcoming section §3.4.3 on modular SDGs. As the impact on runtime is minimal, fined-grained field accesses are enabled by default in Joana.

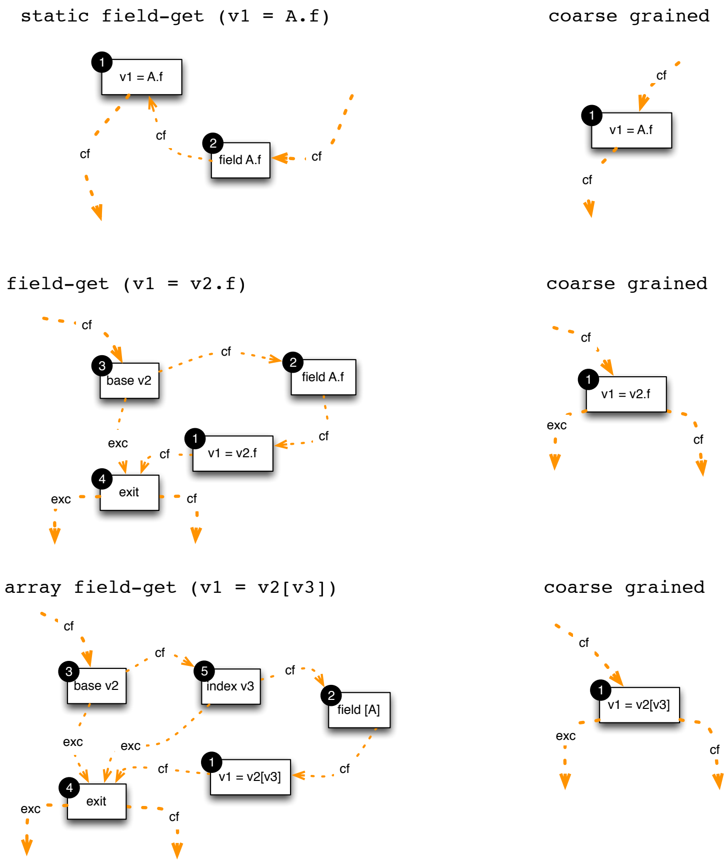


Figure 2.29: The control flow of fine grained field-get instructions.

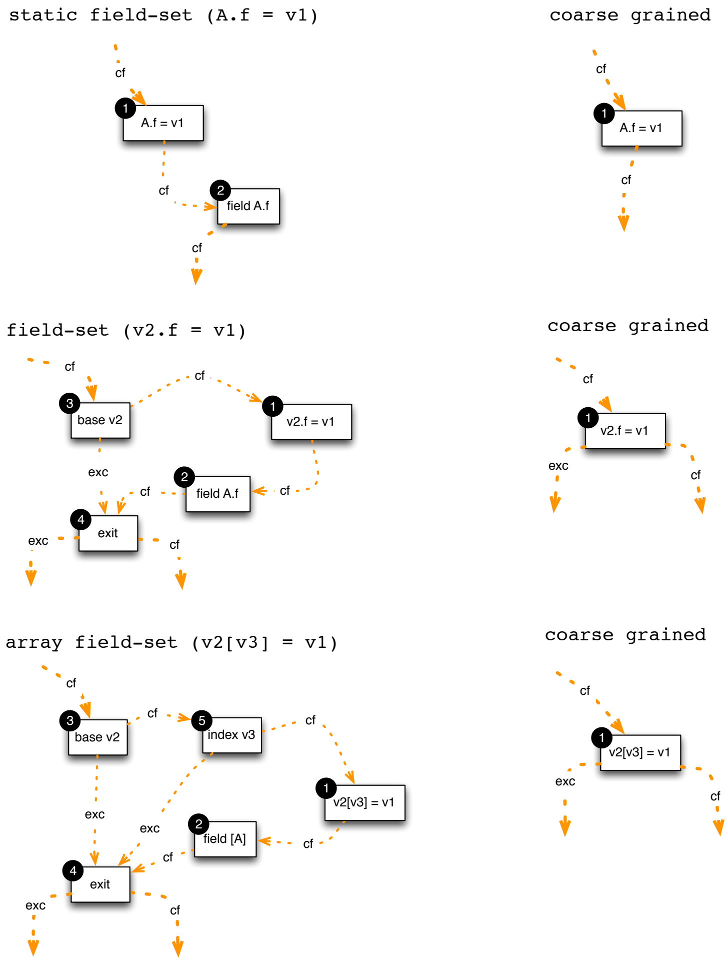


**Figure 2.30:** The control flow of fine grained field-set instructions.

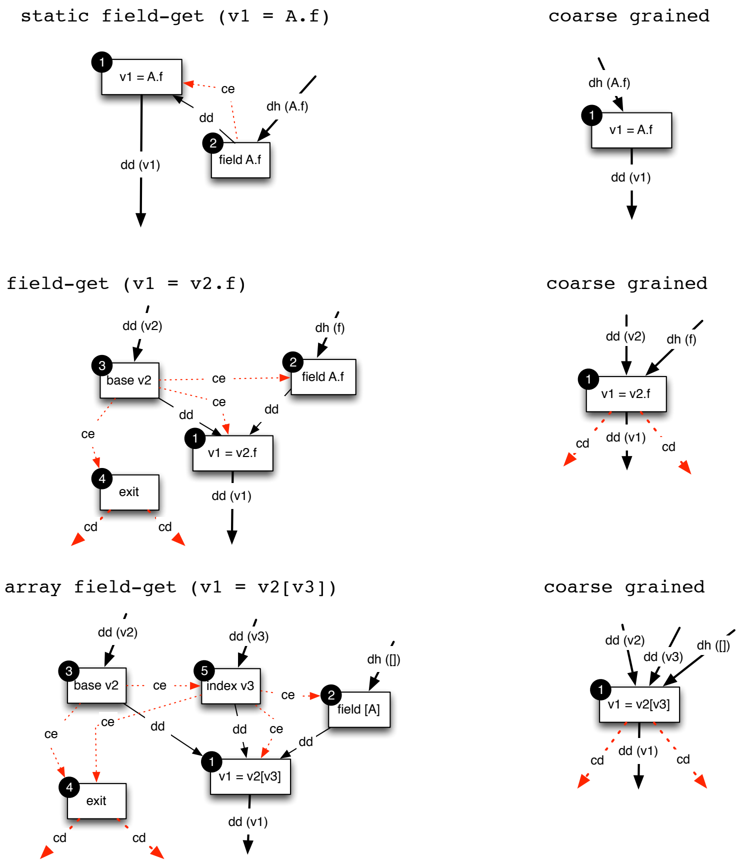


**Figure 2.31:** The control and data dependencies of fine grained field-get instructions.

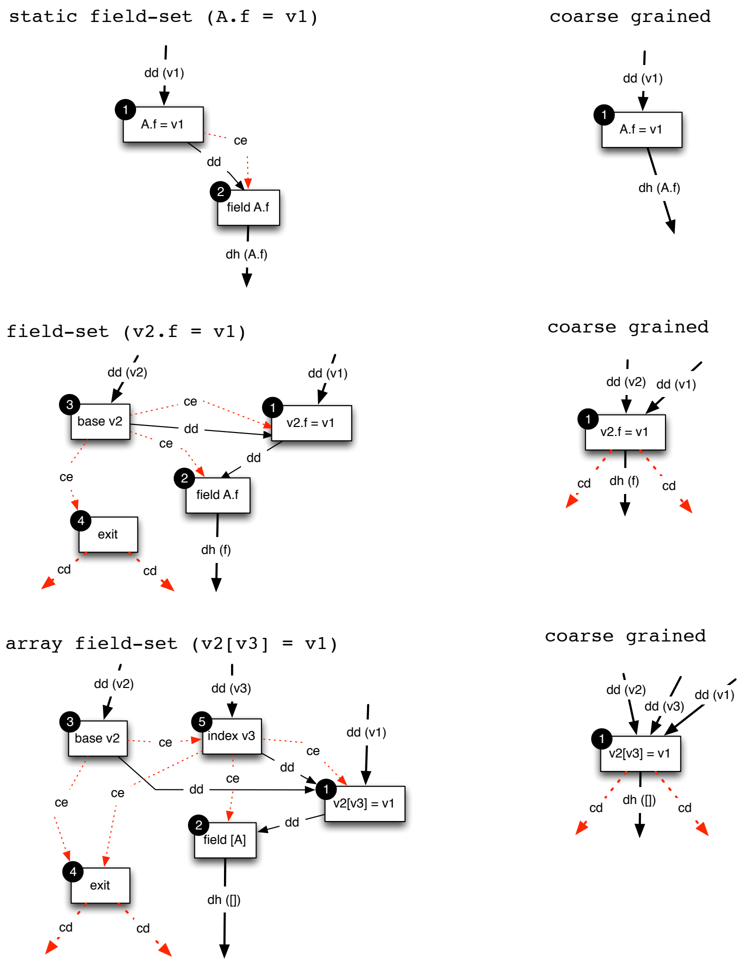


**Figure 2.32:** The control and data dependencies of fine grained field-set instructions.

## 2.5 Interprocedural analysis

This section contains the description of the interprocedural parts of our IFC analysis. Our analysis heavily relies on the results of points-to and call graph analyses, which are introduced in §2.5.1 and §2.5.2. We describe the interprocedural extension of the PDG called *system dependence graph* (SDG) in §2.5.3 and explain in detail how we model precise context-, field- and object-sensitive method side-effects with the help of artificial parameter nodes.

### 2.5.1 Call graph

A *call graph* is an approximation of all methods called during program execution. It shows which methods are executed and where they may be called from. This information is often used to lift an intraprocedural analysis to the interprocedural case: Intraprocedural analyses compute results for each single method and afterwards these results are propagated through the call graph from callee to caller until a fixed-point is reached.

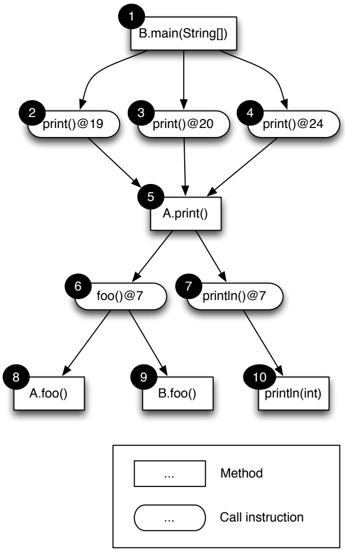
**Definition 2.12** (Call Graph). *A* call graph *of a program P is a graph $G = (N, E)$. The nodes N correspond to methods M and call instructions $I_{call}$ in P. Edges E are of the form $m_1 \xrightarrow{call} c \xrightarrow{call} m_2$ with $m_1, m_2 \in M$ and $c \in I_{call}$, where $m_1$ contains a call instruction c that may call $m_2$. We refer to the tuple $(m_1, c)$ as the* call site *of $m_2$ in $m_1$.*

The structure of a programs call graph depends on the features of programing language it has been written in. E.g. the call graph may contain cycles when recursive calls are allowed. Or —if method calls are only statically bound— each call always contains exactly a single edge to its callee. For dynamically bound methods there may be multiple potential callees. Hence the call graph for a language like Java with dynamic binding and recursive calls can become quite complex. The following example illustrates some of the common problems of call graphs for object-oriented languages.

Figure 2.33 shows a program (2.33a) and a matching call graph (2.33b). The program contains two classes `A` and `B` where `B` is a subclass of `A` that overrides the dynamically bound method `foo`. This has the effect that method `print` in l. 7 may call either `A.foo` or `B.foo` depending on the dynamic type of the **`this`** pointer. The call graph (2.33b) reflects that.

```
1  class A {
2    int foo() {
3      return 42;
4    }
5
6    void print() {
7      println(foo());
8    }
9  }
10
11 class B extends A {
12   int foo() {
13     return 23;
14   }
15
16   void main(String argv[]) {
17     A a = new A();
18     A b = new B();
19     a.print();
20     b.print();
21     if (argv[1].equals("B")) {
22       a = b;
23     }
24     a.print();
25   }
26 }
```

**(a)** program fragment



**(b)** call graph

**Figure 2.33:** A program fragment (2.33a) and its call graph (2.33b). Rectangular nodes correspond to methods, round shapes correspond to call instructions.

Node 5 corresponds to method `A.print` and node 6 represents the call to method `foo`. As both implementations of `foo` may be called, node 6 connects to `A.foo` as well as `B.foo`.

So the graph in Figure 2.33b is an approximation of all possible method calls, but it lacks precision. It does not capture the effect that a call to `A.print` on an object instance of `A` always executes `A.foo` and never `B.foo`. Like in case of the call in l. 19 that refers to an object of instance `A` in every execution of the program. The call in l. 20 is similar as it is always bound to an object of type `B`. Only the call in l. 24 may refer to instances of both classes. Thus the structure of the subgraph that represents subsequent calls from `A.print` depends on the context in which the method executes.

A common approach to increase call graph precision in these cases is to *clone* methods and subgraphs depending on their execution context.

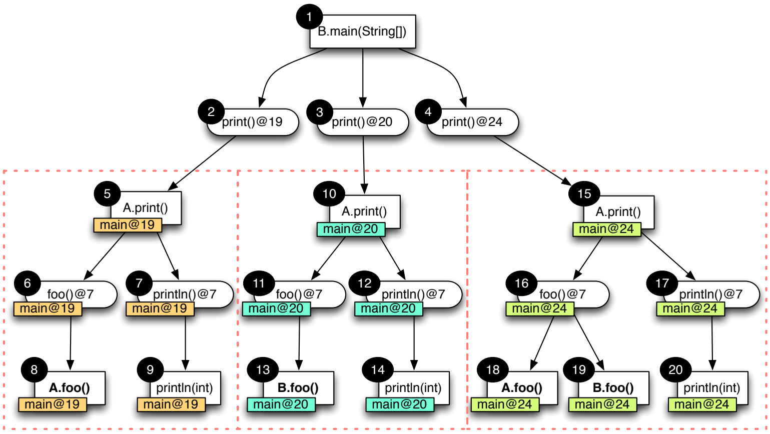


**Figure 2.34:** A call graph for the program in Figure 2.33a with increased precision through cloned methods for each call site of `A.print` in `main`.

Figure 2.34 shows a version of a call graph for the program in Figure 2.33a that cloned the subgraph of each call to `A.print`. These subgraphs no longer contain a call to `A.foo` or `B.foo` if it does not fit the execution context they belong to. E.g. the call in l. 19 is no longer connected to a node representing method `B.foo`. Cloning helps to increase call graph precision, but it comes at a steep price: Even in this small example the number of nodes doubled. In practice cloning has to be used with caution, because the resulting call graph quickly becomes too big to be computed in reasonable time and memory constraints. Cloning can also introduce unnecessary redundancies in the graph. For example the multiple cloned representations of `println` in nodes 9, 14 and 20 do not increase precision. Therefore a more selective approach that only clones relevant nodes can help.

Figure 2.35 shows a call graph that clones methods more selectively. The number of nodes is less than in the call graph with more extensive cloning (2.34), but the precision gain remains. Both graphs can distinguish the different effects of the dynamically bound call to `foo`. Our tool Joana supports call graphs with arbitrarily cloned methods. It features predefined clone strategies as well as an interface to specify custom strategies. Finding and specifying the clone strategy that offers the best

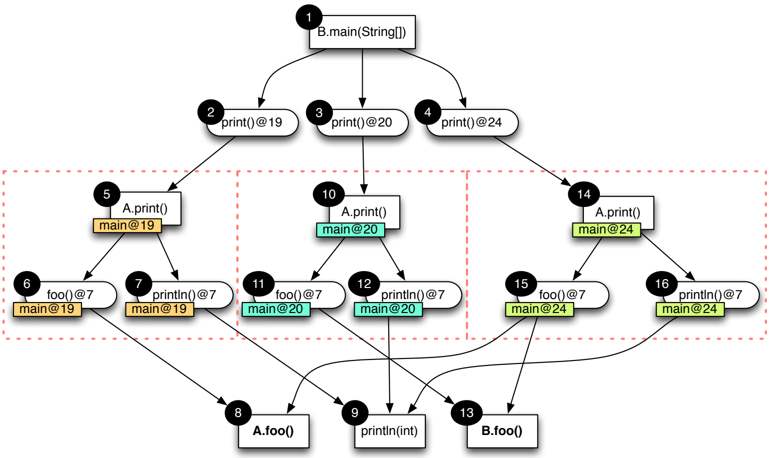


**Figure 2.35:** A call graph for the program in Figure 2.33a with increased precision through selectively cloned methods relevant to the dynamic dispatch of `foo`.

trade-off between precision and scalability is a challenging problem, because it often depends on the nature of the program under analysis and the concrete properties that should be deducible from the call graph.

We have shown what a call graph is and how its precision can be improved through cloning specific parts of the graph. So far we did not discuss how a call graph is built and which information is needed. In order to resolve dynamic dispatch it is important to know upon which concrete object instances a call can occur. This information is retrieved with the help of a so-called *points-to* analysis.

### 2.5.2 Points-to analysis

A static interprocedural IFC analysis needs to detect and overapproximate all effects that may occur during program execution. A *points-to analysis* helps to detect which memory locations may be read and modified. This information is crucial as most of the communication in an object-oriented program relies on passing references to objects, stored in memory, between program parts. In this section we will focus on points-to analysis for object-oriented languages. These points-to analyses compute which object instances references like variables or object attributes may point to. We specifically rely on the following language properties:

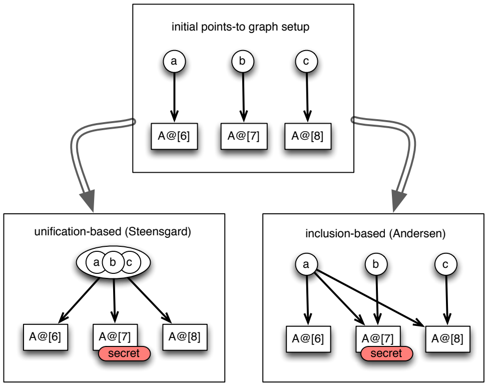
- Type correctness. References point at any time only to object instances of a matching type.
- Object instances may contain attributes referencing other objects, but pointer-to-pointer references and pointer arithmetics are forbidden.
- Only object instances can be stored on the heap and be referenced.
- Function pointers are not allowed. Dynamic calls are allowed.

Points-to analyses provide important information for many interprocedural program analyses. They have been actively researched for over 30 years and many advances were achieved in terms of scalability, precision and modularity. We will discuss more about recent advances in points-to analyses in the upcoming subsections. First we start with the two most basic algorithms invented by Andersen [4] and Steensgaard [122]. Typically all analyses are based on one of these two variants. Both compute finite sets of abstract locations, called *points-to set* (*pts*), for each reference in the program. This set includes any locations the corresponding reference may point to during program execution. The algorithms differ in the way they handle assignments between references. Given references `p` and `q` with the assignment `p = q`, Steensgaard uses an *unification-based* approach to capture the effect of the assignment: $pts'(q) = pts'(p) = pts(p) \cup pts(q)$ where $pts'$ denotes the points-to set after the execution of the assignment. Whereas Andersen proposes a more precise *inclusion-based* approach: $pts'(p) = pts(p) \cup pts(q)$ and $pts'(q) = pts(q)$. Even though Andersen approach is more precise, both approaches are relevant in practice, because Steensgaard algorithm is faster. The runtime complexity of Andersen is $O(n^3)$ while Steensgaard is almost linear with $O(n \cdot \alpha(n))$ ($\alpha$ is the inverse Ackermann function and therefore grows very slowly).

Points-to information is often represented in form of a *points-to graph*, where nodes correspond either to references or locations. References can be program variables as well as object and array fields. When a reference may point to a certain location the points-to graph contains an edge between the node of the reference and the node of the location.

```
 1 class A {
 2
 3   int data;
 4
 5   int ptsExample(int secret) {
 6     A a = new A();
 7     A b = new A();
 8     A c = new A();
 9     a = b;
10     a = c;
11     b.data = secret;
12     return c.data;
13   }
14
15 }
```

**(a)** program fragment


initial points-to graph setup
a
b
c
A@[6]
A@[7]
A@[8]
unification-based (Steensgard)
a
b
c
A@[6]
A@[7]
secret
A@[8]
inclusion-based (Andersen)
a
b
c
A@[6]
A@[7]
secret
A@[8]

**(b)** unification- and inclusion-based points-to graphs

**Figure 2.36:** A program fragment (2.36a) and two corresponding points-to graphs (2.36b).

We are going to use point-to graphs to visualize the differences between various points-to analysis algorithms, starting with the basic inclusion- and unification-based approaches.

Figure 2.36 shows the difference between these two approaches. The program fragment (2.36a) contains a single method that creates three different instances of class `A` in l. 6, l. 7 and l. 8. The initial points-to graph (2.36b) reflects the state of the program after these initializations, where the references `a`, `b` and `c` all point to a different location. Then two assignments in l. 9 and l. 10 occur. The unification-based approach approximates the effect of `a = b` by unifying the points-to sets of `a` and `b` and subsequently unifying `a` and `c` for the following assignment. The result is a points-to graph where all references point to all locations (left side of 2.36b). Hence using this less precise approach we are not able to detect that the assignment of the secret value in l. 11 has no effect on the return value, because we cannot see that `b` and `c` refer to different locations. The more precise inclusion-based approach can detect that the return value is independent of the secret value. Given the assignment `a = b` it only merges the points-to set of `b` into `a` but not vice versa. The same holds for `a = c` where also only the points-to set of `a` is adapted. The resulting points-to graph on the right side of Figure 2.36b shows that while `a` may point to all three instances, `b` and `c` point to distinct locations. Thus we can infer that modifying `b` has no effect on `c`.

**A more formal definition of a points-to analysis** A points-to analysis computes for any variable and object field in the program at which locations they may point-to during execution. The computation of points-to information can be formulated as a constraint system: The computation starts with an empty set of instances for each reference and subsequently iterates over all assignments in the program to add potential referenced instances until a fix point is reached.

Points-to analyses differ in the way they represent locations. In general it is infeasible for a static analysis to compute a precise representation for each location that may be accessed during a program run. E.g. the number of object instances created is potentially unlimited and can depend on statically unknown user input. Therefore every points-to analysis uses equivalence classes of actual heap locations, called *abstract heap locations*. As we focus on points-to analyses for object-oriented programs, all locations that can be referenced are instances of objects.

**Definition 2.13** (Abstract Heap Location). *An abstract heap location $l = T@[ctxi] \in Loc_P$ is an equivalence class of object instances of the same type $T$ created in a program state described by context ctxi.*
*Given an object instance $o_1$ the following holds:*

$$o_1 \in T@[ctxi] := o_1 \text{ is of type } T \wedge o_1 \text{ created in program state matching } ctxi$$

*We also call ctxi the* instance-context *of the object instances. The concrete description of the instance-context is part of the specific points-to analysis.*

Variables are treated in a similar fashion. They can refer to many different locations or values during a program run. The location they refer to depends on the current state of the program —the context it is evaluated in— e.g. the current call stack or the position of the statement that accesses the variable. We call a variable together with a specific description of the context a *variable reference*.

**Definition 2.14** (Variable Reference). *A variable reference $r = v@[ctxr] \in Var_P$ is a pair of the variable name $v$ and a description of the context $ctxr$.*

$$v@[ctxr] := \text{values variable } v \text{ may refer to in a program state matching } ctxr$$

*We call ctxr the* reference-context *of the variable v. The concrete description of the reference-context is part of the specific points-to analysis.*

The choice of the instance- and reference-contexts has a huge influence on the runtime of the points-to analysis and the precision of its results. We are going to show the most common precision variants of pointer analyses in §2.5.2 and explain how these contexts are defined for each variant. E.g. in the example in Figure 2.36 we used the line number of the statement that created the object as the instance-context (`A@6`, `A@7`, `A@8`) and no reference-context at all for variables (`a`, `b`, `c`).

With above definitions in place we can specify the result of a points-to analysis as a mapping between variable references and a set of abstract heap locations. This mapping tells us for each reference which locations it may point to. We call such a mapping the *points-to configuration*.

**Definition 2.15** (Points-to Configuration). *A points-to configuration C maps each program variable and object field to a set of abstract heap locations.*

$$\begin{aligned} C \quad := \quad & \{r \rightarrow l \mid r \in Var_P, l \in Loc_P \wedge r \text{ may point to } l\} \\ \cup \; & \{l.f \rightarrow l' \mid l, l' \in Loc_P \wedge x \rightarrow l \in C \wedge x.f \text{ may point to } l'\} \end{aligned}$$

*We write $pts_C(v)$ for the set of all locations variable v may point to under the given configuration.*

$$pts_C(v) := \{l \mid v'@[ctxr] \rightarrow l \in C \wedge v = v'\}$$

*We refer to $pts_C(v)$ as the points-to set of v.*

*We write $pts_C(v@[ctxr])$ for the set of all locations variable reference $v@[ctxr]$ may point to.*

$$pts_C(v@[ctxr]) := \{l \mid v'@[ctxr'] \rightarrow l \in C \wedge v = v' \wedge ctxr = ctxr'\}$$

*The points-to set of an field access $v.f_1 \ldots f_n$ is defined recursively*

$$pts_C(v.f_1 \ldots f_n) := \begin{cases} \{l \mid \exists l' \in pts_C(v) : l'.f \rightarrow l \in C\} & \text{if } n = 1 \\ \{l \mid \exists l' \in pts_C(v.f_1 \ldots f_{n-1}) : l'.f \rightarrow l \in C\} & \text{else} \end{cases}$$

A main application of points-to analysis is to decide whether two variables may point to the same location. This property is called *may-aliasing*, or simply *aliasing*. The aliasing of variables can be inferred through their points-to sets.

**Definition 2.16** (May-Alias). *Two variables or field accesses $v_1.f_1 \dots f_n$ and $v_2.f'_1 \dots f'_m$ are may-aliased under a given points-to configuration C iff their points-to sets share a common element.*

$$alias_C(v_1.f_1 \dots f_n, v_2.f'_1 \dots f'_m) := pts_C(v_1.f_1 \dots f_n) \cap pts_C(v_2.f'_1 \dots f'_m) \neq \emptyset$$

*We also write*

$$(v_1.f_1 \dots f_n, v_2.f'_1 \dots f'_m) \in C$$

*as abbreviation for*

$$alias_C(v_1.f_1 \dots f_n, v_2.f'_1 \dots f'_m)$$

May-alias information is crucial for our information flow analysis. We use it to compute potential side-effects of methods and data flow through heap locations. The precision of Joana is directly linked to the precision of the may-alias information. Imprecise may-alias information often leads to many false alarms.

### Precision

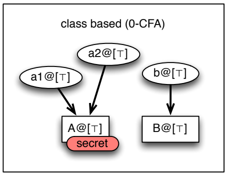
In this section we present the most common precision options of points-to and may-alias analyses and discuss how they can help to improve the results of an IFC analysis. Most of the presented options are available in our Joana tool.

**Class-based (0-CFA)** A points-to analysis is *class-based* if the different object instances are distinguished by their class name and variable references are distinguished by the variable name. It is a special form (context size $k = 0$) of the more general $k$-CFA approach we discuss in the next paragraph and is therefore also referred to as 0-CFA. Essentially the instance- as well as the reference-context is a single element.

$$ctxi_{class} = \top, ctxr_{class} = \top$$

```
1  class A { int data; }
2
3  class B {
4    int data;
5
6    int ptsClass(int secret) {
7      A a1 = new A();
8      A a2 = new A();
9      B b = new B();
10     a1.data = secret;
11     return a2.data;
12   }
13 }
```

**(a)** program fragment



**(b)** points-to graph of a class-based analysis

**Figure 2.37:** A program fragment (2.37a) and the corresponding points-to graph (2.37b) of a class-based analysis.

Figure 2.37 shows an example for a class-based points-to analysis. It contains a program (2.37a) that creates three object instances, two of type `A` and one of type `B`. The abstract heap location for the objects created in l. 7 and l. 8 is the same: $A@[\top]$. This results in the following points-to configuration:

$$C_{ptsClass} = \{a1@[\top] \rightarrow A@[\top], a2@[\top] \rightarrow A@[\top], b@[\top] \rightarrow B@[\top]\}$$

The matching points-to graph (2.37b) contains only one abstract heap location node for the two `A` object instances. This imprecision does not allow us to detect that the information from parameter `secret` is not leaked to the return value, because we cannot distinguish between the access on `a1.data` and `a2.data` as $(a1, a2) \in C_{ptsClass}$.

A class-based points-to analysis yields imprecise results which may lead to many false alarms in an information flow setting. However its computation scales very well, so even large programs with over 100kLoC can be analyzed in a reasonable amount of time. Therefore class-based points-to analyses are still relevant. They can be used whenever speed is more critical then precision or more precise analyses simply can't be computed at all.

**Call-stack (k-CFA / k-l-CFA)** The most common abstractions of heap locations and variable references are call stack based. They distinguish object instances through the instruction used to create the instance and a stack-trace of the methods that led to the execution of the instruction. Variable references are distinguished by the variable name and a stack trace of the methods that lead to the access of the variable. Because a call stack may be of arbitrary length most analyses cut the call stack after a fixed size of $k$ entries. So only the last $k$ methods called are used to distinguish created instances and references. In practice $k$ is mostly restricted to 1 or 2, as higher values lead to increased runtime and memory consumption. The part of the call stack used to identify object instances is also called its *context*. Call-stack based analyses are therefore called *context-sensitive*[12] or *k-level call stack context-sensitive*.

The standard approach for call stack context-sensitive points-to of object-oriented languages is based on the $k$-CFA[13] from Shivers [114]. Vitek et.al. [125] presented the extension for object-oriented languages called $k$-$l$-CFA, that allows different $k$- and $l$-limiting of the call stacks for variable references and object instances. If $k = l$ the result of a $k$-$l$-CFA analysis are similar to $k$-CFA. The reference- and instance-context for a $k$-$l$-level call stack analysis are chosen as follows:

$$ctxi_{\text{l-call}} = c_1 \rightarrow \ldots \rightarrow c_{l-1} \rightarrow i_l$$

The instance-context $ctxi_{\text{l-call}}$ consists of $l-1$ call instructions and a single new-instance instruction $i_l$. It applies to all object instances created by $i_l$ where $c_1, \ldots c_{l-1}$ are the last $l-1$ calls executed to reach $i_l$.

$$ctxr_{\text{k-call}} = c_1 \rightarrow \ldots \rightarrow c_k$$

The reference-context $ctxr_{\text{k-call}}$ of a variable $v$ consists of $k$ call instructions. It describes the last $k$ calls made to access $v$.

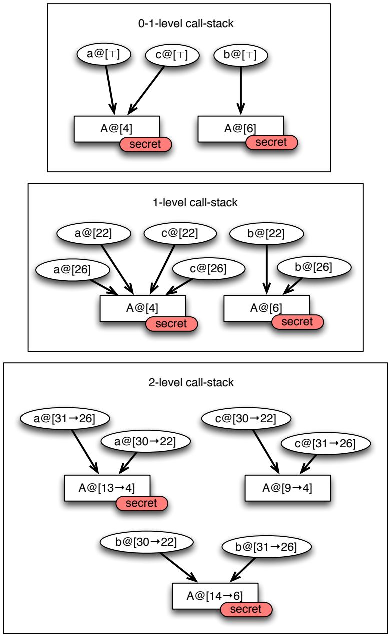
Figure 2.38 shows an example program that illustrates the difference between 1-level, 2-level and a 0-1-level call stack based points-to analysis. The program (2.38a) contains two instructions in l. 4 and l. 6 that create new instances of class `A`. During execution of `ptsStack`, 3 instances of `A` are created, two through direct calls to `create1` and `create2` and one due to an indirect call of `create1` in `callCreate`. When we use the line numbers of the call instructions as their identifier, the following points-to

[12]See §2.2.1 for a general introduction to context-sensitivity

[13]Abbreviation of *k*th-order Control Flow Analysis

```
 1 class A {
 2   int data;
 3
 4   A create1() { return new A(); }
 5
 6   A create2() { return new A(); }
 7
 8   A callCreate() {
 9     return create1();
10   }
11
12   int ptsStack(int secret) {
13     A a = create1();
14     A b = create2();
15     A c = callCreate();
16     b.data = secret;
17     a.data = secret;
18     return c.data;
19   }
20
21   int call1(int secret) {
22     return ptsStack(secret);
23   }
24
25   int call2(int secret) {
26     return ptsStack(secret);
27   }
28
29   int main(int secret) {
30     return call1(secret)
31       + call2(secret);
32   }
33 }
```

**(a)** program fragment



**(b)** three points-to graphs of a call stack based analysis

**Figure 2.38:** A program fragment (2.38a) and the corresponding points-to graphs (2.38b) of a call stack based analysis.

configurations are computed:

$$\begin{aligned} C_{\text{0-1-level}} &= \{a@[\top] \longrightarrow A@[4], b@[\top] \longrightarrow A@[6], c@[\top] \longrightarrow A@[4]\} \\ C_{\text{1-level}} &= \{a@[22] \longrightarrow A@[4], a@[26] \longrightarrow A@[4], \\ &\quad b@[22] \longrightarrow A@[6], b@[26] \longrightarrow A@[6], \\ &\quad c@[22] \longrightarrow A@[4], c@[26] \longrightarrow A@[4]\} \\ C_{\text{2-level}} &= \{a@[30{\rightarrow}22] \longrightarrow A@[13{\rightarrow}4], a@[31{\rightarrow}26] \longrightarrow A@[13{\rightarrow}4], \\ &\quad b@[30{\rightarrow}22] \longrightarrow A@[14{\rightarrow}6], b@[31{\rightarrow}26] \longrightarrow A@[14{\rightarrow}6], \\ &\quad c@[30{\rightarrow}22] \longrightarrow A@[9{\rightarrow}4], c@[31{\rightarrow}26] \longrightarrow A@[9{\rightarrow}4]\} \end{aligned}$$

| | | *k*-level variable reference call stack | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | ... | n |
| *l*-level | 0 | 0-CFA | | | |
| object | 1 | **0-1-CFA** | 1-CFA | | |
| instance | ⋮ | | | ⋱ | |
| call stack | | | | | |
| | n | | | | n-CFA |

**Table 2.2:** Points-to call stack precision options supported by Joana.

The right side of Figure 2.38b shows the three points-to graphs for 0-1-level, 1-level and 2-level call stack sensitivity. The 0-1-level analysis only keeps track of the last entry on the call stack for object instances and therefore cannot distinguish the object created through the call in l. 13 (*ptsStack* → *create*1) from the one in l. 15 (*ptsStack* → *callCreate* → *create*1). This leads to the aliasing of `a` and `c`: $(a, c) \in C_{\text{0-1-level}}$. The same holds for 1-level sensitivity: The additional context-sensitivity for variable references does not improve the precision of the result in this example. E.g. we can distinguish the value of variable `a` for the two different calls to `ptsStack`, but both abstract variable references $(a@[22], a@[26])$ point to the same abstract location $(A@[4])$. Thus 1-level sensitivity still cannot detect the absence of aliasing between `a` and `c`: $(a@[22], c@[22]), \ldots \in C_{\text{1-level}}$. Therefore an IFC analysis reports an illegal flow from parameter `secret` to the return value, as it cannot distinguish between `a.data` and `c.data`. This changes with the 2-level sensitive analysis: Different abstract heap locations exist for the instances of `A` created from the calls *callCreate* → *create*1 and *ptsStack* → *create*1. So we can detect that `a` and `c` are never aliased as $(a@[\ldots], c@[\ldots]) \notin C_{2-level}$. Therefore an IFC analysis with 2-level call stack sensitivity can verify the absence of illegal flow.

Choosing the right points-to precision is a difficult problem on its own. Sometimes the theoretically more precise and slower approach results only in minimal actual gains: In the previous example there is no difference in the results between 0-1-CFA and 1-CFA, the latter is only more complex to compute and needs more space due to the additional abstract variable references. We include several options for n-level and k-l-level call stack sensitivity in Joana—as shown in Table 2.2— so the analysis user can try which option fits best for a concrete problem. The variant of k-l-CFA in Joana is an implementation of the generalized points-to framework from Grove and Chambers [42] that comes with the WALA framework. Evaluation shows that 0-1-level sensitivity provides a good trade-off between scalability and precision, while we have to resort to class-based sensitivity to analyze the largest programs in our evaluation set. Call-stacks with $n > 1$ did only sightly improve the precision compared to its huge impact on analysis runtime.

**Receiver-object / Object-sensitive** Dynamic dispatch —also called dynamic binding— is an integral part of object-oriented languages that enables polymorphism. The call to a dynamically bound method is resolved at runtime and depends on the dynamic type of the *receiver* object. Also the methods behavior often depends on the concrete receiver object: E.g. typical getter and setter methods operate on the attributes of the receiver object. Therefore it makes sense to distinguish variable references and heap locations used in dynamic methods based on the instance of the receiver object. Such a *receiver object based points-to analysis* uses the receiver object instead of the call stack to distinguish between different calls to a method. This enables the analysis to effectively distinguish the effects of dynamically bound methods for different object instances. It is therefore also called *object-sensitive*. The respective instance- and reference-contexts are defined as follows:

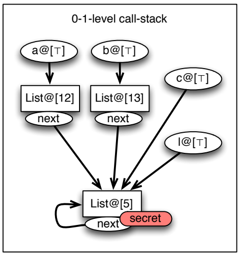
$$ctxi_{obj} = \begin{cases} T[ctxi'_{obj}], i_{new} & i_{new} \text{ is executed in a dynamic method bound} \\ & \text{to } T[ctxi'_{obj}] \\ ctxi_{static} & \text{else use fallback context} \end{cases}$$

Where $i_{new}$ is the new-instance instruction that creates the object instance and $T[ctxi'_{obj}] \in Loc_P$ is the abstract location of the receiver object for the dynamic method that was called to execute $i_{new}$. If $i_{new}$ is part of a static method another context $ctx_{static}$ —typically call stack based— is used instead.

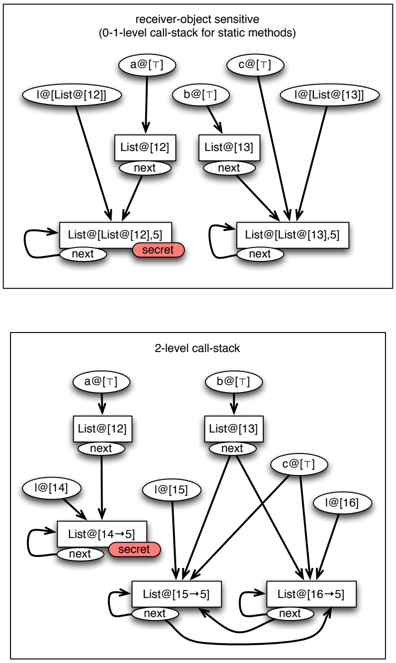
$$ctxr_{obj} = \begin{cases} T[ctxi_{obj}] & v \text{ is referenced in a dynamic method bound} \\ & \text{to } T[ctxi_{obj}] \\ ctxr_{static} & \text{else use fallback context} \end{cases}$$

```
1  class List {
2    int data; List next;
3
4    List add(int data) {
5      List l = new List();
6      l.data = data;
7      l.next = this.next;
8      this.next = l;
9    }
10
11   static int ptsReceiver(int secret) {
12     List a = new List();
13     List b = new List();
14     a.add(secret);
15     b.add(42);
16     b.add(23);
17     List c = b.next;
18     return c.data;
19   }
20 }
```

**(a)** program fragment



**(b)** 0-1-level call stack



**(c)** object and 2-level call stack sensitive

**Figure 2.39:** A program fragment (2.39a) and three matching points-to graphs of varying precision (2.39b, 2.39c).

The reference-context $ctxr_{obj}$ of a variable $v$ contains the abstract location $T[ctxi_{obj}] \in Loc_{\mathcal{P}}$ of the receiver object for the dynamic method that was called to access $v$. If $v$ is accessed in a static method another context definition $ctxr_{static}$ has to be used as a fallback, like the previously introduced call stack sensitive reference-context.

Figure 2.39 contains an example (2.39a) that illustrates the benefits of a object-sensitive analysis over a call stack sensitive approach in a typical situation. The program uses two different `List` objects to store secret (l. 14) as well as non-secret (l. 15, l. 16) data. Both times the same method `add` creates a new entry and adds it to the desired list. A simple 0-1-level or 1-level call stack based analysis cannot distinguish between the entry created in l. 5 for list `a` and the one created for list `b`, because both are created by the same new-instance call. Thus we need at least 2-level call stack sensitivity, but this comes at the price of additional analysis complexity. A receiver-object-sensitive analysis offers a good trade-off in these situations. It does not create distinct variable references and abstract locations for each separate call —only for those that refer to different receiver objects. E.g. the calls to `b.add` in l. 15, l. 16 are not treated separately —like they are with 2-level call stack sensitivity—, but the call to `a.add` is. So the object-sensitive as well as the 2-level call stack sensitive approach can detect that the secret is never leaked to the return value of `ptsReceiver`, while the object-sensitive approach requires less computation time and space. This distinction also shows in the resulting points-to configurations.

$$\begin{aligned}C_{\text{0-1-call}} = \{&a@[\top] \to List@[12], b@[\top] \to List@[13], c@[\top] \to List@[5],\\ &l@[\top] \to List@[5], List@[12].next \to List@[5],\\ &List@[13].next \to List@[5], List@[5].next \to List@[5]\}\end{aligned}$$

$$\begin{aligned}C_{\text{obj}} = \{&a@[\top] \to List@[12], b@[\top] \to List@[13], c@[\top] \to List@[\text{List@[13],5}],\\ &List@[12].next \to List@[\text{List@[12],5}], l@[List@[12]] \to List@[\text{List@[12],5}],\\ &List@[13].next \to List@[\text{List@[13],5}], l@[List@[13]] \to List@[\text{List@[13],5}],\\ &List@[\text{List@[12],5}].next \to List@[\text{List@[12],5}],\\ &List@[\text{List@[13],5}].next \to List@[\text{List@[13],5}]\}\end{aligned}$$

$$\begin{aligned}C_{\text{2-call}} = \{&a@[\top] \to List@[12], b@[\top] \to List@[13], c@[\top] \to List@[15{\to}5],\\ &c@[\top] \to List@[16{\to}5], List@[12].next \to List@[14{\to}5],\\ &l@[14] \to List@[14{\to}5], List@[13].next \to List@[15{\to}5],\\ &l@[15] \to List@[15{\to}5], List@[13].next \to List@[16{\to}5],\\ &l@[16] \to List@[16{\to}5], List@[14{\to}5].next \to List@[14{\to}5],\\ &List@[15{\to}5].next \to List@[15{\to}5], List@[16{\to}5].next \to List@[16{\to}5],\\ &List@[15{\to}5].next \to List@[16{\to}5], List@[15{\to}5].next \to List@[16{\to}5]\}\end{aligned}$$

The corresponding points-to graphs are in Figure 2.39b and Figure 2.39c. While the result for 0-1-level call stack sensitivity is quite small, it is not precise enough to reveal that `c.data` does not contain secret information: $(a.next, c) \in C_{\text{0-1-call}}$. The results of the object-sensitive and 2-level call stack sensitive analysis are precise enough ($(a.next, c) \notin C_{\text{obj}}$, $(a.next, c) \notin C_{\text{2-call}}$), but the call stack result contains an unnecessary distinction between the list elements created from l. 15 and l. 16. The size difference of the points-to configurations is not so huge in this example —as we kept it deliberately small— but it can quickly become worse in real programs. Additional call sites blow up the result quite easily, while they are merged by the object-sensitive approach as long as they are called on the same receiver instance.

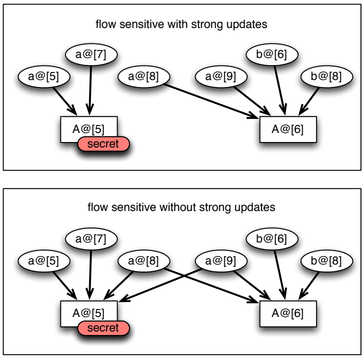
Joana provides support for object-sensitive points-to analysis. Currently it is often the best option for precise points-to analysis available in Joana. While $n$-CFA can be more precise (for $n > 1$) in certain situations, it scales worse and in typical object-oriented programs the object sensitivity does a better job at only distinguishing the relevant parts [91,80]. Evaluation shows that object sensitivity still is quite time and memory consuming, but the increased precision is very much noticeable in an IFC setting. E.g. noninterference of the programs in §4.2 and §4.4 could only be verified with the help of object sensitivity.

**Flow-sensitivity and strong updates** The concept of flow-sensitivity has already been explained in §2.2.4. A flow-sensitive points-to analysis can distinguish between various values of the same variable depending on the point in the control flow where the variable is accessed. It is often used in conjunction with *strong updates* that enable the analysis to detect when the value of a variable has been overwritten.

We use the example in Figure 2.40 to explain how flow-sensitive points-to works in general and subsequently show how it can be combined with strong updates. A flow-sensitive points-to analysis can detect that variable `a` only points-to the instance created in l. 6 after the assignment in l. 8. Therefore the corresponding points-to graph (bottom part of Figure 2.40b) contains multiple nodes for a single variable, depending on the position of the variable in the control flow. There are 4 nodes for variable `a`, one for every usage in l. 5, l. 7, l. 8 and l. 9. The first two represent the object referenced before the assignment in l. 8. They only point to the instance created in l. 5. After the assignment variable `a` points to the instance created in l. 6 and the remaining two nodes reflect that: They potentially point to both instances of `A`.

```
1  class A {
2    int data;
3
4    int ptsFlow(int secret) {
5      A a = new A();
6      A b = new A();
7      a.data = secret;
8      a = b;
9      return a.data;
10   }
11 }
```

**(a)** program fragment



**(b)** two points-to graphs of a flow-sensitive analysis with and without strong updates

**Figure 2.40:** A program fragment (2.40a) and the corresponding points-to graphs (2.40b) of a flow-sensitive analysis with and without strong updates.

However variable a does no longer point to the first instance of A beyond l. 8. A standard inclusion-based approach cannot capture this property, because it merges the points-to sets at the variable assignment ($pts(a)' = pts(a) \cup pts(b)$) instead of replacing it ($pts(a)' = pts(b)$). This behavior is necessary to retrieve valid results for a flow-insensitive points-to analysis, because the points-to set of variable a has to reflect the potential referenced objects at any point in the program, including the parts before l. 8. A flow-sensitive points-to analysis on the other hand distinguishes the various occurrences of variable a and therefore can remove the first instance of A from all points-to sets of variable a after the assignment in l. 8. It performs a *strong update* on the points-to set for a. The top part of Figure 2.40b shows the corresponding points-go graph of a flow-sensitive analysis with strong updates. The variable references $a$@[8], $a$@[9] no longer point to $A$@[5]. The increased precision enables an IFC analysis to detect the absence of information flow from l. 7 to 9 and verify that the value of secret is not leaked.

Joana does not support flow-sensitive points-to analyses with strong updates at this point, we are also not aware of any other IFC analysis tool that does so. However it contains support for an intraprocedural dataflow analysis that detects killing definitions which in many cases produces similar precision enhancements. For example the absence of information flow in the example from Figure 2.40 can also be detected. Additionally —as our analysis builds upon the SSA intermediate representation of the program— we can take advantage of its inherent flow-sensitivity for local variables. Programs are transformed to SSA form through renaming local variables in a way that the value of every variable is only set at a single point in the program. Thus a local variable may correspond to multiple SSA variables depending on which value they refer to at the current point in the program. This effectively turns a flow-insensitive analysis on the SSA form of a program into a flow-sensitive analysis for the original code which is a great advantage of using SSA form for program analysis. However this flow sensitivity does not translate to object fields and static variables, as they are not renamed in standard SSA form. So a full flow-sensitive points-to analysis can still provide improved results.

Flow-sensitive points-to analyses are in general very time and memory consuming. They have been researched for several years and an approach that uses binary decision diagrams (BDDs) seems promising [11, 81]. But in combination with a standard points-to analysis approach —that operates as an iterative dataflow problem on the control flow graph— the analysis is still too complex, so that trade-offs have to be made [136]. Nevertheless Markus Herhoffer incorporated BDD-based points-to analyses into Joana [53] with the help of the DOOP pointer analysis framework [17]. We noticed that additional precision gains are possible, but the scalability improvement through BDDs was not as large as expected. Basically a faster computation time was traded for huge memory and disc space consumption (>100GB of disk space for 100kLoC programs). Also, the DOOP framework relies on a proprietary datalog engine that is not distributed freely and needs to be licensed. Therefore we did opt to not include it into Joana.

So far we have not tried (semi-)sparse flow-sensitive points-to analysis with BDDs [48,49]. They do not operate on the control flow graph like traditional pointer analyses, but rather propagate their intermediate results only along data dependencies. This helps to limit the amount of work as the intermediate results are only propagated to potentially relevant program parts. However they need to run an initial (fast and less precise) points-to analysis in order to compute those data dependencies.

Demand driven points-to analysis uses a similar trick [104,52,121]. It applies a fast and less precise points-to analysis on the whole program for its initial results and runs a more sophisticated analysis on parts of the program only upon request. This allows for a quick initial result were precision can be added later on as needed. A demand driven points-to analysis is included in the WALA framework we build upon. So it is easily available for Joana. Due to the nature of our IFC analysis —we compute a SDG model for the whole program independent of the specific IFC property we want to show— we need to request a precise analysis for any part of the program. This works against the demand driven approach that produces the best results when the precise analysis is only needed for a some parts of the program. However as future work we can try to use the knowledge about the IFC properties already in the SDG computation phase. This enables us to detect which parts of the program are of interest and issue a request for a precise analysis only for these parts.

Recent work [79] combines flow-insensitive with flow-sensitive analysis for special cases in order to enable strong updates without sacrificing scalability. This approach may prove helpful in an IFC setting and its integration in Joana is regarded as future work. In general mixtures between different approaches and precision levels seem to yield the best results in practice. The next section provides a short overview of points-to analyses with mixed sensitivity options that are also available in Joana.

**Mixing different approaches** Different approaches and sensitivity options can be mixed to achieve better analysis results for a given program [90,62]. We present 3 different mixed approaches that are integrated in Joana and have shown to be useful in practice.

**selective class-based with threshold** Typically instances of classes like `String` or `Integer` are created at many points in a program. So when we want to use a precise context-sensitive instance-based points-to analysis the huge number of distinguishable instances tend to compromise the overall performance of the analysis. Falling back to a class-based approach resolves the scalability issues, but also reduces precision. A popular solution is to use *selective class-based* analysis for classes whose instance count exceeds a predefined threshold, e.g.

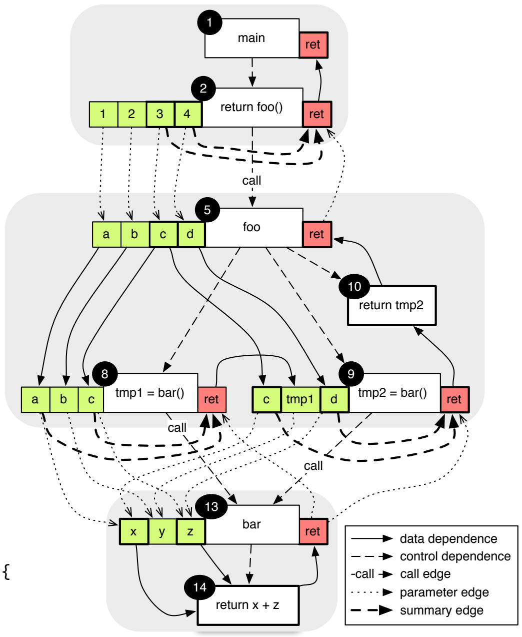
more than 100 instances. So if a certain type is instanced in more then 100 distinguishable contexts, the abstract heap locations are merged to a single location with a class-based context $\top$. This helps to reduce the load on the points-to analysis from heavily used types that often stem from code in the standard library. So the arguably more important object instances of the application specific code are still analyzed with full precision, while only the heavily used types take a hit in precision for the sake of better analysis scalability.

**selective object sensitivity for container classes** The Java library provides a set of predefined container classes such as `LinkedList` or `HashMap`. These classes are often used in programs to store all kind of different objects. This proves to be a burden for points-to analyses, because they have to keep track of which container instance is used to store which data object. Even a 1-level call site sensitive analysis is not enough to achieve the required precision as the internals of container classes are modified indirectly through access methods like `add` or `remove`. However, an object-sensitive analysis —distinguishing calls to these access methods by their receiver instance— is precise enough. As full object sensitivity is often too complex for larger programs a reasonable trade-off is to analyze only container classes with object-sensitivity and default to a less complex sensitivity option for other classes.

**selective object sensitivity for threads** Selective object sensitivity is also used for `java.lang.Thread` and its subclasses. This is necessary due to the way threads are created in Java. To create a thread a programmer declares a subclass of `Thread` and overrides its `run` method with the code he wants the new thread to execute. Then he creates a new instance and calls the inherited method `start` upon this instance. The method spawns a new system level thread that executes the `run` method of the subclass. This setup is problematic for a points-to analysis, because every thread has to call the same inherited `start` method. Without knowing any context `start` potentially may call any `run` method from all threads used in the program. For a precise result the analysis needs to keep track upon which object the `start` method has been called in order to know which `run` method will be executed.

```
1  int main() {
2    return foo(1, 2, 3, 4);
3  }
4
5  int foo(int a, int b,
6          int c, int d)
7  {
8    int tmp1 = bar(a, b, c);
9    int tmp2 = bar(c, tmp1, d);
10   return tmp2;
11 }
12
13 int bar(int x, int y, int z) {
14   return x + z;
15 }
```

**(a)** Program



**(b)** System Dependence Graph

**Figure 2.41:** A program with multiple methods (2.41a) and its corresponding system dependence graph (2.41b).

## 2.5.3 System dependence graph

The system dependence graph (SDG) [56] is the interprocedural extension of the procedure dependence graph (PDG) from §2.3.6 Definition 2.8. It captures the semantics of a whole program and contains a PDG for each method.

**Definition 2.17** (System Dependence Graph). *A system dependence graph $G = (N, E)$ for a program $p$ is a directed graph, where the nodes in $N$ represent statements in $p$, and the edges in $E$ represent dependencies between them [56]. The SDG is partitioned into* procedure dependence graphs *(PDG) that model single procedures.*

*The PDGs are connected at* call sites, *consisting of a call node $c$ that is* *connected with the entry node e of the called procedure through a* call edge $c \xrightarrow{call} e$. *All values and memory locations the called procedure may reference or modify are modeled via synthetic* parameter nodes *(Definition 2.5) and* edges*:*

**referenced values and locations** *For each referenced value and location there exists an* actual-in node $a_i$ *at the call site at node c and a matching* formal-in node $f_i$ *at the entry e of the callee. A* parameter-in edge $a_i \xrightarrow{pi} f_i$ *models the information flow into the callee.*

**modified values and locations** *Values and locations modified by the callee are represented by a* formal-out node $f_o$ *at e and a matching* actual-out node $a_o$ *at the call site c. The information flow from the callee to the call site is modeled with a* parameter-out edge $f_o \xrightarrow{po} a_o$.

*Formal-in and formal-out nodes are control dependent on entry node e, actual-in and actual-out nodes are control dependent on call node c. So-called* summary edges *between actual-in and actual-out nodes of one call site represent transitive flow from a parameter to a return value in the called procedure.*

SDGs enable us to analyze program properties via graph traversal. Moreover, they are purpose-built for *context-sensitive* analyses, which distinguish different invocations of the same procedure. Figure 2.41 contains an example for an SDG composed of multiple PDGs with different call sites. The program consists of 3 methods `foo`, `bar` and the `main` method, thus the SDG also consists of 3 PDGs. Each separate PDG is marked with a gray background. Method `main` calls `foo` — `bar` is called at two different call sites within `foo`. Therefore the SDG contains 3 call edges: $2 \xrightarrow{call} 5$, $8 \xrightarrow{call} 13$ and $9 \xrightarrow{call} 13$. At each call site the values passed to the called method are modeled as actual-in nodes on the left side of the call node. The actual-out node for values returned from the method are on the right side of each call node. The formal-in and -out nodes are placed accordingly at each entry node. Passing values between call and called method is modeled with parameter edges. Aside from call and parameter edges no other edges occur between nodes of different PDGs[14]. This allows us to keep track of the calling context under which an information flow occurs.

[14]This changes with the introduction of interference edges for the multithreaded extension of the SDG.

For example if we want to detect which inputs of the call to `foo` in `main` can influence the return value of `main`, we can directly observe the effect of context-sensitivity on the analysis result:

A naive *context-insensitive* approach applies a simple backwards reachability analysis from the formal-out `ret` of the entry node for `main`. Going backwards from the formal-out of `main` we reach the formal-out of method `bar` through 13→9→10→5→2→1. With the dependencies inside `bar` we reach formal-in nodes `x` and `z`. At this point a simple reachability analysis achieves a less precise result due to the lack of context-sensitivity: From the formal-in nodes of `bar` we can reach the actual-in nodes of both calls from node 8 and 9. So we reach formal-ins `a`, `c` and `d` of `foo` and subsequently detect that the first, third and fourth parameter of the call to `foo` in `main` influence the return value. This result is a conservative approximation of the actual dependencies, as the first parameter in fact has no influence. We can do better with a context-sensitive reachability analysis.

The *context-sensitive* extension of the backwards reachability analysis keeps track of the call site through which a PDG was entered and only traces dependencies back to this call site. This extension is in effect when the formal-in nodes of node 13 are reached. We entered method `bar` through the call in node 9, therefore we only trace back dependencies that return to the same call: Actual-ins `c` and `d` of node 9. We ignore the parameter-in dependencies from the call in node 8. The result is that parameter 1 of the call to `foo` can no longer be reached and therefore only parameter 3 and 4 influence the return value.

In the current example the context-sensitive reachability analysis increases the precision of the result, but it also comes at a price. Keeping track of the context through which a PDG has been entered can become a complex task, given that SDGs are often quite large graphs with thousands of nodes and hundreds of PDGs and call sites. Therefore SDGs include additional so-called *summary edges* between the actual-in and -out nodes of each call. These edges summarize the information flow from the input of the call to its output. Figure 2.41b already contains summary edges at the 3 call sites. They allow us to compute context-sensitive reachability in linear time with a special two-phase algorithm [56]. This algorithm from Horwitz, Reps and Binkley is generally referred to as *HRB slicing* or *context-sensitive slicing*. The downside of HRB slicing is that it depends on summary edges, which are quite complex to compute. A detailed description of the summary edge computation and HRB slicing can be found in the upcoming section §2.5.4.

**SDGs for object-oriented languages** We focus on the computation of SDGs for object-oriented languages. As previously explained, challenges arising from object-orientation are *object- and field-sensitivity*, *exceptions*, *dynamic dispatch* and *objects as parameters*. These features lead to statically undecidable problems that are commonly approximated with the help of a *points-to* analysis as presented in the previous section §2.5.2. One usage of points-to information is to resolve dynamic dispatch. Once it is known to which objects a reference may point to at runtime, one can determine the possible target methods of a dynamic dispatch. Points-to information is also used to achieve object- and field-sensitivity: It is possible to model two objects of the same type separately in the SDG if they are not may-aliasing.

Another application of points-to information is the precise computation of method side-effects. It is used to determine which object fields may be read or modified during method execution and to create the synthetic parameter nodes for these fields in the SDG. The computation of these synthetic nodes has a big influence on the scalability of the SDG computation as a whole, which is investigated in the subsequent section §2.6.

**Improving Scalability** SDGs are built through an intraprocedural phase that covers local control and data dependencies —basically computing a PDG for each method in isolation— and an interprocedural phase that combines the intraprocedural results and models their effects on the global state of the program. The interprocedural phase consists roughly of four steps:

1. Call graph and points-to computation
2. Computation of additional parameter nodes arising from method side-effects
3. Computation of data dependencies for these new parameter nodes
4. Summary edge computation

Step (1) is a prerequisite for the intraprocedural phase, because the call graph is used to detect which methods are called and therefore which PDGs need to be computed. It it is also used to resolve dynamic binding at call sites and to compute method side-effects in step (2). Steps (3) and (4) depend on the results of step (2), because they operate on the additionally created parameter nodes.

Our evaluation revealed that the intraprocedural phase has no scalability issues, whereas the interprocedural phase is the main reason for long runtime and huge memory consumption. The first step of the interprocedural phase is a well-known and extensively researched problem — as many other analyses aside from SDG construction depend on points-to and call graph information. In the previous sections §2.5.1 and §2.5.2 we already showed that there are various options available that let us choose between precision and scalability. The remaining steps are more specific to SDG computation. The step (2) parameter computation seems to be crucial, because its results are used in the subsequent steps (3) and (4). When we apply the parameter computation introduced by Hammer [46] a scalability problem occurs: A faster and less precise point-to analysis in step (1) often leads to greater memory consumption and a longer overall runtime in the subsequent steps — preventing the analysis of larger programs. We determined the way how additional parameters are computed to be responsible. In general a less precise points-to information resulted in a larger number of additionally created parameter nodes and slowed down the subsequent data dependency and summary edge computations. We take a closer look at the computation of those additional parameters and explain the observed behavior in section 2.6. We also introduce our own parameter computation algorithm that fixes this problem.

### 2.5.4 Summary edges and HRB slicing

Summary edges are a crucial prerequisite for the HRB slicer that allows context-sensitive reachability analysis in linear time. They are computed in the last step of the interprocedural SDG computation. They have first been proposed by Horwitz et al. [56] with a computation algorithm based on attribute grammars. The computation algorithm has been significantly improved by Reps et al. [105] to a runtime complexity of $O(n^3)$ where $n$ is the number of nodes in the SDG. Algorithm 2.1 describes an optimized version of the summary computation that has been tweaked with an additional map fragmentPath that stores intermediate results, sacrificing memory usage for improved runtime [44].

**Algorithm 2.1** (Summary edge computation).

```
PROCEDURE ComputeSummaryEdges($sdg$)
INPUT:  A system dependence graph $sdg$
OUTPUT: Set of summary edges $summary$
BEGIN
  Set<Edge> PathEdge = $\emptyset$, $summary$ = $\emptyset$, WorkList = $\emptyset$
  Map<Node, Set<Edge>> fragmentPath = new empty map
  FOREACH (formal-out node $w \in sdg$) DO
    PathEdge = PathEdge $\cup\{w \to w\}$
    WorkList = WorkList $\cup\{w \to w\}$
  DONE

  WHILE (WorkList $\neq \emptyset$) DO
    WorkList = WorkList $\setminus\{v \to w\}$
    IF ($v$ is actual-out) THEN
      FOREACH ($x$ such that $x \to v \in summary \vee x \xrightarrow{cd} v \in sdg$) DO
        Propagate($x \to w$)
      DONE
    ELSE IF ($v$ is formal-in) THEN
      FOREACH (call node $c$ with $\exists c \xrightarrow{call} Entry(w) \in sdg$) DO
        $x$ = matching actual-in for $v$ at call $c$
        $y$ = matching actual-out for $w$ at call $c$
        $summary$ = $summary \cup \{x \to y\}$
        FOREACH ($a$ such that $y \to a \in$ fragmentPath($y$)) DO
          Propagate($x \to a$)
        DONE
      DONE
    ELSE
      FOREACH ($x$ with $x \xrightarrow{dd} v, x \xrightarrow{dh} v$ or $x \xrightarrow{cd} v \in sdg$) DO
        IF (NOT ($x \xrightarrow{cd} v$ and $x$ and $v$ both parameter nodes)) THEN
          Propagate($x \to w$)
        FI
      DONE
    FI
  DONE
  RETURN $summary$
END

PROCEDURE Propagate($e$)
INPUT:  A sdg edge $e = v \to w$
OUTPUT: Potentially added edge $e$ to WorkList, PathEdge and fragmentPath($w$)
BEGIN
  IF ($e \notin$ PathEdge) THEN
    PathEdge = PathEdge $\cup\{e\}$
    WorkList = WorkList $\cup\{e\}$
    IF ($v$ is actual-out) THEN
      fragmentPath($w$) = fragmentPath($w$) $\cup\{e\}$
    FI
  FI
END
```

The summary edge computation in Algorithm 2.1 computes for each formal-out node which formal-in nodes of the same method can reach it through a same-level path. This information is transferred to summary edges between the matching actual-in and -out nodes at the corresponding call sites. The algorithm is described in detail in [105] and evaluated in [70]. We include the pseudocode of the algorithm in this thesis as Chapter 3 contains an extension of this algorithm in §3.4.4. The extension is tailored for modular analysis and incorporates precomputed information into the computation in order to improve computation time.

We introduced slicing in §1.2. In general a program slice is a set of SDG nodes that potentially can influence the *slicing criterion*. The slicing criterion is also a set of nodes — in an IFC setting these are often program statements related to public output and we want to detect if some statement that handles secret information is in the slice. A program slicer uses a reachability check on the dependence graph in order to check which statements can be influenced. Ideally this check should be context-sensitive to minimize false alarms. The standard approach to context-sensitive slicing is the so-called HRB slicer [56] invented by Horwitz, Reps and Binkley.

The HRB slicer in Algorithm 2.2 takes a SDG with summary edges and a set of nodes called slicing criterion. It computes context-sensitive reachability for all nodes in the criterion in $O(\#\text{edges})$. The algorithm consists of two phases. Phase 1 traverses the graph "upwards" only ascending to calling methods, while using summary edges to bypass call statements. The actual-out nodes of call statements are connected to corresponding formal-out nodes of the callees through parameter-out edges. These formal-outs are stored for later usage in phase 2 in list $W_2$. Phase 2 then traverses the graph "downwards" only descending into called methods. It uses all nodes in $W_2$ as a starting point. The result is a context-sensitive slice for the given criterion.

**Algorithm 2.2** (Two-phase HRB slicer). *From Horwitz, Reps and Binkley [56] with an asymptotic runtime of $O(\#edges\ in\ sdg)$.*

```
PROCEDURE hrbSlice($sdg$, $crit$)
INPUT:
  A system dependence graph with summary edges $sdg$
  A set of nodes as slicing criterion $crit$
OUTPUT:
  The slice for the criterion $crit$: $slice$
BEGIN
  // two worklists and the result set
  $W_1 = \{n \mid n \in crit\}$, $W_2 = \{\}$, $slice = \{n \mid n \in crit\}$

  /* phase 1 */
  DO
    $W_1 = W_1 \setminus \{n\}$ // process the next node in $W_1$
    // handle all incoming edges of n
    FORALL ($n' \xrightarrow{e} n \in sdg$) DO
      // $n'$ has not been visited yet
      IF ($n' \notin slice$) THEN
        $slice = slice \cup \{n'\}$
      FI
      // if $e$ is not a parameter-out edge, add $n'$ to $W_1$, otherwise, add $n'$ to $W_2$
      IF ($e$ is parameter-out) THEN
        $W_1 = W_1 \cup \{n'\}$
      ELSE
        $W_2 = W_2 \cup \{n'\}$
      FI
    DONE
  WHILE ($W_1 \neq \emptyset$)

  /* phase 2 */
  DO
    $W_2 = W_2 \setminus \{n\}$ // process the next node in $W_2$
    // handle all incoming edges of $n$
    FORALL ($n' \xrightarrow{e} n \in sdg$)  DO
      // $n'$ has not been visited yet
      IF ($n' \notin slice$) THEN
        // if $e$ is not parameter-in or call, add $n'$ to $W_2$ and to the slice
        IF ($e$ is parameter-in or call) THEN
          $W_2 = W_2 \cup \{n'\}$
          $slice = slice \cup \{n'\}$
        FI
      FI
    DONE
  WHILE ($W_2 \neq \emptyset$)

  RETURN $slice$
END
```

## 2.6 Parameter-model

In an object-oriented language such as Java method calls can have side-effects that change the state of an object passed as a parameter — e.g. adding an element to an existing list or setting a new value for a field. These side-effects can influence the program behavior at the call site and therefore need to be modeled in the SDG. The *parameter model* describes how these side-effects are represented in the SDG. In general they are represented through additional parameter nodes at the call site and the PDG of the callee. The choice of the parameter model has a huge impact on the precision, scalability and soundness of the resulting SDG. In our work we build upon the work of Liang and Harrold [82] as well as Hammer [46] who suggested using *object-trees* to model objects passed as parameters, where a node represents an object and its children represent its fields. While this model presents a sound and precise approximation of the side-effects, we show that object-trees cause severe performance problems: Their size grows with declining precision (and thus improved runtime) of the points-to analysis. An imprecise points-to analysis leads to huge object-trees, reducing the performance of SDG computation. In order to get the object-tree sizes under control, one has to employ highly precise points-to analyses, which do not scale for larger programs. We suggest to model parameters as *object-graphs* [36, 35] instead of trees and merge duplicate information from different subtrees into a single representation.

In this section we introduce object-graphs as a new parameter model for SDGs of object-oriented languages. We explain in detail how object-graphs are computed and show their scalability benefits in an evaluation that compares the results to other parameter models. Therefore we also briefly introduce alternative parameter models such as object-trees and the unstructured instruction-based approach as used in the SDG computation that comes with the WALA framework. Our evaluation consists of 20 small (100LoC) to medium (60kLoC) sized programs and covers various options: Four points-to analyses provide different levels of precision. The results show that object-graphs are the best choice for imprecise points-to analyses and also work well with more precise points-to information. In general points-to precision in most cases has a moderate influence on the overall precision —measured through the average size of a slice— (around 4%, but up to 24%) while it has a huge

```
 1 class Data {
 2   Node list1 = new Node();
 3   Node list2 = new Node();
 4 }
 5
 6 class Node {
 7   int i = 1;
 8   Node next;
 9 }
10
11 static void main() {
12   Data d = new Data();
13   append(d.list1, 4);
14   d.list2 = d.list1;
15   int sum = sumData(d);
16   println(sum);
17 }
18
19 static int sumData(Data d) {
20   Node sum1 = sum(d.list1);
21   Node sum2 = sum(d.list2);
22   return sum1.i + sum2.i;
23 }
24 static void append(Node node, int len) {
25   for (int i = 0; i < len; i++) {
26     node.next = new Node();
27     node = node.next;
28   }
29 }
30
31 static Node sum(Node n) {
32   int sum = 0;
33   while (n != null) {
34     sum += n.i;
35     n = n.next;
36   }
37   Node res = new Node();
38   res.i = sum;
39   return res;
40 }
```

**Figure 2.42:** Small program performing list operations with recursive data structures. It contains side-effects and aliasing to showcase the differences between the parameter models.

impact on the analysis runtime.

In the following we start with the description and overview of three different parameter models —unstructured, object-tree and object-graph— before we continue with a detailed examination of the differences between object-tree and -graph and their computation. We conclude with an evaluation and a discussion of the results.

**Common example** Figure 2.42 shows a program that builds a data structure with two single-linked lists of integers and subsequently computes the sum of all entries in both. We use this example to showcase the differences the parameter model makes on the structure of the resulting SDG. The program starts in l. 11, it creates a new `Data` object and appends 4 new elements to `list1` in l. 13. The assignment in l. 14 introduces aliasing between `list1` and `list2`. Finally the sum of both lists is computed and printed to console. During the computation method `sum` creates a new `Node` instance to hold the result. This temporary object never leaves the boundaries of `sumData`. The object-tree and -graph

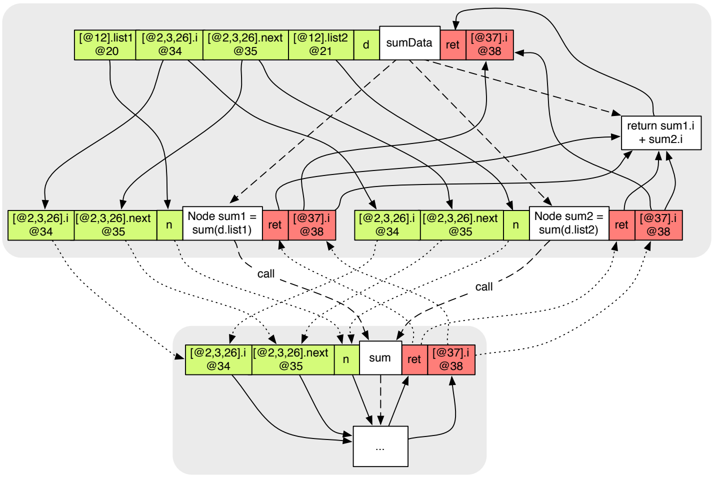


**Figure 2.43:** Part of the SDG with an unstructured parameter model for the example in Figure 2.42 analyzed with a context-insensitive points-to analysis.

approach can detect that `sumData` has no visible side-effects and therefore no additional parameter nodes need to be created for it. The unstructured parameter model however creates unnecessary nodes. The differences between object-tree and -graph model is visible in the way they model nodes for the two aliased lists.

### 2.6.1 Unstructured model

The unstructured parameter model is closely tied to the points-to analysis used. It creates a new parameter node for each distinguishable memory location that may be accessed during method execution. The amount of distinguishable locations depends on the precision of the points-to analysis: The more precise it is, the more nodes will be created. Its main advantage is that it is easy to compute which nodes need to be created — a simple reachability analysis on the call graph can detect which methods may be called transitively and a one-time pass through the instructions of each method suffices to extract points-to information for all encountered field access instructions.

Figure 2.43 shows a part of the SDG for the common example

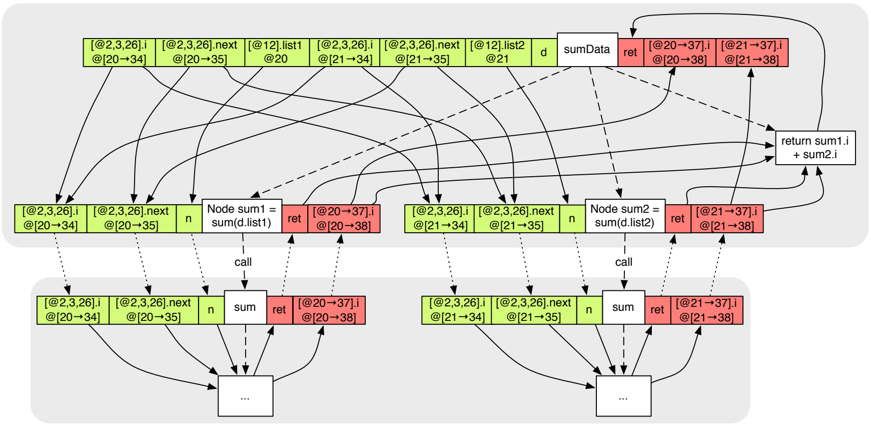


**Figure 2.44:** Part of the SDG with an unstructured parameter model for the example in Figure 2.42 analyzed with a 1-level call stack sensitive points-to analysis.

in Figure 2.42. The SDG contains parameter nodes computed with the unstructured parameter model and a context-insensitive points-to analysis. The part shown focuses on method `sumData` and its calls to `sum`. The additional parameter nodes created with the unstructured approach are labeled with the line numbers of the object instantiation sites and field access instructions that triggered their creation. E.g. the input parameter labeled "*[@2,3,26].next@35*" refers to the access of field `next` in l. 35 on the object created either in l. 2, l. 3 or l. 26. Method `sum` reads from fields `i` and `next` in l. 34 and l. 35. Thus two additional input-parameter nodes are created. Method `sum` also creates a new instance of `Node` and writes the computed sum to field `i`. This is represented by an additional output-parameter node. These nodes are propagated through the call sites to `sumData`. In addition to the propagated nodes, `sumData` also has two input-parameter nodes for the field accesses to `list1` and `list2`. In this example the unstructured parameter model results in a total of 5 additional parameter nodes — 4 input and 1 output node — for method `sumData`.

This approach is used by the SDG builder that comes with the WALA framework. Its computation scales quite well in theory, but in practice it is not very usable — especially for precise points-to analyses where it creates a large amount of additional parameters. While the additional parameters can be computed very fast, their number can become large and present a struggle for the successive phases of the interprocedural SDG computation, like summary edge computation.

As shown in the example for context-insensitive points-to analyses, the number of additional parameters is bound by the number of object instantiation instructions in the program. Typically each instantiation instruction is modeled as a single separate distinguishable location. For context-sensitive analyses the number of additional parameters grows: It is bound by the number of object instantiations times the number of distinguishable contexts they may appear in.

The impact of a slightly more precise points-to analysis can be seen in Figure 2.44: It shows the same part of the SDG as the previous figure, but computed with a 1-level call site sensitive points-to analysis. Due to the call stack sensitivity the calls to `sum` in l. 20 and l. 21 are analyzed separately. Now each call produces 3 additional parameter nodes for `sumData`, resulting in a total of 8 additional parameter nodes in this scenario.

While this approach computes new parameters fast, it fails to group parameters with a similar effect together and therefore creates unnecessary many parameters. In contrast to the object-tree and -graph model it also does not include an implicit *escape analysis* that detects method local memory accesses that are not visible to the caller. These invisible accesses still produce additional parameter nodes in the unstructured model, while they could have been omitted. E.g. in the current example the modification of field `i` of the object created in method `sum` is not visible outside of `sumData`, as the created object is not returned. So the additional output nodes could have been omitted. Our evaluation shows that the additional costs of the escape analysis almost always pays off and that less nodes increase the precision of the result and also help to reduce runtime of summary edge computation.

### 2.6.2 Object-tree model

The object-tree model structures additional parameters as trees beneath the nodes of the normal method parameters. Each node represents a field and its parent node represents the object that contains it. The tree

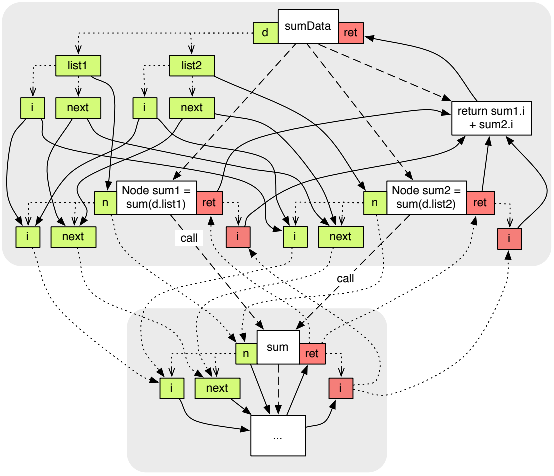


**Figure 2.45:** Part of the SDG with an object-tree parameter model analyzed with a context-insensitive points-to analysis for the example in Figure 2.42.

structure shows —starting from the normal root-parameter nodes— how a certain value can be reached through subsequent field access operations. This is an advantage to the unstructured model, where the access path of a value is not visible. It allows us to group nodes together that represent a value accessed in the same way and also to detect and ignore nodes of values that are not accessible.

Figure 2.45 shows a part of the SDG computed with a context-insensitive points-to analysis and the object-tree model. The additional parameters are connected through *parameter-structure* edges and form a tree. E.g. at the interface of method `sum` fields `i` and `next` of parameter `n` are read, while field `i` of the return value object is written. The structure of the additional parameters allows us to infer if a certain effect is visible outside of a method boundary. In the current example this is the case with the field `i` modified by method `sum`. While the return value of `sum` is an object that contains a reference to `i`, the return value of `sumData` is a primitive integer. Thus the return value of `sumData` cannot not contain a reference to the modified field and the matching output-parameter node must not be included in the interface of `sumData`. In total the object-tree approach creates 6 additional input parameters for `sumData`.

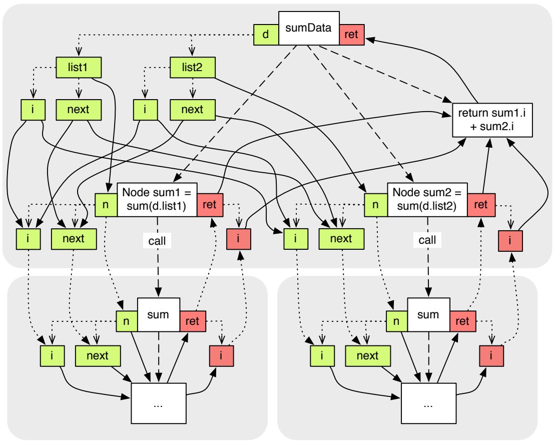


**Figure 2.46:** Part of the SDG with an object-tree parameter model analyzed with a 1-level call stack sensitive points-to analysis for the example in Figure 2.42.

In contrast to the unstructured approach, the number of additional parameter nodes in the example does not increase with a more precise points-to analysis. Figure 2.46 shows the SDG part with object-trees for the 1-level call site sensitive analysis. While the two calls to `sum` are now treated separately, both calls result in similar parameter nodes, thus the number of additional nodes at the caller `sumData` does not change and remains at 6.

A disadvantage of the object-tree approach however is that it can produce unnecessarily many nodes for a single location in case the location is reachable through different access paths. In the current example this is the case for elements of `list1` and `list2` passed to `sumData`. Before the call to `sumData` both elements are set to the same location in l. 14. So all fields referenced through `list1` refer to the same location when referenced through `list2` (e.g. `d.list1.i` == `d.list2.i`). Yet separate nodes are created for them. E.g. in Figure 2.46 there is a node `i` reachable from node `list1` and another node `i` reachable from `list2`. Both nodes however have the same outgoing dependencies, as they effectively refer to the same memory location.

These duplicate nodes can occur for two reasons: Either a location can in fact be reached through different access paths or —in case the

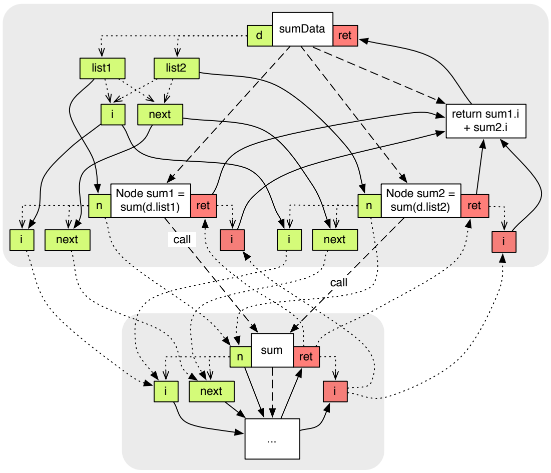


**Figure 2.47:** Part of the SDG with an object-graph parameter model analyzed with a context-insensitive points-to analysis for the example in Figure 2.42.

underlying points-to analysis is not precise enough— two access paths point-to locations that cannot be differentiated by the points-to analysis. In the current example the first reason applies, but often —when analyzing larger programs— we need to apply a fast and less precise points-to analysis that tends to group many actual locations into a single abstract location. The new object-graph approach tackles this problem by using arbitrary graphs instead of trees to structure parameter nodes. The graph structure merges sub-trees of nodes that refer to the same location and thereby removes the duplicate nodes.

### 2.6.3 Object-graph model

The object-graph model refines the object-tree approach. It allows parameter nodes to share subtrees in case they refer to the same location. Object-graphs remove the bottleneck introduced by object-trees for less precise points-to analyses and therefore are well suited to analyze larger programs. As only nodes that correspond to the same location are combined, no precision is lost when compared to the object-trees.

Figure 2.47 shows the SDG part of the common example with the

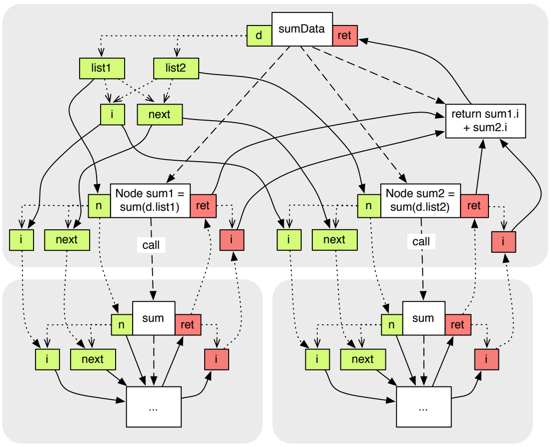


**Figure 2.48:** Part of the SDG with an object-graph parameter model analyzed with a 1-level call stack sensitive points-to analysis for the example in Figure 2.42.

object-graph parameter model and context-insensitive points-to analysis. In contrast to the object-tree model, we use the information that `list1` and `list2` point-to the same location. Therefore all fields reachable from `list1` is also reachable from `list2` and vice versa. Hence we can share the sub-trees and hereby reduce the amount of additional nodes. In this example the object-graph parameter model creates 4 additional input parameter nodes. Like the object-tree approach it can detect that no additional output nodes need to be created.

Unlike the unstructured model, the object-graph model does not produce a huge amount of additional nodes when the points-to analysis precision is increased. Figure 2.48 shows the resulting SDG part of the common example for a more precise 1-level call stack sensitive points-to analysis. There the number of additional nodes for the interface of `sumData` did not change compared to the context-insensitive variant. Still only additional 4 nodes are created.

We have argued that object-graphs combine the benefits of the object-tree and the unstructured model: (1) They do not produce unnecessary duplicate nodes for aliased location like the object-trees. (2) They maintain the benefit of object-trees when used with more precise points-to where the unstructured approach struggles. Also we can detect which side-effects are not visible outside of a methods boundary and remove corresponding nodes from the methods interface. In the following we take a closer look at the computation algorithm of object-trees before we explain in detail the changes and adjustments made for the object-graph model.

### 2.6.4 Computation

#### Object-tree computation

The object-tree model uses an object- and field-sensitive representation for all fields a method may read or modify. A method holds an object-tree for each parameter: The parameter itself corresponds to the root node of its object-tree, while the child nodes match the accessed fields of the parameter object. The computation is initialized with a single root node for each parameter and subsequently adds new child nodes through a fixed-point computation that consists of one initial and two mutually iterating steps:

1. Create root nodes for each method in the program.
2. Repeat until object-trees no longer grow:
   (a) Intraprocedural for each method: Examine all field access instructions and add new child nodes to the current root nodes or already existing child nodes that match the field access. Ignore the field access if no current node in the interface qualifies as a parent.
   (b) Interprocedural: Propagate the effects of method calls from callee to caller. Check for each child node of a called method if it can be added to the interface of the caller.

The interprocedural step (a) and the intraprocedural step (b) are repeated until a fixed-point is reached, as they may add new child nodes to the interface of a method that could act as a parent for previously ignored field accesses from the other step.

Our computation algorithm differs from the initial algorithm developed by Hammer [44] because we do not compute new parameter nodes directly, but rather compute an intermediate representation of so-called *field-candidates*. This approach has the benefit that it is more flexible and allows us to compute the upcoming object-graph with only minor changes to the algorithm. Another optimization to Hammers approach is that we keep input and output parameters in a single data structure to avoid redundancy and extract the specific input and output parameter trees later on in a separate step.

We compute *field-candidates* for each field access in the program. If an existing parameter node qualifies as a parent node for a candidate, a new parameter-node is created for the candidate.

**Definition 2.18** (Field-Candidate). *A field-candidate $fc$ is a 4-tuple $fc = (pts_{base}, f, pts_f, acc)$ that describes the location and effects of one or more field access operations. It contains the points-to set $pts_{base}$ of the base-pointer, the field name $f$ of the accessed field, the points-to set $pts_f$ of field $f$ and an access qualifier $acc \in \{ref, mod, both\}$. The access qualifier qualifies if the location has been referenced ($ref$), modified (mod) or both (both).*

Each method in the program has its own set of object-trees —one for each parameter— that describes the visible side-effects of the method. The object-tree parameter nodes computed from the field-candidates are defined as follows.

**Definition 2.19** (Tree-Parameter). *A tree-parameter tp is a 4-tuple $tp = (parent, f, pts_f, acc)$ that describes the memory location of a field accessed during execution of a method. It contains a reference to its parent parameter. The reference is $\top$ (or **`null`**) for root parameters. Field name $f$ and points-to set $pts_f$ contain the name and points-to set of the accessed field. The access qualifier shows if the field has been referenced ($ref$), modified (mod) or both (both).*

During computation we add a new child to tree-parameter *tp* for every candidate $fc$ where $fc.pts_{base}$ and $tp.pts_f$ may refer to the same location and the type of $tp.f$ holds a reference to field $fc.f$. There may be multiple candidates for the same field that qualify as child of the same tree-parameter —e.g. candidates from different access operations to the same field with distinguishable points-to sets. In this case the points-to sets of all candidates are merged to a single tree-parameter. This helps to reduce the size of the resulting trees, but may harm precision.

A problem of this approach is that recursive data structures can lead to trees of infinite depth. So a special treatment of recursive structures is required.

```
1  class Node {
2    Node next;
3
4    static void append(Node node, int len) {
5      node.next = new Node();
6      node = node.next;
7      for (int i = 1; i < len; i++) {
8        node.next = new Node();
9        node = node.next;
10     }
11   }
12 }
```

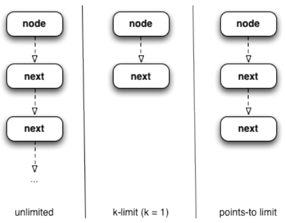


**Figure 2.49:** Example for object-trees with potential infinite depth.

**Unfolding recursive data structures** When we represent the parameter structure in form of trees, recursive data structures pose a problem. Figure 2.49 contains an example with a linked list defined as recursive data structure. The length of the list created by append is in general not statically known, but the object-tree has to be limited somehow. Liang and Harrold [82] proposed to $k$-limit the tree depth with an arbitrary number $k$. This approach prevents an infinite expansion, but can sacrifice soundness. Hammer [46] introduced a *unfolding criterion* that limits the size of object-trees and maintains soundness without the loss of precision. It uses points-to information to decide how far an object-tree is allowed to expand. The unfolding criterion works as follows:

**Definition 2.20** (Object-Tree Unfolding Criterion). *A new child parameter node $c$ is only added to parameter node $p$, if no other node $n$ on the path from the root node to $p$ has the same points-to set as $c$. So given $p_1 \cdots p_n$ is the path to $p$ with $p.parent = p_n$ and $p_i.parent = p_{i-1}.parent$, $c$ is only added if $\nexists n \in \{p_1 \cdots p_n, p\}$ with $n.pts_f \subseteq c.pts_f$ and $n.f = c.f$.*

In our example in Figure 2.49 a fairly precise points-to analysis is able to distinguish between the object instance created in l. 5 and the one created in l. 8. Depending on the value of parameter len, l. 8 is executed multiple times and thus may create more than one new object. No points-to analysis is not able to distinguish all of them, because the number of loop iterations is in general not detectable for a static analysis. Typically only the nodes created by the first and the second **new**-statement can be distinguished. So the unfolding criterion limits the depth of the object-tree for parameter node to two children. The right side of Figure 2.49 displays how the object-tree for node looks like using the k-limiting from Liang and Harrold and the points-to limited unfolding criterion from Hammer. In Joana we use the unfolding criterion of Definition 2.20 as it can guarantee soundness [44].

**Algorithm** We present an object-tree computation algorithm that improves upon the original approach by Hammer. Our approach is based on field-candidates as intermediate representation and a single data structure to represent referenced and modified fields. Later —in a separate step— we extract the resulting input- and output-tree-parameter nodes. Our approach performs better due to the smaller memory footprint and fewer nodes during propagation. In the upcoming section we show how —exploiting the intermediate representation with field-candidates— the algorithm can be adapted to compute object-graphs with only small modifications.

**Algorithm 2.3** (Object-tree computation).

```
PROCEDURE build_objecttree($cg$)
INPUT:
  Call-graph $cg$
OUTPUT:
  Mapping of method to object-tree $map$
BEGIN
  Map<Method, Tree> $map$ = new empty map
  FORALL (methods $m \in cg$) DO
    $t_m$ = emit_root($m$)
    $t_m$ = adjust_local_interface($m$, $t_m$)
    $map$.put($m$, $t_m$)
  DONE
  DO
    FORALL (calls $m_1 \to m_2 \in cg$) DO
      $t_{m_1}$ = $map$.get($m_1$), $t_{m_2}$ = $map$.get($m_2$)
      $t_{m_1}$ = coalesce($t_{m_1}$, emit_from_tree($t_{m_2}$))
      $t_{m_1}$ = adjust_local_interface($m_1$, $t_{m_1}$)
      $map$.put($m_1$, $t_{m_1}$)
    DONE
  WHILE (object-trees changed)
  RETURN $map$
END

PROCEDURE emit_root($m$, $t$)
INPUT:
  Method $m$
OUTPUT:
  Object-tree $t$
BEGIN
  $t$ = empty object-tree
  add root node $\top$ to $t$
  FORALL (parameters $p$ of $m$) DO
    $pts_p$ = points-to set of parameter $p$
    Add $(\top, p, pts_p, both)$ to $t$
  DONE
  $pts_{static}$ = artificial unique set for static fields
  add $p_{static} = (\top, static, pts_{static}, both)$ to $t$
  IF ($m$ has a return value $ret$) THEN
    $pts_{ret}$ = points-to set of return value
    add $p_{ret} = (\top, ret, pts_{ret}, ref)$ to $t$
  FI
  IF ($m$ can throw an exception) THEN
    $pts_{exc}$ = points-to set of all exceptions thrown by $m$
    add $p_{exc} = (\top, exc, pts_{exc}, ref)$ to $t$
  FI
  RETURN $t$
END

PROCEDURE coalesce($t$, $cands$)
INPUT:
  Object-tree $t$
  Set of candidates $cands$
OUTPUT:
  Object-tree $t'$
BEGIN
  $t'$ = copy of $t$;
  FORALL (candidates $cand \in cands$, $p \in t$ with $p.acc \in ref, both$) DO
    IF ($p.pts_f \cap cand.pts_{base} \neq \emptyset$ AND
        type of $p.f$ can reference field $cand.f$ AND
        $\nexists n \in$ path from $\top$ to $p$ where $n.pts_f \subseteq cand.pts_f \wedge n.f = cand.f$)
    THEN
      IF ($p$ has no child for field $cand.f$) THEN
        add $(p, cand.f, cand.pts_f, cand.acc)$ to $t'$
      ELSE
        get child $ch$ of $p$ with $ch.f = cand.f$
        $ch.pts_f = ch.pts_f \cup cand.pts_f$
        IF ($ch.acc \neq cand.acc$) THEN
          $ch.acc$ = both
        FI
      FI
    FI
  DONE
  RETURN $t'$
END

PROCEDURE adjust_local_interface($m$, $t$)
INPUT:
  Method $m$
  Object-tree $t$
OUTPUT:
  Object-tree $t'$
BEGIN
  $t'$ = copy of $t$
  DO
    FORALL (instructions $i \in m$) DO
      $t'$ = coalesce($t'$, emit_from_instruction($i$))
    DONE
  WHILE (fixed-point of $t'$ is not reached)
  RETURN $t'$
END

PROCEDURE emit_from_instruction($i$)
INPUT:
  Instruction $i$
OUTPUT:
  Set of emitted candidates
BEGIN
  IF ($i$ is field access) THEN
    create new candidate $cand = (pts_{base}, f, pts_f, acc)$
    IF ($i$ is static access) THEN
      $cand.pts_{base} = pts_{static}$
    ELSE
      $cand.pts_{base}$ = points-to of base pointer of $i$
    FI
    $cand.f$ = accessed field of $i$
    $cand.pts_f$ = points-to of accessed field of $i$
    IF ($i$ is field-get) THEN
      $cand.acc$ = ref
    ELSE
      $cand.acc$ = mod
    FI
    RETURN $\{cand\}$
  FI
  RETURN $\emptyset$
END

PROCEDURE emit_from_tree($t$)
INPUT:
  Object-tree $t$
OUTPUT:
  Set of emitted candidates $cands$
BEGIN
  $cands = \emptyset$
  FORALL (nodes $n \in t$ where $n \neq \top$ AND $n.parent \neq \top$) DO
    create new candidate $cand = (pts_{base}, f, pts_f, acc)$ with
    $cand.pts_{base} = n.parent.pts_f$
    $cand.f = n.f$
    $cand.pts_f = n.pts_f$
    $cand.acc = n.acc$
    $cands = cands \cup \{cand\}$
  DONE
  RETURN $cands$
END
```

Algorithm 2.3 computes the object-tree parameters for every method contained in a given call graph. In addition to the call graph it uses points-to and type information to create new field-candidates and tree-parameters. The computation starts in procedure `build_objecttree` with the initial computation of root nodes for each parameter. The object-trees for every parameter of a method are stored in a single tree structure with

$\top$ as artificial root node. Procedure `emit_root` adds root parameters for each method parameter, as well as for the return value. It also creates special root nodes for static field accesses and possible exceptions of the analyzed method. Procedure `adjust_local_interface` adds new child nodes that correspond to method local field accesses to the provided tree. Procedure `coalesce` guarantees that only one child node per field exists and the unfolding criterion is met when new tree-parameters are added. The interprocedural propagation takes place in the **`do while`**-loop of `build_objecttree`: The algorithm repeatedly iterates over all method calls and tree-nodes from the interface of the callee and propagate to the caller, iff a matching parent node exists. Propagation is finished when no additional tree-parameters have been added during the previous iteration.

In the next step —shown in Algorithm 2.4— the input- and output-parameters for each method are extracted from the computed object-tree data structure.

**Algorithm 2.4** (Extract input and output nodes from object-tree).

```
PROCEDURE extract_output_nodes($t$)
INPUT:
  Object-tree $t$
OUTPUT:
  Object-tree of all output-parameter nodes $t_{mod}$
BEGIN
  $t_{mod}$ = empty tree
  add all root parameters $rp \in t$ with $rp.parent = \top$ to $t_{mod}$
  FORALL (nodes $f \in t$ where $f.acc \in \{mod, both\}$) DO
    add all nodes on the path from $\top$ to $f$ in $t$ to $t_{mod}$
  DONE
  RETURN $t_{mod}$
END

PROCEDURE extract_input_nodes($t$)
INPUT:
  Object-tree $t$
OUTPUT:
  Object-tree of all input-parameter nodes $t_{ref}$
BEGIN
  $t_{chop}$ = copy of $t$
  remove $p_{ret}$, $p_{exc}$ and all their children from $t_{chop}$
  $t_{ref}$ = empty tree
  add all root parameters $rp \in t_{chop}$ with $rp.parent = \top$ to $t_{ref}$
  FORALL (nodes $f \in t_{chop}$ where $f.acc \in \{ref, both\}$) DO
    add all nodes on the path from $\top$ to $f$ in $t_{chop}$ to $t_{ref}$
  DONE
  RETURN $t_{ref}$
END
```

## Object-graph computation

The object-graph computation is closely related to the object-tree algorithm already presented. Again field-candidates of Definition 2.18 are used to compute a similar intermediate result. In contrast to the tree-parameters used in object-trees, the object-graph is directly built from field-candidates. During computation we maintain a set of mod-ref field-candidates and build an object-graph from these sets afterwards. This separation allows us to minimize the memory footprint, as we do not need to create new graph-parameter objects during the memory-critical propagation phase. We introduce two variants of the algorithm: a slower and more precise variant called *standard* and a faster less precise variant called *fast*. We start with the standard variant.

1. Add root field-candidates to each methods mod-ref set.
2. Compute local field-candidates for each method.
3. Repeat until mod-ref sets no longer change.
    (a) Intraprocedural for each method: Add local field-candidates created from the instructions of the current method to its mod-ref set. Add only new candidates if they can be reached through an element already in the set.
    (b) Interprocedural: Propagate effects of method calls from callee to caller. Add all field-candidates in the mod-ref set of the callee to the mod-ref set of the caller. Add only new candidates that can be reached from elements already in the callers set.
4. Build object-graph structure from mod-ref sets.
5. Extract input- and output-nodes from object-graph.

**Algorithm (standard)** The standard variant of the object-graph computation is shown in Algorithm 2.5.

**Algorithm 2.5** (Object-graph computation).

```
PROCEDURE build_objectgraph($cg$)
INPUT:
  Call-graph $cg$
OUTPUT:
  Mapping of method to object-graph $map_{graph}$
BEGIN
  Map<Method, Graph of candidates> $map_{graph}$ = new empty map
  Map<Method, Set of candidates> $map_{modref}$, $map_{local}$, $map_{root}$ = new empty maps

  FORALL (methods $m \in cg$) DO
    $root_m$ = emit_root($m$)
    $cands_m$ = build_local_interface($m$, $root_m$)
    $map_{local}$.put($m$, $cands_m$)
    $map_{root}$.put($m$, $root_m$)
    $modref_m$ = copy of $root_m$
    $map_{modref}$.put($m$, $modref_m$)
  DONE

  DO
    FORALL (methods $m \in cg$) DO
      $cands_m$ = $map_{local}$.get($m$)
      $modref_m$ = $map_{modref}$.get($m$)
      $modref'_m$ = add_reachable_to_set($modref_m$, $cands_m \setminus modref_m$)
      $map_{modref}$.put($m$, $modref'_m$)
    DONE
    FORALL (calls $m_1 \rightarrow m_2 \in cg$) DO
      $root_{m_2}$ = $map_{root}$.get($m_2$)
      $modref_{m_1}$ = $map_{modref}$.get($m_1$)
      $modref_{m_2}$ = $map_{modref}$.get($m_2$)
      $modref'_{m_1}$ = add_reachable_to_set($modref_{m_1}$, $modref_{m_2} \setminus root_{m_2}$)
      $map_{modref}$.put($m_1$, $modref'_{m_1}$)
    DONE
  WHILE (mod-ref sets change)

  FORALL (methods $m \in cg$) DO
    $modref_m$ = $map_{modref}$.get($m$)
    $g$ = extract_graph($modref_m$)
    $map$.put($m$, $g$)
  DONE

  RETURN $map$
END

PROCEDURE emit_root($m$)
INPUT:
  Method $m$
OUT:
  Candidates for root nodes $roots$
BEGIN
  $roots$ = new empty set
  $pts_\top$ = artificial unique points-to set for root candidates
  FORALL (parameters $p$ of $m$) DO
    $pts_p$ = points-to set of parameter $p$
    add $(pts_\top, p, pts_p, both)$ to $roots$
  DONE
  $pts_{static}$ = artificial unique set for static fields
  add $p_{static} = (pts_\top, static, pts_{static}, both)$ to $roots$
  IF ($m$ has a return value $ret$) THEN
    $pts_{ret}$ = points-to set of return value
    add $p_{ret} = (pts_\top, ret, pts_{ret}, ref)$ to $roots$
  FI
  IF ($m$ can throw an exception) THEN
    $pts_{exc}$ = points-to set of all exceptions thrown by $m$
    add $p_{exc} = (pts_\top, exc, pts_{exc}, ref)$ to $roots$
  FI
  RETURN $roots$
END

PROCEDURE build_local_interface($m$, $roots$)
INPUT:
  Method $m$
  Set of root candidates $roots$
OUTPUT:
  Set of local candidates $locals$
BEGIN
  $locals$ = copy of $roots$
  FORALL (instructions $i \in m$) DO
    $locals$ = $locals \cup$ emit_from_instruction($i$)
  DONE
  RETURN $locals$
END

PROCEDURE emit_from_instruction($i$)
INPUT:
  Instruction $i$
OUTPUT:
  Set of emitted candidates
BEGIN
  IF ($i$ is field access) THEN
    create new candidate $cand = (pts_{base}, f, pts_f, acc)$
    IF ($i$ is static access) THEN
      $cand.pts_{base}$ = $pts_{static}$
    ELSE
      $cand.pts_{base}$ = points-to of base pointer of $i$
    FI
    $cand.f$ = accessed field of $i$
    $cand.pts_f$ = points-to of accessed field of $i$
    IF ($i$ is field-get) THEN
      $cand.acc$ = ref
    ELSE
      $cand.acc$ = mod
    FI
    RETURN $\{cand\}$
  FI
  RETURN $\emptyset$
END

PROCEDURE add_reachable_to_set($modref$, $cands$)
```

```
INPUT:
  Set of already reachable candidates $modref$
  Set of candidates $cands$
OUTPUT:
  New set of reachable of candidates $modref'$
BEGIN
  $modref'$ = copy of $modref$
  DO
    FORALL (nodes $n \in cands$) DO
      IF ($\exists n' \in modref' : n'.pts_f \cap n.pts_{base} \neq \emptyset$
          AND type of $n'.f$ can reference $n$)
      THEN
        add $n$ to $modref'$
      FI
    DONE
  WHILE (fixed-point of $modref'$ not reached)
  RETURN $modref'$
END

PROCEDURE extract_graph($modref$)
INPUT:
  Set of candidates $modref$
OUTPUT:
  Object-graph of the provided candidates $g$
BEGIN
  $g$ = new graph of candidates
  $roots = \{gc \mid gc \in modref \wedge gc.pts_{base} = pts_{\top}\}$
  FORALL ($gc_1, gc_2 \in modref$ with $gc_1 \neq gc_2$) DO
    IF ($gc_1.pts_f \cap gc_2.pts_{base} \neq \emptyset$ AND type of $gc_1.f$ can reference $gc_2$) THEN
      add $gc_1 \rightarrow gc_2$ to $g$    // $gc_1$ is potential parent of $gc_2$
    FI
  DONE
  // optionally merge candidates with same parent and same field
  $g'$ = merge_candidates($g$)
  RETURN $g'$
END

PROCEDURE merge_candidates($g$)
INPUT:
  Graph of candidates $g$
OUTPUT:
  Graph of merged candidates $g$
BEGIN
  DO
    FORALL ($gc_1, gc_2 \in$ nodes of $g$ where $gc_1 \neq gc_2$) DO
      IF ($gc_1.f = gc_2.f$ AND $\exists gc \in$ nodes of $g$ with $gc \rightarrow gc_1, gc \rightarrow gc_2 \in g$
          AND $gc_1.acc = gc_2.acc$ OR $gc_1.acc = both$)
      THEN
        $gc_{merge} = (gc_1.pts_{base} \cup gc_2.pts_{base}, gc_1.f, gc_1.pts_f \cup gc_2.pts_f, gc_1.acc)$
        replace all occurrences of $gc_1$ and $gc_2$ in $g$ with $gc_{merge}$
      FI
    DONE
  WHILE ($g$ changes)
  RETURN $g$
END
```

Algorithm 2.5 computes an object-graph for each method contained in the provided call graph. Similar to the object-tree algorithm it also uses points-to and type information to create field-candidates and propagates them along the call graph. The computation starts in procedure `build_objectgraph`. The resulting object-graph for each method is stored in $map_{graph}$. The other maps contain the intermediate results for each method: $map_{modref}$ stores the current set of mod-ref candidates, $map_{local}$ the set of all field-candidates created from instructions of the corresponding method and $map_{root}$ holds the root field-candidates of each method.

Initially all root and local field-candidates are created for each method and stored to $map_{root}$ and $map_{local}$. The mod-ref sets in $map_{modref}$ are initialized with the root field-candidates of the corresponding method. Then the **do while**-loop begins and the fixed-point based intra- and interprocedural propagation starts. We apply a reachability based escape analysis during each step of the intra- and interprocedural propagation phase. Procedure `add_reachable_to_set`$(modref, cands)$ creates a new set of field-candidates that contains all elements of $modref$ and all candidates in *cands* that the are reachable from a candidate already in $modref$. This procedure filters all candidates that correspond to a side-effect that is not visible outside the scope of the current method. It basically acts as an integrated escape analysis during propagation. We use `add_reachable_to_set` during both intra- and interprocedural propagation phases. After the fixed-point is reached, we extract an object-graph from the mod-ref set of each method with procedure `extract_graph`: We create a node for each field-candidate and add edges iff one candidate qualifies as the parent of the other. Optionally a call to `merge_candidates` combines field-candidates referring to the same field and share a common parent. In contrast to the object-tree algorithm there is no need to check for the unfolding criterion, as it is automatically met: Field-candidates are identified by their field-name and points-to sets, so no two candidates with a same field and points-to set can exists. No additional nodes are created during graph construction . If merging is enabled, the number of nodes is even reduced. The resulting object-graph contains input- as well as output-parameter nodes of the corresponding method. In a final step —analog to the extraction of input and output nodes for object-trees described in Algorithm 2.4— we extract and create the relevant input- and output-nodes from the shared structure.

**Algorithm (fast propagation)** We introduce a less precise but faster variant of the standard object-graph algorithm. This approach does not apply reachability checks during the propagation phase, but rather uses a separate phase afterwards that prunes field-candidates of non-escaping unreachable side-effects. The algorithm is quite similar to the standard approach, only the reachability analysis is now done in a separate step afterwards. This allows us to propagate the field-candidates from callee to caller without checking if a potential parent node is already in the candidates of the caller. The changes to the standard algorithm are marked in *italic*.

1. Add root field-candidates to each methods mod-ref set.
2. Compute local field-candidates for each method.
3. *Intraprocedural for all methods: Add local field-candidates created from the instructions of the current method to its mod-ref set. Add all local candidates to the set.*
4. Repeat until mod-ref sets no longer change.
    (a) Interprocedural: Propagate effects of method calls from callee to caller. Add all field-candidates in the mod-ref set of the callee to the mod-ref set of the caller. *Add all candidates to the set of the caller without checking.*
5. *Prune unreachable candidates from mod-ref sets.*
6. Build object-graph structure from mod-ref sets.
7. Extract input- and output-nodes from object-graph.

Algorithm 2.6 shows a variant of the standard approach that applies the escape analysis in a separate step after the propagation phase. This helps to improve the runtime of the propagation phase, but it can produce additional unnecessary parameter nodes, as the separate escape analysis step can only detect a subset of the non-escaping side-effects detected by the integrated approach. The pseudocode shown contains only the methods that changed compared to the standard approach in Algorithm 2.5.

**Algorithm 2.6** (Object-graph computation with faster but less precise interprocedural propagation).

```
PROCEDURE build_objectgraph($cg$)
INPUT:  Call-graph $cg$
OUTPUT: Mapping of method to object-graph $map_{graph}$
BEGIN
  Map<Method, Graph of candidates> $map_{graph}$ = new empty map
  Map<Method, Set of candidates> $map_{modref}$, $map_{root}$ = new empty maps

  FORALL (methods $m \in cg$) DO
    $root_m$ = emit_root($m$)
    $cands_m$ = build_local_interface($m$, $root_m$)
    $map_{modref}$.put($m$, $cands_m$)
    $map_{root}$.put($m$, $root_m$)
  DONE

  DO
    FORALL (calls $m_1 \rightarrow m_2 \in cg$) DO
      $cands_{m_1}$ = $map_{modref}$.get($m_1$)
      $cands_{m_2}$ = $map_{modref}$.get($m_2$)
      $root_{m_2}$ = $map_{root}$.get($m_2$)
      $cands'_{m_1}$ = $cands_{m_1} \cup (cands_{m_2} \setminus root_{m_2})$
      $map_{modref}$.put($m_1$, $cands'_{m_1}$)
    DONE
  WHILE (mod-ref sets change)

  FORALL (methods $m \in cg$) DO
    $modref_m$ = $map_{modref}$.get($m$)
    $g$ = extract_graph($modref_m$)
    $map$.put($m$, $g$)
  DONE

  RETURN $map$
END

PROCEDURE extract_graph($modref$)
INPUT:  Set of candidates $modref$
OUTPUT: Object-graph of the provided candidates $g$
BEGIN
  $g$ = new graph of candidates
  $roots = \{gc \mid gc \in modref \wedge gc.pts_{base} = pts_{\top}\}$
  FORALL ($gc_1, gc_2 \in modref$ with $gc_1 \neq gc_2$) DO
    IF ($gc_1.pts_f \cap gc_2.pts_{base} \neq \emptyset$ AND type of $gc_1.f$ can reference $gc_2.f$) THEN
      add $gc_1 \rightarrow gc_2$ to $g$
    FI
  DONE
  // prune non-escaping - extra step
  remove all candidates in $g$ that are not reachable from an element in $roots$
  // optionally merge candidates with same parent and same field
  $g'$ = merge_candidates($g$)
  RETURN $g'$
END
```

Computation starts in procedure `build_objectgraph`. In contrast to the standard approach, the mod-ref set of each method $map_{modref}$ is initialized with all local field-candidates of the method. The following propagation phase in the **`do-while`** loop is much simpler and only contains the interprocedural step that propagates from callee to caller. Without the reachability check, the propagation can be done with simple set operations. That allows us to use a fast standard data-flow framework based on bitvectors for the interprocedural propagation[15]. After the propagation phase the object-graph is extracted by `extract_graph`. The algorithm performs a separate escape analysis on the resulting graph and continues with the optional merge operation: All nodes not reachable from a root node can be pruned from the graph. Next candidates referring to the same field and parents can be merged optionally, before the object-graph is finally returned. Analog to Algorithm 2.4 of the standard approach we extract input- and output-parameter nodes from the shared object-graph structure.

### 2.6.5 Evaluation

We integrated the previously introduced parameter models — unstructured (§2.6.1), object-tree (§2.6.4) and object-graph (§2.6.4) — into the Joana framework and evaluated their performance in combination with points-to analyses of varying precision. Our tool is based on the WALA framework that supports a variety different points-to analyses as described in §2.5.2. For this evaluation we focus on three widely used variants:

1. *Context-insensitive type-based (0-CFA)* analysis that disambiguates objects according to declared types.

2. *Context-insensitive instance-based (0-1-CFA)* analysis that disambiguates objects according to instantiation sites.

3. *Object-sensitive instance-based* analysis that enables object-sensitivity for all instantiation sites.

[15]For sake of brevity this optimization is not part of the pseudocode.

| program | #instructions |
|---|---|
| *JavaCard* | |
| corporatecard | 1186 |
| purse | 10807 |
| safeapplet | 1295 |
| wallet | 116 |
| *JRE* | |
| battleship | 937 |
| cloudstorage | 1972 |
| clientserver | 601 |
| hybrid | 1479 |
| *JavaGrande (11)* | *14176* |
| freecs | 49396 |
| hsqldb | 126860 |
| *J2ME* | |
| barcode | 11406 |
| bexplore | 14041 |
| j2mesafe | 5519 |
| keepass | 16427 |
| onetimepass | 13034 |

**Table 2.3:** Size of analyzed programs in number of bytecode instructions. Library code not included.

The points-to variants from 1) to 3) are enumerated from least to most precise. We evaluated the performance and precision of SDG creation[16] on 26 Java programs with up to 3 different points-to analyses and three different parameter models. All programs were analyzed including the library methods. Native methods were conservatively approximated through hand written method stubs — a huge difference to other approaches where parts of the runtime library are not analyzed at all [124]. The parts included from the runtime library tend to be big and have an enormous impact on the runtime of the analysis. But when left out, the result of the analysis is no longer a conservative approximation and some illegal information flow may be undetected.

We evaluate our SDG computation on 26 example programs that vary in program size as well as runtime library size: 4 programs (Corporate Card, Purse, Wallet and Safe) are JavaCard applications. JavaCard is

[16]We executed all tests on a computer with a Core i7 processors, 32GB of RAM running Windows 10 with Java 8 64-bit.

built for applications that run on smart cards and other devices with very limited memory and processing capabilities, so its runtime library is quite small with about 500LoC. 5 programs (Barcode, bExplore, J2MESafe, KeePass and OneTimePass) are J2ME applications. J2ME is a Java environment for mobile phones with a medium sized runtime library comprised of about 30kLoC. The remaining 17 programs (Battleship, CloudStorage, ClientServer, Hybrid, FreeCS, HSQLDB and the 11 examples of the JavaGrande suite[17]) use the Java 1.4 library with about 100kLoC. Table 2.3 shows the size of all analyzed programs in number of bytecode instructions. These numbers do not include additional libraries. The 11 programs of the JavaGrande suite share a common codebase, therefore the individual program sizes cannot be distinguished.

We canceled computation when the memory usage exceeded 30GB or the analysis ran for more than 4 hours. The JavaCard programs and most of the Java 1.4 programs —except MonteCarlo, FreeCS and HSQLDB— can be analyzed with all available options. They are used to compare the effects of the different parameter models and points-to analyses. All J2ME programs and the large FreeCS and HSQLDB applications could only be analyzed with some of the available options. Sometimes we could not apply a more precise points-to analysis because the points-to computation itself was too costly. Other times we could not use other parameter models as they did not scale: The parameter computation of the object-tree model often consumed too much time and memory and did not finish. The unstructured approach finished parameter computation in many cases, but created so many additional parameter nodes that the subsequent summary edge computation struggled.

As expected object-graphs scale better then object-trees or the unstructured approach. Object-graphs are a good choice when analyzing larger programs as they can deal with imprecise points-to analyses more efficiently, yet they also cope well with more precise points-to information.

**Runtime** Table 2.4 shows the runtime of the SDG computation in milliseconds for each evaluated program. The results are grouped by parameter model and for each model three columns show the results

[17]EPCC University of Edinburgh. The Java Grande benchmarking suite (`http://www.epcc.ed.ac.uk/research/activities/java-grande/`).

| runtime | unstructured | | | object-tree | | | object-graph | | |
|---|---|---|---|---|---|---|---|---|---|
| in ms | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| *JavaCard* | | | | | | | | | |
| corporatecard | 892 | 1034 | 1359 | 1056 | 1111 | 1689 | 427 | 485 | 821 |
| purse | 5814 | 14426 | 14160 | 16577 | 10361 | 13047 | 6004 | 6844 | 13165 |
| safeapplet | 902 | 965 | 1408 | 1090 | 991 | 1641 | 376 | 384 | 1332 |
| wallet | 955 | 1167 | 1519 | 1218 | 1182 | 1982 | 504 | 507 | 924 |
| *JRE* | | | | | | | | | |
| battleship | 1093 | 3199 | 6702 | 1621 | 1673 | 2259 | 408 | 400 | 1078 |
| cloudstorage | 1583 | 2546 | 4955 | 1998 | 2071 | 2333 | 563 | 563 | 1030 |
| clientserver | 661 | 775 | 703 | 565 | 624 | 757 | 204 | 220 | 298 |
| hybrid | 1477 | 1814 | 1882 | 1775 | 1743 | 1962 | 454 | 475 | 729 |
| *JavaGrande suite* | | | | | | | | | |
| barrier | 3400 | 115016 | 138895 | 11300 | 11174 | 15912 | 553 | 638 | 2055 |
| crypt | 3501 | 218500 | 160353 | 11841 | 12368 | 17465 | 1816 | 1842 | 5810 |
| forkjoin | 3189 | 109919 | 149240 | 11177 | 11637 | 16751 | 553 | 582 | 2097 |
| lufact | 3405 | 130934 | 120168 | 11323 | 11988 | 16456 | 962 | 1046 | 2340 |
| moldyn | 4410 | 151784 | 198994 | 15171 | 15605 | 24120 | 1920 | 1995 | 7854 |
| montecarlo | 14255 | 86065* | - | 13807 | 14995 | 41044 | 1798 | 2286 | 12636 |
| raytracer | 4062 | 215234 | 360410 | 12196 | 12188 | 22846 | 1259 | 1408 | 5998 |
| series | 3227 | 148995 | 132424 | 11238 | 11937 | 16431 | 560 | 666 | 1871 |
| sor | 3565 | 180225 | 146968 | 11974 | 12261 | 17717 | 1160 | 1215 | 3031 |
| sparsematmult | 3587 | 180475 | 147296 | 11554 | 13011 | 18099 | 2860 | 2972 | 4360 |
| sync | 3244 | 118664 | 154422 | 11104 | 11444 | 18398 | 562 | 603 | 2308 |
| *J2ME* | | | | | | | | | |
| barcode | 7168 | 10247 | - | 6829 | 5872 | - | 1307 | 1506 | - |
| bexplore | 32240 | 208014 | - | - | - | - | 21538 | 31977 | - |
| j2mesafe | 8204 | 10293 | - | 31549 | 38167 | - | 3937 | 4063 | - |
| keepass | 73583 | 2091700 | - | 73202 | 196643 | - | 9495 | 22411 | - |
| onetimepass | 24105 | 776573 | - | - | - | - | 48729 | 52805 | - |
| *JRE (large)* | | | | | | | | | |
| freecs | - | - | - | - | - | - | 1301072 | 6468649 | - |
| hsqldb | - | - | - | - | - | - | 4129274 | 17166274 | - |

**Table 2.4:** Runtime of SDG computation including summary edges. Points-to analysis variants: (1) type-based, (2) instance-based and (3) object-sensitive. Unfinished computation runs (timeout >4 hours) are marked with "-". Numbers marked with * are computation times without summary edges.

| runtime relative to | unstructured | | | object-tree | | | object-graph | | |
|---|---|---|---|---|---|---|---|---|---|
| object-graph (1) | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| *JavaCard* | | | | | | | | | |
| corporatecard | 2.09 | 2.42 | 3.18 | 2.47 | 2.60 | 3.96 | 1.00 | 1.14 | 1.92 |
| purse | 0.97 | 2.40 | 2.36 | 2.76 | 1.73 | 2.17 | 1.00 | 1.14 | 2.19 |
| safeapplet | 2.40 | 2.57 | 3.74 | 2.90 | 2.64 | 4.36 | 1.00 | 1.02 | 3.54 |
| wallet | 1.89 | 2.32 | 3.01 | 2.42 | 2.35 | 3.93 | 1.00 | 1.01 | 1.83 |
| *JRE* | | | | | | | | | |
| battleship | 2.68 | 7.84 | 16.43 | 3.97 | 4.10 | 5.54 | 1.00 | 0.98 | 2.64 |
| cloudstorage | 2.81 | 4.52 | 8.80 | 3.55 | 3.68 | 4.14 | 1.00 | 1.00 | 1.83 |
| clientserver | 3.24 | 3.80 | 3.45 | 2.77 | 3.06 | 3.71 | 1.00 | 1.08 | 1.46 |
| hybrid | 3.25 | 4.00 | 4.15 | 3.91 | 3.84 | 4.32 | 1.00 | 1.05 | 1.61 |
| *JavaGrande suite* | | | | | | | | | |
| barrier | 6.15 | 207.99 | 251.17 | 20.43 | 20.21 | 28.77 | 1.00 | 1.15 | 3.72 |
| crypt | 1.93 | 120.32 | 88.30 | 6.52 | 6.81 | 9.62 | 1.00 | 1.01 | 3.20 |
| forkjoin | 5.77 | 198.77 | 269.87 | 20.21 | 21.04 | 30.29 | 1.00 | 1.05 | 3.79 |
| lufact | 3.54 | 136.11 | 124.91 | 11.77 | 12.46 | 17.11 | 1.00 | 1.09 | 2.43 |
| moldyn | 2.30 | 79.05 | 103.64 | 7.90 | 8.13 | 12.56 | 1.00 | 1.04 | 4.09 |
| montecarlo | 7.93 | 47.87* | - | 7.68 | 8.34 | 22.83 | 1.00 | 1.27 | 7.03 |
| raytracer | 3.23 | 170.96 | 286.27 | 9.69 | 9.68 | 18.15 | 1.00 | 1.12 | 4.76 |
| series | 5.76 | 266.06 | 236.47 | 20.07 | 21.32 | 29.34 | 1.00 | 1.19 | 3.34 |
| sor | 3.07 | 155.37 | 126.70 | 10.32 | 10.57 | 15.27 | 1.00 | 1.05 | 2.61 |
| sparsematmult | 1.25 | 63.10 | 51.50 | 4.04 | 4.55 | 6.33 | 1.00 | 1.04 | 1.52 |
| sync | 5.77 | 211.15 | 274.77 | 19.76 | 20.36 | 32.74 | 1.00 | 1.07 | 4.11 |
| **average** | **3.48** | **91.04** | **103.26** | **8.59** | **8.81** | **13.43** | **1.00** | **1.08** | **3.03** |
| *J2ME* | | | | | | | | | |
| barcode | 5.48 | 7.84 | - | 5.22 | 4.49 | - | 1.00 | 1.15 | - |
| bexplore | 1.31 | 9.66 | - | - | - | - | 1.00 | 1.48 | - |
| j2mesafe | 2.08 | 2.61 | - | 8.01 | 9.69 | - | 1.00 | 1.03 | - |
| keepass | 7.75 | 220.29 | - | 7.71 | 20.71 | - | 1.00 | 2.36 | - |
| onetimepass | 0.49 | 15.94 | - | - | - | - | 1.00 | 1.08 | - |
| *JRE (large)* | | | | | | | | | |
| freecs | - | - | - | - | - | - | 1.00 | 4.97 | - |
| hsqldb | - | - | - | - | - | - | 1.00 | 4.16 | - |

**Table 2.5:** Runtime of SDG computation relative to the object-graph option with type-based points-to. Points-to analysis variants: (1) type-based, (2) instance-base and (3) object-sensitive.

| % summary computation time | (1) | (2) | (3) |
|---|---|---|---|
| unstructured | 19.27 % | 69.83 % | 67.54 % |
| object-tree | 10.42 % | 7.00 % | 6.01 % |
| object-graph | 6.56 % | 7.10 % | 10.94 % |

**Table 2.6:** Average percent of runtime spent on summary computation for JavaCard, JavaGrande suite and small JRE examples. Points-to analysis variants: (1) type-based, (2) instance-base and (3) object-sensitive.

for the respective points-to precision: (1) type-based, (2) instance-based and (3) object-sensitive. The object-graph numbers are measured for the standard approach with integrated escape analysis and merging of parameter field nodes with the same name. In order to improve comparability, Table 2.5 displays analysis runtime relative to the object-graph approach with type-based points-to.

Almost in any case object-graphs prove to be the fastest option. As expected, analysis runtime increases with points-to precision and program size. The parameter-model has a huge impact on scalability: The unstructured approach scales relatively well in combination with a less precise type-based points-to analysis. In average it is 3.48 times slower then object-graphs. This gets significantly worse with instance-based and object-sensitive points-to where its runtime is 91.04 and 103.26 times slower compared to the fastest variant. In contrast —using the same points-to options— the object-graph approach only increases runtime by a factor of 1.08 and 3.03 on average. The object-tree approach scales better than the unstructured model by a significant margin in most cases. In combination with instance-based and object-sensitive points-to it is on average only 8.81 and 13.43 times slower then object-graphs. However, in case of the least precise points-to option object-trees are on average 8.59 times slower, the unstructured model is considerably faster with a smaller slowdown of 3.48.

Overall we observe that program size and points-to precision are not the only factors that influence analysis runtime. Program structure and the amount of library code used is also relevant. This shows the most in the J2ME examples. These programs are roughly all about the same size and use the same library, but still analysis runtime differs

| SDG size in | unstructured | | | object-tree | | | object-graph | | |
|---|---|---|---|---|---|---|---|---|---|
| number of nodes | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| *JavaCard* | | | | | | | | | |
| corporatecard | 9973 | 10874 | 19033 | 11453 | 11198 | 22019 | 10212 | 10494 | 20316 |
| purse | 73193 | 138097 | 127641 | 110184 | 105765 | 139401 | 69928 | 75950 | 130884 |
| safeapplet | 9973 | 10874 | 19033 | 11453 | 11198 | 22019 | 10212 | 10494 | 20316 |
| wallet | 12428 | 14238 | 22687 | 14342 | 13939 | 25926 | 12421 | 12944 | 24077 |
| *JRE* | | | | | | | | | |
| battleship | 28797 | 145949 | 201622 | 17437 | 17385 | 54912 | 7114 | 7087 | 18514 |
| cloudstorage | 20408 | 46231 | 77035 | 15996 | 16156 | 35939 | 10102 | 9919 | 17876 |
| clientserver | 2066 | 2728 | 4745 | 1718 | 1817 | 3546 | 1783 | 1787 | 3743 |
| hybrid | 18085 | 43481 | 65924 | 12481 | 12305 | 27700 | 7299 | 7269 | 10577 |
| *JavaGrande suite* | | | | | | | | | |
| barrier | 24267 | 127972 | 377873 | 16029 | 15519 | 86684 | 11843 | 12076 | 29294 |
| crypt | 30249 | 183692 | 434412 | 19850 | 19169 | 97085 | 14658 | 15153 | 29100 |
| forkjoin | 24663 | 137910 | 402237 | 16225 | 15723 | 90737 | 11850 | 12080 | 29130 |
| lufact | 26992 | 149058 | 361205 | 18629 | 18116 | 83186 | 13868 | 14327 | 25682 |
| moldyn | 31003 | 162289 | 489979 | 21734 | 21177 | 109189 | 17250 | 17894 | 43503 |
| montecarlo | 91463 | 813347 | - | 42808 | 39083 | 277066 | 31493 | 34328 | 101375 |
| raytracer | 34720 | 196956 | 725573 | 23605 | 22528 | 139994 | 18249 | 20001 | 62646 |
| series | 26363 | 154064 | 376339 | 17421 | 16877 | 84319 | 12618 | 13048 | 24538 |
| sor | 29588 | 173651 | 405312 | 18931 | 18298 | 90658 | 13893 | 14363 | 26412 |
| sparsematmult | 29806 | 174291 | 405815 | 19536 | 18824 | 90957 | 14268 | 14846 | 26676 |
| sync | 23950 | 132746 | 410565 | 16063 | 15523 | 92540 | 12055 | 12300 | 31839 |
| *J2ME* | | | | | | | | | |
| barcode | 21905 | 33531 | - | 16864 | 15915 | - | 14437 | 14580 | - |
| bexplore | 307761 | 1812063 | - | - | - | - | 115744 | 118694 | - |
| j2mesafe | 34145 | 45417 | - | 57283 | 89731 | - | 19968 | 20309 | - |
| keepass | 106968 | 371326 | - | 144775 | 180344 | - | 55388 | 56260 | - |
| onetimepass | 81889 | 2790301 | - | - | - | - | 50329 | 51005 | - |
| *JRE (large)* | | | | | | | | | |
| freecs | - | - | - | - | - | - | 1606970 | 1374689 | - |
| hsqldb | - | - | - | - | - | - | 3321630 | 3262836 | - |

**Table 2.7:** SDG size in number of nodes. Points-to analysis variants: (1) type-based, (2) instance-base and (3) object-sensitive.

| size relative to | unstructured | | | object-tree | | | object-graph | | |
|---|---|---|---|---|---|---|---|---|---|
| object-graph (1) | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| *JavaCard* | | | | | | | | | |
| corporatecard | 0.98 | 1.06 | 1.86 | 1.12 | 1.10 | 2.16 | 1.00 | 1.03 | 1.99 |
| purse | 1.05 | 1.97 | 1.83 | 1.58 | 1.51 | 1.99 | 1.00 | 1.09 | 1.87 |
| safeapplet | 0.98 | 1.06 | 1.86 | 1.12 | 1.10 | 2.16 | 1.00 | 1.03 | 1.99 |
| wallet | 1.00 | 1.15 | 1.83 | 1.15 | 1.12 | 2.09 | 1.00 | 1.04 | 1.94 |
| *JRE* | | | | | | | | | |
| battleship | 4.05 | 20.52 | 28.34 | 2.45 | 2.44 | 7.72 | 1.00 | 1.00 | 2.60 |
| cloudstorage | 2.02 | 4.58 | 7.63 | 1.58 | 1.60 | 3.56 | 1.00 | 0.98 | 1.77 |
| clientserver | 1.16 | 1.53 | 2.66 | 0.96 | 1.02 | 1.99 | 1.00 | 1.00 | 2.10 |
| hybrid | 2.48 | 5.96 | 9.03 | 1.71 | 1.69 | 3.80 | 1.00 | 1.00 | 1.45 |
| *JavaGrande suite* | | | | | | | | | |
| barrier | 2.05 | 10.81 | 31.91 | 1.35 | 1.31 | 7.32 | 1.00 | 1.02 | 2.47 |
| crypt | 2.06 | 12.53 | 29.64 | 1.35 | 1.31 | 6.62 | 1.00 | 1.03 | 1.99 |
| forkjoin | 2.08 | 11.64 | 33.94 | 1.37 | 1.33 | 7.66 | 1.00 | 1.02 | 2.46 |
| lufact | 1.95 | 10.75 | 26.05 | 1.34 | 1.31 | 6.00 | 1.00 | 1.03 | 1.85 |
| moldyn | 1.80 | 9.41 | 28.40 | 1.26 | 1.23 | 6.33 | 1.00 | 1.04 | 2.52 |
| montecarlo | 2.90 | 25.83 | - | 1.36 | 1.24 | 8.80 | 1.00 | 1.09 | 3.22 |
| raytracer | 1.90 | 10.79 | 39.76 | 1.29 | 1.23 | 7.67 | 1.00 | 1.10 | 3.43 |
| series | 2.09 | 12.21 | 29.83 | 1.38 | 1.34 | 6.68 | 1.00 | 1.03 | 1.94 |
| sor | 2.13 | 12.50 | 29.17 | 1.36 | 1.32 | 6.53 | 1.00 | 1.03 | 1.90 |
| sparsematmult | 2.09 | 12.22 | 28.44 | 1.37 | 1.32 | 6.37 | 1.00 | 1.04 | 1.87 |
| sync | 1.99 | 11.01 | 34.06 | 1.33 | 1.29 | 7.68 | 1.00 | 1.02 | 2.64 |
| **average** | **1.93** | **9.34** | **20.35** | **1.39** | **1.36** | **5.43** | **1.00** | **1.03** | **2.21** |
| *J2ME* | | | | | | | | | |
| barcode | 1.52 | 2.32 | - | 1.17 | 1.10 | - | 1.00 | 1.01 | - |
| bexplore | 2.66 | 15.66 | - | - | - | - | 1.00 | 1.03 | - |
| j2mesafe | 1.71 | 2.27 | - | 2.87 | 4.49 | - | 1.00 | 1.02 | - |
| keepass | 1.93 | 6.70 | - | 2.61 | 3.26 | - | 1.00 | 1.02 | - |
| onetimepass | 1.63 | 55.44 | - | - | - | - | 1.00 | 1.01 | - |
| *JRE (large)* | | | | | | | | | |
| freecs | - | - | - | - | - | - | 1.00 | 0.86 | - |
| hsqldb | - | - | - | - | - | - | 1.00 | 0.98 | - |

**Table 2.8:** SDG size relative to the object-graph option with type-based points-to. Points-to analysis variants: (1) type-based, (2) instance-base and (3) object-sensitive.

significantly between them. For example bExplore contains about 1.2× the instructions of Barcode, yet the analysis takes about 20× longer. This inconsistency also shows when we compare the J2ME programs to JavaGrande examples. All programs have roughly about the same size of around 10000 bytecode instructions. While the JRE1.4 runtime library of the JavaGrande examples is far bigger then the J2ME library, still we were not able to analyze the J2ME programs with the object-sensitive points-to precision, while all JavaGrande programs except montecarlo could be analyzed. We explain this effect through the different program structures: The J2ME programs make heavy use of object-orientation due to GUI elements included from the runtime library. The JavaGrande programs focus on the computation of mathematical problems and make less use of object-orientation and the runtime library. Therefore the object-sensitive points-to analysis of the J2ME programs is more complex —due to more objects used— resulting in a longer runtime and more distinguishable points-to elements. This in turn leads to more parameter nodes and ultimately prohibits SDG computation in this scenario.

Taking a closer look at how much of the runtime is spent in the different phases of SDG computation, we notice that specifically the summary edge computation takes up a significant chunk. Table 2.6 shows which fraction of the analysis runtime is spent on summary computation. The unstructured parameter model leads to very long summary computation phases of 70% on average for the medium sized programs in our evaluation. Even with the least precise points-to analysis the summary phase takes up 19% of the computation time. The object-tree approach spends less time for summary computation. However, it shows that in contrast to the unstructured and object-graph model, summary computation takes more time with the less precise points-to. This is directly related to the cloning of sub-trees in case of aliasing. As previously mentioned we identified a problem in the object-tree algorithm as it creates additional parameter nodes whenever a potential aliasing is detected — which in turn occurs more often the less precise a points-to analysis is. Hence, with more parameter nodes in the graph the summary computation takes longer. The object-graph model performs as expected, where the summary computation time increases with the precision of the points-to analysis, as naturally more distinguishable memory locations lead to more parameter nodes in this approach.

In general, SDGs computed with the object-graph approach contain the fewest nodes. Table 2.7 shows the size of all computed SDGs and Table 2.8 shows the SDG size relative to the object-graph SDG with type-based points-to precision. The overall trend is similar to the runtime statistics —as expected, the larger the graph the longer it takes to compute. The unstructured model produces the largest SDGs: With the least precise points-to variant they are on average about 2 times larger then an SDG built with the object-graph. With increasing precision the difference becomes even more visible. SDG size is about 9× and 20× the size of the smallest variant, compared to an increase of only 1.03× and 2.21× with the object-graph model. Object-trees produce a smaller SDG, but are still on average about 1.4× to 5.4× the size of object-graph SDGs —depending on points-to precision. We expected larger object-tree SDGs for less precise points-to due to the cloning of subtrees, as explained earlier. However SDGs with the type-based points-to are only slightly larger then with instance-based points-to. This is because increased precision has many effects that result in very different outcomes depending on the parameter-model. On the one hand, with increased precision we can reduce the amount of potential targets detected for dynamic calls —resulting in fewer methods called. On the other hand more distinguishable points-to elements are computed, hence points-to sets become larger. A larger points-to set has no visible effect on object-trees, as parameter nodes are grouped by field-names. Less precise points-to can lead to cloning of subtrees which increases the size of the object-tree. However a less precise points-to analysis can also reduce the size of the resulting SDG, as call targets are distinguished in fewer contexts —leading to less cloned nodes in the call graph. Hence object-tree SDGs are not always strictly larger when computed with less precise points-to. However, with the unstructured approach —that creates a node for each points-to element— a precise points-to analysis with larger result sets is very noticeable and leads to a significant increase of the SDG size.

Overall object-graphs produce the smallest SDGs in the least amount of time. They enable us to analyze significantly larger programs as previously possible, as the freecs and hsqldb examples show. Aside from the improved runtime, the object-graph model is also able to provide precise results. In the following section we will take a closer look at the effects of parameter models on analysis precision.

| average size of | unstructured | | | object-tree | | | object-graph | | |
|---|---|---|---|---|---|---|---|---|---|
| random slice | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| *JavaCard* | | | | | | | | | |
| corporatecard | 45.54 % | 45.07 % | 44.13 % | 42.72 % | 39.91 % | 23.94 % | 19.25 % | 15.49 % | 8.45 % |
| purse | 68.86 % | 68.64 % | 68.64 % | 70.73 % | 66.45 % | 56.01 % | 14.06 % | 9.23 % | 2.91 % |
| safeapplet | 45.54 % | 45.07 % | 43.66 % | 44.13 % | 41.31 % | 23.94 % | 19.25 % | 15.49 % | 8.45 % |
| wallet | 53.58 % | 53.21 % | 52.45 % | 48.68 % | 44.53 % | 22.26 % | 22.26 % | 20.75 % | 9.06 % |
| *JRE* | | | | | | | | | |
| battleship | 51.90 % | 51.90 % | 50.63 % | 39.24 % | 34.18 % | 30.38 % | 46.84 % | 45.57 % | 41.77 % |
| cloudstorage | 63.72 % | 63.72 % | 63.26 % | 68.84 % | 67.44 % | 55.35 % | 63.26 % | 59.53 % | 57.21 % |
| clientserver | 66.29 % | 66.29 % | 62.92 % | 62.92 % | 60.67 % | 52.81 % | 59.55 % | 57.30 % | 47.19 % |
| hybrid | 42.11 % | 40.35 % | 39.18 % | 32.75 % | 19.88 % | 19.88 % | 28.65 % | 21.05 % | 21.05 % |
| *JavaGrande suite* | | | | | | | | | |
| barrier | 40.68 % | 39.83 % | 39.83 % | 33.90 % | 16.95 % | 10.17 % | 14.41 % | 14.41 % | 14.41 % |
| crypt | 24.14 % | 17.73 % | 16.75 % | 33.99 % | 17.24 % | 17.24 % | 18.23 % | 17.24 % | 16.26 % |
| forkjoin | 40.23 % | 40.23 % | 40.23 % | 59.77 % | 22.99 % | 14.94 % | 14.94 % | 14.94 % | 14.94 % |
| lufact | 24.80 % | 21.26 % | 20.87 % | 70.08 % | 16.54 % | 16.54 % | 17.72 % | 16.93 % | 16.93 % |
| moldyn | 29.94 % | 28.53 % | 28.53 % | 80.23 % | 37.57 % | 34.75 % | 22.32 % | 21.19 % | 21.19 % |
| montecarlo | 39.37 % | - % | - | 50.13 % | 27.30 % | 20.73 % | 11.02 % | 11.55 % | 9.97 % |
| raytracer | 29.31 % | 26.59 % | 25.38 % | 73.11 % | 23.56 % | 22.66 % | 23.87 % | 23.56 % | 21.45 % |
| series | 24.56 % | 16.67 % | 16.67 % | 22.81 % | 8.77 % | 7.89 % | 10.53 % | 10.53 % | 10.53 % |
| sor | 27.08 % | 20.83 % | 20.83 % | 20.14 % | 9.72 % | 9.72 % | 12.50 % | 12.50 % | 12.50 % |
| sparsematmult | 21.38 % | 14.48 % | 13.79 % | 46.90 % | 8.97 % | 8.97 % | 9.66 % | 8.28 % | 8.28 % |
| sync | 51.96 % | 51.96 % | 50.98 % | 64.71 % | 27.45 % | 15.69 % | 15.69 % | 14.71 % | 14.71 % |
| **average** | 41.63 % | 39.58 % | 38.82 % | 50.83 % | 31.13 % | 24.41 % | 23.37 % | 21.59 % | 18.80 % |
| *J2ME* | | | | | | | | | |
| barcode | 19.60 % | 18.15 % | - | 9.44 % | 6.35 % | - | 4.54 % | 4.17 % | - |
| bexplore | 61.23 % | 43.81 % | - | - | - | - | 43.72 % | 30.73 % | - |
| j2mesafe | 25.06 % | 17.81 % | - | 23.46 % | 14.86 % | - | 14.25 % | 13.39 % | - |
| keepass | 58.88 % | 57.93 % | - | 48.75 % | 26.65 % | - | 51.60 % | 36.88 % | - |
| onetimepass | 38.85 % | 10.05 % | - | - | - | - | 22.57 % | 9.78 % | - |
| *JRE (large)* | | | | | | | | | |
| freecs | - | - | - | - | - | - | 27.45 % | 16.39 % | - |
| hsqldb | - | - | - | - | - | - | 42.76 % | 28.95 % | - |

**Table 2.9:** Percent of application code on average in a backward slice. Points-to analysis variants: (1) type-based, (2) instance-base and (3) object-sensitive.

**Precision** Finding a good metric to measure and compare the precision of SDG computation algorithms is tricky. Counting the number of nodes or edges in the result does not work, as the number varies with parameter model and configuration and is only loosely related to precision. Sometimes more nodes is good, when they actually represent different locations or instructions in a different context. Other times —e.g. in case of subtree cloning of the object-tree model— multiple nodes exists that in fact represent the same location and thus do not enhance the quality of the result.

We chose to look at the slices that can be computed from the given SDGs, as slicing is the main application of dependency graphs [56,98, 82,70] and many advanced analyses —like our information flow control tool— depend on it. Naturally our goal is to build SDGs well suited to precise slicing. Hence it makes sense to judge the quality of our SDG computation with the average precision of a random slice. We measure the precision of a slice by counting the average number of source code lines it covers. Therefore we group all nodes in the graph by the source code line they correspond to. Then we run a backward slice for each source code line found. Finally we compute the percentage of source code lines on average in such a slice. So, a lower percentage implies better precision of the SDG. Table 2.9 shows the measured precision for each evaluated program. The numbers are quite low compared to the evaluation in our previous work [35,36], because we choose to ignore the effects of exceptions. In §2.4.1 we discussed the effects of exceptions on our analysis. It showed that the indirect information flow through exception often overshadows precision improvements in other areas, like pointer-analyses and parameter model. As we are interested in the effect of parameter-model and points-to analysis alone, we chose to run the evaluation with exceptions turned off.

**Impact of parameter model** Overall the difference in precision between parameter models is very visible. Object-graphs produce the best results in most cases with an average of 23% to 19% of source code lines in a random slice —depending on points-to precision. As expected, SDG precision increases when a better points-to analysis is used. Nevertheless the object-graph approach already provides good results in combination with the least precise points-to. Object-trees specifically struggle in this case with about 51% lines in a random slice for the least precise points-to option. The results of object-trees improve significantly with a better points-to analysis, up to a point where they have similar precision as object-graphs with an average of 31% and 24% compared to 22% and 19% of object-graphs. The unstructured model produces the worst results in most cases and also benefits the least from a better points-to analysis. The difference between the average of 42% to 39% seems not worth the effort, especially when the huge impact on runtime is also taken into account. Only in combination with the least precise points-to analysis, the unstructured approach has a better average precision than object-trees with 42% compared to 51%. However, this is not consistently the case, as seen in the J2ME examples where object-trees are the better option —as long as computation can be finished.

We explain the observed differences mostly through (1) the way field-candidates are grouped together through merging of similar candidates and (2) the reachability based escape analysis that removes parameter nodes corresponding to side-effects that are not visible outside the method scope.

Impact of (1): All three parameter models differ in the way parameter nodes are computed and hence candidates for parameter nodes are handled. The unstructured model directly creates nodes from field accesses and does not merge candidates at all, while object-trees merge nodes corresponding to the same field on-the-fly and object-graphs merge candidates after propagation. We notice that merging reduces node count which in turn improves runtime especially for larger programs, but it can also harm precision. When two candidates are merged information can be lost: Previously distinguishable side-effects are then represented by a single candidate. Merging at a later stage after propagation proves to be beneficial for precision of the result, as object-graphs produce the most precise result in most cases measured. While the unstructured model does not merge at all, it still usually fails to deliver precise results. We are going to show that this is due to the missing escape analysis.

Impact of (2): In contrast to the unstructured approach, object-trees and -graphs use an integrated escape analysis. Object-trees check whether a matching parent for a new tree-parameter exists, before adding it to the tree and object-graphs only add candidates to the mod-ref set reachable from a candidate already contained in the set. Both approaches help to improve precision. We measured the precision of object-graphs with a deactivated reachability analysis and observed that slice sizes were on average 3× to 5× as big, depending on the points-to precision. The better the points-to analysis the more visible this effect becomes.

**Impact of points-to precision** Points-to precision appears to have a large effect on overall SDG precision —especially in case of the object-tree model, which suffers the most from less precise points-to information. As expected, the average precision of object-tree SDGs is significantly worse in case of the least precise points-to analysis used. The SDG precision improves significantly with a better pointer-analysis. The unstructured model benefits only sightly from improved points-to information. We suspect the missing escape analysis as a cause: In contrast to object-trees and -graphs —where the escape analysis makes heavy use of points-to information to detect which nodes can be omitted— the unstructured model does use points-to information to reduce the number of nodes and thus the enhanced precision is not utilized.

Overall the effect of improved points-to precision differs from program to program. Sometimes —when critical program parts and locations can be distinguished— a huge improvement is possible, other times there seems to be almost no effect. Hence a single best option for all cases does not exists. Sometimes the enhanced precision due to object-sensitive points-to is worth the increased computation time, other times choosing a simple type-based approach yields the same result. For object-graphs and -trees the instance-based points-to analysis should almost always be preferred to the type-based approach, as the average runtime difference is not huge, but precision improves in many cases. The unstructured model works only well with the least precise type-based approach. All other options take significantly longer to compute, yet only slightly enhance the SDG precision.

### 2.6.6 Conclusion

We have shown that the parameter model has a huge influence on precision and scalability of the SDG computation. Overall, the object-graph model we introduced in this work (see Algorithm 2.5) proves to be the best option in most cases. It works well with different points-to analyses and almost always produces the smallest graphs in the least amount of time —all while the precision of the resulting SDG does not suffer. In fact, in most cases object-graphs achieved the most precise result of all available options. Hence we opted to use them as the standard option in Joana.

The object-tree model (see Algorithm 2.3) works reasonably well with smaller programs and precise points-to analysis. As expected this model has problems when aliasing —two variables or fields pointing to the same memory location— occurs, as it creates many additional nodes in this case. The reason for aliasing can either be that the aliasing in fact exists in the program or that the static approximation —the points-to analysis— issues a false alarm. Hence object-trees struggle with imprecise points-to information and larger programs. They also struggle with larger programs when a precise points-to analysis is used, as the number of aliases that are no false alarms can still be too much to handle. However, they remain a viable option for small to medium sized programs and precise points-to information.

The unstructured model of §2.6.1 works reasonably well with imprecise points-to information, as the number of nodes it creates is directly related to the number of distinguishable points-to set elements. So the more precise the points-to information gets, the more nodes are created and subsequently the longer the analysis takes. In addition the missing escape analysis hurts the precision of the result. Apart from its simple and straight forward implementation there are few reasons to use this model.

The precision gained through an advanced points-to analysis can be quite large, but varies depending on the nature and size of the program. We assume that some of the programs analyzed with a precise points-to analysis were too small and used too few objects to benefit from the precise information. Other, larger programs —like the J2ME examples— could have benefited from a more precise points-to analysis, but the analysis was not able to scale. We experimented with other promising BDD-based approaches to pointer-analysis, but those algorithms appeared to trade faster computation time for increased space requirements due to large intermediate results [53]. For example for a program as large as HSQLDB a object-sensitive analysis would take up several hundred gigabytes of disk space. These results pose a problem to the scalability of the subsequent parameter model and summary edge computation. As of today, the scalability of object-sensitive points-to analyses limits their application to smaller programs. Current ongoing work in our group focuses on points-to analysis specifically tailored to a given IFC problem.

This tailored analysis is a hybrid of an imprecise points-to approach used for most program parts and an advanced, precise analysis that applies only to specific program parts relevant for the IFC request. We expect this approach to provide a good trade-off between precision and scalability. As of now we suggest using the instance-based points-to approach as default —it combines good scalability with decent overall precision.

*I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone.*

Bjarne Stroustrup

# 3 A Modular Approach to Information Flow with SDGs

The SDG based algorithms presented so far in the previous chapter are whole-program analyses. Their disadvantage is that in order to work they need to analyze the whole-program —including all libraries and the runtime-system— at once. This can lead to significant scalability issues, due to the sheer size of a given program and libraries. It may even be impossible to do in the first place, as some of the libraries used may not be available at the time the analysis is run. For example one may want to analyze an encryption library for noninterference properties without knowing in which contexts and programs it will be used.

This chapter describes how precise SDG based IFC analyses can be extended to reason about code in unknown context. At first a motivating example shows the limits of current approaches, then we introduce our approach to remove these limitations. Section §3.1 contains a detailed overview of our approach and a discussion of its limitations. Section §3.2 covers information flow in unknown context in general, defines common notation and introduces the *monotonicity property* for context configurations. In §3.3 we present our language *FlowLess* that allows the specification of information flow properties for methods in unknown context and explain how existing analysis techniques can be extended to check these properties. Section §3.4 introduces techniques specifically tailored for our SDG based analyses: *conditional data dependencies* that only hold in certain contexts, an enhanced summary edge computation that accompanies conditional dependencies and a new way to precompute dependencies through so-called *access paths*.

```
1  class User {
2    char[] name;
3    char[] passwd;
4
5    public User(char[] n, char[] p) {
6      this.name = n;
7      this.passwd = p;
8    }
9
10   // does it return u.passwd?
11   static char[] getName(User u) {
12     char[] orig = u.name;
13     int len = orig.length;
14     char[] copy = new char[len];
15     for (int i = 0; i < len; i++) {
16       char ch = orig[i];
17       copy[i] = ch;
18     }
19     return copy;
20   }
21 }
22 class Context {
23   static void legal() {
24     char[] in1 = "public".toCharArray();
25     char[] in2 = "secret".toCharArray();
26     User u = new User(in1, in2);
27     printInfo(u);
28   }
29
30   static void illegal() {
31     char[] in1 = "public".toCharArray();
32     char[] in2 = "secret".toCharArray();
33     in1 = in2; // bug or malicious code
34     User u = new User(in1, in2);
35     printInfo(u);
36   }
37
38   static void printInfo(User u) {
39     char[] name = User.getName(u);
40     System.out.println(name); // maybe unsafe
41   }
42 }
```

**Figure 3.1:** An example of a method (`User.getName()`) that returns secret information depending on the context in which it is called.

**Information flow in unknown context — a motivating example** Figure 3.1 contains a program where the information flow properties of a method depend on the context in which it is used. Assume we want to guarantee that the method `getName()` of class `User` only returns the name of the user and never leaks information about the password. At first glance the method never touches the attribute `passwd` and therefore it should not reveal its content. However, this is not true if attributes `name` and `passwd` refer to the same location in memory. Then `getName()` also returns the content of `passwd` and subsequently method `printInfo` of class `Context` could reveal secret information. Class `Context` contains 2 methods that call `printInfo()`: Method `legal()` calls it with two separate parameters, so `getName()` acts as expected. Method `illegal()` contains an additional statement in l. 33 that sets both parameters to the same location leading to the aliasing of attributes `name` and `passwd`. So if class `User` is part of a library we have to analyze for information flow security without knowledge in which programs and contexts it will be used, theses effects have to be detected.

In general these effects are often very subtle and easily overlooked on manual inspection. Hence an automated analysis that guarantees to detect all possible effects is beneficial. We are now going to propose some general steps on how to tackle this problem for languages that allow aliasing, before we explain in more detail our solution that uses modular SDGs specifically tailored for the object-oriented language Java.

**Related work** IFC Analysis of program components without the knowledge of the whole system has been researched by several other projects. Components can occur in the form of *mobile code*, that is downloaded and plugged into an existing system. Examples are browser plug-ins or mobile phone applications.

The Mobius project [6] is one of the earliest and largest projects that aimed to provide a security infrastructure for Java programs in mobile phones. It was based on *proof carrying code* (PCC) and security type systems. The PCC approach delivers a program together with a formal proof of certain security properties, in the form of pre- and postconditions. After the program is downloaded, the proof is then checked at client side against the actual code. PCC works best, when the verification of the proof is significantly more efficient then building it from scratch. This way the client side has to do less work and can still verify the security properties without the need of a full-blown analysis. The IFC checker in Mobius uses a security type system for Java bytecode, hence it comes with the usual pros and cons of type system based approaches: It lacks the precision —especially flow-, context- and object-sensitivity— of more advanced program analysis techniques as used in Joana, but on the other hand its relatively simple nature leads to good scalability.

*Compositionality* is a property that is closely related to the analysis of mobile code. An analysis is called compositional if its intermediate results for each component of a large system can automatically guarantee the security property for the whole system. So if each component itself is found to be noninterferent, the system composed through language constructs such as if-then-else or parallel execution of the individual components is also noninterferent. This is a very nice property to have, but it does not work for all forms of noninterference [86] and it can lead to overly restrictive results which harms its practical usability.

More practical solutions sacrifice compositionality as well as soundness in order to achieve good scalability and few false alarms. One example for such an approach is the TAJ system [124] from IBM, which can analyze 500kLOC written in full Java. It is based on *thin slicing* —a special variant of slicing that ignores certain implicit flows. So TAJ misses illegal implicit flow, but the authors argue that those are rarely critical and the user benefits if the reported violations contain mostly true information leaks. Other studies [58] also suggest that a large number of false alarms can have similar negative effects on the system security as missed leaks due to soundness problems: If too many false alarms are reported, the user tends to ignore those security warnings and therefore misses also the valid ones. False alarms can either be reduced through unsound analysis techniques —such as ignoring implicit flows— or through improving the precision of the analysis [48]. In Joana we focus on improving the analysis precision, but we also include heuristics that categorize detected leaks. We can show for each of the detected leaks if it can only occur due to implicit flow, exceptions, or thread interferences. Its up to the user to decide on the severity of the leak based on its category.

In general only few of the available research tools for IFC analysis can handle a realistic language like full Java or C. Often, they focus on a small subset of a real language, where specific constructs such as exceptions or dynamic dispatch or aliasing are not allowed. Those tools also often lack precision: they are rarely flow-, context- or object-sensitive. Some can handle only small programs of 500 LoC instead of 500 kLoC, or they demand a high degree of user interaction in form of many annotations (e.g. JIF [94]). For IFC in mobile code precise interprocedural program analysis —to our best knowledge— has never been used.

## 3.1 Overview — Goals and limitations

We focus our efforts on the IFC analysis for enclosed modules of a program, such as library methods. Our goal is to provide information about the information flow inside a module in general and about how the flow is dependent on the modules context. When the module is to be integrated into a program, the programmer can lookup its IFC properties and decide if it is save to use. We also include the possibility to automatically use precomputed IFC summaries of the module to speed up the process of the IFC analysis for the whole program.

Figure 3.2 shows a high-level view of a program that uses a module. It passes two inputs to the module, an uncritical public (*low*) input and a secret (*high*) input, and it requires the output of the module to only

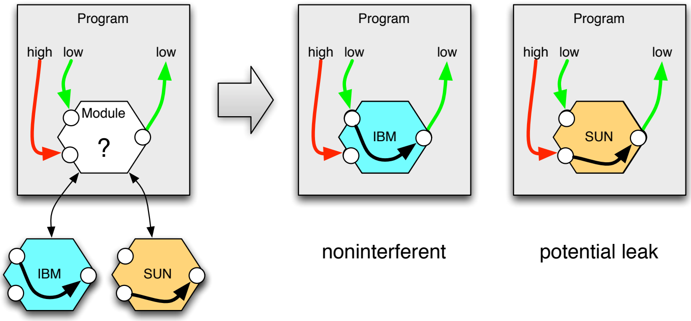


**Figure 3.2:** A program with two possible variants of a module. Only one variant can guarantee noninterference.

contain low information. Now the programmer has to choose which implementation of the module he is going to use in the program. He may choose between various libraries, in this example one from IBM and another from SUN. When plugged in the IBM variant does not leak high information to the low output and the program is noninterferent. Using the SUN variant may result in an information leak.

Our goal is to provide tools that help support the decision of the programmer. We introduce a new language called *FlowLess* in §3.3 that allows the programmer to specify information flow requirements the module needs to meet. In the current example he may specify that the second input is not allowed to influence the value of the output in any way. This specification can then be used to check which modules fulfill the requirements. We also include an inference analysis in §3.2.3 that detects under which minimal context preconditions the requirements are met by a given module. The example in Figure 3.1 shows that unforeseen aliasing can introduce unwanted information flow. We can detect that this unintentional leak only occurs in case of the aliasing of both input parameters and provide this information to the programmer.

Thus, we need to analyze the module in unknown alias context. One approach is to analyze the module in all possible *context configurations*. A context configuration describes the points-to configuration at the entrypoint of the module —the call site of the interface or entry method of the module. We can detect and enumerate all context configurations with the algorithm presented in §3.2.1. While enumerating all context configurations is theoretically possible, it is not feasible in practice due to the huge amount of possibilities. As a remedy we present a partial order on the context configurations of a module together with a *monotonicity property* in §3.2.2 that builds a connection between context order and information flow properties. We show that given two configurations $C_1 \leq C_2$ of a module $m$, all information flow inside $m$ in $C_1$ also may happen in $C_2$. Thus only additional flow can occur in $C_2$. This helps us search the space of context configurations in an efficient way, e.g. by using binary search, and it also allows us to detect the minimal and maximal flow inside a module. Therefore the monotonicity property greatly reduces the amount of context configurations we need to analyze, which in turn makes the automated inference of information flow relevant context conditions presented in §3.2.3 feasible.

In Section §3.3 we present the language *FlowLess* that is used to describe the information flow requirements of a module in detail. FlowLess allows the programmer to specify unwanted information flow between the input and output of a method as well as restrictions on the context configuration. In §3.3.2 we show how to compute the minimal and maximal context configurations that fulfill those restrictions and automatically generate stubs that call the module in these contexts. Aside from our own IFC analysis this allows us to run arbitrary whole-program analyses on modules in unknown context, as long as they do not break the monotonicity property.

Finally in §3.4 we extend our SDG format from §2.5.3 to include so-called *conditional data dependencies* that only hold in certain context configurations. This modular SDG contains precomputed information flow that applies to every possible context configuration together with conditional flow depending on the context. When we need to extract the information flow for a specific context, the matching conditional dependencies are activated and the adapted algorithm for summary edge computation in §3.4.4 is run. It computes the context specific summary in reduced runtime compared to a full-blown analysis and returns a SDG corresponding to the context. The precomputed modular SDG reduces the runtime and memory needed to compute a context specific SDG, which further reduces the overall runtime of our context inference analysis. Section §3.4.3 describes the suggested algorithm to compute the modular SDG for a component. It uses so-called *access paths* as an abstract context-independent description of memory locations. These paths allow us to detect context dependent data dependencies in the SDG and to extract the required context condition for each individual dependency.

**Limitations** Our modular analysis currently does not cover modules that include one or more of the following features.

**Callbacks** We do not support modules that contain callbacks to unknown code. The module needs to be enclosed in the sense that every method that is called from within the module has to be available at analysis time. Due to the nature of the Java language a single method can introduce almost arbitrary side-effects through static fields or use of reflection — please see §2.1 for details. This makes it hard to compute any meaningful information flow properties for a module that contains unknown code. However it seems possible to pose restrictions on the unknown code —like forbid visible side-effects or any information sources and sinks within the code— that allow us to narrow down the possible impact on the information flow of the module. These restrictions then would need to be checked as soon as the unknown code is available. This is future work that is not included in the scope of this thesis.

**Downcasts** We don't allow explicit downcasts in module code. Downcasts do not pose a principal problem to our analysis, but they can massively increase the number of potential side-effects the module may have and therefore harm the analysis scalability and precision. For example if we have to approximate the potential modifications a method can execute on a parameter of type `Object`, we only have to consider that attributes of `Object` may have been changed. However, if the method is allowed to downcast the parameter, it could potentially modify any attribute of any class, as all classes in Java directly or indirectly extend `Object`. An additional analysis that detects downcasts in the module can help tackle this problem, but as downcasts are in general a bad practice and should be avoided —especially in the security critical scenarios we are aiming at— we choose to forbid them.

**Reflection** As discussed in §2.1.4 reflection is a huge threat to security related applications in Java and should be avoided. It is possible to allow some less critical functionality —like loading of classes by name— and detect its effects with a sophisticated string analysis [83] or record the actual values with a dynamic analysis [16]. However these analyses only work in special cases and thus can not guarantee a conservative result in general. Therefore we currently do not support reflection in our analysis.

## 3.2 Information flow in unknown context

In this section we explain how we apply our static whole-program analysis to components in unknown alias context.

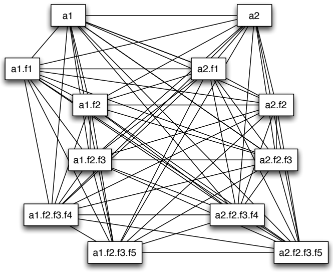
### 3.2.1 Enumerating all contexts

One idea to deal with components in unknown context is to analyze them in any context possible. This approach is straight forward and works at least in theory as long as the number of contexts is finite. Figure 3.3 illustrates the various potential alias contexts of a component and how the number of different contexts can be minimized through implicit constraints of object type and structure. In this example method `foo` has two parameters of the same type `A`. These parameters or any of their fields may be aliasing when `foo` is called. In order to find out which different alias configurations are possible we start by detecting fields reachable from each parameter. For recursive data structures such as linked lists the number can potentially be infinite, but in the current example each parameter can reach up to 5 fields. Including the parameters themselves method `foo` may access up to 12 different references. These references may be aliasing in certain contexts and be unique in others. The number of different context configurations depends on the aliasing possible between those references. We are now going to discuss 3 different approaches —with increasing finesse— that can be used to enumerate all different aliasing contexts.

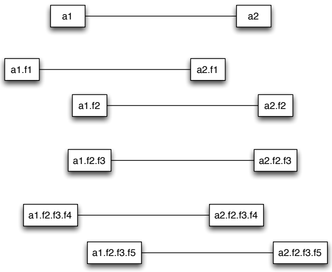
**Case (a)** In a first naive approach we can assume that each of these locations may be aliased to any other location. This results in the situation depicted at case (a) in Figure 3.3: Every location reachable is

```
class A { B f1; C f2; }
class B {}
class C { D f3; }
class D { E f4; F f5; }
class E {}
class F {}

static void foo(A a1, A a2) {
  ...
}
```

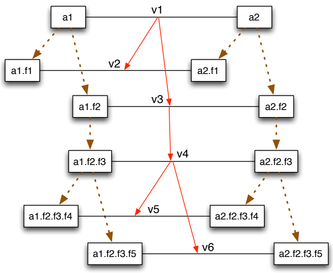
Example: A method with two parameters in unknown context



(a) Naive aliasing of locations: fully connected



(b) Type aware aliasing



(c) Type and structure aware aliasing

**Figure 3.3:** Example for enumerating alias contexts with different approaches.

shown as a node in the graph and edges between nodes show potential aliasing. Without restrictions on the aliasing the graph is complete with $\binom{\#nodes}{2}$ edges. For each edge we can decide independently if the aliasing is present or not, which results in a total of $2^{\#edges}$ context variations.

$$2^{\#edges} = 2^{\binom{\#nodes}{2}} = 2^{\binom{12}{2}} = 2^{66} \approx 1.6 * 10^{71}$$

The number of variations is obviously far too big in this case to analyze the component in each one, but if we only allow aliasing between locations whose types are compatible, the number is greatly reduced.

**Case (b)** The graph in case (b) of Figure 3.3 contains only edges between nodes of locations that refer to compatible types, where the static type checker allows both references to point to an object of the same type. In Java this is the case if both references have the same static type, or one refers to a subtype of the other reference. In the current example only aliasing between fields of the same name are possible, leading to a graph with only 6 edges. According to our previously derived formula, we get a total of 64 different aliasing contexts. While this number is still quite high for such a small example, it is already feasible to analyze the component in all variants.

$$2^{\#edges} = 2^{\sum_{\#types} \binom{\#nodes\_per\_type}{2}} = 2^{1+1+1+1+1+1} = 2^6 = 64$$

However we can still improve if we incorporate the object structure into the computation of different aliasing contexts.

**Case (c)** The graph in case (c) of Figure 3.3 visualizes the access structure of the parameter objects. It shows the possible aliasing between compatible types as in case (b) and the dependencies between the aliasing due the object structure. In case (b) we still included some alias configurations that are not possible in practice, because whenever two references point to the same location, all their fields must also be aliasing. It is impossible that `a1.f2` and `a2.f2` are aliasing, while `a1.f2.f3` and `a2.f2.f3` are not. In general we can rule out any alias configuration that violates the following statement.

$$alias(a,b) \implies \forall f \in \text{ Fields in type } a \text{ or } b : alias(a.f, b.f)$$

The number of valid alias configurations is the number of all permutations minus the number of violating permutations. In the current example we can rule out 51 of all 64 potential configurations, leaving only 13 valid configurations.

$$2^{\#edges} - \#violating = 2^6 - 51 = 13$$

The number of actual valid alias configurations is far less than the first naive approach suggests. Hence the analysis of a component in all possible configurations is not only theoretically possible but also practically feasible in some situations. However our evaluation shows that there are still components with more then $2^{10}$ in rare cases $2^{100}$ valid configurations, even when compatible types and object structure are incorporated.

**Computation of violation permutations** We can map the problem of finding violating permutations to the more general boolean satisfiability problem with disjunctions that have at most 2 elements (2-SAT). We model each potential alias edge in the graph as a boolean variable $v$ and add an implication from each variable $v$ to $v'$ if the corresponding aliases are dependent due to object structure. So in the current example we have variables $v_1, \ldots, v_6$ and we derive the implications from the respective object structure:

$$(v_1 \implies v_2) \wedge (v_1 \implies v_3) \wedge (v_3 \implies v_4) \wedge (v_4 \implies v_5) \wedge (v_4 \implies v_6)$$

Which resolves straight forward to a conjunction of disjunctions.

$$(\neg v_1 \vee v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_3 \vee v_4) \wedge (\neg v_4 \vee v_5) \wedge (\neg v_4 \vee v_6)$$

We can compute 13 different assignments for $v_1 \ldots v_6$ that satisfy the above formula. Therefore we know that 13 different valid and 51 invalid alias configurations exist.

We have shown that the explicit enumeration of alias configurations is possible and the number of valid configurations can be greatly reduced with the help of type and object structure information. However the number can still be impracticably large, so we suggest an alternative approach that allows us to infer information flow properties for all configurations where only few need to be analyzed.

### 3.2.2 Context configurations: Order and monotonicity

During our evaluation and upon a closer look at the underlying algorithms of our IFC analysis we noticed that more may-aliases in the context configuration imply more flow, that is more edges or paths in the PDG. Indeed, the set of may-alias relations forms a partial order (e.g. $\{(a,b)\} \subseteq \{(a,b),(b,c)\}$) with minimal and maximal elements, which induces an order on the resulting flows (e.g. $\{a.i \to^* c.i\} \subseteq \{a.i \to^* c.i, a.i \to^* b.i\}$), again with minimal and maximal elements. The latter provide limit cases for potential flow that can be used to quickly check for existing context configurations of a component that fulfills a given security property.

However this observation is not valid for any IFC analysis or the noninterference property in general. The analysis needs to meet certain requirements that lead to monotone analysis results with respect to the initial context configuration.

- The points-to analysis does not use strong updates and thus fulfills the forthcoming Lemma 3.1 and Lemma 3.2.
- Interprocedural killing-definitions that rely on global must-alias information are not applied. Local context independent killing-definitions however are allowed.
- The computation of control flow is purely intraprocedural.
- Aside from may-alias information no other context specific information is used during component analysis.

We observed that these requirements are met by most practical program analyses, especially those that build upon the well known static analysis frameworks WALA or Soot. More specialized approaches like the KeY framework would need to artificially relax their precision in order to achieve monotone analysis results.

We define a lattice on the context configuration elements and show a *monotonicity property* for any given configuration $C_1$, $C_2$ that if $C_1$ is smaller then $C_2$ ($C_1 \sqsubseteq C_2$) the analysis result for $C_1$ contains at most the same data dependencies as $C_2$. This monotonicity property can then be used by the concrete analysis to reason about its own results. It can be leveraged by any program analysis that applies interprocedural data dependence computation through points-to and alias analysis. In our IFC scenario we show that if a component does not contain leaks for context $C_2$ then it also does not contain leaks for any context $C_1 \sqsubseteq C_2$. By choosing $C_2$ as the greatest element of the context lattice we can also show the absence of security leaks for any possible context.

Given the context order $\sqsubseteq$ and a program component $P\|_m$ for method $m$ we will show that the following is always true for a noninterference analysis (§1.1.3) that meets above mentioned requirements.

$$C_1 \sqsubseteq C_2 \wedge P\|_m \text{ in } C_2 \text{ is noninterferent } \implies P\|_m \text{ in } C_1 \text{ is noninterferent}$$

We also write $C \models NI(P\|_m)$ to express that $P\|_m$ is noninterferent in context configuration $C$ .

$$C_1 \sqsubseteq C_2 \wedge C_2 \models NI(P\|_m) \implies C_1 \models NI(P\|_m)$$

In the following we proof the upcoming Theorem 3.2 on monotonicity of program slices which directly implies above statement. Therefore we start with a formal definition of the order of context configurations and show how a program component can be derived for a given method. Then we go into more detail about the properties a points-to and alias analysis has to fulfill in order for the monotonicity theorem to hold. In general we allow any form of points-to analysis as long as it does not apply strong updates. We define how initial context configurations are modified to derive the context configuration at a given statement and show how this information is used to compute data dependencies. With the help of these definitions we can proof that without strong updates the result of points-to computation is monotone with respect to the initial configuration (Lemma 3.1 and Lemma 3.2). Then we leverage this result for points-to computation to show monotonicity for dependencies in a PDG. Therefore we briefly recapitulate the different dependency types inside a PDG —control, direct data and heap data dependence— and show that only heap data dependence is influenced by the initial points-to configuration. As all other dependencies remain the same, it suffices to show that heap data dependencies are also monotone with respect to initial points-to configuration to finally proof the monotonicity for program slices.

**Order on context configurations** In the following we refer to points-to sets, points-to configurations and may-aliasing as defined by Definition 2.15 and Definition 2.16 in the previous Chapter 2 in §2.5.2. We define an order on context configurations through the aliasing properties they imply.

**Definition 3.1** (Context Order). *Given two points-to context configurations $C_1$ and $C_2$ the relation ($\sqsubseteq$ ) defines an order.*

$$C_1 \sqsubseteq C_2 := \forall v_1, v_2 \in Var_P : alias_{C_1}(v_1, v_2) \implies alias_{C_2}(v_1, v_2)$$

*The subset relation between the points-to sets implies the matching order of the contexts.*

$$\forall v \in Var_P : pts_{C_1}(v) \subseteq pts_{C_2}(v) \implies C_1 \sqsubseteq C_2$$

We are going to show the effect of initial context configurations on the information flow inside components. Therefore we start with a more detailed definition of components derived from a whole program. We call those components *program* and *method parts*.

**Program and method parts as components** We call a subset $P||_{s_1 \to s_2}$ of all statements in program $P$ a *program part* between $s_1$ and $s_2$ iff $P||_{s_1 \to s_2}$ contains all statements that may be executed after the execution of $s_1$ and before $s_2$ while $s_2$ is always executed after $s_1$.

$$P||_{s_1 \to s_2} := \{s \mid s \in P \wedge s \text{ may be executed after } s_1 \text{ and before } s_2\}$$

**Definition 3.2** (Method Part)**.** *A program part can be defined for every method $m$ by choosing $s_1$ as the entry statement of $m$ and $s_2$ as the set of all return statements in $m$.*

$$P||_m := P||_{entry(m) \to exit(m)}$$

*The program part $P||_m$ of a method $m$ is also called the* method part *of $m$.*

```
1  class A {
2      int i;
3  }
4
5  A v1; A v2; int v3;
6
7  void main() {
8      v1 = new A(); // $l_1$
9      v2 = new A(); // $l_2$
10     foo(); // $C_{foo@10}$
11     v3 = readInt; // secret input
12     if (v3 > 23)
13         v2 = v1;
14     foo(); // $C_{foo@14}$
15 }
16
17 void foo() {
18     v2.i = 42;
19     output();
20 }
21
22 void output() {
23     printInt v1.i; // public output
24 }
```

**Figure 3.4:** A program that calls method $foo$ in different points-to context configurations.

Note that the method part $P||_m$ does not only contain all statements of method $m$, but also all statements of methods that may be called during execution of $m$. For example the method part for method `foo` in Figure 3.4 contains all statements in `foo` as well as the statements in `output`.

**Points-to and aliasing** A points-to analysis as defined in §2.5.2 is a static analysis that decides for all variables $v \in Var_P$ and fields $l'.f$ of a program $P$ at which memory locations $l \in Loc_P$ they may point to during all possible executions of $P$. Due to the potentially unbounded number of memory locations a points-to analysis has to use *abstract heap locations* —which are equivalence classes of actual memory locations— during computation. The precision of the analysis is strongly related to the granularity of the instance and variable context that defines these equivalence classes. We will use Definition 2.13 of abstract heap locations, Definition 2.15 of points-to configurations and Definition 2.16 of may-aliasing in the latter.

If we assume a standard 0-1-CFA based points-to analysis, the example in Figure 3.4 contains two abstract heap locations $l_1 = A@[8]$ and $l_2 = A@[9]$. Location $l_1$ refers to the instance of class $A$ created at l. 8 and $l_2$ to the instance created at l. 9. The result of a points-to and may-alias analysis is a conservative approximation. So if two variables are may-aliased, they may point to the same location, but if they are not may-aliased, they are guaranteed to point to distinct locations. False positives may occur as locations are combined to equivalence classes. So a single element $l \in Loc_P$ may refer to multiple actual locations $(h_1, h_2)$ in memory. Even if two variables $v_1$ and $v_2$ point to the same abstract location $pts_C(v_1) = pts_C(v_2) = \{l\}$ they still may never point to the same actual memory location during program execution. For example with a type-based points-to analysis the instances of `A` created at l. 8 and l. 9 would be treated as the same abstract location $l = A@[\top]$.

**Points-to configuration for method parts** The points-to information for a method part $P\|_m$ depends on the context in which $m$ is called. We distinguish the *initial context configuration* —that describes the state of the memory upon method invocation— from the *intermediate contexts* for each statement to the *final context* at the end of the execution of $P\|_m$. The points-to analysis computes the intermediate and final contexts for $P\|_m$ from the initial context through a data flow analysis on the statements in the control flow graph of $P\|_m$.

For example the program in Figure 3.4 contains two calls to method $foo$. One in l. 10 and the other in l. 14. Depending on the context in which $foo$ is called, the effects vary. If $v1$ and $v2$ refer to the same memory location then $v2.i = 42$ also changes the value of $v1.i$ and thus $output()$

prints 42, if not $v1.i$ is left unchanged. At the call in l. 10 $v1$ and $v2$ are referring to two distinct locations while they may point to the same location at the call in l. 14. So method part $P||_{foo}$ is called in two different initial contexts: $C_{\mathrm{foo@10}}$ for the call in l. 10 is $\{v_1 \rightarrow l_1, v_2 \rightarrow l_2\}$ and $C_{\mathrm{foo@14}}$ for the call in l. 14 is $\{v_1 \rightarrow l_1, v_2 \rightarrow l_1, v_2 \rightarrow l_2\}$. Thus for the context configurations $C_{\mathrm{foo@10}}$, $C_{\mathrm{foo@14}}$ we get $C_{\mathrm{foo@10}} \sqsubseteq C_{\mathrm{foo@14}}$. This observation supports the suspected monotonicity property we are going to proof, as $P||_{foo}$ in context $C_{\mathrm{foo@14}}$ contains additional flow that results in a security leak.

In order to show how the initial points-to configuration influences the result of a points-to computation for a method part, we go into more detail about the actual computation. Therefore we define a *points-to update operation* for each instruction in the program according to Andersens subset-based approach. Note that the less precise approach from Steensgaard —that unifies the points-to sets of variables in an assignment— computes points-to sets that are always a superset of the result from Andersens approach. So our monotonicity proof also carries over to Steensgaard based points-to analyses.

**Definition 3.3** (Points-to Update Operation). *Given a points-to configuration C and an instruction s, the points-to update operation that incorporates the effect of s on the program memory described by C is as follows.*

$$update(C,s) := \begin{cases} C \cup \{l.f \rightarrow l' \mid a \rightarrow l, b \rightarrow l' \in C\} & \textit{if } s \equiv \text{"}a.f = b\text{"} \\ C \cup \{a \rightarrow l \mid b \rightarrow l', l'.f \rightarrow l \in C\} & \textit{if } s \equiv \text{"}a = b.f\text{"} \\ C \cup \{a \rightarrow l \mid b \rightarrow l \in C\} & \textit{if } s \equiv \text{"}a = b\text{"} \\ C \cup \{a \rightarrow l \mid l = Loc_P(s)\} & \textit{if } s \equiv \text{"}a = new \ldots\text{"} \\ C & \textit{else} \end{cases}$$

When we compute the points-to configuration for a given statement $s$ in the method part $P||_m$, we have to incorporate any possible effects that may have occurred before the execution of $s$. Therefore we apply all the effect of all statements that may be executed between the entry of $P||_m$ and $s$. We compute the points-to configuration for the call to foo in l. 10 of Figure 3.4 as follows: Given method part $P||_{main}$ with the initial empty points-to configuration $C_{\text{main}} = \{\}$ we get

$$\begin{aligned} C_{\text{foo@10}} &= \mathit{update}(\mathit{update}(C_{\text{main}}, \text{"}\texttt{v1 = new A()}\text{"}), \text{"}\texttt{v2 = new B()}\text{"}) \\ &= \mathit{update}(\{v_1 \rightarrow l_1\}, \text{"}\texttt{v2 = new B()}\text{"}) \\ &= \{v_1 \rightarrow l_1, v_2 \rightarrow l_2\} \end{aligned}$$

For any points-to analysis that uses this style of subset based update operations we can show the following two lemmas on monotonicity and order preservation.

**Lemma 3.1** (Monotone Update Operation). *For any context C and any statement s the following holds for the update operation defined in Definition 3.3.*

$$C \sqsubseteq \mathit{update}(C, s)$$

*Proof.* If $s$ is a field-update operation $''a.f = b''$, we know that $\mathit{update}(C, s) = C \cup \{l.f \rightarrow l' \mid a \rightarrow l, b \rightarrow l' \in C\}$. Obviously $C \sqsubseteq C \cup \ldots$ is always true. If $s$ is a field-get operation $''a = b.f''$ then $\mathit{update}(C, s) = C \cup \{a \rightarrow l \mid b \rightarrow l', l'.f \rightarrow l \in C\}$. As $C \sqsubseteq C \cup \ldots$ the lemma holds in this case. If $s$ is an assignment $''a = b''$ then $\mathit{update}(C, s) = C \cup \{a \rightarrow l \mid b \rightarrow l \in C\}$. Again $C \sqsubseteq C \cup \ldots$ is always true. The same argument holds if $s$ creates a new instance of a class $''a = \mathit{new} \ldots''$. The resulting configuration is again extending $C$: $\mathit{update}(C, s) = C \cup \{a \rightarrow l \mid l = \mathit{Loc}_P(s)\} = C \cup \ldots$. In all other cases we know that $\mathit{update}(C, s) = C$. So $C \sqsubseteq \mathit{update}(C, s) = C$ holds. □

Another important property is that the context order is preserved for any two initial contexts as long as the same update operations are performed upon both.

**Lemma 3.2.** *Given two contexts $C_1$, $C_2$ and a statement s the following is always true.*

$$C_1 \sqsubseteq C_2 \implies \mathit{update}(C_1, s) \sqsubseteq \mathit{update}(C_2, s)$$

*Proof.* We show that any element $x \in \mathit{update}(C_1, s)$ must also be present in $\mathit{update}(C_2, s)$. If $x \in C_1$ then $x \in C_2$ due to $C_1 \sqsubseteq C_2$. Thus $x \in \mathit{update}(C_2, s)$ due to Lemma 3.1. Otherwise if $x \notin C_1$ we distinguish 5 cases for operation $s$.

**field-update** If $s$ is a field-update operation $''a.f = b''$ we know that $x$ is of the form $l.f \to l'$, where $l \in pts_{C_1}(a)$ and $l' \in pts_{C_1}(b)$. Due to $C_1 \sqsubseteq C_2$ we also know that $l \in pts_{C_2}(a)$ and $l' \in pts_{C_2}(b)$. Therefore $l.f \to l'$ must also be part of $update(C_2, ''a.f = b'')$ due to Definition 3.3.

**field-get** If $s$ is a field-get operation $''a = b.f''$ we know that $x$ is of the form $a \to l$, where $l \in pts_{C_1}(l'.f)$ and $l' \in pts_{C_1}(b)$. Due to $C_1 \sqsubseteq C_2$ we also know that $l \in pts_{C_2}(l'.f)$ and $l' \in pts_{C_2}(b)$. Therefore $a \to l$ must also be part of $update(C_2, ''a = b.f'')$ due to Definition 3.3.

**assignment** If $s$ is an assignment $''a = b''$ we know that $x$ is of the form $a \to l$, where $l \in pts_{C_1}(b)$. Due to $C_1 \sqsubseteq C_2$ we also know that $l \in pts_{C_2}(b)$. Therefore $a \to l$ must also be part of $update(C_2, ''a = b'')$ due to Definition 3.3.

**new instance** If $s$ creates a new instance $''a = new \ldots''$ we know that $x$ is of the form $a \to l$, where $l = Loc_P(s)$. $a \to l$ is added to the context independent of the elements already present. So we get $x = a \to l \in update(C_2, ''a = new \ldots'')$.

**other operations** For all other operations we know $update(C_1, s) = C_1$. Therefore $x \notin C_2$ cannot occur given $x \in update(C_1, s) = C_1 \sqsubseteq C_2$.

□

We write $C[s]$ for the context configuration at statement $s$ computed by the points-to analysis with the initial configuration $C$. It is defined as follows.

**Definition 3.4** (Context at a Statement). *Given a statement $s$, an initial context $C$ and an initial statement $s_{init}$ we define $C[s]$ as the union of the results for each execution path from $s_{init}$ to $s$.*

$$C[s] := \bigcup_{p: s_{init} -cf\to^* s} update(C, p)$$

*with*

$$update(C, s_1 \text{-}cf\text{→} s_2 \ldots \text{-}cf\text{→} s_n) := update(update(C, s_1), s_2 \ldots \text{-}cf\text{→} s_n)$$

**Theorem 3.1** (Preservation of Context Order). *Given any two context configurations $C_1$, $C_2$ for method part $P||_m$ the following is always true.*

$$C_1 \sqsubseteq C_2 \implies \forall s \in P||_m : C_1[s] \sqsubseteq C_2[s]$$

*Proof.* Given two configurations $C_1$, $C_2$ with $C_1 \sqsubseteq C_2$ it suffices to show that $update(C_1, p) \sqsubseteq update(C_2, p)$ holds for any execution path $p$. This directly implies that the context order is also preserved on the union of the result for each path:

$$\begin{aligned} &\forall p : update(C_1, p) \sqsubseteq update(C_2, p) \\ \implies &\bigcup_{\forall p} update(C_1, p) \sqsubseteq \bigcup_{\forall p} update(C_2, p) \end{aligned}$$

Thus we assume an execution path $p = s_1 \text{-}cf\text{→} s_2 \ldots \text{-}cf\text{→} s_n$ and show that $update(C_1, p) \sqsubseteq update(C_2, p)$ holds by induction over $n$.

$n = 1$ If $p = s_1$ we get

$$\begin{aligned} update(C_1, p) &= update(C_1, s_1) \\ update(C_2, p) &= update(C_2, s_1) \end{aligned}$$

From Lemma 3.2 it directly follows that $update(C_1, s_1) \sqsubseteq update(C_2, s_1)$.

$n \to n+1$ With $p = s_1 \text{-}cf\text{→} \ldots s_n$ we know that

$$update(C_1, p) \sqsubseteq update(C_2, p)$$

holds. For $p' = s_1 \text{-}cf\text{→} \ldots s_n \text{-}cf\text{→} s_{n+1}$ we get

$$\begin{aligned} update(C_1, p') &= update(update(C_1, p), s_{n+1}) \\ update(C_2, p') &= update(update(C_2, p), s_{n+1}) \end{aligned}$$

So we can follow from Lemma 3.2 that

$$update(update(C_1, p), s_{n+1}) \sqsubseteq update(update(C_2, p), s_{n+1})$$

and thus $update(C_1, p') \sqsubseteq update(C_2, p')$.

□

**Dependencies in the PDG** We described the computation of a standard PDG in detail in Chapter 2. Therefore we only briefly recapitulate the parts relevant for this proof. Dependencies in the PDG are either control or data dependencies. Control dependencies (§2.3.4) are computed from the control flow graph (§2.3.3). They capture the dependencies that arise when the outcome of one statement controls if another statement is executed, e.g. the condition of the if statement trigger the execution of all statements in the if-block. Data dependencies are computed through a data flow analysis on the CFG (§2.3.5). They capture the dependencies between statements that potentially access the same memory location. Whenever a statement $s_1$ modifies a memory location that is read by another statement $s_2$ and $s_2$ is executed after $s_1$, $s_2$ is data dependent on $s_1$. We distinguish two different forms of data dependence: *direct data dependence* (dd) and *heap data dependence* (dh).

**Definition 3.5** (Direct Data Dependence)**.** *Direct data dependencies arise from definition and usage of variables. Let $def(s_1)$ be the set of variables defined by statement $s_1$ and $use(s_2)$ the set of variables used by $s_2$. Then $s_2$ is direct data dependent on $s_2$ ($s_1 \text{-}dd\text{→} s_2$), iff*

- *$s_1$ defines a variable $v$ that $s_2$ uses: $\exists v \in def(s_1) : v \in use(s_2)$*
- *A control flow path from $s_1$ to $s_2$ exists on which no other statement redefines $v$: $\exists s_1 \text{-}cf\text{→}^* s_2 \in ICFG : \nexists s' \in s_1 \text{-}cf\text{→}^* s_2 : v \in def(s')$*

Heap data dependencies occur only between statements that read or modify heap memory. In Java the only operations that can access heap values are field-get and set operations. For brevity we omit the special type of field access operations for arrays.

**Definition 3.6** (Field Access Operation)**.** *A field access is a tuple $(v, f)$ where $v$ is a variable and $f$ is a field name. The value of $v$ is used as a pointer to a memory cell in the heap and $f$ is interpreted as an offset specific to the field. The fields that can be accessed depend on the type of the object that $v$ refers to. Let $Fields_v$ be the set of all fields possible for the type of $v$. We distinguish two kinds of field access operations.*

**Field-get operation**

$$\text{field-ref}(v, f) : v, v' \in Var_P, f \in Fields_v : v' = v.f$$

**Field-set operation**

$$\text{field-mod}(v, f) : v \in Var_P, f \in Fields_v : v.f = \textit{<expr>}$$

**Definition 3.7** (Heap Data Dependence). *Heap data dependencies cover the dependencies between field access operations. Two statements $s_1, s_2$ are heap data dependent ($s_1 \xrightarrow{dh} s_2$) iff*

- *$s_1$ is a* field-mod$(v, f)$ *operation*
- *$s_2$ is a* field-ref$(v', f')$ *operation*
- *Both operations access the same field at potentially the same location: $f = f' \wedge alias(v, v')$*
- *There is a realizable path from $s_1$ to $s_2$ in the interprocedural control flow graph: $\exists s_1 \xrightarrow{cf}{}^{*} s_2 \in ICFG$*
- *There is no killing write access to the same location along every path between $s_1$ and $s_2$: $\forall p : s_1 \xrightarrow{cf}{}^{*} s_2 \nexists s_{kill} \in p$ where $s_{kill}$ is a* field-mod$(v_{kill}, f)$ *with must-alias$(v_{kill}, v')$.*

In order to incorporate killing definitions in the computation of heap data dependencies a so-called *must-alias* analysis is needed. A global must-alias analysis for a real world language like Java is due to its complexity not part of general program analysis frameworks. As previously mentioned in §3.2.2 it is in general impossible to decide if two operations always refer to the same memory location, as the locations computed by the points-to analysis are equivalence classes of the actual locations. Therefore in our analysis setup we do not apply a global must-alias analysis and always assume that no killing definition exists. We may however use local context independent optimizations that detect killing definitions for the intraprocedural case.

Figure 3.5 shows the direct and heap data dependencies for our example program. It contains a single heap data dependency from $v2.i = 42$ to *printInt* $v1.i$. We are going to show that the presence of this dependency depends on the context in which the method part $P||_{foo}$ is executed.

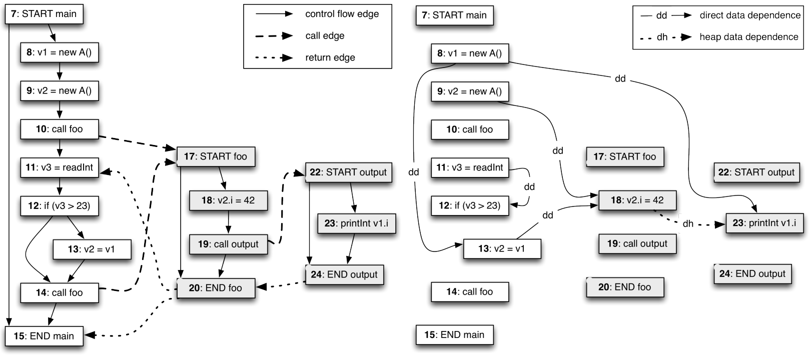


**Figure 3.5:** The left side shows the control flow graph for the example of Figure 3.4. The grey nodes belong to method part $P||_{foo}$. The right side shows direct and heap data dependencies.

**Data dependence in varying contexts** The points-to context has an influence on the data dependencies of a method part. Specifically the heap data dependencies use alias information to compute if two statements may access the same location. In our example program the method part $P||_{foo}$ contains a heap data dependency from statement $s_{18} : v2.i = 42$ to $s_{23} : printInt\ v1.i$. This dependency holds as long as $alias(v2, v1)$ holds. For the previously defined context configurations $C_{\text{foo@10}}$, $C_{\text{foo@14}}$ we get the following alias results: $alias_{C_{\text{foo@11}}}(v2, v1)$:

$$pts_{C_{\text{foo@10}}}(v2) \cap pts_{C_{\text{foo@10}}}(v1) = \{l_2\} \cap \{l_1\} = \emptyset$$

and $alias_{C_{\text{foo@14}}}(v2, v1)$:

$$pts_{C_{\text{foo@14}}}(v2) \cap pts_{C_{\text{foo@14}}}(v1) = \{l_1, l_2\} \cap \{l_1\} = \{l_1\}$$

So the heap data dependency is only present in the context $C_{\text{foo@14}}$. We write $s_{18} \xrightarrow{\cancel{dh}}_{C_{\text{foo@10}}} s_{23}$ and $s_{18} \xrightarrow{dh}_{C_{\text{foo@14}}} s_{23}$ as abbreviation for heap data dependencies under the given context configurations.

**Monotonicity in PDGs** We are now going to show that the monotonicity properties (Lemma 3.1 and Lemma 3.2) of the applied points-to computation implies a similar property for the dependencies in the computed PDG.

**Lemma 3.3** (Monotonicity of Dependencies in a PDG)**.** *Given a program part $P\|_m$ for method m and two initial context configurations $C_1$ and $C_2$ the following holds.*

$$C_1 \sqsubseteq C_2 \implies \forall s_1 \rightarrow s_2 \in PDG_{C_1}(P\|_m) : s_1 \rightarrow s_2 \in PDG_{C_2}(P\|_m)$$

*Proof.* We distinguish the different type of dependencies inside a PDG.

**control dependence** If $s_1 \xrightarrow{cd} s_2$ we know due to the Definition 2.3 of control dependence that the dependency is computed from the control flow graph. The computation of the CFG is independent of the initial context, therefore the dependency is present in any variant of $PDG(P\|_m)$.

**local data dependence** If $s_1 \xrightarrow{dd} s_2$ we know due to the Definition 2.3 of local data dependence that the dependency is independent of any points-to information. Therefore it is present in any variant of $PDG(P\|_m)$.

**heap data dependence** If $s_1 \xrightarrow{dh} s_2$ the dependency is only present if $s_1$ and $s_2$ are aliasing. Lets assume that the heap data dependency is not part of $PDG_{C_2}(P\|_m)$ but part of $PDG_{C_1}(P\|_m)$:

$$s_1 \xrightarrow{dh}_{C_1} s_2 \land s_1 \xrightarrow{\cancel{dh}}_{C_2} s_2$$

As $s_1 \xrightarrow{\cancel{dh}}_{C_2} s_2$ at least one condition of Definition 3.7 is violated. Due to $s_1 \xrightarrow{dh}_{C_1} s_2$ we know that $s_1$ is a field-mod operation $"v_1.f_1 = v_3'"$ and $s_2$ is a field-read operation $"v_4 = v_2.f_2"$. Also $f_1 = f_2$ and $alias_{C_1}(v_1, v_2)$ as well as a path in the $ICFG\|_m$ exists ($\exists s_1 \xrightarrow{cs} s_2 \in ICFG\|_m$). As all these conditions are also true in context $C_2$ only the aliasing $alias_{C_2}(v_1, v_2)$ may have changed. Because $s_1 \xrightarrow{\cancel{dh}}_{C_2} s_2$ has to hold we have to assume that $\neg alias_{C_2}(v_1, v_2)$ is true.

Due to Theorem 3.1 we know that

$$C_1[s_1] \sqsubseteq C_2[s_1] \land C_1[s_2] \sqsubseteq C_2[s_2]$$

and therefore

$$alias_{C_1}(v_1, v_2) \implies alias_{C_2}(v_1, v_2)$$

Leading to a contradiction as $\neg alias_{C_2}(v_1, v_2)$ as well as $alias_{C_1}(v_1, v_2)$ has to hold. So $s_1 \xrightarrow{\cancel{dh}}_{C_2} s_2$ is not true and $s_1 \xrightarrow{dh}_{C_2} s_2$ has to exist.

□

**Theorem 3.2** (Monotonicity of Slices in a PDG). *Given a program part $P\|_m$ for method m and two initial context configurations $C_1$ and $C_2$ the following holds for any PDG computed with a points-to analysis that fulfills Lemma 3.1 and Lemma 3.2.*

$$C_1 \sqsubseteq C_2 \implies \forall s \in P\|_m : slice(PDG_{C_1}(P\|_m), s) \subseteq slice(PDG_{C_2}(P\|_m), s)$$

*Proof.* From Lemma 3.3 we can deduce that all paths $s_1 \to \ldots \to s_n \to s \in PDG_{C_1}(P\|_m)$ must also be present in $PDG_{C_2}(P\|_m)$. Subsequently all elements in the slice of $s$ in $PDG_{C_1}(P\|_m)$ need to be contained in the slice of $s$ in $PDG_{C_2}(P\|_m)$. □

**Conclusion** We have shown through Theorem 3.2 that slices of method parts are monotone with respect to the initial points-to configuration if the underlying analysis fulfills certain very common properties — as our analysis tool Joana does. So whenever $C_1 \sqsubseteq C_2$ by the context order $\sqsubseteq$ defined in Definition 3.1, we know that every dependency in $C_1$ is also present in $C_2$. This enables us to reason about dependencies in other contexts $C_3$ without the need to run the analysis for $C_3$: If $C_3 \sqsubseteq C_2$ and we want to know if $s_1 \xrightarrow{dh}_{C_3} s_2$, we only need to look at the result for $C_2$ and see if $s_1 \xrightarrow{dh}_{C_2} s_2$ is present. If $s_1 \xrightarrow{\cancel{dh}}_{C_2} s_2$ we know for sure that $s_1 \xrightarrow{\cancel{dh}}_{C_3} s_2$ is also true. On the other hand if $C_1 \sqsubseteq C_3$ and we know that $s_1 \xrightarrow{dh}_{C_1} s_2$ we also know that $s_1 \xrightarrow{dh}_{C_3} s_3$ holds. So before we run an analysis for the context $C_3$ we can lookup if two context $C_1$, $C_2$ with $C_1 \sqsubseteq C_3 \sqsubseteq C_2$ exists and if analysis results for $C_1$ and $C_2$ have already been computed. If so we can use them to approximate the result for $C_3$. In the next section we show how to use this property to infer relevant context conditions that can guarantee the absence of illegal information flow.

### 3.2.3 Inferring relevant context conditions

In this section we show how the monotonicity property helps us to detect if and under which conditions a method part $P\|_m$ is guaranteed to be noninterferent. These so-called *relevant context conditions* allow us to decide if $P\|_m$ is noninterferent for any given context configuration $C$ without the need to analyze $P\|_m$ in configuration $C$ . Therefore we propose the following 3 basic analysis steps.

1. Check if $P\|_m$ is noninterferent independent of its context configuration. If true we don't need to infer any conditions, as $P\|_m$ is always noninterferent. If false we continue with step 2.
2. Check if $P\|_m$ is noninterferent for some context configurations. If false no condition exists under which the noninterference of $P\|_m$ can be guaranteed. If true we continue with step 3.
3. Infer relevant conditions —e.g. "parameter $a$ and $b$ are not aliasing"— on the context configurations that tell us if a configuration can guarantee noninterference for $P\|_m$.

The monotonicity property allows us to quickly check if (1) $P\|_m$ is noninterferent independent of the context it is called in and (2) if there exists at least some context configurations in which we can verify its noninterference. For (1) we analyze $P\|_m$ in the *maximal context configuration* $C_{\text{max}}$ with $\forall C : C \sqsubseteq C_{\text{max}}$. If $P\|_m$ is noninterferent in $C_{\text{max}}$ we automatically know that it is noninterferent in any other context. For (2) we analyze $P\|_m$ in the *minimal context configuration* $C_{\text{min}}$ with $\forall C : C_{\text{min}} \sqsubseteq C$. If $P\|_m$ is noninterferent in $C_{\text{min}}$ we know that at least a single configuration exists in which $P\|_m$ is noninterferent. Thus further analysis that detects which configurations exactly make or break the noninterference of $P\|_m$ should be applied.

This approach only works if a maximal and minimal context configuration exist. Therefore we define $C_{\text{min}}$, $C_{\text{max}}$ as follows and show that they are indeed the maximal and minimal element of all context configurations for $P\|_m$

**Definition 3.8** (Minimal and Maximal Context Configurations)**.** *The minimal $C_{min}$ and maximal context configuration $C_{max}$ for a component $P\|_m$ are*

$$C_{min} := \{\} \qquad\qquad C_{max} := \bigcup_{\forall \text{ valid configuration } C \text{ of } P\|_m} C$$

*A* valid configuration *is a context configuration that only contains aliases that fulfill the conditions presented in §3.2.1 case c). Note that the unification of valid configurations always results in a valid configuration.*

**Lemma 3.4.** *$C_{min}$ and $C_{max}$ are the minimal and maximal elements of all context configurations for $P\|_m$.*

$$\begin{aligned} C_{min} \not\models NI(P\|_m) &\implies \forall C : C \not\models NI(P\|_m) \\ C_{max} \models NI(P\|_m) &\implies \forall C : C \models NI(P\|_m) \end{aligned}$$

*Proof.* If $C_{\text{min}}$ is not minimal then $\exists C : C \sqsubseteq C_{\text{min}}$ and thus $\exists (a, b) \in C_{\text{min}} : (a, b) \notin C$. As $C_{\text{min}} = \{\}$ no such element can exist. Therefore $C_{\text{min}}$ is the minimal configuration.

Given $C_{\text{min}} \not\models NI(P\|_m)$: If $\exists C : C \models NI(P\|_m)$ then $C_{\text{min}} \sqsubseteq C$ and due to the monotonicity property $C_{\text{min}} \models NI(P\|_m)$ must be true, which contradicts the premise. So $C$ cannot exist.

If $C_{\text{max}}$ is not maximal then $\exists C' : C_{\text{max}} \sqsubseteq C'$. As $C$ needs to be a valid context configuration of $P\|_m$ we know that $C_{\text{max}}$ contains all elements of $C\,'$ per definition. Thus $C' \sqsubseteq C_{\text{max}}$ and therefore $C' = C_{\text{max}}$.

Given $C_{\text{max}} \models NI(P\|_m)$: If $\exists C : C \not\models NI(P\|_m)$ then also $C \sqsubseteq C_{\text{max}}$ and due to the monotonicity property $C_{\text{max}} \not\models NI(P\|_m)$ must be true, which contradicts the premise. So $C$ cannot exist. □

When we detect through (2) that $P\|_m$ is noninterferent in some context configurations, our goal is to (3) infer under which concrete context configurations $P\|_m$ is guaranteed noninterferent. We define $\mathbb{C}_{valid}$ as the set of all valid context configurations that fulfill the noninterference property for $P\|_m$. Subsequently we show how to detect if a given configuration $C$ is part of $\mathbb{C}_{valid}$ without the need to compute all elements of $\mathbb{C}_{valid}$.

**Definition 3.9** (Valid Fulfilling Configurations). *Given a method part $P\|_m$ the set of valid fulfilling context configurations $\mathbb{C}_{valid}$ is defined as*

$$\mathbb{C}_{valid} := \{C \mid C \models NI(P\|_m)\}$$

Given $\mathbb{C}_{valid}$ we can verify that $P\|_m$ is noninterferent in a configuration $C$ by checking if $C \in \mathbb{C}_{valid}$. Due to the monotonicity property we don't need to find an exact match in $\mathbb{C}_{valid}$, it suffices if we are able to find a configuration $C' \in \mathbb{C}_{valid}$ where $C \sqsubseteq C'$. Therefore we only need to find the *maximal fulfilling context configurations* in $\mathbb{C}_{valid}$ and are still able to check noninterference for any given configuration.

**Definition 3.10** (Maximal Fulfilling Configurations). *Given a component $P\|_m$ the set maximal fulfilling context configurations $\uparrow \mathbb{C}_{valid}$ contains only the maximal elements of $\mathbb{C}_{valid}$.*

$$\uparrow \mathbb{C}_{valid} := \{C \mid C \in \mathbb{C}_{valid} \wedge \nexists C' \in \mathbb{C}_{valid} : C \neq C' \wedge C \sqsubseteq C'\}$$

**Lemma 3.5.** *Checking if a configuration $C$ is in $\mathbb{C}_{valid}$ is equivalent to checking if $C$ is a subcontext of any element in $\uparrow \mathbb{C}_{valid}$.*

$$C \in \mathbb{C}_{valid} \equiv \exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$$

*So we can use $\uparrow \mathbb{C}_{valid}$ to verify noninterference.*

$$\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C' \equiv C \models NI(P\|_m)$$

*Proof.* First part of the lemma: $\implies$ : Given $C \in \mathbb{C}_{valid}$ we distinguish two cases:

$\nexists C' \in \mathbb{C}_{valid} : C \neq C' \wedge C \sqsubseteq C'$ Due to Definition 3.10 $C \in \uparrow \mathbb{C}_{valid}$. As $C \sqsubseteq C$ we get that $\exists C'' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C''$ with $C = C''$.

$\exists C' \in \mathbb{C}_{valid} : C \neq C' \wedge C \sqsubseteq C'$ If $C' \in \uparrow \mathbb{C}_{valid}$ obviously $\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$ is true. If $C' \notin \uparrow \mathbb{C}_{valid}$ through Definition 3.10 $C'' \in \mathbb{C}_{valid}$ exists with $C' \sqsubseteq C''$. As the number of elements in $\mathbb{C}_{valid}$ are finite $\exists C_n \in \mathbb{C}_{valid}$ with $C \sqsubseteq C' \sqsubseteq C'' \sqsubseteq \ldots \sqsubseteq C_n$ where $\nexists C_{n+1} \in \mathbb{C}_{valid}$ with $C_n \neq C_{n+1} \wedge C_n \sqsubseteq C_{n+1}$. Thus $C_n \in \uparrow \mathbb{C}_{valid}$ and $C \sqsubseteq C_n$.

$\impliedby$ : Given $\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$ we know due to Definition 3.10 that $C' \in \mathbb{C}_{valid}$ and thus $C' \models NI(P\|_m)$. As $C \sqsubseteq C'$ due to the monotonicity property $C \models NI(P\|_m)$ is true.

Second part: $\implies$ : Given $\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$ we know that $C' \in \mathbb{C}_{valid}$ and $C' \models NI(P\|_m)$. Therefore $C \models NI(P\|_m)$.

$\impliedby$ : Given $C \models NI(P\|_m)$ we know that $C \in \mathbb{C}_{valid}$. Due to the first part of this lemma $\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$ must hold. □

We further minimize the effort for checking a given configuration $C$ with so-called *alias conditions* derived from the configurations in $\uparrow \mathbb{C}_{valid}$. These conditions are a boolean formula of aliasing predicates. If $C$ satisfies these conditions we know that $C' \in \uparrow \mathbb{C}_{valid}$ exists where $C \sqsubseteq C'$ and thus $C \models NI(P\|_m)$.

**Definition 3.11** (Alias Conditions of Context Configurations). *An alias condition $Cond(C)$ of a given context configuration $C$ is a conjunction of not-alias predicates of all variables and fields that are not aliased in $C$.*

$$Cond(C) := \bigwedge_{(a,b)\in C_{max}\setminus C} \neg alias(a,b)$$

*The alias condition for a set of configurations $\mathbb{C}$ is the disjunction of the conditions for all elements.*

$$Cond(\mathbb{C}) := \bigvee_{C\in\mathbb{C}} Cond(C) = \bigvee_{C\in\mathbb{C}} \bigwedge_{(a,b)\in C_{max}\setminus C} \neg alias(a,b)$$

The alias conditions derived from a single context configuration maintain the monotonicity property. This allows us to check if $C_1$ is a subcontext of $C_2$ by verifying that $C_1$ satisfies $Cond(C_2)$.

**Lemma 3.6** (Monotonicity of Alias Conditions).

$$C_1 \sqsubseteq C_2 \equiv Cond(C_1) \implies Cond(C_2)$$

*Proof.* $\implies$ : If $C_1 \sqsubseteq C_2$ we also know that $C_{\max} \setminus C_2 \sqsubseteq C_{\max} \setminus C_1$. Therefore

$$Cond(C_1) = Cond(C_2) \wedge \bigwedge_{(a,b)\in C_2\setminus C_1} \neg alias(a,b)$$

Thus $Cond(C_1) \implies Cond(C_2)$.

$\impliedby$ : If $Cond(C_1) \implies Cond(C_2)$ we know that

$$\bigwedge_{(a,b)\in C_{\max}\setminus C_1} \neg alias(a,b) \implies \bigwedge_{(a,b)\in C_{\max}\setminus C_2} \neg alias(a,b)$$

Therefore $C_{\max} \setminus C_2 \sqsubseteq C_{\max} \setminus C_1$. Which leads to $C_1 \sqsubseteq C_2$. □

We write $C_1 \models Cond(C_2)$ if $C_1$ satisfies the alias conditions derived from $C_2$.

$$C_1 \models Cond(C_2) := \bigwedge_{(a,b)\in C_{\max}\setminus C_2} \neg alias_{C_1}(a,b)$$

From Lemma 3.6 it follows that

$$C_1 \models Cond(C_2) \equiv C_1 \sqsubseteq C_2$$

Hence we can use the alias conditions $Cond(\uparrow \mathbb{C}_{valid})$ to check if a configuration $C$ is a subcontext of any configuration in $\uparrow \mathbb{C}_{valid}$.

**Lemma 3.7.** *Iff a configuration $C$ fulfills the alias conditions of a set of configurations $\mathbb{C}$, the set must contain a configuration $C'$ with $C \sqsubseteq C'$.*

$$C \models Cond(\mathbb{C}) \equiv \exists C' \in \mathbb{C} : C \sqsubseteq C'$$

*Proof.* $\Longrightarrow$ : If $C \models Cond(\mathbb{C})$ we know due to Definition 3.11 that $C$ satisfies the alias condition of at least a single configuration $C' \in \mathbb{C}$. So $C \models Cond(C')$ and thus $C \sqsubseteq C'$.

$\Longleftarrow$ : If $\exists C' \in \mathbb{C} : C \sqsubseteq C'$ we know that $C \models Cond(C')$ and therefore $C \models Cond(\mathbb{C})$. □

**Theorem 3.3.** *In order to check $P\|_m$ for noninterference in context configuration $C$ , it is sufficient to check if $C$ satisfies the alias conditions derived from the maximal fulfilling configurations $\uparrow \mathbb{C}_{valid}$.*

$$C \models Cond(\uparrow \mathbb{C}_{valid}) \equiv C \models NI(P\|_m)$$

*Proof.* $\Longrightarrow$ : If $C \models Cond(\uparrow \mathbb{C}_{valid})$ we know due to Lemma 3.7 that $\exists C' \in \uparrow \mathbb{C}_{valid} : C \sqsubseteq C'$. By Lemma 3.5 $C \models NI(P\|_m)$ must hold.

$\Longleftarrow$ : If $C \models NI(P\|_m)$ we know due to Definition 3.9 that $C \in \mathbb{C}_{valid}$. Due to Lemma 3.5 $C' \in \uparrow \mathbb{C}_{P\|_m}$ exists with $C \sqsubseteq C'$. Thus due to Lemma 3.7 $C \models Cond(\uparrow \mathbb{C}_{P\|_m})$. □

The alias conditions in $Cond(\uparrow \mathbb{C}_{valid})$ tell us exactly for which configurations the method part $P\|_m$ is noninterferent. Therefore we also call $Cond(\uparrow \mathbb{C}_{valid})$ the *relevant context conditions* of $P\|_m$. Note that this implies that the conditions derived from the whole set of valid fulfilling configurations $Cond(\mathbb{C}_{valid})$ is equivalent to the conditions derived only from the set of maximal fulfilling configurations $Cond(\uparrow \mathbb{C}_{valid})$ .

In general we can remove any context configuration $C_1$ from a set of configurations $\mathbb{C}$ without changing $Cond(\mathbb{C})$ as long as $\mathbb{C}$ still contains a configuration $C_2$ with $C_1 \sqsubseteq C_2$.

**Lemma 3.8.** *Non-maximal context configurations do not change the alias condition of a set of configurations.*

$$C_1, C_2 \in \mathbb{C} : C_1 \neq C_2 \wedge C_1 \sqsubseteq C_2 \implies Cond(\mathbb{C}) \equiv Cond(\mathbb{C} \setminus \{C_1\})$$

*Proof.* $Cond(\mathbb{C}) = Cond(\mathbb{C} \setminus \{C_1, C_2\} \cup \{C_1\} \cup \{C_2\}) = Cond(\mathbb{C} \setminus \{C_1, C_2\}) \vee Cond(C_1) \vee Cond(C_2)$. As $C_1 \sqsubseteq C_2$ due to Lemma 3.6 $Cond(C_1) \implies Cond(C_2)$ and thus $Cond(C_1) \vee Cond(C_2) = Cond(C_2)$.
So we get $Cond(\mathbb{C} \setminus \{C_1, C_2\}) \vee Cond(C_1) \vee Cond(C_2) = Cond(\mathbb{C} \setminus \{C_1, C_2\}) \vee Cond(C_2) = Cond(\mathbb{C} \setminus \{C_1, C_2\} \cup \{C_2\}) = Cond(\mathbb{C} \setminus \{C_1\})$ □

This allows us to reason about the conditions derived from a set of context configurations without the need to look at all elements of the set. Our goal is to compute the relevant context conditions $Cond(\mathbb{C}_{valid})$ that tell us for every configuration if $P\|_m$ is noninterferent. Through Theorem 3.3 we reduce this problem to compute only the conditions derived from the maximal fulfilling configurations $Cond(\uparrow \mathbb{C}_{valid})$ . Therefore we only need to compute all elements in $\uparrow \mathbb{C}_{valid}$ instead of all elements in $\mathbb{C}_{valid}$. While this reduces the amount of work, it is still a quite complex task. However we can use any element $C \in \mathbb{C}_{valid}$ to approximate the relevant context conditions. While not all elements in $\mathbb{C}_{valid}$ may satisfy $Cond(C)$, still any context configuration that does guarantees noninterference of $P\|_m$. This allows us to use an approach that starts with very restrictive context conditions derived from a small subset of $\mathbb{C}_{valid}$ and subsequently refine these conditions with newly discovered members of $\mathbb{C}_{valid}$ until only the relevant context conditions remain. If needed —due to space or time constraints— we can cancel the computation at any point in time and are still left with conditions that, albeit being more restrictive, still can guarantee noninterference.

**Lemma 3.9** (Conservative Approximation of Alias Conditions)**.** *The alias conditions derived from $C_1$ are stronger than the conditions of $C_2$ if $C_1 \sqsubseteq C_2$.*

$$\forall C_1, C_2 \notin \mathbb{C} \text{ where } C_1 \sqsubseteq C_2 : Cond(\mathbb{C} \cup \{C_1\}) \implies Cond(\mathbb{C} \cup \{C_2\})$$

*So $Cond(\mathbb{C} \cup \{C_1\})$ can be used to conservatively approximate the conditions in $Cond(\mathbb{C} \cup \{C_2\})$ .*

*Proof.* Due to Lemma 3.6 $Cond(C_1) \implies Cond(C_2)$ and obviously $Cond(\mathbb{C}) \implies Cond(\mathbb{C})$. So $Cond(\mathbb{C}) \vee Cond(C_1) \implies Cond(\mathbb{C}) \vee Cond(C_2)$ is also true. □

**Theorem 3.4** (Alias Conditions Preserve the Monotonicity Property). *Given two alias conditions $Cond(C_1)$, $Cond(C_2)$ for a method part $P\|_m$ the following holds.*

$$\begin{aligned}&(Cond(C_1) \implies Cond(C_2)) \land Cond(C_2) \models NI(P\|_m)\\ \implies &Cond(C_1) \models NI(P\|_m)\end{aligned}$$

*As well as*

$$\begin{aligned}&(Cond(C_1) \implies Cond(C_2)) \land Cond(C_1) \not\models NI(P\|_m)\\ \implies &Cond(C_2) \not\models NI(P\|_m)\end{aligned}$$

*Proof.* Part (1): As $Cond(C_1) \implies Cond(C_2)$ we know

$$\forall C : C \models Cond(C_1) \implies C \models Cond(C_2)$$

and therefore $C \models NI(P\|_m)$.

Part(2): As $Cond(C_1) \not\models NI(P\|_m)$ we know that there exists a context $C$ with $C \models Cond(C_1)$ and $C \not\models NI(P\|_m)$. Because $Cond(C_1) \implies Cond(C_2)$ it holds that $C \models Cond(C_2)$ and therefore $Cond(C_2) \not\models NI(P\|_m)$. □

In the following we show our approach that uses Lemma 3.9 and Theorem 3.4 to approximate the relevant context conditions without the need to fully detect all elements of $\uparrow \mathbb{C}_{valid}$.

**Inference algorithm for relevant context conditions** Given a method part $P\|_m$, its root parameters *roots* and a classification of secret input and public output, the inference algorithm computes a set of alias conditions that can guarantee noninterference of $P\|_m$. The resulting alias conditions are in the form of Definition 3.11: Each alias condition is a conjunction of not-alias predicates on the input variables of $P\|_m$. We distinguish atomic from non-atomic not-alias predicates. An atomic predicate forbids exactly a single alias relation (e.g. $\neg alias(a.f, b.i)$) while a non-atomic predicate contains a wildcard '*' and in general can refer to multiple relations (e.g. $\neg alias(a.*, b.*)$).

**Algorithm 3.1** (Inference algorithm for relevant context conditions).

```
MAIN inferRelevantConditions($roots$)
IN:  Set of method root parameters $roots$
OUT: Set of alias conditions that guarantee noninterference $valid$
BEGIN
  Set<Predicate> $initialPreds$ = initialPredicates($roots$)
  Set<Condition> $valid$ = refine($\emptyset$, $initialPreds$, true)
  WHILE ($\exists c \in valid \wedge c$ contains non-atomic predicate) DO
    $valid$.remove(c)
    // modify $valid$ with refined and expanded conditions derived from $c$
    refineAndExpand($valid$, $c$, $initialPreds$)
  DONE
END

PROCEDURE refineAndExpand($valid$, $cond$, $initialPreds$)
IN:  Set of already valid condition $valid$
     Condition to be refined and expanded $cond$
     Set of initial predicates $initialPreds$
OUT: Set of valid conditions including the newly refined and expanded ones $valid$
BEGIN
  Predicate p = some non-atomic predicate $\in cond$        // 1. split
  Condition noP = $cond$ without predicate p
  Set<Predicate> splitP = split(p);
  Set<Condition> refineValid = refine($valid$, splitP, noP)   // 2. refine
  Set<Condition> expandValid = copy of refineValid            // 3. expand
  Set<Predicates> maxInitial = $\{p_{initial} \mid p_{initial} \in initialPreds \wedge p_{initial} \not\Longrightarrow p\}$
  Condition maxNoP = noP $\wedge \bigwedge_{p_{initial} \in \text{maxInitial}} p_{initial}$
  FOR (i: 1 to #elements in spiltP) DO
    checkAndAddIfRelevant(expandValid, i, splitP, maxNoP)
  DONE
  expandValid.removeAll(refineValid)
  FORALL (c $\in$ expandValid) DO
    refine($valid$, maxInitial, noP $\wedge$ c)
  DONE
END

PROCEDURE checkAndAddIfRelevant($valid$, $numPreds$, $preds$, $base$)
IN:  Set of valid conditions $valid$
     Number of predicates in the conditions to check $numPreds$
     Set of predicates to build the condition from $preds$
     Base condition thats added to the built conditions $base$
OUT: Set of valid conditions extended with newly found conditions $valid$
BEGIN
  FORALL $\{p_1, ..., p_{\text{numPreds}}\} \in \{P \mid P \in 2^{preds} \wedge |P| = numPreds\}$ DO
    Condition c = $\bigwedge_{i:1...numPred} p_i$
    IF ($\nexists$c'$\in valid$ with c $\Longrightarrow$ c') THEN
      IF (checkNoninterference(c $\wedge base$)) THEN
        $valid$.add(c)
      FI
    FI
  DONE
END
```

```
PROCEDURE refine($valid$, $preds$, $base$)
IN:  Set of valid conditions the result is added to $valid$
     Set of predicates to build the condition from $preds$
     Base condition thats added to the built conditions $base$
OUT: Adapted set of valid conditions $valid$
     Set of newly added conditions without base condition $refined$
BEGIN
  Set<Condition> $refined = \emptyset$
  FOR (i: 1 to #elements in $preds$) DO
    checkAndAddIfRelevant($refined$, i, $preds$, $base$)
  DONE
  FORALL ($c \in refined$) DO $valid$.add($c \wedge base$) DONE
  RETURN $refined$
END

PROCEDURE split($p$)
IN:  Non-atomic predicate $p$
OUT: Set of finer-grained predicates (derived from $p$) $split$
BEGIN
  Set<Predicate> $split = \emptyset$
  $p$ is of the form $\neg alias(r.f_1 \dots f_n.*, r'.f'_1 \dots f'_m.*)$
  FORALL (non-primitive fields $f_{n+1}$ reachable from $r.f_1 \dots f_n$) DO
    FORALL (non-primitive fields $f'_{m+1}$ reachable from $r'.f'_1 \dots f'_m$) DO
      addIfPotentialRelevant($split$, $r.f_1 \dots f_n.f_{n+1}$, $r'.f'_1 \dots f'_m.f'_{m+1}$)
    DONE
  DONE
  RETURN $split$
END

PROCEDURE addIfPotentialRelevant($preds$, $ref_1$, $ref_2$)
IN:  Set of predicates to extended $preds$
     Parameter references $ref_1$, $ref_2$
OUT: Set of predicates extended if references were relevant $preds$
BEGIN
    IF ($ref_1$ and $ref_2$ are potential aliasing) THEN
      IF ($ref_1$ contains only primitive types) THEN
        IF ($ref_1 \neq ref_2$) THEN $preds$.add("$\neg alias(ref_1, ref_2)$") FI
      ELSE
        $preds$.add("$\neg alias(ref_1.*, ref_2.*)$")
      FI
    FI
END

PROCEDURE initialPredicates($roots$)
IN:  Method root parameters $roots$
OUT: Set initial predicates $initialPreds$
BEGIN
  Set<Predicate> $initialPreds = \emptyset$
  FOREACH ($(r_1, r_2) \mid r_1, r_2 \in roots$) DO
    addIfPotentialRelevant($initialPreds$, $r_1$, $r_2$)
  DONE
  RETURN $initialPreds$
END
```

The inference algorithm in Algorithm 3.1 initially computes a set of coarse-grained predicates that forbid any aliasing. Then it detects for which minimal combinations of predicates noninterference can be guaranteed and stores them in the set of *valid* conditions. A fixed-point iteration refines and expands any conditions in *valid* that contain a non-atomic predicate until only conditions with atomic predicates remain. This fixed-point iteration can be aborted at any time as the result in *valid* contains a conservative approximation of all relevant alias conditions during each iteration.

Procedure `initialPredicates` computes the set of coarse-grained predicates from the method root parameters.

Procedure `refine` applies the monotonicity property of Theorem 3.4 to compute the minimal conditions from a set of given predicates *preds* and a base condition *base*. It uses procedure `checkAndAddIfRelevant` to compute which combination of predicates is relevant.

Procedure `checkAndAddIfRelevant` checks for any combination of *numPreds* predicates which of them form a condition that guarantees noninterference and currently cannot be inferred through elements in *valid*. It skips any call to `checkNoninterference` where the outcome of the computation can be inferred and thus only adds conditions to *valid* that increase the number of detected valid context configurations.

The main part of the inference is in procedure `refineAndExpand`. It converts a single coarse-grained condition *cond* into a set of finer-grained conditions in a 3-step process: *split*, *refine* and *expand*.

1. (*split*) A non-atomic predicate *p* in *cond* is split into a set of finer-grained predicates *splitP* through procedure `split`.
2. (*refine*) A refinement step computes which combinations of those finer-grained predicates form —together with the rest of condition *cond* excluding *p*— a valid condition and adds them to the set of *valid* conditions.
3. (*expand*) The last step checks if combinations of non-valid finer-grained predicates with previously skipped coarse-grained predicates can yield new valid conditions. The *expand* step itself is also a 3-step process.
   (a) Compute a maximal alias condition *maxNoP* that includes all predicates besides *p*.

(b) Check if the finer-grained parts of *p* in *splitP* can be combined with *maxNoP* into conditions that currently cannot be inferred by the conditions already found in the *refine* phase.

(c) Use a separate *refine* step for each newfound condition to detect which parts of *maxNoP* are necessary and add those to the set of *valid* conditions.

**Approximating relevant context conditions — an example** In this paragraph we show how Algorithm 3.1 infers the relevant conditions for the example in Figure 3.6. We consider the example component $P||_{bar}$ noninterferent if information about value of parameter `secret` is not leaked to a print statement. Thus the component may leak information if the secret value read at l. 2 can influence the print statements in l. 6 and l. 7. This is possible either through a direct flow from l. 2 to l. 7 —in case `a1.b1` and `a2.b1` are aliased— or through an indirect flow caused by the conditional if-statement l. 2→l. 3→l. 4→l. 6 —in case `a1.b1` and `a1.b2` as well as `a2.b2` and `a1.b1` are aliased. So the expected result of the analysis of this component is that $P||_{bar}$ is noninterferent as long as

$$\neg alias(a1.b1, a2.b1) \wedge (\neg alias(a1.b1, a2.b2) \vee \neg alias(a1.b1, a1.b2))$$

holds. Our inference algorithm can detect this condition as follows.

**Initial setup** In the first step we check if $P||_{bar}$ is noninterferent independent of its context configuration. Therefore we have to compute the maximal alias configuration $C_{\text{max}}$:

1. Identify all variable and field references that can be accessed by $P||_{bar}$.
2. Add an aliasing between each two references with compatible types to $C_{\text{max}}$.

As result of (1) we get the references $\{a1, a1.b1, a1.b2, a2, a2.b1, a2.b2\}$. For (2) we assume all references to type `A` may be aliased as well as all references to type `B`.

$$\begin{aligned} C_{\text{max}} = \{ & (a1, a2), (a1.b1, a1.b2), (a1.b1, a2.b1), (a1.b1, a2.b2), \\ & (a1.b2, a2.b1), (a1.b2, a2.b2), (a2.b1, a2.b1)\} \end{aligned}$$

```
1 static void bar(A a1, A a2, int secret) {  9 class A {
2   a1.b1.l = secret;                        10   B b1;
3   if (a1.b2.l > 0) {                       11   B b2;
4     a2.b2.k = 23;                          12 }
5   }                                        13 class B {
6   print(a1.b1.k);                          14   int l;
7   print(a2.b1.l);                          15   int k;
8 }                                          16 }
```

**Figure 3.6:** A small yet somewhat complex example that illustrates how we infer relevant context conditions for the noninterference of method `bar`.

Our analysis shows that $C_{\max} \not\models NI(P||_{bar})$. Due to the potential aliasing of $(a1.b1, a2.b1)$ l. 2 may directly influence the value read at l. 7.

In the next step we check if $P||_{bar}$ is noninterferent in any context, by analyzing it in the minimal context $C_{\min} = \{\}$. The analysis result shows that $C_{\min} \models NI(P||_{bar})$, because only $a1.b1.l$ holds the secret value and is never referenced directly in the rest of the component.

**Inference of relevant conditions** As

$$C_{\max} \not\models NI(P||_{bar}) \text{ and } C_{\min} \models NI(P||_{bar})$$

we start to infer the relevant alias conditions as described in Algorithm 3.1. Given the two root parameters $a1, a2$ method `initialPredicates` computes

$$initialPreds = \{\neg alias(a1.*, a1.*), \neg alias(a1.*, a2.*), \neg alias(a2.*, a2.*)\}$$

The resulting set contains 3 non-atomic not-alias predicates. Before entering the fixed-point iteration we compute the initial set of *valid* conditions through a call to `refine`. Method `refine` checks which minimal combinations of those predicates can guarantee noninterference. Table 3.1 shows the result of the necessary IFC checks. Each row represents a single alias configuration where all not-alias predicates with entry '1' are part of the conjunction for the alias condition of the configuration. For example the alias condition for row 5 —with the entries '1', '1', '0'— is $\neg alias(a1.*, a1.*) \wedge \neg alias(a1.*, a2.*)$. The last column shows the result of the noninterference analysis of $P||_{bar}$ for the given configuration: A '✓' marks that noninterference could be verified, while '✗' marks that

| $\neg(a1.*, a1.*)$ | $\neg(a1.*, a2.*)$ | $\neg(a2.*, a2.*)$ | $NI$ |
|---|---|---|---|
| 0 | 0 | 0 | ✗($C_{\max}$) |
| 1 | 0 | 0 | ✗ |
| 0 | **1** | 0 | ✓ |
| 0 | 0 | 1 | ✗ |
| 1 | 1 | 0 | (✓) |
| 1 | 0 | 1 | ✗ |
| 0 | 1 | 1 | (✓) |
| 1 | 1 | 1 | ✓($C_{\min}$) |

**Table 3.1:** Root-level IFC checks for all possible not-alias combinations. ✓- marks combinations that guarantee $NI(P||_{bar})$ , ✗- marks combinations that do not and (✓)- marks noninterference implied by the monotonicity property.

$NI(P||_{bar})$ could not be shown in the given configuration. Parentheses '(✓)' and '(✗)' represent noninterference properties that can be inferred —with the help of Theorem 3.4— through previous analysis results. For example noninterference of row 5 is automatically implied by the result in row 3 as $\neg alias(a1.*, a1.*) \wedge \neg alias(a1.*, a2.*) \implies \neg alias(a1.*, a1.*)$. Therefore we can check all 8 different combinations of not-alias conditions with 4 analysis runs —assuming the analysis for $C_{\min}$ and $C_{\max}$ has already previously been run— and the resulting approximated valid alias condition is $\neg alias(a1.*, a2.*) \models NI(P||_{bar})$. Thus

$$valid = \{\neg alias(a1.*, a2.*)\}$$

We now know that $P||_{bar}$ is guaranteed noninterferent as long as no field of `a1` is aliased to any field of `a2`. This condition is already usable and we could stop our interference analysis at this point. However if we invest more computation time we can further improve the result through the following fixed-point iteration.

The condition in *valid* contains a non-atomic predicate, so we remove it from the set and call `refineAndExpand`.

1. `split(`$\neg alias(a1.*, a2.*)$`)`:

$$\begin{aligned} splitP = \{&\neg alias(a1.b1, a2.b1), \neg alias(a1.b1, a2.b2),\\ &\neg alias(a1.b2, a2.b1), \neg alias(a1.b2, a2.b2)\} \end{aligned}$$

| $\neg(a1.b1, a2.b1)$ | $\neg(a1.b1, a2.b2)$ | $\neg(a1.b2, a2.b1)$ | $\neg(a1.b2, a2.b2)$ | *NI* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | (✗) |
| 1 | 0 | 0 | 0 | ✗ |
| 0 | 1 | 0 | 0 | ✗ |
| 0 | 0 | 1 | 0 | ✗ |
| 0 | 0 | 0 | 1 | ✗ |
| **1** | **1** | 0 | 0 | ✓ |
| 1 | 0 | 1 | 0 | ✗ |
| 1 | 0 | 0 | 1 | ✗ |
| 0 | 1 | 1 | 0 | ✗ |
| 0 | 1 | 0 | 1 | ✗ |
| 0 | 0 | 1 | 1 | ✗ |
| 1 | 1 | 1 | 0 | (✓) |
| 1 | 1 | 0 | 1 | (✓) |
| 1 | 0 | 1 | 1 | ✗ |
| 0 | 1 | 1 | 1 | ✗ |
| 1 | 1 | 1 | 1 | (✓) |

**Table 3.2:** Refinement of the $\neg alias(a1.*, a2.*)$ condition from Table 3.1. Checking alias variants of object-fields.

2. `refine`$(\{\}, splitP, true)$: We check which minimal combinations of the split predicate result in valid conditions. Table 3.2 shows the results of the various analysis runs. After 13 runs we know that $\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a2.b2)$ —and any condition implied by it— can guarantee noninterference. So only aliasing between those object fields is relevant. This is a quite significant enhancement of our previous result, as we now know that only 3 of 16 possible combinations can guarantee noninterference and we can discard any context configuration that is only valid in one of the other 13 combinations.

$$\begin{aligned} refineValid &= \{\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a2.b2)\} \\ valid &= \{\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a2.b2)(\wedge true)\} \end{aligned}$$

3. The expand step checks if it is still possible to guarantee noninterference when only the valid split predicates are left out of the condition. The result of step 2 in Table 3.2 shows that 13 combinations of the split predicate cannot guarantee noninterference by

| $\neg(a1.b1, a2.b1)$ | $\neg(a1.b1, a2.b2)$ | $\neg(a1.b2, a2.b1)$ | $\neg(a1.b2, a2.b2)$ | *NI* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | (✗) |
| **1** | 0 | 0 | 0 | ✓ |
| 0 | 1 | 0 | 0 | ✗ |
| 0 | 0 | 1 | 0 | ✗ |
| 0 | 0 | 0 | 1 | ✗ |
| 1 | 1 | 0 | 0 | (✓) |
| 1 | 0 | 1 | 0 | (✓) |
| 1 | 0 | 0 | 1 | (✓) |
| 0 | 1 | 1 | 0 | ✗ |
| 0 | 1 | 0 | 1 | ✗ |
| 0 | 0 | 1 | 1 | ✗ |
| 1 | 1 | 1 | 0 | (✓) |
| 1 | 1 | 0 | 1 | (✓) |
| 1 | 0 | 1 | 1 | (✓) |
| 0 | 1 | 1 | 1 | ✗ |
| 1 | 1 | 1 | 1 | (✓) |

**Table 3.3:** Expanding failed alias conditions from Table 3.2 to check if noninterference is possible in combination with $\neg alias(a1.*, a1.*) \wedge \neg alias(a2.*, a2.*)$.

themselves, however some of them may be sufficient when combined with other alias conditions. Therefore we run an IFC analysis for each insufficient condition in Table 3.2 in combination with the additional alias conditions $\neg alias(a1.*, a1.*) \wedge \neg alias(a2.*, a2.*)$.

$$\begin{aligned} maxInitial &= \{\neg alias(a1.*, a1.*), \neg alias(a2.*, a2.*)\} \\ maxNoP &= (true\wedge)\neg alias(a1.*, a1.*) \wedge \neg alias(a2.*, a2.*) \end{aligned}$$

The following loop tries any combination of the split predicates in conjunction with *maxNoP*. Table 3.3 shows the result of these analysis runs. We observe that condition $\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.*, a1.*) \wedge \neg alias(a2.*, a2.*)$ in row 2 is sufficient to guarantee noninterference and therefore also all combinations that include $\neg alias(a1.b1, a2.b1)$ —as in rows 6-8 and 12-14— are also sufficient due to Theorem 3.4. In total 8 additional analysis runs were needed to detect which of the 13 alias conditions are sufficient. As all other conditions can be implied by the one in row 2, we only get a single

| $\neg(a1.*, a1.*)$ | $\neg(a2.*, a2.*)$ | $NI$ |
|---|---|---|
| 0 | 0 | (✗) |
| **1** | 0 | ✓ |
| 0 | 1 | ✗ |
| 1 | 1 | (✓) |

**Table 3.4:** Refining the valid alias condition $\neg alias(a1.b1, a2.b1) \land \neg alias(a1.*, a1.*) \land \neg alias(a2.*, a2.*)$ from Table 3.3. Given $\neg alias(a1.b1, a2.b1)$ we test which combinations of $\neg alias(a1.*, a1.*) \land \neg alias(a2.*, a2.*)$ can guarantee $NI(P\|_{bar})$ .

additional condition as the result.

$$expandValid = \{\neg alias(a1.b1, a2.b1)\}$$

We now know that any element in *expandValid* in conjunction with all elements of *maxNoP* can guarantee noninterference, but it may be that only some elements of *maxNoP* are needed. In the last phase of the expand step we refine the newfound conditions by checking which parts of *maxNoP* are needed. Table 3.4 shows that $\neg alias(a1.b1, a2.b1) \land \neg alias(a1.*, a1.*)$ is sufficient and $\neg alias(a2.*, a2.*)$ is not relevant for a valid alias condition. So the call to refine adds

$$valid := valid \cup \{\neg alias(a1.b1, a2.b1) \land \neg alias(a1.*, a1.*)(\land true)\}$$

to the set of valid conditions.

After the first iteration of the *while*-loop in `inferRelevantConditions` the set of valid conditions contains two entries:

$$\begin{aligned} valid = \{&\neg alias(a1.b1, a2.b1) \land \neg alias(a1.b1, a2.b2), \\ &\neg alias(a1.b1, a2.b1) \land \neg alias(a1.*, a1.*)\} \end{aligned}$$

Any of these conditions can guarantee $NI(P\|_{bar})$ so the disjunction of all conditions in *valid* is also a valid condition.

$$\neg alias(a1.b1, a2.b1) \land (\neg alias(a1.b1, a2.b2) \lor \neg alias(a1.*, a1.*))$$

We could abort the fixed-point iteration at this point and use above condition as a conservative approximation of the relevant conditions, but as *valid* still contains a condition with a non-atomic predicate $\neg alias(a1.*, a1.*)$, a second iteration of the *while*-loop is possible. In the second iteration `refineAndExpand` is called with the arguments:

$$\begin{aligned} valid =& \{\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a2.b2)\} \\ c =& \neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.*, a1.*) \\ initialPreds =& \{\neg alias(a1.*, a1.*), \neg alias(a1.*, a2.*), \neg alias(a2.*, a2.*)\} \end{aligned}$$

The subsequent call to `splitP` splits the predicate only into a single non-atomic predicate, as no other aliasing is possible.

$$\texttt{split}(\neg alias(a1.*, a1.*)) = \{\neg alias(a1.b1, a1.b2)\}$$

The following *refine* and *expand* steps therefore only need a single additional IFC-check to assure that

$$\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a1.b2)$$

is also a valid condition. After the second and final iteration of the *while*-loop *valid* only contains conditions with atomic predicates and the inference algorithm terminates with

$$\begin{aligned} valid =& \{\neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a2.b2), \\ & \neg alias(a1.b1, a2.b1) \wedge \neg alias(a1.b1, a1.b2)\} \end{aligned}$$

as the final result. Which translates to the expected condition:

$$\neg alias(a1.b1, a2.b1) \wedge (\neg alias(a1.b1, a2.b2) \vee \neg alias(a1.b1, a1.b2))$$

**Conclusion** We have shown how to apply the monotonicity property to infer relevant alias conditions for a method part. We can use these conditions to check for any given context configuration if the method part is noninterferent when executed in this context. However in order to analyze a method part for noninterference we have to know which statements or variables are considered sources and sinks. In the following section we introduce an annotation language called *FlowLess* that allows us to specify forbidden information flow for a method part at the head of the method declaration. This way the programmer is able to specify expected information flow properties for arbitrary methods. Our tool then automatically infers the relevant alias conditions that guarantee that the specified properties are satisfied.

```
1  interface Lib {
2    //@ifc: !{a, b} => x -!-> \result
3    int call(A a, A b, A c, int x);
4  }
5
6  class Library implements Lib {
7    int call(A a, A b, A c, int x) {
8      c.i = a.i;
9      a.i = x;
10     return b.i;
11   }
12 }
13 class A { int i; }
```

```
14 void prepareContext(Random r) {
15   A a = new A(); A b = new A();
16   A c = new A(); int x;
17   A ac = new A(); A bc = new A();
18   if (r.nextInt(2) == 0) a = ac;
19   if (r.nextInt(2) == 0) c = ac;
20   if (r.nextInt(2) == 0) b = bc;
21   if (r.nextInt(2) == 0) c = bc;
22   (new Library()).call(a, b, c, x);
23 }
```

**Figure 3.7:** A component with alias and information flow specification and the generated context stub calling the component in the context $C_{\neg alias(a,b)}$

## 3.3 FlowLess: A language for information flow annotations

In this section we introduce *FlowLess* — a specification language for information flow properties of program components. FlowLess can be used to annotate method declarations with restrictions on the information flow that have to be met during method execution. These restrictions can be paired together with requirements on the context in which the method is executed. For example: "In any context where parameter `a` and `b` are not aliasing, the method execution does not result in information flow between parameter `x` and the return value.". The matching FlowLess specification for this IFC-condition is "$\neg alias(a,b) \implies x \not\rightarrow \backslash result$". Figure 3.7 shows an example component annotated with this condition using a JML-like syntax (see l. 2).

Given an IFC-condition for a method we compute if it can be guaranteed that the condition is valid. Due to the monotonicity property we are able to check the condition without knowledge of the concrete calling context. In the given example we can verify that method `call` of class `Library` never leaks information about the value of parameter `x` to the return value as long as parameters `a` and `b` are not aliased. Therefore we infer the maximal alias configurations allowed by the alias conditions on the methods parameters and generate code in form of *context stubs* that call the method in said configuration. Method `prepareContext` on the right side of Figure 3.7 shows a generated code that prepares the

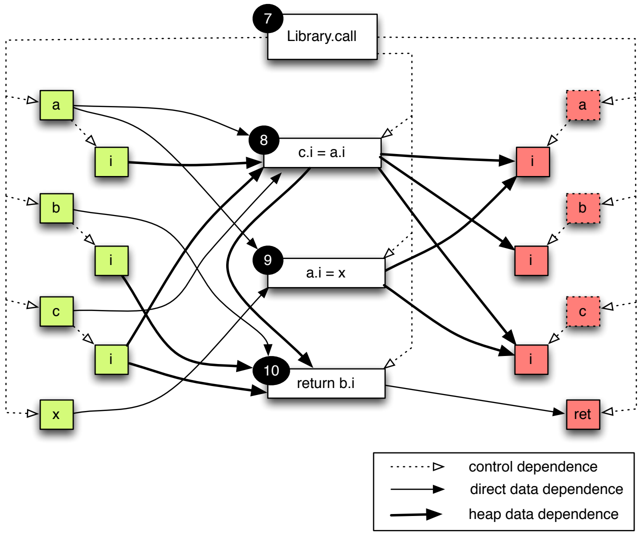


**Figure 3.8:** The PDG of `Library.call` in the context $C_{\neg alias(a,b)}$.

maximal allowed alias configuration for `Library.call`. We analyze the information flow that occurs inside of method `call` when called from `prepareContext` and can verify the absence of flow between `x` and the return value. Figure 3.8 shows the resulting PDG for the method part $P\|_{call}$ in the context of $C_{\neg alias(a,b)} = \{(b,c),(a,c)\}$. The PDG contains no path from input parameter $x$ to output parameter $ret$, therefore the IFC-condition is valid. Note that if `a` and `b` were aliased, the modification of `a.i` in l. 9 would also affect field `b.i` read in the following line. Thus $\neg alias(a,b)$ is a relevant context condition, as in every context that allows $alias(a,b)$ an illegal flow occurs.

With FlowLess it is also possible to infer the relevant context conditions for a given flow restriction with the algorithm presented in §3.2.3. Changing the IFC-condition in the example to "$? \implies x \nrightarrow \backslash result$" triggers the inference algorithm and our analysis automatically infers if and under which context conditions `x` does not influence the return value. The result of the inference is as expected $\neg alias(a,b)$.

In the following subsections we present an introduction to the syntax of FlowLess in §3.3.1, before we explain the generation of context stubs in §3.3.2. §3.3.3 concludes with an example that shows step by step how a context stub is build from a given alias condition.

## 3.3.1 Overview and syntax

The *FlowLess* specification language enables the software engineer to specify required information flow properties at the public methods of a component. He may annotate each method with a set of so-called *IFC-conditions*. Each IFC-condition contains a description of the aliasing contexts it applies to and restrictions to the information flow inside the component that need to be guaranteed in these contexts.

**Definition 3.12** (IFC-Condition). *An information flow control condition (IFC-condition)*

$$c \underset{m}{\Longrightarrow} f$$

*of a method m is comprised of an alias-condition c describing the context configurations to which it applies and a flow-restriction f that poses required restrictions on the information flow inside the component $P\|_m$.*

The IFC-condition $c \underset{m}{\Longrightarrow} f$ is satisfied iff for any context configuration $C$ that meets the alias condition $c$, the method $m$ fulfills the flow-restriction $f$ when called in context $C$ .

$$\forall C : C \models c \implies C[P\|_m] \text{ fulfills } f$$

An alias condition $c$ —as defined in Definition 3.11— is a disjunction of conjunctions of not-alias predicates on input parameters[18] $\text{IN}(m)$ of method $m$.

$$c := \bigvee_i \bigwedge_j \neg alias(p_{ij}, p'_{ij}) \text{ with } p_{ij}, p'_{ij} \in \text{IN}(m)$$

We allow not-alias predicate with > 2 parameters for convenience. They translate to atomic predicates as follows.

$$\neg alias(p_1, p_2, \ldots, p_i) := \bigwedge_{1 \geq k, l \geq i, k \neq l} \neg alias(p_k, p_l)$$

[18]Note that input parameters are normal method parameters as well as static and object fields read by the method.

We also allow wildcards '*' to forbid aliasing of all reaching fields.

$$\neg alias(p.*, p'.*) := \bigwedge_{p.f_1...f_k, p'.f'_1...f'_l \in \text{IN}(m)} \neg alias(p.f_1 \dots f_k, p'.f'_1 \dots f'_l)$$

**Definition 3.13** (Flow Restriction). *A flow restriction $f$ for a component $P\|_m$ restricts the allowed information flow between the input parameters $IN(m)$ and the output parameters $OUT(m)$ of $P\|_m$. We call $f := p^{in} \nrightarrow p^{out}$ an* atomic flow-restriction *when it restricts direct and indirect flow from a single input to a single output parameter. An atomic flow-restriction is satisfied if no information flow between $p^{in}$ and $p^{out}$ is possible. In general a flow-restriction is a conjunction of atomic flow restrictions.*

$$f := \bigwedge_{i,j} p_i^{in} \nrightarrow p_j^{out} \textit{ with } p_i^{in} \in IN(m), p_j^{out} \in OUT(m)$$

We say $f$ is fulfilled in context $C$

$$C \models f$$

if $P\|_m$ does not contain any information flow from $p_{in}$ to $p_{out}$ when called in context $C$. We say $f$ is fulfilled in general

$$\models f$$

if it is fulfilled in any context.

Similar to the not-alias predicate we allow flow-restrictions on multiple input and output parameters at once.

$$(p_1^{in}, p_1^{in}, \dots, p_i^{in}) \nrightarrow (p_1^{out}, p_1^{out}, \dots, p_j^{out}) := \bigwedge_{1 \geq k \geq i, 1 \geq l \geq j} p_k^{in} \nrightarrow p_l^{out}$$

Our tool supports IFC-conditions in form of a JML style syntax inside the comment block for method declarations. Figure 3.9 contains the grammar of a single IFC-condition in extended backus-naur form (EBNF). An example of an IFC-condition at a method declaration has already been shown in Figure 3.7. For convenience we allow several shortcuts:

- Alias conditions (rule *alias-cond*) do not need to be declared in disjunctive normal form. They are automatically transformed to DNF in a precomputation step.

```
        ifc-cond   →   alias-cond => flow-rest
                       | ? => flow-rest
      alias-cond   →   alias-and [ | alias-cond ]
                       | ( alias-cond )
       alias-and   →   alias-pred [ & alias-pred ]
      alias-pred   →   !{ param , param-list }
      param-list   →   param [ , param-list ]
           param   →   <ident> [ . access-path ]
     access-path   →   field [ . access-path ]
           field   →   <ident> | [] | *
       flow-rest   →   in-params -!-> out-params [ , flow-rest ]
       in-params   →   param | ( param-list )
      out-params   →   out-param | ( out-param-list )
       out-param   →   out-root [ . access-path ]
        out-root   →   <ident> | \result | \exception
  out-param-list   →   out-param [ , out-param-list ]
```

**Figure 3.9:** EBNF-Grammar of *FlowLess* – the description language for conditional information flow requirements.

- Not-alias predicates (rule *alias-pred*) can be declared with a list of parameters. Single parameter declarations can use a wildcard * that expands to a list of reachable parameters.

- Flow-restrictions (rule *flow-rest*) can be declared declared in form of a comma separated list. The conjunction of atomic flow-restrictions is built automatically from each list entry. For example

  ```
  a.f -!-> b.i, b -!-> c.f2
  ```

  translates to

  $$a.f \nrightarrow b.i \land b \nrightarrow c.f2$$

- Single entries of the flow-restriction list can refer to multiple in- and out-parameters. When multiple parameters are referenced flow from any in- to any out-parameter is forbidden. For example

  ```
  (a.f, b, c.i) -!-> (a.k, b.i)
  ```

translates to the following atomic flow-restrictions

$$a.f \nrightarrow a.k \wedge a.f \nrightarrow b.i \wedge b \nrightarrow a.k \wedge b \nrightarrow b.i \wedge c.i \nrightarrow a.k \wedge c.i \nrightarrow b.i$$

- Special root out-parameter names exist for the return value (*\result*) and exception object (*\exception*). For example the flow-restriction in Figure 3.7 `x -!-> \result` translates to $x \nrightarrow p_{exit}$.

Joana parses the FlowLess annotation of a method and transforms it into a so-called *basic* IFC-conditions that use only alias conditions in disjunctive normal form with atomic predicates and flow-restrictions that only contain a conjunction of atomic restrictions. Given these basic IFC-conditions Joana then computes if the conditions may hold for the given method. This is a two step process, where in the first step we compute the maximal context configurations allowed by the IFC-condition and in the second step we check if the component satisfies the flow-restriction in those configurations. The following subsection §3.3.2 explains how to generate *context stubs* that call the component in the maximal context configuration allowed. These stubs can then be used to analyze the information flow inside the component through a standard SDG-based information flow analysis. The structure of the components SDG obviously depends on the context configuration. However independent of the context, all SDGs of the component share a common structure that can be exploited to compute a modular context independent representation. Section §3.4 introduces the *modular* variant of the SDG that allows an efficient computation of the actual context dependent SDG. With a precomputed modular SDG the full recomputation for each context is no longer needed. So checking flow-restrictions in different context configurations becomes more efficient.

### 3.3.2 Building context stubs from annotation

One way to verify if a given IFC-condition ($c \underset{m}{\Longrightarrow} f$) holds is to analyze the corresponding component $P\|_m$ in the maximal context that still satisfies alias-condition $c$. Therefore we extract and normalize the alias-condition of the IFC-condition and use it to build maximal points-to sets that satisfy $c$. Then we generate a *context stub* method that creates a context according to the points-to sets and calls the component $P\|_m$.

Finally we apply our standard whole-program IFC analysis on the generated context stub to see if $P\|_m$ satisfies flow-condition $f$ in all contexts allowed by $c$.

### Normalizing alias conditions

The alias-conditions in a FlowLess annotation may contain arbitrarily nested conjunctions and disjunctions of atomic and non-atomic alias predicates. This improves convenience for the user, but at the same time complicates an automated analysis. Therefore we transform the alias-condition of an IFC-condition into the *disjunctive normal form (DNF)* of atomic alias predicates. We remove superfluous parts from the condition and then analyze each part of the remaining disjunction separately. Every part of the disjunction only contains a conjunction of atomic alias predicates. For every conjunction there exists a single maximal context configuration that satisfies all predicates and therefore a single analysis run suffices to check if the flow-restrictions can be guaranteed in the contexts allowed by the conjunction. Therefore we create a separate context stub for each conjunction and check if the flow-restrictions hold in any of them. Note that we can use a lazy approach because of the disjunction of conjunctions: We can stop the analysis as soon as we verify a single conjunction.

Given an IFC-condition $c \underset{m}{\Longrightarrow} f$ we apply the following steps:

1. Expand non-atomic alias predicates in $c$.

$$\neg alias(p_1, \ldots, p_n) \rightarrow \bigwedge_{1 \leq i,j \leq n, i \neq j} \neg alias(p_i, p_j)$$

2. Transform to disjunctive normal form.

$$c \rightarrow DNF(c) \equiv \bigvee_i c_i \equiv \bigvee_i (\bigwedge_j c_{i,j}) \equiv \bigvee_i (\bigwedge_j \neg alias(p_{i,j}, p'_{i,j}))$$

3. Remove superfluous conjunctions of predicates.

$$\mathfrak{C}(c) = \{c_j \mid c_j \in \bigvee_i c_i \wedge \nexists c_k \in \bigvee_i c_i : c_k \neq c_j \wedge c_k \implies c_j\}$$

4. Build disjunction of *basic IFC-conditions*. If any of those conditions can be met, the originating condition $c \underset{m}{\Longrightarrow} f$ is also met.

$$\bigvee_{c' \in \mathfrak{C}(c)} (c' \underset{m}{\Longrightarrow} f)$$

Consider for example the following IFC-condition in FlowLess syntax:

```
(!{x, y} | !{a, b, c, d}) & (!{a, b} | !{c, d}) => c -!-> d
```

We parse the alias-condition part and get:

$$(\neg alias(x,y) \vee \neg alias(a,b,c,d)) \wedge (\neg alias(a,b) \vee \neg alias(c,d))$$

Then we transform to disjunctive normal form and remove superfluous conditions.

$$\begin{array}{ll} & \neg alias(x,y) \wedge \neg alias(a,b) \\ \vee & \neg alias(x,y) \wedge \neg alias(c,d) \\ \vee & \neg alias(a,b,c,d) \wedge \neg alias(a,b) \\ \vee & \neg alias(a,b,c,d) \wedge \neg alias(c,d) \\ \Longrightarrow & \\ & \neg alias(x,y) \wedge \neg alias(a,b) \\ \vee & \neg alias(x,y) \wedge \neg alias(c,d) \\ \vee & \neg alias(a,b) \wedge \neg alias(a,c) \wedge \neg alias(a,d) \wedge \neg alias(b,c) \\ & \wedge \neg alias(b,d) \wedge \neg alias(c,d) \end{array}$$

Now we are left with three different conjunctions of atomic alias-predicates that we need to check separately. Therefore we are going to build initial points-to sets for all parameters of $P||_m$ that satisfy these conditions.

**Building initial points-to sets**

We start with the maximal alias configuration $C_{max}$ of the component. This configuration allows aliasing between all parameters of the components interface only restricted by type information. For example an

**int** array can never point to the same location as a **char**. Objects may only alias if one is a subtype of the other. Given a conjunction of atomic alias-predicates ($c = \bigwedge_i \neg alias(a_i, b_i)$) we use them to remove the specified non-aliasing parameters from the configuration.

$$\begin{aligned} C_{\neg alias(a,b)} &:= C_{max} \setminus \{(a,b)\} \\ C_c = C_{\bigwedge_i \neg alias(a_i,b_i)} &:= \bigcap_i C_{\neg alias(a_i,b_i)} \end{aligned}$$

The configuration $C_c$ contains a pair of every parameters potentially aliased. By the definition of aliasing we know that the respective points-to sets of these parameters must share at least a single common element.

$$(a,b) \in C_c \implies \exists l \in pts_c(a) : l \in pts_c(b)$$

We use this observation to create point-to sets from the given alias configuration. A straight forward approach would be to create a new points-to element for each pair in the configuration.

$$\forall a \in Interface(m) : pts_c(a) := \{l_{a,b} \mid \exists (a,b) \in C_c\}$$

However the number of potential aliasing parameter pairs can become large and thus the number of individual points-to elements too. We can do better by detecting *complete groups of aliasing parameters*.

$$\begin{aligned} \overline{(a,b)_c} &:= \{(a,b)\} \cup \{(x,y) \mid (x,y) \in C_c \\ &\qquad \wedge \forall (a',b') \in \overline{(a,b)_c} : (a',x),(a',y),(b',x),(b',y) \in C_c\} \end{aligned}$$

Every parameter in a complete group of aliasing parameters is aliased to any other parameter in this group.

$$(a',\_),(b',\_) \in \overline{(a,b)_c} \implies (a',b') \in C_c$$

Therefore it suffices to create only a single points-to element for each group. We define $\overline{C_c}$ as the set of all complete groups under the alias-condition $c$. Then we define the points-to sets with respect to these groups.

$$\forall a \in Interface(m) : pts_c(a) := \{l_{\overline{a}} \mid \exists\, \overline{a} = \overline{(a',x')_c} \in \overline{C_c} \;\wedge\; (a,\_) \in \overline{(a',x')_c}\}$$

This definition reduces the amount of points-to elements significantly compared to the naive approach. It shows even in a small example where three parameters $a, b, c$ may be aliased with each other. The minimal alias configuration for this example is $\{(a, b), (a, c), (b, c)\}$ and thus the naive approach creates three different points-to elements $\{l_{(a,b)}, l_{(a,c)}, l_{(b,c)}\}$ with $pts(a) = \{l_{(a,b)}, l_{(a,c)}\}$, $pts(b) = \{l_{(a,b)}, l_{(b,c)}\}$ and $pts(c) = \{l_{(a,c)}, l_{(b,c)}\}$. The refined approach detects that $a, b, c$ form a complete group of alias parameters: $\overline{a} = \overline{(a, b)} = \overline{(a, c)} = \overline{(b, c)}$. Therefore we only need to create a single points-to element $l_{\overline{a}}$ with $pts(a) = pts(b) = pts(c) = \{l_{\overline{a}}\}$.

We integrated initial points-to sets through complete aliasing groups into our Joana tool. To compute complete groups efficiently we build a structure called *alias graph* and search for *maximal complete subgraphs* — also called *maximal cliques* — in this graph.

**Definition 3.14** (Alias Graph). *An* alias graph *is an undirected graph $G = (N, E)$ that represents the aliasing configuration of a component $P\|_m$. The nodes of the graph correspond to input parameters of the component $N := Interface(m)$. Two nodes $n_1, n_2 \in N$ are connected if they are may-aliased: $(n_1, n_2) \in E \equiv alias(n_1, n_2)$.*

A maximal clique in the alias graph corresponds to a complete alias group. Therefore we can leverage well known approaches to compute maximal cliques in an undirected graph. For arbitrary graphs this problem is hard to compute [107] with an approximated runtime of $O(2^{\frac{n}{3}}) \approx O(1.3^n)$. We found that the alias graphs in our evaluation computed in reasonable time. However it is still possible to fall back to the naive approach and trade memory — the number of points-to elements — in favor of runtime.

While maximal cliques provide the best result in terms of memory usage, it is also possible to mix it with the naive approach and only identify some cliques. It does not guarantee a minimal number of points-to elements but proved to be the best option in practice. We suggest a greedy algorithm that starts at a random node with a high degree and searches a maximal clique for it. Then we remove all edges between nodes in the clique — which may remove the possibility to detect other maximal cliques — and the process starts again with a different node. Algorithm 3.2 contains the pseudocode of the greedy algorithm that runs in $O(n^3)$. The naive approach needs to touch every node and edge once

**Algorithm 3.2** (Compute points-to sets from alias graphs in $O(n^3)$).
*No optimal solution, but a good trade-off.*

```
MAIN AliasToPts($G$, $minDeg$)
IN:  An alias graph $G = (N, E)$ for a component and
     a threshold $minDeg$ for the minimal degree of the nodes we search cliques for.
OUT: A mapping $pts : N \to Loc_m^2$ for each node of $G$ to a set of points-to elements $Loc_m$.
BEGIN
  Initialize $pts$ with empty sets for each node.
  Initialize $workset := \{n \mid n \in N \wedge degree(n) \geq minDeg\}$
  WHILE ($workset \neq \emptyset$) DO
    Remove random element $n$ from $workset$
    $clique := \{n\}$
    Add $n$ to empty $worklist$

    WHILE ($worklist \neq \emptyset$) DO
      Remove first element $e$ from $worklist$
      FORALL (neighbors $n'$ of $e$ in $G$ with $degree(n') \geq max(minDeg, \|clique\|)$) DO
        IF ($n' \notin clique \wedge n'$ is neighbor to all nodes in $clique$) THEN
          Add $n'$ to $clique$
          Add $n'$ to $worklist$
        FI
      DONE
    DONE

    Create a new points-to element $l$
    FORALL ($n' \in clique$) DO
      $pts(n') = pts(n') \cup \{l\}$
    DONE
    Remove all edges between nodes of $clique$ in $G$
    Remove all nodes $n'' \in clique$ with $degree(n'') < minDeg$ from $workset$
  DONE

  FORALL (remaining edges $(n_1, n_2) = e \in E$) DO
    Create a new points-to element $l$
    $pts(n_1) = pts(n_1) \cup \{l\}$
    $pts(n_2) = pts(n_2) \cup \{l\}$
  DONE
END
```

and therefore runs in $O(n + e) \approx O(n^2)$. Note that it is possible to abort the greedy algorithm during any iteration of the *workset* and finish the remaining graph with the naive approach, in case the computation still takes too long. However we have never found the need to do so.

The result of our points-to set computation is a mapping from each variable and field to an initial points-to set. These initial points-to sets are the largest points-to configuration that still can satisfy the alias conditions of the component. Thus if we use this configuration to analyze the component for internal information flow, we can be sure that when no illegal flow is detected, there will be no illegal flow for any other points-to configuration that satisfy the alias conditions.

In order to run an IFC analysis of the given component with the precomputed initial points-to sets it is possible to insert the points-to sets as initial states into the points-to computation of our whole program analysis. However our IFC framework —as well as other program analysis frameworks— supports multiple flavors of points-to analyses. We would have to integrate initial states in each, as the integration is very much dependent on the nature of the points-to analysis. Therefore we decided to decouple initial points-to sets from the concrete points-to analysis and generate stubs that call the component in the desired context instead. This way we can run our analysis with an unaltered points-to analysis directly on the stub and the called component is automatically analyzed in the right context.

**Generating stubs from given points-to sets**

This section describes how we generate a stub that calls a given component in an alias configuration specified by initial points-to sets for each parameter. The presented approach in general only works under the previously stated restrictions that the component contains no callbacks and the dynamic types of all method parameters are equal to their static type. However in our implementation we can treat dynamic types, because the WALA framework we use to generate the instructions of the stub supports a special flavor of new-instructions that does not only create an object of a specific type, but potentially creates a new-instance of any subtype of the specified type.

Algorithm 3.3 contains the pseudocode of the context-stub creation for a given context configuration. The input is a mapping $pts : Interface(m) \rightarrow Loc_m^2$ from parameter nodes $n \in Interface(m)$ of component $m$ to a set of points-to elements $l \in Loc_m$. At first we generate new instance instructions for each element of the points-to set $Loc_m$ and assign a separate local variable $v_l$ to each instance. Then we start at the root parameter nodes $r \in Root(m)$, create a variable $v_r$ for each root node and add a statically undecidable conditional assignment to $v_r$ for each element of the points-to set of $r$. Then we traverse to the parameter fields

**Algorithm 3.3** (Build a context-stub for a given points-to configuration).

```
MAIN PtsToStub($pts$)
IN:  A mapping $pts : N \to Loc_m^2$ for each node of that alias graph $G$ of component $m$
     to a set of points-to elements $Loc_m$.
OUT: A method stub $s$ that calls component $m$ with the points-to configuration
     described by the input.
BEGIN
  Initialize empty map $pts2var : Loc_m \to V$
  Initialize empty set of alias nodes $visited = \{\}$

  FORALL ($l \in Loc_m$) DO
    Create new local variable $v_l$
    // Add new-instance instruction $i$ for pts-element $l$ to stub $s$:
    Emit('$v_l$ = new $type(l)$')
    $pts2var(l)$ := $v_l$
  DONE

  FORALL ($r \in Root(m)$) DO
    Create new local variable $v_r$
    FORALL ($l \in pts(r)$) DO
      $v_l$ := $pts2var(l)$
      // Add conditional assignment to stub $s$:
      Emit('if (System.timeInMillis() % 23 == 0) $v_r$ = $v_l$')
    DONE
    AssignFields(r)
  DONE

  // Finally add call to component with root node variables $v_r$ as params
  Emit('call m($v_{r_1}, \dots, v_{r_n}$)')
END

PROCEDURE AssignFields($n \in Interface(m)$)
BEGIN
  Add $n$ to $visited$
  FOREACH (non-primitive field $f$ of $n$) DO
    FOREACH ($c \in Interface$ with $c$ corresponds to field $f$) DO
      FORALL ($l \in pts(c)$) DO
        $v_l$ := $pts2var(l)$
        // Add conditional assignment to stub $s$:
        Emit('if (System.timeInMillis() % 23 == 0) $n.f$ = $v_l$')
      DONE
      IF ($c \notin visited$) THEN
        AssignFields(c)
      FI
    DONE
  DONE
END

PROCEDURE Emit(Instruction i)
BEGIN
  Add $i$ at the end of stub $s$
END
```

```
1  class A {
2    B b1;
3    B b2
4  }

5  class B {
6    int i;
7  }

8  class Library {
9    //@ifc: !{a1, a2} && !{a1.b1, a1.b2, a2.b2}
10   static void component(A a1, A a2, B b) { ... }
11 }
```

**Figure 3.10:** An example program with an alias condition (l. 9) for method `component`.

reachable from those root nodes and also create variables and conditional assignments for them. Finally we add a call to the component that takes the matching variables of the root nodes as argument.

We implemented this approach in our Joana framework and use it to compute information flow inside a component for a given points-to configuration. However this approach is in general independent from our IFC and dependency graph analysis and can be used to set up stubs for other analysis purposes as well. A limitation of this approach is that the resulting stub is not executable in a virtual machine. The instructions in the stub are tailored to support arbitrary points-to analyses and contain as few clutter as possible —we omit additional methods calls. The generated field accesses can violate access restrictions and the new-Instance instructions miss the subsequent call to the constructor of the type. This leaves the code non-executable but —to our experience— it yields the best results for points-to analyses in terms of analysis runtime and precision.

### 3.3.3 Example

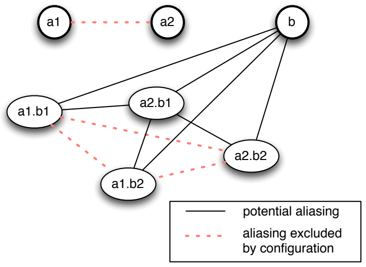
This example illustrates how we create a stub calling a component with a given alias condition. The relevant code for the component is shown in Figure 3.10 and contains an alias condition in l. 9.

We start by parsing the alias condition and convert it into disjunctive normal form:

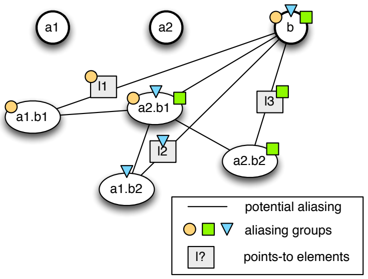
$$\neg alias(a1, a2) \wedge \neg alias(a1.b1, a1.b2, a2.b2)$$

$$\Longrightarrow$$

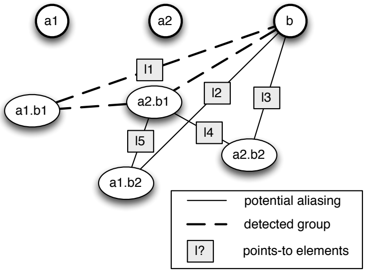
$$\neg alias(a1, a2) \wedge$$
$$\neg alias(a1.b1, a1.b2) \wedge \neg alias(a1.b1, a2.b2) \wedge \neg alias(a1.b2, a2.b2)$$



**(a)** All potential aliasing relations. Dotted edges show aliasing forbidden by the alias conditions.



**(b)** Optimal approach



**(c)** Greedy approach

**Figure 3.11:** Alias graphs for method `component` of the example in Figure 3.10. Nodes with bold outline are root parameters, others are field parameters.

Then we compute the maximal aliasing possible for the input parameters of method `component` in form of an alias graph. We use type information and the additional restrictions to remove impossible aliasing. Figure 3.11a shows the resulting alias graph. The alias relations that would be possible due to type information but are forbidden by the restrictions are drawn as dotted edges.

We use the alias graph to compute the necessary points-to elements. Therefore we can use either the slow but optimal approach (see Figure 3.11b) to detect all complete groups in the graph, or the faster

$$\begin{aligned}
pts(a1) &= \{l_{a1}\} \\
pts(a2) &= \{l_{a2}\} \\
pts(b) &= \{l_b, l_1, l_2, l_3\} \\
pts(a1.b1) &= \{l_{a1.b1}, l_1\} \\
pts(a1.b2) &= \{l_{a1.b2}, l_2\} \\
pts(a2.b1) &= \{l_{a2.b1}, l_1, l_2, l_3\} \\
pts(a2.b2) &= \{l_{a2.b2}, l_3\}
\end{aligned}$$

**(a)** Optimal approach

$$\begin{aligned}
pts(a1) &= \{l_{a1}\} \\
pts(a2) &= \{l_{a2}\} \\
pts(b) &= \{l_b, l_1, l_2, l_3\} \\
pts(a1.b1) &= \{l_{a1.b1}, l_1\} \\
pts(a1.b2) &= \{l_{a1.b2}, l_2, l_5\} \\
pts(a2.b1) &= \{l_{a2.b1}, l_1, l_4, l_5\} \\
pts(a2.b2) &= \{l_{a2.b2}, l_3, l_4\}
\end{aligned}$$

**(b)** Greedy approach

**Figure 3.12:** Resulting points-to sets for the optimal and the greedy approach.

non-optimal greedy approach (see Figure 3.11c). Figure 3.12 shows the resulting points-to sets for both approaches. Besides a points-to element for each detected group and edges in the graph, those sets also contain an individual points-to element for each separate parameter and field. This helps to guarantee that every parameter has at least a single element in its points-to set and that no two parameters share the exact same points-to set —making the may-aliasing more explicit.

In the final step we use the computed points-to sets from Figure 3.12a to build a stub that calls method `component` in the defined alias configuration. Figure 3.13 shows the resulting stub. It contains instructions to create instances for each individual parameter, as well as for each points-to element induced by aliasing constraints. Then the values of all fields are set according to their corresponding points-to set and finally the component is called.

## 3.4 Modular SDG

In the previous sections we argued that some whole-program analyses —like our IFC analysis— can also be used to analyze components in unknown context, if they fulfill the requirements for the monotonicity property. This property allows us to draw conclusions about the outcome

```
public static void stub() {
  // individual elements
  A v1  = new A(); // l_a1
  A v2  = new A(); // l_a2
  B v3  = new B(); // l_b
  B v4  = new B(); // l_a1.b1
  B v5  = new B(); // l_a1.b2
  B v6  = new B(); // l_a2.b1
  B v7  = new B(); // l_a2.b2
  // aliasing induced elements
  B v8  = new B(); // l_1
  B v9  = new B(); // l_2
  B v10 = new B(); // l_3
  // set b
  if (System.timeInMillis() % 23 == 0) v3 = v8;
  if (System.timeInMillis() % 23 == 0) v3 = v9;
  if (System.timeInMillis() % 23 == 0) v3 = v10;
  // set a1.b1
  v1.b1 = v4;
  if (System.timeInMillis() % 23 == 0) v1.b1 = v8;
  // set a1.b2
  v1.b2 = v5;
  if (System.timeInMillis() % 23 == 0) v1.b2 = v9;
  // set a2.b1
  v2.b1 = v6;
  if (System.timeInMillis() % 23 == 0) v2.b1 = v8;
  if (System.timeInMillis() % 23 == 0) v2.b1 = v9;
  if (System.timeInMillis() % 23 == 0) v2.b1 = v10;
  // set a2.b2
  v2.b2 = v7;
  if (System.timeInMillis() % 23 == 0) v2.b1 = v10;
  // call method
  Library.component(v1, v2, v3);
}
```

**Figure 3.13:** Generated stub for the component in Figure 3.10 and the alias configuration in Figure 3.12a.

of the analysis in a given context through previously obtained results in other contexts. We introduced a new language to specify conditions on the components context and showed how to generate context-stubs from those conditions. These context-stubs allow us to use whole-program analyses to analyze a component for any context that fulfills these conditions. With this approach however we still need to run a full-blown analysis for every context variant.

In this section we introduce the *modular SDG* —a special variant of the SDG presented in §2.5.3. The modular SDG of a component can be computed independent of the components context. Given a specific context we can extract a standard SDG from the modular SDG in significantly less computation time and with lower memory usage than a standard SDG computation would take. The modular SDG exploits the fact that many dependencies —like control (Definition 2.3) and direct data dependencies (Definition 3.5)— are not influenced by the context of the respective component. Only heap data dependencies (Definition 3.7) depend on the alias configuration at the call site of the component. Therefore it includes a new form of heap data dependencies —called *conditional dependencies*— annotated with alias conditions that must be met for the dependencies to occur. In the following section §3.4.1 we provide an overview of the steps needed to compute a modular SDG, §3.4.2 contains a detailed description of the new conditional dependencies, §3.4.4 contains the adjustments to the summary computation phase for the modular SDG. Section §3.4.3 introduces *access paths* and the accompanying computation algorithm. Access paths allow us to compute the conditional dependencies of the modular SDG. Finally section §3.4.5 concludes with an evaluation of the modular IFC approach.

### 3.4.1 Overview

In order to compute the modular SDG of a component we approximate its structure through the SDGs in the minimal and maximal context configuration possible. Any dependency that is part of the minimal SDG is also present in any other context. Therefore any dependency that is part of the maximal SDG, but not part of the minimal SDG, is obviously a conditional dependency.

In the next step we compute the actual conditions of the conditional dependencies. This can be achieved through a brute force approach that enumerates all possible contexts or with a more sophisticated approach similar to the inference algorithm for relevant context conditions (Algorithm 3.1). However, we propose a new algorithm based on access paths that takes the maximal SDG as input, automatically detects conditional heap dependencies and subsequently computes an approximation of the matching alias conditions for them. While the access path algorithm can detect alias conditions for single heap dependencies, it cannot be used to compute conditions for the summary edges between the components input and output parameters.

When we extract the SDG for a given context configuration from the modular SDG with conditional heap dependencies, we trigger all conditional dependencies whose condition is satisfied by the context. Then we recompute the summary edges for the specific context at the plug-in site. As we know the minimal and maximal SDGs of the component, this process can be sped up significantly: We inject the summary edges of the minimal SDG at the starting point of the computation and run an adapted version of the summary computation from there.

When the summary computation finished, we analyze the resulting SDG for information flow properties in the given context configuration.

### 3.4.2 Conditional data dependencies

The modular system dependence graph $SDG^*(m)$ of a component $m$ contains conditions for all heap data dependencies that only occur in specific contexts. These *conditional dependencies* are annotated with may-alias conditions. In contrast to (not-)alias conditions as defined in Definition 3.11, may-alias conditions represent a positive condition that holds for a context $C$ if the dependency occurs in the dependence graph for the component in this context. We write $SDG_C(m)$ for the dependence graph of $P\|_m$ in context $C$ and $\mathit{May\text{-}Cond}(x)$ to denote the may-alias conditions implied by $x$.

**Definition 3.15** (Conditional Dependence). *A conditional dependence inside a modular dependence graph $SDG^*(m)$ for a component $m$ is a heap data dependency $n_1 \xrightarrow{dh} n_2$ annotated with a may-alias condition $\mathit{May\text{-}Cond}(n_1 \xrightarrow{dh} n_2)$ that evaluates to true for all contexts $C$ in which $n_1 \xrightarrow{dh} n_2 \in SDG_C(m)$.*

*We define $\mathbb{C}_{n_1,n_2}$ as the set of all valid contexts of $m$ in which the heap data dependence $n_1 \xrightarrow{dh} n_2$ occurs.*

$$\mathbb{C}_{n_1,n_2} := \{C \mid C \in \mathbb{C}_m \wedge n_1 \xrightarrow{dh} n_2 \in SDG_C(m)\}$$

*The may-alias conditions implied by $\mathbb{C}_{n_1,n_2}$ are used to annotate $n_1 \xrightarrow{dh} n_2$.*

$$\mathit{May\text{-}Cond}(n_1 \xrightarrow{dh} n_2) := \bigvee_{\forall C \in \mathbb{C}_{n_1,n_2}} \bigwedge_{(a,b) \in C} \mathit{alias}(a,b)$$

These conditions can be simplified by standard logic operations, but we can also reduce the size of these conditions with a limited subset of $\mathbb{C}_{n_1,n_2}$ that only contains relevant configurations. When we use $\downarrow \mathbb{C}_{n_1,n_2}$ similar to Definition 3.10 as the set of all minimal elements in $\mathbb{C}_{n_1,n_2}$

$$\downarrow \mathbb{C}_{n_1,n_2} := \{C \mid C \in \mathbb{C}_{n_1,n_2} \wedge \nexists C' \in \mathbb{C}_{n_1,n_2} : C \neq C' \wedge C' \sqsubseteq C\}$$

the resulting conditions are still equivalent[19] to $\mathit{May\text{-}Cond}(n_1 \xrightarrow{dh} n_2)$ .

[19]Using a similar reasoning as in Theorem 3.3

$$
\begin{aligned}
\mathit{May\text{-}Cond}(\mathbb{C}_{n_1,n_2}) &= \mathit{May\text{-}Cond}(\downarrow \mathbb{C}_{n_1,n_2}) \\
&= \bigvee_{\forall C \in \downarrow \mathbb{C}_{n_1,n_2}} \mathit{May\text{-}Cond}(C) \\
&= \bigvee_{\forall C \in \downarrow \mathbb{C}_{n_1,n_2}} \bigwedge_{(a,b) \in C} \mathit{alias}(a,b)
\end{aligned}
$$

Due to the monotonicity property we can guarantee that for any context $C$ where $\exists C' \in \downarrow \mathbb{C}_{n_1,n_2}$ with $C \sqsubseteq C'$ the dependency $n_1 \text{-}dh\!\rightarrow n_2$ is not present in $SDG_C(m)$ and so $\mathit{May\text{-}Cond}(\downarrow \mathbb{C}_{n_1,n_2})$ is necessary for $n_1 \text{-}dh\!\rightarrow n_2$.

We argued in §3.2.1 that enumeration of all valid context configurations is possible, but it can become rather complex. So computing $\mathbb{C}_{n_1,n_2}$ through enumeration is only feasible in some cases. Our evaluation results have shown that the number of type-correct context configurations can become huge, often around $2^{10}$ and up to $2^{100}$. We suggest an approach using so-called *access paths* to compute these conditions more efficiently.

### 3.4.3 Access paths

*Access paths* describe referenced locations inside a component through a syntactic representation of subsequent field accesses needed to reach the location. Access paths start from the components input parameters, return values of method calls or newly created objects. They include a path of field names that need to be dereferenced. This description of locations uses a syntactic representation of parameters and fields that is independent of aliasing contexts and specific points-to information. We are going to compute access paths for all statements in a given SDG. Then we use them to decide which statements are referring to the same location independent of aliasing context or if two different statements may refer to the same location depending on initial aliasing. This approach enables us to infer aliasing conditions for heap data dependencies without recomputing the component in various alias contexts, as previously suggested in §3.4.2.

```
class Library implements Lib {
  int call(A a, A b, A c, int x) {
    c.i = a.i;
    a.i = x;
    return b.i;
  }
}
```

**Figure 3.14:** Relevant part of the code from Figure 3.7.

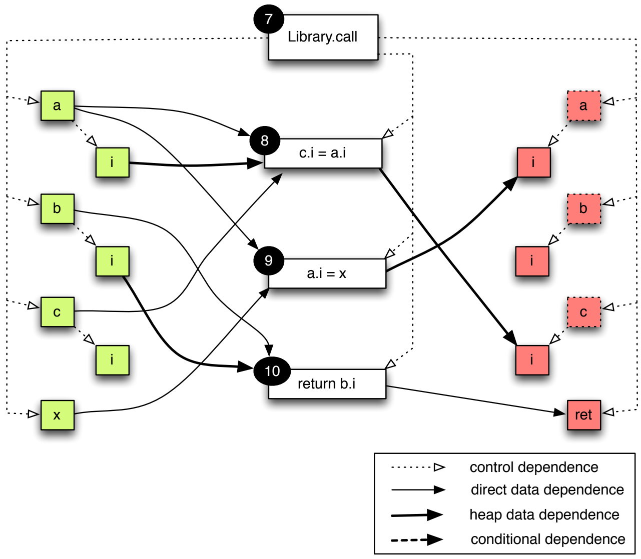


**Figure 3.15:** SDG in minimal alias context $C_{\text{min}}$ of method `call` from Figure 3.14.

For example method `Library.call` from Figure 3.14 may be called in various contexts. Depending on the calling context different dependencies occur:

- In $\mathbb{C}_{min}$ all three statements are independent from each other and parameter `x` does not influence the return value. Figure 3.15 shows the matching $SDG_{C_{\text{min}}}$ for the minimal alias context.
- In $\mathbb{C}_{max}$ —with input parameters `a`, `b` and `c` potentially aliased—

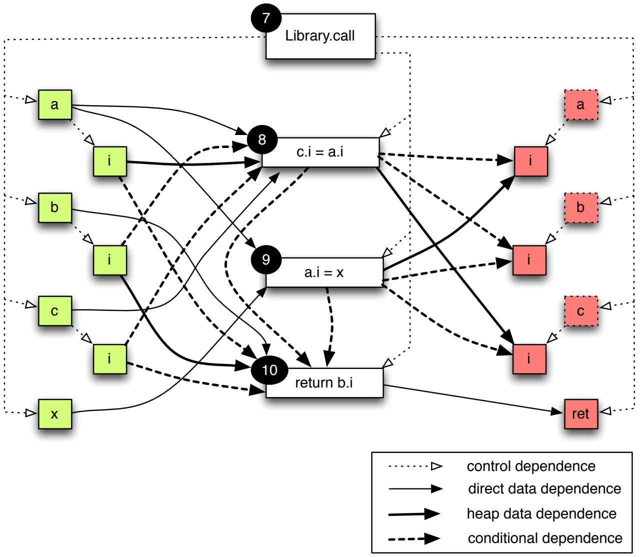


**Figure 3.16:** SDG in maximal alias context $C_{\max}$ of method `call` from Figure 3.14. Additional conditional heap dependencies compared to Figure 3.15 are dashed.

the statements are no longer independent and a flow from `x` to the return value exists. Figure 3.16 shows the $SDG_{C_{\max}}$ where additional heap dependencies —compared to $SDG_{C_{\min}}$— are drawn with a dashed line. Those dependencies are conditional dependencies, as they are not present in $SDG_{C_{\min}}$ and therefore clearly depend on the initial alias configuration.

Our goal is to compute the necessary alias conditions for the conditional dependencies. If these alias conditions do not hold in a given initial context configuration, the conditional dependency is not part of the SDG specific to this configuration. Figure 3.17 shows the SDG with annotated access paths for each node and alias conditions for each conditional dependency. The graph omits all unconditional data dependencies already present in $SDG_{C_{\min}}$ to improve readability and only shows conditional dependencies. For example statement 10 reads the value set in statement 9 only if parameters `a` and `b` are initially aliased.

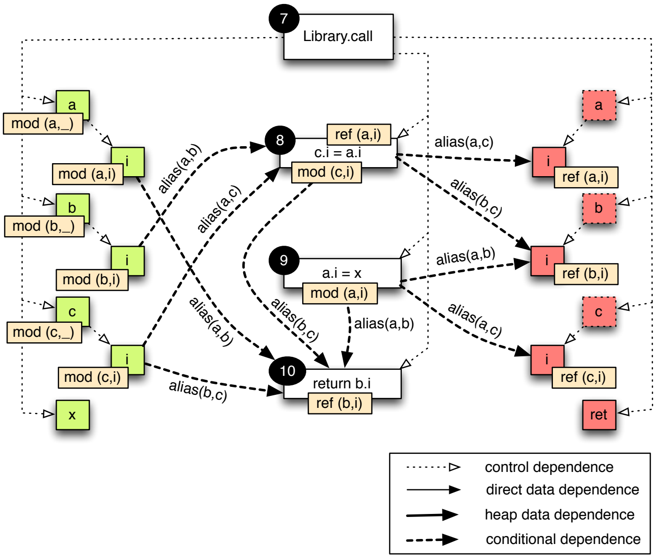


**Figure 3.17:** Conditional dependencies in the modular SDG for Figure 3.14. Nodes are annotated with matching access paths.

In case we want to extract the SDG where only a and b are aliasing ($\mathbb{C}_{alias(a,b)} = \{(a, b)\}$) from the modular SDG, we have to check each conditional dependency if it is valid in $C_{alias(a,b)}$. Invalid dependencies are removed, valid dependencies are replaced with standard heap data dependencies. The resulting $SDG_{C_{alias(a,b)}}$ —shown in Figure 3.18— is a standard non-modular SDG suitable for IFC analysis.

The conditional dependencies in Figure 3.17 have been inferred from the access paths of the source and destination nodes for each dependency. They are also shown in the Figure as small boxes above or beneath each node. An access path $(r, f_1 \to \ldots \to f_n)$ consists of a root parameter $r$ and an access path of fields $f_i$. The path of fields describes the field access operations needed to reach the referenced location starting from $r$. A "mod" denotes that the location reachable by the access path is modified, while "ref" marks referenced locations. Whenever the access paths between source and sink do not share a common element the

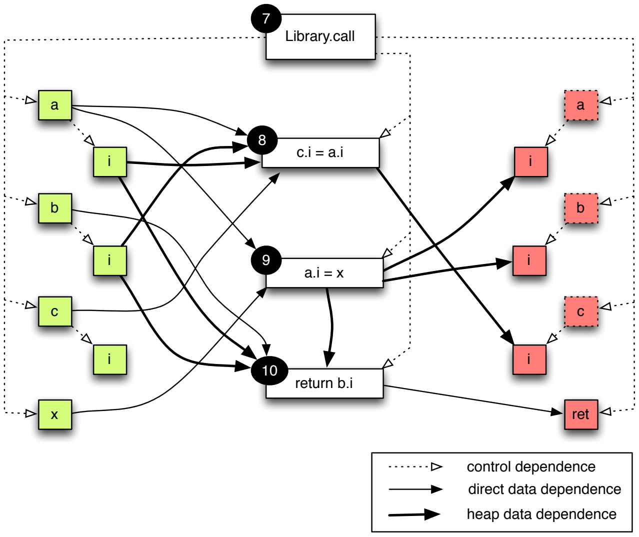


**Figure 3.18:** The SDG for the code in Figure 3.14 in the context $C_{alias(a,b)}$.

dependency is conditional. The condition is built from the elements of the access paths: The dependency occurs if any element described by the "mod" access paths of the source node is initially aliased to any element of the "ref" access paths from the destination node. Again looking at statement 9 and 10, 10 reads data modified by statement 9 if $(a, i)$ and $(b, i)$ refer to the same location. Thus condition $alias(a, b)$ is inferred. In practice we do not need to distinguish mod and ref access paths if we use the fine-grained field access representation from §2.4.3. With fine-grained field accesses, no node holds mod and ref access paths at the same time, so the type of access can be inferred from the node type.

We start with the definition of access paths before we explain in detail how they are computed for a given SDG. The access paths of a node contain a path for any potential trace of field accesses in the program that may have been used to reach the location.

**Definition 3.16** (Access Path). *An* access path *ap is a tuple $(r, fp)$ of a root node r and a field path $fp$. It identifies a set of locations on the heap that can be reached from r through field accesses given by $fp$. A* root node *corresponds to values passed to the current method or created within. It is either a method parameter, static variable, return value of a call or a new-instruction. A* field path *is a possibly empty list of object fields. It describes the fields that need to be dereferenced in order to reach the desired location.*

$$fp \quad = \quad \begin{cases} \_ & \textit{no field accesses, direct value} \\ f_1 \to \ldots \to f_n & \textit{subsequent accesses: } i \in [1 \ldots n], f_i \in \textit{Fields} \end{cases}$$

Some access paths are only viable in case some conditions are met. These conditions come in the form of aliasing between statements.

**Definition 3.17** (Conditional Access Paths). *We write $AP_n$ for the set of conditional access paths of node $n \in SDG$. A conditional access path $cap = (cond, ap)$ consists of a condition cond and an access path $ap = (r, fp)$ with a root node r and a field path $fp$. The condition cond is a boolean formula of alias conditions for access paths of nodes. It describes in which aliasing configurations the access path exists.*

*An alias condition alias-ap$(n_1, n_2)$ for access paths of nodes $n_1$, $n_2$ is recursively defined through the aliasing of their conditional access paths.*

$$\textit{alias-ap}(n_1, n_2) := \bigvee_{\substack{(cond_1, ap_1) \in AP_{n_1} \\ (cond_2, ap_2) \in AP_{n_2}}} cond_1 \wedge cond_2 \wedge alias(ap_1, ap_2)$$

The aliasing of access paths, given a concrete context configuration $C$ , computes directly from the points-to information of the described locations (Definition 2.15).

$$alias_C((r, f_1 \to \cdots f_n), (r', f'_1 \to \cdots f'_m)) := alias_C(r.f_1 \ldots f_n, r'.f'_1 \ldots f'_m)$$

As previously shown in the example from Figure 3.17 we use access paths to infer for each heap data dependence $n_1 \text{-}dh\text{→} n_2$ in which alias configuration it may hold and when it can be removed: If the access paths of $n_1$ and $n_2$ share a common path, the dependence may be present in any alias configuration else it only holds in certain configurations.

When there are no common paths, any aliasing between access paths of $n_1$ and $n_2$ is a possible condition for the dependence. This leads to the *access path theorem.*

**Theorem 3.5** (Access Path Theorem). *Given a heap data dependency $n_1 \xrightarrow{dh} n_2$ in the maximal $SDG_{C_{max}}$ of a component, the dependency is not present in any other $SDG_C$ with an initial alias configuration $C$ if the access paths $AP_{n_1}, AP_{n_2}$ of $n_1, n_2$ are not aliasing in $C$ .*

$$\begin{aligned}(\forall(cond_1, ap_1) \in AP_{n_1} \forall(cond_2, ap_2) \in AP_{n_2} :\\ \neg(cond_1 \wedge cond_2 \wedge alias_C(ap_1, ap_2))) \implies n_1 \xrightarrow{dh} n_2 \notin SDG_C\end{aligned}$$

In the following we are going to show the computation of access paths in the intra- and interprocedural case and how the result can be used to derive conditional data dependencies. Additionally we sketch a proof of the access path theorem for the intraprocedural setting. The proof for the interprocedural part is considered future work and not in the scope of this thesis.

## Computation

The access path computation is split in an intraprocedural (Algorithm 3.4) and an interprocedural (Algorithm 3.5) part. The intraprocedural part uses the system dependence graph *sdg* in the maximal alias context as input and produces a mapping *n2ap* of nodes to their access paths as output. Access paths are propagated unconditional along local data dependencies and parameter structure edges, propagation along heap data dependencies adds an alias condition to each propagated access path.

The intraprocedural computation (Algorithm 3.4) takes the following steps separately for each methods *pdg*.

1. Create initial conditional access paths for parameter and new-instance nodes in `initialAccessPaths`.
2. Propagate the access paths along data dependency and parameter structure edges $e$ until a fixed-point is reached in `propagateIntraproc`.

   Given edge $e = n_1 \rightarrow n_2 \in SDG_{C_{\max}}$, propagate access paths from source $n_1$ to target $n_2$.

(a) **Case** $n_1 \xrightarrow{ps} n_2$**:** All access paths of $n_1$ are extended with the field represented by $n_2$ and propagated to $n_2$.

(b) **Case** $n_1 \xrightarrow{dd} n_2$**:** If $n_2$ is a field access and $e$ is a data dependency to the base pointer of the field access then every access path of $n_1$ is extended by the field accessed in $n_2$. In any other case the access paths of $n_1$ are simply propagated to $n_2$.

(c) **Case** $n_1 \xrightarrow{dh} n_2$**:** All access paths of $n_1$ are propagated to $n_2$ with the additional alias condition $\mathit{alias}(n_1, n_2)$.

Step 2 is repeatly executed until the access paths of all nodes no longer change. The result of the intraprocedural propagation captures all method local changes on the access paths between method entry and exit. So the access paths of the formal-out nodes of each method represent the method local effects that are visible outside the method itself. These effects need to be propagated from callee to call site in the interprocedural computation in order to capture the effects for a whole component.

**Algorithm 3.4** (Compute intraprocedural access paths)**.**

```
MAIN intraprocAccessPaths($sdg$, $n2ap$)
IN:  System dependence graph $SDG_{C_{max}}$ in maximal alias configuration $sdg$
OUT: Mapping of nodes $n \in sdg$ to intraprocedural conditional access paths $AP_n$ $n2ap$
BEGIN
  $n2ap$ = map all nodes to an empty set.
  FORALL (PDGs $pdg \in sdg$) DO
    initialAccessPaths($pdg$, $n2ap$) // step 1
    propagateIntraproc($pdg$, $n2ap$) // step 2
  DONE
END

PROCEDURE propagateIntraproc($pdg$, $n2ap$)
IN:  Procedure dependence graph $pdg$
     Mapping from nodes $n \in pdg$ to conditional access paths $n2ap$
OUT: Returns true iff $n2ap$ mapping changed
BEGIN
  DO
    FORALL (edges $e = n_1 \rightarrow n_2 \in pdg$) DO
      SWITCH (type of edge $e$)
        CASE parameter structure edge: // step 2a
          // $n_2$ must be a field node
          // $f$ = corresponding field of field node $n_2$
          $n2ap(n_2)$ = $n2ap(n_2) \cup$ expand($n2ap(n_1)$, $f$)
        CASE local data dependence: // step 2b
          // $n_2$ is a field-get ("$v_2 = v_1.f$") or
          //          field-set ("$v_1.f = v_2$") operation on field $f$
          IF ($e$ is data dependency for field base pointer $v_1$) THEN
            $n2ap(n_2)$ = $n2ap(n_2) \cup$ expand($n2ap(n_1)$, $f$)
          ELSE
            $n2ap(n_2) = n2ap(n_2) \cup n2ap(n_1)$
          FI
        CASE heap data dependence: // step 2c
          FORALL $(cond, (r, fp)) \in n2ap(n_1)$ DO
            Add $(alias(n_1, n_2) \wedge cond, (r, fp))$ to $n2ap(n_2)$
          DONE
      ESAC
    DONE
  WHILE ($n2ap$ changed)
  RETURN true iff $n2ap$ changed
END

PROCEDURE expand($paths$, $f$)
IN:  Set of access paths $paths$
     Field $f$
OUT: Set of all conditional access paths in $paths$ extended with field $f$
BEGIN
  $expanded = \{\}$
  FORALL (access paths $ap = (cond, (r, f_1 \rightarrow \ldots f_n)) \in paths$) DO
    IF ($\exists f_i \in \{f_1, \ldots f_n\}$ with $f_i = f$) THEN
      $expanded$.add($(cond, (r, f_1 \rightarrow \ldots f_i))$)
    ELSE
      $expanded$.add($(cond, (r, f_1 \rightarrow \ldots f_n \rightarrow f))$)
    FI
  DONE
  RETURN $expanded$
END

PROCEDURE initialAccessPaths($pdg$, $n2ap$)
IN:  Procedure dependence graph $pdg$
OUT: Added mapping from nodes $n \in pdg$ to initial conditional access paths in $n2ap$
BEGIN
  FORALL (nodes $n \in pdg$) DO
    IF (($n$ is root parameter AND ($n$ is formal-in/out OR actual-out))
        OR $n$ is new-instance operation)
    THEN
      $n2ap(n) = \{(true, (n, \_))\}$
    FI
  DONE
END
```

Figure 3.19 shows a schematic overview of the interprocedural propagation at each call site. The algorithm is described in detail in Algorithm 3.5. It starts with an initial run of the intraprocedural access path computation for each method and then subsequently propagates method local effects from callee to call site until a fixed-point is reached. Method accessPaths contains these two basic steps. The interprocedural propagation takes place in `propagateInterprocedural`. It applies the following steps until all access paths no longer change and a fixed-point is reached.

1. Propagate access paths intraprocedural for each method
2. Substitute access paths of callee input parameters with access paths of parameters at call site
3. Reiterate intraprocedural propagation at callee
4. Transfer access paths from callee output parameters to call site
5. Reiterate intraprocedural propagation at caller

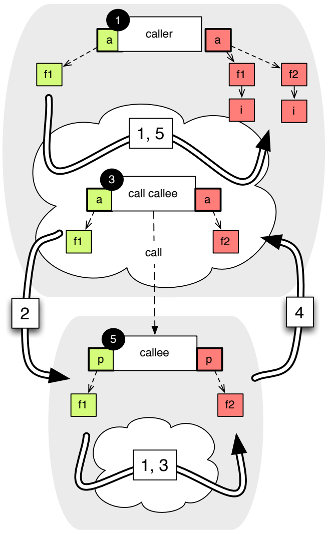


**Figure 3.19:** Schematic interprocedural propagation of access path information at a single call site.

1. For each call site *call* and each called target *tgt* in the SDG it propagates the effects inside *tgt* so the call site.
   (a) Search the matching formal-in nodes of the target for the actual-in nodes of the call.
   (b) Temporarily replace the access paths of the formal-in node with the access paths of the actual-in node for all nodes of *tgt*.
   (c) Propagate the temporarily replaced access paths intraprocedural.
   (d) Search the matching actual-out nodes of the call for each formal-out of the target.
   (e) Permanently replace all access paths of the actual-out node with the access paths of the formal-out node for all nodes in the PDG of the call site.
2. For all methods with changed access paths the intraprocedural propagation is run again.

**Algorithm 3.5** (Compute interprocedural access paths)**.**

```
PROCEDURE accessPaths($sdg$)
IN:  System dependence graph $SDG_{C_{max}}$ in maximal alias configuration $sdg$
OUT: Mapping of nodes $n \in sdg$ to set of interprocedural access paths $AP_n$ $n2ap$
BEGIN
  $n2ap$ = empty mapping from nodes $\in sdg$ to sets of access paths
  intraprocAccessPaths($sdg$, $n2ap$)
  propagateInterprocedural($sdg$, $n2ap$)
END

PROCEDURE propagateInterprocedural($sdg$, $n2ap$)
IN:  System dependence graph $SDG_{C_{max}}$ in maximal alias configuration $sdg$
     Mapping of nodes $n \in sdg$ to set of intraprocedural access paths $AP_n$ $n2ap$
OUT: Mapping of nodes $n \in sdg$ to set of interprocedural access paths $AP_n$ $n2ap$
BEGIN
  DO
    FORALL (call nodes $call \in sdg$) DO
      FORALL (entry nodes $tgt \in sdg$ where $call$ calls $tgt$) DO
        // propagate effects to call site
        propagateFromCalleeToSite($call$, $tgt$, $n2ap$, $sdg$)
      DONE
    DONE

    FORALL (PDGs $pdg \in sdg$ where $\exists n \in pdg$ with $n2ap(n)$ has changed) DO
      propagateIntraproc($pdg$, $n2ap$)
    DONE
  WHILE ($n2ap$ changed)
END

PROCEDURE propagateFromCalleeToSite($call$, $tgt$, $n2ap$, $sdg$)
IN:  Call node $call$
     Entry node of callee $tgt$
     Mapping of nodes $n \in sdg$ to set of access paths $AP_n$ $n2ap$
     System dependence graph $SDG_{C_{max}}$ in maximal alias configuration $sdg$
OUT: Adjusted mapping including effects inside callee at the call site $n2ap$
BEGIN
  $n2apSubst$ = empty mapping from nodes to set of access paths
  // search matching form-in nodes for act-ins of call and
  // replace callee access paths with access paths found at actual-ins
  FORALL (actual-in nodes $aIn$ of $call$) DO
    $fIn$ = matching formal-in node of $aIn$ for callee $tgt$
    substitute($tgt$, $sdg$, $n2ap$, $n2ap(fIn)$, $n2ap(aIn)$, $n2apSubst$)
  DONE
  // propagate temporarily replaced access paths inside callee
  propagateIntracproc($tgt$, $n2apSubst$)
  // search matching act-out nodes for form-out of callee
  // replace access paths of act-outs with those of form-outs
  $callerEntry$ = entry node for the method containing node $call$
  FORALL (actual-out nodes $aOut$ of $call$) DO
    $fOut$ = matching formal-out node of $aOut$ for callee $tgt$
    substitute($callerEntry$, $sdg$, $n2ap$, $n2ap(aOut)$, $n2apSubst(fOut)$, $n2ap$)
  DONE
END
```

```
PROCEDURE substitute($entry$, $sdg$, $n2ap$, $apOrig$, $apNew$, $n2apSubst$)
IN:  Entry node of a method $entry$
     System dependence graph $SDG_{C_{max}}$ in maximal alias configuration $sdg$
     Mapping of nodes to set of access paths $n2ap$
     Set of original access paths $apOrig$
     Set of access paths to replace with $apNew$
OUT: Mapping of nodes to set of substituted access paths $n2apSubst$
BEGIN
  FORALL (nodes $n \in sdg$ in the same method as $entry$) DO
    FORALL ($ap = (cond, (r, f_1 \rightarrow \dots f_l)) \in n2ap(n)$) DO
      IF ($\exists i : 0 >= i <= l : (cond_{orig}, (r, f_1 \rightarrow \dots f_i)) \in apOrig$) THEN
        FORALL ($ap' = (cond_{new}, (r', f'_1 \rightarrow \dots f'_k)) \in apNew$) DO
          $n2apSubst(n) = n2apSubst(n)$
              $\cup\{(cond_{orig} \wedge cond_{new}, (r', f'_1 \rightarrow \dots f'_k \rightarrow f_{i+1} \dots f_l))\}$
        DONE
      FI
    DONE
  DONE
END
```

### From access path to conditional data dependencies

Given a heap data dependency $n_1 \text{-}dh\!\rightarrow n_2$ and the result of the access path computation —a mapping from each node $n$ to a set of conditional access paths $AP_n$— we can decide if and under which context configurations the dependency will be part of the SDG using the alias condition for $n_1$ and $n_2$.

$$n_1 \text{-}dh\!\rightarrow n_2 \in SDG_C \implies alias_C(n_1, n_2) \implies alias\text{-}ap_C(n_1, n_2)$$

The computed access paths are a conservative approximation, hence $alias\text{-}ap_C(n_1, n_2)$ may evaluate to true even when the aliasing is in fact not present for the given context $C$ . However the opposite is never the case: Whenever $alias\text{-}ap_C(n_1, n_2)$ evaluates to false the aliasing and thus the data dependency is definitely not existing in configuration $C$ .

$$alias\text{-}ap(n_1, n_2) \not\models C \implies n_1 \text{-}dh\!\rightarrow n_2 \notin SDG_C$$

With Definition 3.17 the alias condition for nodes can be transformed to alias conditions for access paths.

$$alias\text{-}ap(n_1, n_2) = \bigvee_{\substack{(cond_1, ap_1) \in AP_{n_1} \\ (cond_2, ap_2) \in AP_{n_2}}} cond_1 \wedge cond_2 \wedge alias(ap_1, ap_2)$$

Conditions *cond*$_1$ and *cond*$_2$ can be transformed to alias conditions for access paths accordingly. In practice —as we aim for a scalable conservative approximation— it is safe to substitute a condition with *true* in case the formula becomes too complicated to evaluate within existing time- and memory-constraints.

We approximate $\mathbb{C}_{n_1,n_2}$ (Definition 3.15), the set of all contexts $C$ where $n_1 \text{-}dh\!\rightarrow n_2$ occurs in the matching $SDG_C$, with the help of the conditional access paths as $\mathbb{C}^{AP}_{n_1,n_2}$.

$$\mathbb{C}^{AP}_{n_1,n_2} := \{C \mid C \in \mathbb{C} \land \textit{alias-ap}_C(n_1, n_2)\}$$

Obviously —given access paths are a conservative approximation— $\mathbb{C}_{n_1,n_2} \subseteq \mathbb{C}^{AP}_{n_1,n_2}$ holds, so $\downarrow\mathbb{C}_{n_1,n_2}$ —the set of all minimal elements in $\mathbb{C}_{n_1,n_2}$— can be approximated in a similar way as $\downarrow\mathbb{C}^{AP}_{n_1,n_2}$.

$$\downarrow\mathbb{C}^{AP}_{n_1,n_2} := \{C \mid C \in \mathbb{C}^{AP}_{n_1,n_2} \land \nexists C' \in \mathbb{C}^{AP}_{n_1,n_2} : C \neq C' \land C' \sqsubseteq C\}$$

By definition it follows that

$$\forall C \in \downarrow\mathbb{C}_{n_1,n_2} \exists C' \in \downarrow\mathbb{C}^{AP}_{n_1,n_2} : C' \sqsubseteq C$$

The minimal satisfying condition for $n_1 \text{-}dh\!\rightarrow n_2$ derived from the approximated set of minimal contexts $\downarrow\mathbb{C}^{AP}_{n_1,n_2}$ therefore is weaker than the actual condition.

$$\begin{aligned} &\textit{May-Cond}(n_1 \text{-}dh\!\rightarrow n_2) = \textit{May-Cond}(\downarrow\mathbb{C}_{n_1,n_2}) \\ \Longrightarrow\ &\textit{May-Cond}(\downarrow\mathbb{C}^{AP}_{n_1,n_2}) \end{aligned}$$

So we can still guarantee that if a conditional dependency $n_1 \text{-}dh\!\rightarrow n_2$ is removed from $SDG_C$ due to $C$ not satisfying $\textit{May-Cond}(\downarrow\mathbb{C}^{AP}_{n_1,n_2})$ , $C$ also does not satisfy $\textit{May-Cond}(\downarrow\mathbb{C}_{n_1,n_2})$ . Therefore the resulting SDGs remain a conservative approximation.

**Correctness**

We are going to show that the computation of access paths is correct, meaning Theorem 3.5 holds for the access paths computed by Algorithm 3.4. We restrict this proof to the intraprocedural case and a simpler language without array accesses and static fields. Thus only object field accesses are allowed. Extending this proof to full Java is considered future work. The outline is as follows: We start off with definition of execution traces, trace based dependencies, their connection to dependencies in the SDG and dynamic access paths. Then we show that for each dynamic access path there always exists a matching (static) access path. We argue that a special form of dynamic access paths called *origin access paths* describe how locations can be reached from the initial memory configuration —also referred to as the context of the component. This helps to connect trace-based heap dependencies to aliasing in the initial memory configuration. Finally we show that whenever an additional trace-based heap dependency is triggered by initial aliasing of access paths, the relevant access paths are also found in the static approximation and therefore the resulting dependence conditions (Theorem 3.5) are sound.

**Traces, dynamic dependencies and static approximation** We define traces in a similar fashion to Giffhorn [29] as an ordered infinite list of actions[20].

**Definition 3.18** (Trace). *A trace $T$ for a program $P$ is a potential infinite list of ordered actions $a_0, a_1, \dots a_j, \dots$*

$$(\overline{m}_0, o_0, m_0), (\overline{m_1}, o_1, m_1), \dots (\overline{m_j}, o_j, m_j), \dots$$

*where each action $(\overline{m}_i, o_i, m_i)$ consists of an operation $o_i$ from program $P$, the state of the memory $\overline{m}_i$ before the execution of $o_i$ and afterwards $m_i$. For two consecutive actions $a_i, a_{i+1}$ the states of the memory match each other: $m_i = \overline{m}_{i+1}$.*

*We write $v[m]$ to refer to the value of variable $v$ evaluated at the memory state $m$. As a shorthand we use $v[\overline{a}]$ and $v[a]$ to refer to the value of $v$ before and after the execution of action $a$.*

*We refer to the first trace action $a_0$ in each trace also as the initial trace action $a_{init}$.*

The memory state of a trace action $a$ specifies which access paths point-to the same location —are aliased— before or after $a$ executes. We

[20]We chose to name single trace elements *action* instead of *configuration* to prevent confusion with alias configurations.

write $alias_a$ for the alias configuration after execution of $a$ and $alias_{\overline{a}}$ before execution of $a$. We are going to define heap data dependence in traces based on aliasing, but first we have to start off with data dependence through local variables.

**Definition 3.19** (Data Dependence for Traces). *Two actions $a, a' \in T$ are data dependent ($a \text{-}dd\text{→} a'$) iff*

- *Operation $o$ of $a$ defines a variable $v$ that $o'$ of $a'$ reads.*
- *$a$ happens before $a'$ in $T$.*
- *No action between $a$ and $a'$ contains an operation that redefines $v$.*

*We write $def(a', v)$ for the action $a$ that defines variable $v$ used in $a'$. Note that $def(a', v)$ is unambiguous and always exists. It is the first action preceding $a'$ that defines $v$.*

Data dependencies in traces are related to data dependencies in the corresponding system dependence graph of the program independent of the initial aliasing configuration.

**Corollary 3.1** (SDGs and Data Dependence in Traces). *If action $a_1, a_2$ of a trace $T$ of program $P$ are data dependent then the nodes of the operations $o_1, o_2$ are also data dependent in the corresponding system dependence graph $SDG_P$ of $P$.*

$$\forall a_1 = (\overline{m_1}, o_1, m_1), a_2 = (\overline{m_2}, o_2, m_2) \in T :$$
$$a_1 \text{-}dd\text{→} a_2 \implies o_1 \text{-}dd\text{→} o_2 \in SDG_P$$

*Proof Argument.* A system dependence graph contains a conservative approximation of all local data dependencies in the program. This property has already been shown as part of the proof of the correctness of slicing from Wasserrab [129, 127]. □

Besides data dependencies through local variables, heap data dependencies can also occur through values stored on the heap.

**Definition 3.20** (Heap Data Dependence in Traces). *Two trace actions $a_1 \to \ldots \to a_2 \in T$ are heap data dependent ($a_1 \text{-}dh\text{→} a_2$) iff $a_1$ writes a value to a heap location that $a_2$ reads and no action $a_3$ in between $a_1$ and $a_2$ overwrites* *the value. Heap data dependencies can occur only between actions that modify and reference a value stored on the heap. So operation $o_1$ of action $a_1$ is a field-set operation and $o_2$ of $a_2$ is a field-get operation.*

$$\begin{aligned} o_1 &:= v^1_{base}.f^1 = v^1_{val} \\ o_2 &:= v^2_{ref} = v^2_{base}.f^2 \end{aligned}$$

*As $a_1$ and $a_2$ refer to the same location, we know that also the base pointers refer to the same location and the accessed fields must be the same.*

$$v^1_{base}[\overline{m_1}] = v^2_{base}[\overline{m_2}] \wedge f^1 = f^2$$

Heap data dependencies in traces also have corresponding dependencies in the statically computed dependence graph.

**Corollary 3.2** (SDGs and Heap Data Dependence in Traces)**.** *If two actions $a_1, a_2$ of a trace $T$ of program $P$ are heap data dependent ($a_1 \xrightarrow{dh} a_2$) then the system dependence graph computed in the maximal alias configuration $SDG^{max}_P$ contains a heap data dependence edge between the corresponding operations $o_1, o_2$.*

$$\begin{gathered} \forall a_1 = (\overline{m_1}, o_1, m_1), a_2 = (\overline{m_2}, o_2, m_2) \in T : \\ a_1 \xrightarrow{dh} a_2 \implies o_1 \xrightarrow{dh} o_2 \in SDG^{max}_P \end{gathered}$$

*Proof Argument.* Aside from Wasserrabs correctness proof [129, 127], the semantics of slicing including heap data dependencies have already been extensively studied [100, 106, 14]. In combination with the monotonicity property from §3.2.2, we know that the maximal SDG needs to contain all heap data dependencies possible. □

With the definition of local and heap data dependencies for traces in place, we can define the notion of a dynamic data slice for traces.

**Definition 3.21** (Dynamic Data Slice)**.** *The dynamic backward data slice $BS^{dyn}_{data}(a)$ of a trace action $a$ in trace $T$ contains all actions in the transitive closure of heap and data dependent actions leading to $a$.*

$$BS^{dyn}_{data}(a) := \{a\} \cup \{a' \mid \exists a'' \in BS^{dyn}_{data}(a) : a' \xrightarrow{dd} a'' \in T \vee a' \xrightarrow{dh} a'' \in T\}$$

From Corollary 3.1 and Corollary 3.2 it follows that the static backward slice of the system dependence graph is a conservative approximation of the dynamic slice.

**Corollary 3.3** (Static and Dynamic Data Slice). *For every action $a' = (\overline{m}', o', m')$ in the dynamic backward data slice of $a = (\overline{m}, o, m)$ the matching operation $o'$ is also in the static backward data slice of $o$ when the maximal aliasing configuration is assumed.*

$$(\overline{m}', o', m') \in BS^{dyn}_{data}((\overline{m}, o, m)) \implies o' \in BS_{data}(SDG^{max}_{P}, o)$$

As static data slices in the maximal SDG are a conservative approximation, this can be extended to standard slices including control dependencies. Therefore we define *control dependence for traces* similar to dynamic control dependence from Xin and Zhang [132]. Two actions are trace control dependent if (1) one action happens after the other, (2) their respective operations are statically control dependent and (3) there if no action in between that also fulfills those requirements.

**Definition 3.22** (Control Dependence for Traces). *Two trace actions $a_1 = (\dots, o_1, \dots)$, $a_2 = (\dots, o_2, \dots)$ of trace $T$ are dynamic control dependent ($a_1 \text{-}cd\!\rightarrow a_2$) iff*

1. $a_1 \rightarrow^* a_2 \in T$
2. $o_1 \text{-}cd\!\rightarrow o_2 \in SDG^{max}_{P}$
3. $\nexists a' = (\dots, o', \dots) \in T : a_1 \rightarrow^* a' \rightarrow^* a_2 \land o' \text{-}cd\!\rightarrow o_2 \in SDG^{max}_{P}$

With dynamic data dependence and control dependence in place we define the *dynamic slice* as follows.

**Definition 3.23** (Dynamic Slice). *The dynamic backward slice $BS^{dyn}(a)$ of trace action $a$ in trace $T$ contains all actions in the transitive closure of heap, data and control dependent actions leading to $a$.*

$$BS^{dyn}(a) := \{a\} \cup \{a' \mid \exists a'' \in BS^{dyn}(a) : a' \in BS^{dyn}_{data}(a'') \lor a' \text{-}cd\!\rightarrow a'' \in T\}$$

With these definitions in place we can extend Corollary 3.3 to the standard backward slice that includes data dependencies as well as control dependencies.

**Corollary 3.4** (Static and Dynamic Slice). *For every action $a' = (\overline{m}', o', m')$ in the dynamic backward slice of $a = (\overline{m}, o, m)$ the matching operation $o'$ is also in the static backward data slice of $o$ when the maximal aliasing configuration is assumed.*

$$(\overline{m}', o', m') \in BS^{dyn}((\overline{m}, o, m)) \implies o' \in BS(SDG_P^{max}, o)$$

*Proof Argument.* Given $a' \in BS^{dyn}(a)$ we know that $a'$ is connected to $a$ through dynamic data and control dependencies $a' \to \cdots \to a$. Hence there exists an $a''$ with $a' \to a'' \to \cdots \to a$ where $a'$ reaches $a''$ either through (1) a control dependency ($a' \text{-}cd\text{→} a''$) or (2) a data dependency ($a' \text{-}dd\text{→} a''$ or $a' \text{-}dh\text{→} a''$). In case (1) we know from Definition 3.22 that also the matching operations $o'$ and $o''$ must be statically control dependent: $o' \text{-}cd\text{→} o'' \in SDG_P^{max}$. Hence $o' \in BS(SDG_P^{max}, o'')$. In case (2) we know trough Corollary 3.3 that $o' \in BS(SDG_P^{max}, o'')$.

If $a'' = a$ we are done. Otherwise the same argument is used for $a'' \in BS^{dyn}(a) \implies o'' \in BS(SDG_P^{max}, o)$ until by induction $a'' = a$ finally holds. □

**Location and aliasing of access paths for traces** We want to show how access paths can be used to decide if a heap data dependency is present in a certain alias configuration. Therefore we start with access paths for traces and a definition of aliasing between them.

**Definition 3.24** (Location of Access Paths in Traces). *An access path $ap = (r, f_1 \to \ldots f_n)$ describes a location through subsequent field accesses $f_i$ starting from root $r$. The location reached through these accesses depends on the memory state of the program. In the context of traces the location of an access path depends on the memory state of the trace actions $\vec{a} = (a_1, \ldots, a_n)$ used to dereference fields $f_1, \ldots, f_n$.*

$$loc((r, f_1 \to \ldots f_n), (a_1, \ldots, a_n)) := [[\ldots[r.f_1]_{a_1} \ldots]_{a_{n-1}}.f_n]_{a_n}$$

*We also write $ap[\vec{a}] = (r_1, f_1 \to \ldots f_n)[a_1, \ldots, a_n]$ to denote that the field accesses $f_i$ of the access path $ap$ evaluate at the respective trace action $a_i$.*

**Definition 3.25** (Aliasing of Access Paths in Traces). *Two access path $ap_1 = (r_1, f_1 \to \dots f_n)$ and $ap_2 = (r_2, f'_1 \to \dots f'_m)$ are aliasing if they refer to the same location. Given two trace action vectors $\vec{a} = (a_1, \dots, a_n)$, $\vec{a'} = (a'_1, \dots a''_m)$ we can decide if $ap_1$ and $ap_2$ are aliasing.*

$$alias(ap_1[\vec{a}], ap_2[\vec{a'}]) := loc(ap_1, \vec{a}) = loc(ap_2, \vec{a'})$$

We are especially interested in the aliasing of access paths at the initial memory configuration $\overline{m}_{init}$ at the start of each trace, as the alias conditions we are going to proof impose restrictions on the aliasing of access paths in the initial configuration. We write $\vec{a}_{init}$ for the vector that contains only the initial trace action and consequently $ap[\vec{a}_{init}]$ for the location of the access path *ap* evalutated at the initial memory configuration.

**Access path from traces** We define a special access path called *origin access path* that is computed from a given trace $T$, a variable $v$ and a trace action $a$. The origin access path $oap_v(a)$ of $v$ at $a$ reflects how the location of $v$ at $a$ can be reached at the initial memory state $\overline{m}_{init}$ before method execution. Or —in case the referenced location is of an object created during method execution— the origin access path refers to the memory state of the action that created the object.

**Definition 3.26** (Origin Access Path). *The origin access path $oap_v(a)$ of a variable $v$ in trace action $a \in T$ describes how the value can be reached at the initial memory state $\overline{m}_{init}$. The origin access path is built as follows. Instead of starting at action $a$, we start directly at last action in the trace before $a$ that defines $v$: $a_v = def(a, v)$ as $oap_v(a) = oap_v(a_v)$. Depending on the type of the operation of action $a_v$ the origin access path is built recursively until a* new-instance *operation or the* method-entry *($a_{init}$) is reached.*

**field-get** *$oap_v(a_v)$ of a field-get action "$v = v_{base}.f$" is*

$$oap_v(a_v) := \begin{cases} oap_{val}(a_{mod}) & \text{if } \exists a_{mod} \xrightarrow{dh} a_v \in T \\ expand(oap_{base}(a_{base}), f) & \text{else with } a_{base} = def(a_v, v_{base}) \end{cases}$$

*For field-get operations we refer to $oap_v(a_v)$ also as $oap_{ref}(a_v)$ and to*

*$oap_{v_{base}}(a_v)$ of the base pointer as $oap_{base}(a_v)$.*

$$\begin{aligned} oap_{base}(a_v) &:= oap_{v_{base}}(a_{base}) \text{ with } a_{base} = def(a_v, v_{base}) \\ oap_{ref}(a_v) &:= oap_v(a_v) \end{aligned}$$

**field-set** *A field-set action $a = (\overline{m_a}, \text{"}v_{base}.f = v_{val}\text{"}, m_a)$ is special in that it does not define the value of a variable. However it modifies a value on the heap and uses the values of two variables. Therefore we distinguish between 3 different access paths: $oap_{base}(a)$, $oap_{mod}(a)$ and $oap_{val}(a)$. $oap_{base}(a)$ is the access path of the base pointer $v_{base}$, $oap_{mod}(a)$ is the access path of the field that is modified. It can be inferred by expanding the access path of the base pointer.*

$$\begin{aligned} oap_{mod}(a) &:= expand(oap_{base}(a), f) \\ oap_{base}(a) &:= oap_{v_{base}}(a_{base}) \text{ with } a_{base} = def(a, v_{base}) \\ oap_{val}(a) &:= oap_{v_{val}}(a_{val}) \text{ with } a_{val} = def(a, v_{val}) \end{aligned}$$

**assignment** *$oap_v(a_v)$ of an assignment "$v = v'$" is*

$$oap_v(a_v) := oap_{v'}(a_{v'}) \text{ with } a_{v'} = def(a_v, v')$$

**new-instance** *$oap_v(a_v)$ of a new-instance operation $o$ :="$v = new\ C$" is*

$$oap_v(a_v) := (r_o, \_) \text{ with } r_o \text{ root param for operation } o$$

*If $v$ is defined at a new-instance operation, it refers to the instances created by this operation.*

**method entry** *If $a_v$ is the method entry, then $v$ is a method parameter.*

$$oap_v(a_v) := (r_v, \_) \text{ with } r_v \text{ root param for variable } v$$

*If $v$ is defined at method entry, it refers to a method parameter and thus is directly related to a root parameter.*

**Lemma 3.10** (Location of Origin Access Paths). *The location $loc(v, a)$ variable $v$ points-to at trace action $a = (\overline{m}, o, m)$ is the determined through the value of $v$ evaluated in memory $\overline{m}$: $loc(v, a) = v[\overline{m}]$. It can also be computed with the help of the origin access path.*

$$loc(v,a) = \begin{cases} oap_v(a)[\vec{a}_{new}] & \textit{if } v \textit{ refers to a new instance} \\ & \textit{created at } a_{new} \\ oap_v(a)[\vec{a}_{init}] & \textit{else} \end{cases}$$

*Proof.* If $a$ is a

**field-get** action "$v = v_{base}.f$" then $oap_v(a)$ has two cases:

Case (1) $oap_{val}(a_{mod})$ if $\exists a_{mod} \xrightarrow{dh} a \in T$ then $oap_{val}(a_{mod})[\vec{a}] = oap_{val}(a_{mod})[\vec{a}_{mod}]$. $a_{mod}$ is a field-set action "$v_{base}.f = v_{val}$". We follow $oap_{val}(a_{mod})$ to the action that defines $v_{val}$: $a_{val} = def(a, v_{val})$. Then $oap_{val}(a_{mod}) = oap_{v_{val}}(a_{val})$ hence $oap_{val}(a_{mod})[\vec{a}_{mod}] = oap_{v_{val}}(a_{val})[\vec{a}_{val}]$.

Case (2) $expand(oap_{base}(a_{base}), f)$ with $a_{base} = def(a_v, v_{base})$. As no action overwriting the value of $f$ in between $a_{base} \rightarrow^* a$ exists, it holds that $expand(oap_{base}(a_{base}), f)[\vec{a}] = expand(oap_{base}(a_{base}), f)[\vec{a}_{base}]$.

**assignment** action "$v = v'$" then $oap_v(a_v) = oap_{v'}(a_{v'})$ with $a_{v'} = def(a_v, v')$. Hence $oap_v(a_v)[\vec{a}_v] = oap_{v'}(a_{v'})[\vec{a}_{v'}]$.

**new-instance** action "$v = new\ C$" then $oap_v(a_v) = (r_o, \_)$ with $r_o$ as root parameter for the new-instance operation. Hence $loc(v, a) = oap_v(a)[\vec{a}]$.

**method entry** then $a = a_{init}$ and $v$ is a method parameter defined at $a_{init}$. Hence $loc(v, a) = oap_v(a)[\vec{a}_{init}]$.

**other** actions are not possible, as they cannot define a value for $v$.

As the trace $a_{init} \rightarrow^* a$ is finite, the evaluation of the origin access path location always stops at either $a_{init}$ or a new-instance action. □

**Observation 3.1.** *Given $oap_v(a) = (r, f_1 \rightarrow \ldots \rightarrow f_n)$. If $loc(v, a) = oap_v(a)[\vec{a}_{new}]$ where the access path starts at a new-instance trace action $a_{new}$ then the access path consists only of the root node for action $a_{new}$ : $oap_v(a) = (r_{a_{new}}, \_)$.*

*Proof Argument.* Given $n \neq 0$ then $f = f_1$ exists. Therefore a field-get action $a_f = (\overline{m}_f, \text{“}v_f = v_{base}.f\text{”}, m_f)$ with $a \rightarrow^* a_f$ exists. Also $def(v_{base}, a_f) = a_{new}$. Then field $f$ needs to be written after $a_{new}$ and before $a_f$, because field $f$ of a newly created instance needs to be initialized: $\exists a' = (\overline{m}', \text{“}v'_{base}.f = v_{val}\text{”}, m)$ where $a_{new} \rightarrow^* a' \rightarrow^* a_f$ and $def(v'_{base}, a') = a_{new}$. Due to the Definition 3.26 $oap_{v_f}(a_f) = oap_{v_{val}}(a')$. Hence if $n > 0$ and $f$ exists, then $f$ needs to be initialized after $a_{new}$. If $f$ is initialized after $a_{new}$ a different access path is computed from $v_{val}$. Thus $n > 0$ is not possible. □

Lemma 3.10 allows us to reason about the aliasing of access path in the initial context configuration. Note that $alias_C(oap_{mod}(a_1), oap_{ref}(a_2))$ includes $oap_{mod}(a_1) = oap_{ref}(a_2)$ as same access paths are aliased in any configuration.

**Lemma 3.11** (Origin Access Paths and Initial Context Configuration). *Given initial context configuration C and the set of all possible traces $\mathcal{T}_C$ with initial configuration C then*

$$\forall T \in \mathcal{T}_C : \forall a_1, a_2 \in T : a_1 \text{-}dh\!\!\rightarrow a_2 \implies alias_C(oap_{mod}(a_1), oap_{ref}(a_2))$$

*Proof.* Given $a_1 \text{-}dh\!\!\rightarrow a_2$ we know that $a_1$ is a field-set action “$v_1.f = v_2$” and $a_2$ a field-get action “$v_3 = v_4.f$”. Hence $loc(v_4, a_2) = loc(v_1, a_1)$. Due to Lemma 3.10 4 combinations are possible:

Case (1) $loc(v_4, a_2) = oap_{v_4}(a_2)[\vec{a}_{init}]$, $loc(v_1, a_1) = oap_{v_1}(a_1)[\vec{a}_{init}]$: Hence $oap_{v_4}(a_2)[\vec{a}_{init}] = oap_{v_1}(a_1)[\vec{a}_{init}]$.
Therefore $alias_C(oap_{v_4}(a_2), oap_{v_1}(a_1))$ holds.

Case (2) $loc(v_4, a_2) = oap_{v_4}(a_2)[\vec{a}_{init}]$, $loc(v_1, a_1) = oap_{v_1}(a_1)[\vec{a}_{new}]$: Due to Observation 3.1 $oap_{v_1}(a_1) = (r_a, \_)$. So $oap_{v_1}(a_1)$ refers to a location that is not existing at $a_{init}$. Hence there can't be a heap data dependence between $a_1$ and $a_2$. Thus this case is impossible to occur given $a_1 \text{-}dh\!\!\rightarrow a_2$.

Case (3) $loc(v_4, a_2) = oap_{v_4}(a_2)[\vec{a'}_{new}]$, $loc(v_1, a_1) = oap_{v_1}(a_1)[\vec{a}_{init}]$: Due to similar reasoning as in (2) this case cannot occur given $a_1 \text{-}dh\!\!\rightarrow a_2$.

Case (4) $loc(v_4, a_2) = oap_{v_4}(a_2)[\vec{a'}_{new}]$, $loc(v_1, a_1) = oap_{v_1}(a_1)[\vec{a}_{new}]$: Both origin access paths are evaluated at a trace action of a new-instance operation. Due to Observation 3.1 the origin access paths contain only a root node: $oap_{v_4}(a_2) = (r_{a'}, \_)$ and $oap_{v_1}(a_1) = (r_a, \_)$.

Given $loc((r_{a'},\_)) = loc((r_a,\_))$, $r_{a'} = r_a$ and finally $a' = a$. Obviously $alias_C(ap_1, ap_2)$ holds for any $C$ if $ap_1 = ap_2$. □

We have shown that the dynamically computed origin access paths are always aliased in the initial context configuration if a heap data dependency is present. Next we need to show that every possible origin access path is included in the set of statically computed access paths from the corresponding SDG node.

As prerequisite we show that ignoring edges in the static computation always yields weaker conditions and less access paths.

**Lemma 3.12** (Monotonicity of Static Access Paths)**.** *The static access paths $AP_n$ of a node $n$ computed from the dependence graph $SDG = (N, E)$ include all access paths $AP'_n$ computed from a subgraph $SDG' = (N', E')$ with $N' \subseteq N$ and $E' \subseteq E$.*

*We write $AP'_n \leq AP_n$ iff all access paths in $AP'_n$ are implied by access paths in $AP_n$:*

$$AP'_n \leq AP_n \equiv \forall(cond', ap') \in AP'_n : \exists(cond, ap) \in AP_n : \\ (cond \implies cond') \wedge ap = ap'$$

*Given a subgraph, all resulting access paths are implied by the access paths of the original graph.*

$$\forall n \in SDG' \subseteq SDG : AP'_n \leq AP_n$$

*Proof.* We proof that the invariant $AP'_n \leq AP_n$ holds during each step of the computation. Initially at computation start, the access path set for each node is empty. Hence $\forall n \in N : AP'_n \leq AP_n$.

Given $e = n_1 \rightarrow n_2 \in E \setminus E'$ is an edge missing in $E'$. By Algorithm 3.4 the computation of `initialAccessPaths` is unchanged due to $N = N'$ every root-parameter and new-instance node in $SDG$ is also part of $SDG'$. Hence $\forall n \in N : AP_n = AP'_n$ after computation of initial access paths. So at the start of `propagateIntraproc` $\forall n \in N : AP'_n \leq AP_n$ holds. This is unchanged up to the first iteration of the `FORALL` loop that propagates edge $e \notin E'$. Edge $e$ may be one of three types: $\xrightarrow{ps}$, $\xrightarrow{dd}$ or $\xrightarrow{dh}$.

If $n_1 \xrightarrow{ps} n_2$ then $AP_{n_2} = AP_{n_2} \cup expand(AP_{n_1}, f)$. Hence $AP_{n_2}$ before the propagation step is a subset of $AP_{n_2}$ afterwards. Given $AP'_{n_2} \leq AP_{n_2}$ before propagation of $e$, $AP'_{n_2} \leq AP_{n_2}$ holds afterwards.

In case $n_1 \xrightarrow{dd} n_2$ the access paths of $n_1$ are merged into those of $n_2$: $AP_{n_2} = AP_{n_2} \cup AP_{n_1}$. Hence $AP'_{n_2} \leq AP_{n_2}$ holds afterwards. If $e$ is a data dependency for a field base pointer, the same argument as in case $n_1 \xrightarrow{ps} n_2$ applies.

In case $n_1 \xrightarrow{dh} n_2$ the access paths of $n_1$ are merged into those of $n_2$ with the additional condition $alias(n_1, n_2)$:

$$AP_{n_2} = AP_{n_2} \cup \{(alias(n_1, n_2) \wedge cond, ap) \mid (cond, ap) \in AP_{n_1}\}$$

Given $AP'_{n_1} \leq AP_{n_1}$ and $AP'_{n_2} \leq AP_{n_2}$ before propagation. As $AP_{n_1}, AP'_{n_1}$ are unchanged $AP'_{n_1} \leq AP_{n_1}$ is trivial. As $AP_{n_2}$ contains additional access paths after propagation. A no existing path is removed, it still holds that $AP_{n_2} \leq AP'_{n_2}$.

Subsequent iterations of edge not contained in $E'$ further expand the existing access paths, while $AP'_n \leq AP_n$ always holds. □

**Theorem 3.6.** *Given program P in initial context configuration C . For any trace action $a = (\overline{m}, o, m)$ of $T \in \mathcal{T}_C$ and corresponding node $o \in SDG_P^{max}$ it holds that:*

$$\exists (cond, ap) \in AP_o : alias_C(ap, oap(a)) \wedge C \models cond$$

*Especially it holds that*

$$\exists (cond, ap) \in AP_o : ap = oap(a) \wedge C \models cond$$

*Proof Argument.* Given a specific trace $T \in \mathcal{T}_C$ and trace action $a \in T$. We write $T_{|a} = a_{init} \rightarrow \ldots \rightarrow a$ for the part of the trace up to $a$. Let $SDG' = (N', E')$ be the dependency graph that contains only nodes corresponding to trace actions up to $a$:

$$n \in N' \equiv \exists (\overline{m}', o', m') = a' \in T_{|a} : n = o'$$

$SDG'$ only includes edges that are also present in corresponding actions of the trace up to $a$.

$$\begin{aligned} n_1 \xrightarrow{dd} n_2 \in E' \equiv \quad & \exists a_1, a_2 \in T_{|a} : a_1 \xrightarrow{dd} a_2 \wedge o_1 = n_1 \wedge o_2 = n_2 \\ n_1 \xrightarrow{dh} n_2 \in E' \equiv \quad & \exists a_1, a_2 \in T_{|a} : a_1 \xrightarrow{dh} a_2 \wedge o_1 = n_1 \wedge o_2 = n_2 \end{aligned}$$

Due to Corollary 3.4 $SDG_P^{max} = (N, E)$ is a conservative approximation of all dependencies in any trace. Hence for all dependencies between two actions $(\ldots, o_1, \ldots) \rightarrow (\ldots, o_2, \ldots)$ in trace $T$ there is a matching dependency edge $o_1 \rightarrow o_2 \in E$. So $E' \subseteq E$ and thus $SDG' \subseteq SDG_P^{max}$.

Due to Lemma 3.12 any access path computed from $SDG'$ is also contained in $SDG_P^{max}$. We show that (1) $oap(a)$ is part of the static access paths of $AP_o$ computed from $SDG'$ and that (2) the access path conditions computed from the subgraph $SDG'$ are valid in $C$ .

For (1) we show that the dependencies taken to build the $oap(a)$ also trigger the static computation of a similar access path. Let $v$ be the variable defined at $a$ then —depending on the type of operation $o$— we build the origin access path by Definition 3.26 as follows:

**field-get** *"$v = v_{base}.f$"* — $oap_v(a)$ is built from one of two options: (a) In case a heap data dependency $a_{mod} \text{-}dh\text{→} a$ exists, $oap_{val}(a_{mod})$ is computed. If the heap data dependency is present in the trace then $o_{mod} \text{-}dh\text{→} o$ is also present in the $SDG'$. Hence the access paths from $o_{mod}$ are propagated to $o$.

(b) The origin access path $oap_{v_{base}}(a_{base})$ is computed from $a_{base} = def(a, v_{base})$ and extended with field $f$. As the value of $v_{base}$ is used in $a$ and defined in $a_{base}$ the data dependency $a_{base} \text{-}dd\text{→} a$ exists and hence $o_{base} \text{-}dd\text{→} o \in SDG'$. Operation $o$ is a field access, so the incoming data dependency of the base pointer connects to the *base* node (see §2.4.3). Hence the *base* node contains all access paths from $o_{base}$. The *base* node connects to the *field* node of $o$ with a parameter structure edge. Due to Algorithm 3.4 the *field* node contains all access paths of the *base* node extended with field $f$.

**field-set** *"$v_{base}.f = v_{val}$"* As a field-set action does not define a variable, the origin access path computation enters it only via a heap-data dependency of a previously visited field-get operation. Hence $oap_{v_{val}}(a)$ is built. In the static computation with the fine-grained field access model (see §2.4.3) the heap-data dependency leaves the operation at the *field* node. The main node is connected to the *field* node with a data dependency. Hence all access paths computed for the main node are also part of the *field* node paths. The main node contains the access paths computed from the value read $v_{val}$, as it is connected to the definition of $v_{val}$ via an incoming data dependency. In a similar fashion $oap_{v_{val}}(a)$ is computed from $oap_{v_{val}}(a_{val})$ with $a_{val} = def(a, v_{val})$. Hence $a_{val} \text{-}dd\text{→} a$ exists and therefore $o_{val} \text{-}dd\text{→} o \in SDG'$.

**assignment** *"$v = v'$"* then $oap_v(a)$ is built from $oap_{v'}(a')$ with $a' = def(a, v')$. Hence $a' \xrightarrow{dd} a$ and therefore $o' \xrightarrow{dd} o \in SDG'$. The static access path is propagated accordingly from $o'$ to $o$.

**new-instance** *"$v = new\ C$"* then $oap_v(a)$ is $(r_o, \_)$. In the static computation a similar access path is added initially in `initialAccessPaths`.

**method entry** If the method entry is reached, then $oap_v(a)$ is $(r_v, \_)$ and $v$ refers to a root parameter. In the static computation all root parameters are initialized with a similar access path.

For (2) we show that the conditions are valid during each step of the computation. During computation of access path conditions in Algorithm 3.4 all conditions are initialized with *true*. Additional conditions are build traversing heap data dependence edges. The first iteration of the `FORALL`-loop that propagates a $n_1 \xrightarrow{dh} n_2$ edge builds new conditions by extending the existing conditions of the access paths of the source node $n_1$:

$$cond_2 = cond_1 \wedge alias(n_1, n_2)$$

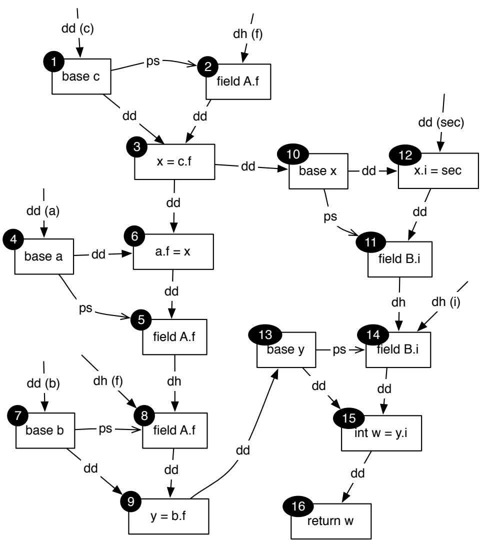
Due to initialization $cond_1 = true$. Because $SDG'$ only contains data dependence edges also present in trace $T$, we know $\exists a_1 \xrightarrow{dh} a_2 \in T$ where $o_1 = n_1$ and $o_2 = n_2$. Hence $(\ldots, oap_{mod}(o_1)) \in AP_{n_1}$ and $(\ldots, oap_{ref}(o_2)) \in AP_{n_2}$. We also know by Lemma 3.11 that $alias_C(oap_{mod}(a_1), oap_{ref}(a_2))$. Thus $C \models alias(n_1, n_2)$ and $C \models cond_2$.

In the following iterations of heap data dependence edges, before propagation $C \models cond_1$ and hence by induction $C \models alias(n_1, n_2)$ and therefore $C \models cond_2$ for each propagation step. □

Theorem 3.6 guarantees that —given the initial context $C$ — it is safe to remove all computed access paths from the nodes in $SDG_P^{max}$ whose condition is not valid in $C$ . So we can safely remove heap data dependencies from the maximal SDG where the access paths of source and target node no longer share a common element. The resulting graph $SDG_P^C$ remains a conservative approximation of all possible information flow in context $C$ .

```
 1 class A {
 2   B f;
 3 }
 4
 5 class B {
 6   int i;
 7 }
 8
 9 //@ifc ? => sec -!-> \result
10 int compute(A a, A b,
11             A c, int sec)
12 {
13   B x = c.f;
14   a.f = x;
15   B y = b.f;
16   x.i = sec;
17   int w = y.i;
18
19   return w;
20 }
```

**(a)** Sourcecode



**(b)** Fine-grained data dependence graph

**Figure 3.20:** Example for intraprocedural access path propagation. Sourcecode (3.20a) and the relevant data dependence part of the SDG (3.20b). Control dependencies and formal parameters are omitted for brevity.

## Example: Intraprocedural access paths

In the previous sections we introduced the algorithms for intra- and interprocedural access path computation and showed how access paths help to approximate conditional dependencies for the modular SDG. This section contains a step-by-step example leading from the intraprocedural propagation phase to the resulting conditional dependencies. Figure 3.20 shows a method (Figure 3.20a) and the relevant parts of its matching PDG (Figure 3.20b). The method is annotated with a FlowLess specification in l. 9 that forbids information flow from parameter sec to the method result. The context in which the method is called is unknown, so we infer in which configurations the flow restriction is met. Therefore we compute a modular SDG with the access path approach. The PDG part in Figure 3.20b shows only the edges relevant to the intraprocedural access path propagation algorithm: *dd*-edges for direct data dependencies, *dh*-edges

| Node | Access Paths |
|---|---|
| 1 | $(\top, (c, \_))_1$ |
| 2 | $(\top, (c, f))_{2b}$ |
| 3 | $(\top, (c, f))_{2a}$ |
| 4 | $(\top, (a, \_))_1$ |
| 5 | $(\top, (a, f))_{2b}$ |
| 6 | $(\top, (a, f))_{2a}$, $(\top, (c, f))_{2a}$ |
| 7 | $(\top, (b, \_))_1$ |
| 8 | $(\top, (b, f))_{2b}$, $(alias(5, 8), (a, f))_{2c}$ |
| 9 | $(\top, (b, f))_{2a}$, $(alias(5, 8), (a, f))_{2b}$ |
| 10 | $(\top, (c, f))_{2a}$ |
| 11 | $(\top, (c, f.i))_{2a}$ |
| 12 | $(\top, (c, f.i))_{2a}$ |
| 13 | $(\top, (b, f))_{2a}$, $(alias(5, 8), (a, f))_{2b}$ |
| 14 | $(\top, (b, f.i))_{2b}$, $(alias(11, 14), (c, f.i))_{2c}$, $(alias(5, 8), (a, f.i))_{2a}$ |
| 15 | $(\top, (b, f.i))_{2a}$, $(alias(11, 14), (c, f.i))_{2b}$, $(alias(5, 8), (a, f.i))_{2b}$ |
| 16 | $(\top, (b, f.i))_{2a}$, $(alias(11, 14), (c, f.i))_{2b}$, $(alias(5, 8), (a, f.i))_{2b}$ |

**Table 3.5:** Access paths of the example from Figure 3.20 after the intraprocedural propagation described in §3.4.3. The subscript of each access path element shows the propagation step in which it was added.

for heap data dependencies and *ps*-edges for the parameter structure of field accesses. The graph contains fine-grained field access instructions (§2.4.3) as they are necessary for the propagation Algorithm 3.4.

Table 3.5 shows the access paths computed after the intraprocedural propagation. The access paths are annotated with the subscript of the specific step in Algorithm 3.4 that computed them. The flow restriction forbids any flow from sec to the result. A closer look at Figure 3.20b reveals that illegal flow can only occur if there is a path from $12 \rightarrow^* 16$. Specifically the heap data dependency between 11 and 14 needs to be present. Therefore we compute the context condition that guarantees the absence of $11 \xrightarrow{dh} 14$ by resolving its alias condition:

$$\begin{aligned} alias(11, 14) &= \top \wedge \top \wedge alias((c, f, i), (b, f.i)) \\ &\bigvee \top \wedge \bot \wedge alias((c, f.i), (c, f.i)) \\ &\bigvee \top \wedge alias(5, 8) \wedge alias((c, f.i), (a, f.i)) \end{aligned}$$

As $alias(5,8)$ is referenced we continue with

$$\begin{aligned} alias(5,8) &= \top \wedge \top \wedge alias((a,f),(b,f)) \\ &\bigvee \top \wedge \bot \wedge alias((a,f),(a,f)) \\ &= alias((a,f),(b,f)) \end{aligned}$$

Finally we get

$$\begin{aligned} alias(11,14) &= alias((c,f,i),(b,f.i)) \\ &\quad \vee (alias((a,f),(b,f)) \wedge alias((c,f.i),(a,f.i))) \end{aligned}$$

So a path from `sec` to the method result ($12 \rightarrow^* 16$) is only possible if $alias(11,14)$ holds. Therefore method `compute` is safe as long as its calling context satisfies

$$\neg alias(c.f.i, b.f.i) \wedge (\neg alias(a.f, b.f) \vee \neg alias(c.f.i, a.f.i))$$

We have shown how access paths help to compute if a heap data dependency is present in a certain context configuration. Given a specific context we can enable the matching dependencies in the modular SDG through these alias conditions. However afterwards the summary edges still need to be adjusted. In the following section we discuss how the information in the modular SDG can be used to precompute some parts of the summary computation in order to safe a significant amount of time compared a full-blown recomputation.

### 3.4.4 Precomputation of summary information

The summary edge computation originally introduced in Chapter 2 section §2.5.4 computes summary edges between actual-in and -out nodes for each call site in the SDG. It summarizes the information flow that may occur through each call and enables unlimited call site context-sensitive slicing in linear time $O(\#edges)$ with the two-phase slicing algorithm.

Summary edges in the modular SDG depend on the context the component is called in. Thus we either need to recompute them once a standard SDG is extracted for a given context or we compute a conditional variant of summary edges that can be triggered similar to conditional data dependencies. However we noticed that —due to the nature of summary edges— precomputing a conditional variant in the spirit of access paths is rather complex. The conditions have to include a disjunction over all paths that lead from a formal-in to a formal-out node computed for each pair of formal-ins and -outs. This would add another layer of complexity to the already time consuming summary computation. In contrast to the standard computation, the conditional computation can no longer abort once a single path between formal-in and -out has been found. It needs to find all paths in order to enumerate all potential conditions —which further hinders computation. Therefore we focus on improving the recomputation of summary edges for a given context rather than computing conditional summary edges.

The recomputation speed of summary edges can be improved, because we know the minimal and maximal SDGs that can result from a modular SDG. Thus we can precompute a minimal set of summary edges —that occur in every configuration— and a maximal set that helps us to decide which edges are even possible. We extend the summary edge algorithm from §2.5.4 with several options that allow us to include precomputed information from the maximal and minimal SDGs. We identified 3 optimizations to the standard algorithm.

**minimal edges** Using the minimal $SDG_{C_{\text{min}}}$ we compute the minimal set of summary edges that will always be present in each variant of the modular SDG. We use the minimal set instead of an empty set of summary edges as initial configuration.

**influenced methods** We keep track of the added conditional edges when we extract a SDG for a given context. We detect all methods that contain activated conditional dependency edges as well as all methods that can reach them through calls. The summary edge computation only has to propagate additional summary edges along those methods. It can ignore all other parts of the SDG that have not changed.

**shortcut for maximal dependencies** We use the maximal set of dependency edges to detect which parts of the SDG will no longer change and can be excluded from the propagation. We also detect sooner when to abort the fixed-point-based algorithm.

Algorithm 3.6 shows the enhanced version of Algorithm 2.1 that contains above optimizations. It takes the additional arguments *initialSum, relevantPDGs* and *out2in*:

*initialSum* contains the set of minimal summary edges. We add these edges to the SDG at the start of the computation and also include them in the result set.

*relevantPDGs* is a set of all PDGs inside the given SDG that may change during the summary computation — thus PDGs not contained in this set will not change and can be ignored.

*out2in* is a mapping of actual-out and formal-out nodes to the corresponding actual-in or formal-in nodes of the same call or PDG. For each -out node that is already fully connected we directly store the -in nodes it depends on. It allows us to skip the search through the call or PDG and propagate directly all dependencies that can occur. This information is retrieved by checking the differences between $SDG_{C_{\min}}$ and $SDG_{C_{\max}}$: All -out nodes that depend on the same -in nodes in both SDGs can be included in this mapping.

**Algorithm 3.6** (Summary edge computation for modular SDGs)**.**

```
PROCEDURE ComputeSummaryEdges($sdg$, $initialSum$, $relevantPDGs$, $out2in$)
INPUT:
  A system dependence graph $sdg$
  Initial minimal set of summary edges $initialSum$
  Set of PDGs where additional summary changes may occur $relevantPDGs$
  Map of actual and formal-out to -in nodes already fully connected $out2in$
OUTPUT:
  Set of summary edges $summary$
BEGIN
  FOREACH (Edge $e \in initialSum$) DO // enhanced
    Add $e$ to $sdg$
  DONE
  Set<Edge> PathEdge = $\emptyset$, $summary$ = $initialSum$, WorkList = $\emptyset$
  Map<Node, Set<Edge>> fragmentPath = new empty map
  FOREACH (formal-out node $w \in sdg$) DO
    IF ($w$ is part of a PDG in $relevantPDGs$) THEN // enhanced
      PathEdge = PathEdge $\cup \{w \to w\}$
      WorkList = WorkList $\cup \{w \to w\}$
    FI
  DONE

  WHILE (WorkList $\neq \emptyset$) DO
    WorkList = WorkList $\setminus \{v \to w\}$
    IF ($v$ is actual-out) THEN
      IF ($v \in out2in$) THEN // enhanced
```

```
        FOREACH ($a_{in} \in out2in(v)$) DO
          Propagate($a_{in} \rightarrow v$, $relevantPDGs$)
        DONE
      ELSE
        FOREACH ($x$ such that $x \rightarrow v \in summary \vee x \xrightarrow{cd} v \in sdg$) DO
          Propagate($x \rightarrow w$, $relevantPDGs$)
        DONE
      FI
    ELSE IF ($v$ is formal-in) THEN
      FOREACH (call node $c$ with $\exists c \xrightarrow{call} Entry(w) \in sdg$) DO
        $x$ = matching actual-in for $v$ at call $c$
        $y$ = matching actual-out for $w$ at call $c$
        $summary = summary \cup \{x \rightarrow y\}$
        FOREACH ($a$ such that $y \rightarrow a \in$ fragmentPath($y$)) DO
          Propagate($x \rightarrow a$, $relevantPDGs$)
        DONE
      DONE
    ELSE
      FOREACH ($x$ with $x \xrightarrow{dd} v, x \xrightarrow{dh} v$ or $x \xrightarrow{cd} v \in sdg$) DO
        IF (NOT ($x \xrightarrow{cd} v$ and $x$ and $v$ both parameter nodes)) THEN
          Propagate($x \rightarrow w$, $relevantPDGs$)
        FI
      DONE
    FI
  DONE
  RETURN $summary$
END

PROCEDURE Propagate($e$, $relevantPDGs$)
INPUT:
  A SDG edge $e = v \rightarrow w$
  Set of PDGs where additional summary changes may occur $relevantPDGs$
OUTPUT:
  Potentially added edge $e$ to WorkList, PathEdge and fragmentPath($w$)
BEGIN
  IF ($v$ and $w$ not part of a PDG in $relevantPDGs$) THEN RETURN FI //   enhanced
  IF ($e \notin$ PathEdge) THEN
    PathEdge = PathEdge $\cup \{e\}$
    WorkList = WorkList $\cup \{e\}$
    IF ($v$ is actual-out) THEN
      fragmentPath($w$) = fragmentPath($w$) $\cup \{e\}$
    FI
  FI
END
```

| Program | original | +minimal | +influenced | +shortcut |
|---|---|---|---|---|
| tests.Mantel00Page10 | 442 | 287 | 67 | 56 |
| conc.ds.DiskSchedulerDriver | 407 | 251 | 66 | 55 |
| tests.ThreadSpawning | 371 | 222 | 54 | 44 |
| tests.Synchronization | 353 | 201 | 60 | 49 |
| tests.ThreadJoining | 333 | 187 | 43 | 37 |
| conc.sq.SharedQueue | 315 | 176 | 50 | 41 |
| conc.daisy.DaisyTest | 339 | 183 | 36 | 30 |
| tests.HammerDistributed | 401 | 209 | 52 | 44 |
| tests.IndirectRecursiveThreads | 460 | 244 | 62 | 50 |
| conc.bb.ProducerConsumer | 459 | 248 | 69 | 59 |
| conc.cliser.kk.Main | 476 | 258 | 66 | 59 |
| conc.kn.Knapsack5 | 482 | 269 | 70 | 59 |
| tests.RecursiveThread | 487 | 265 | 39 | 34 |
| conc.TimeTravel | 553 | 295 | 81 | 70 |
| conc.lg.LaplaceGrid | 584 | 311 | 75 | 64 |
| tests.ProbPasswordFile | 565 | 311 | 76 | 69 |
| tests.ConcPasswordFile | 564 | 310 | 73 | 65 |
| tests.Hammer | 550 | 295 | 73 | 63 |
| conc.dp.DiningPhilosophers | 618 | 335 | 70 | 56 |
| conc.ac.AlarmClock | 555 | 293 | 100 | 85 |
| conc.ac.AlarmClockSync | 616 | 314 | 71 | 73 |
| | 9930 | 5464 | 1353 | 1162 |
| speedup from original | 0% | 44,70% | 86,27% | 88,27% |

**Table 3.6:** Runtime evaluation of enhanced summary edge computation of Algorithm 3.6. Times are shown in milliseconds.

### 3.4.5 Evaluation

**Enhanced summary computation** We evaluated the impact of the optimizations to summary computation presented in §3.4.4 on 21 different programs and found that the average runtime improvement is about 88%. Table 3.6 shows the result of the evaluation. The enhanced algorithm is 44% faster with the minimal summary edges added alone. In combination with the detection of potentially influenced methods the runtime is reduced by 86%. Finally the last 2% are gained through the shortcuts for the fully connected parameter-out nodes. The impact of the shortcuts is not large, because part of their effect is already included in the minimal summary edges: The shortcuts already exist in the SDG in form of summary edges, only the unnecessary traversal of non-summary edges leading to those nodes is prohibited by this optimization.

In conclusion, the enhanced summary computation significantly reduces the time needed to extract a context specific SDG from a given modular SDG. Due to small speed-up from the shortcut optimization the additional memory required may not be worth it, but the other two optimizations work well in practice.

**Modular SDGs with access paths** We randomly selected 151 different method parts —with less then 100 methods in each part— from the set of example programs already used in §2.6.5 Table 2.3. The resulting SDGs varied in size between 10 and 5000 nodes with an average of about 210 nodes. We computed the modular SDG using access paths as well as the standard SDG in minimal context configuration. Table 3.7 shows the result of the evaluation. We accumulated the results of the analysis according to the size of the dependency graphs in 5 groups: Starting from very small method parts with 10-50 nodes going to 51-100, 101-200, 201-500 and graphs with 501-5000 nodes. We were not able to compute a modular SDG for graphs larger then 5000 nodes due to scalability issues. The first column "*modular vs. std*" shows the relative execution time of the modular SDG computation with access paths, compared to the standard SDG computation in the minimal context configuration with no aliases. We see that for small graphs the access path computation is about 2.5× slower then the standard SDG computation. As expected it gets worse with increasing graph size. For the group of largest graphs we were able to compute, access paths are on average 60.5× slower. The second column "*std vs. adjust*" shows the relative execution time of standard SDG computation from scratch compared to adjusting a precomputed modular SDG to a given context configuration. For small graphs adjusting the modular SDG to a given context yields a significant speedup 313× then a full blown recomputation. However the speedup shrinks with increased graph sizes down to only a 20× speedup for the group of graphs between 500 and 5000 nodes. The last column "*% alias edges*" shows the average percentage of alias edges in an SDG of the corresponding group. We notice that with increased SDG size the amount of contained alias edges increases disproportionately —in the group of the biggest graphs on average 64% of all contained edges are alias edges. We expect that the percentage increases even further with larger graphs.

In conclusion we see that —once the modular SDG is computed— a speed-up is present when we extract and adjust a SDG for a specific

| SDG size #nodes | modular vs. std | std vs. adjust | % alias edges |
|---|---:|---:|---:|
| 10-50 | 2.54 × | 313.20 × | 28% |
| 51-100 | 4.54 × | 93.46 × | 34% |
| 101-200 | 5.28 × | 88.46 × | 28% |
| 201-500 | 6.29 × | 35.01 × | 55% |
| 500-5000 | 60.54 × | 20.42 × | 64% |

**Table 3.7:** Modular SDG computation with access paths.

context. However, the initial computation of the modular SDG is quite complex and finishes only for smaller method parts. In addition, the speed-up shrinks for larger modular SDGs that contain many alias edges, because the adjustment to a specific context needs to compute for each edge if it is valid in the context. The complicated adjustment operation currently limits the application of the modular SDG to smaller method parts, for larger parts we suggest to use the standard SDG computation in combination with a context specific stub, as presented in §3.3.2.

*There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.*

C.A.R. Hoare

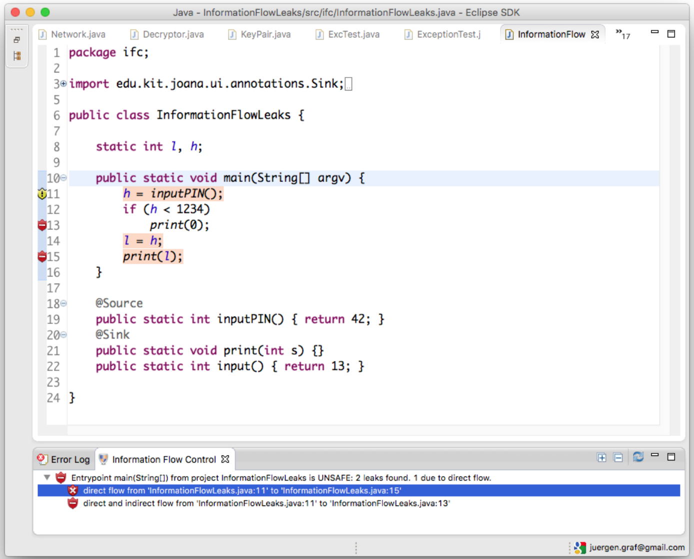
# 4 Applications of Information Flow Control

This chapter presents how our information flow control tool Joana is use and showcases several successful applications.

## 4.1 Information flow control in practice with Joana

Joana contains a plug-in for the Eclipse platform that integrates into the user interface of the development environment. This plug-in can be used to trigger an IFC analysis of the current project. It displays the results in a separate view and marks relevant or critical parts of the code directly in the editor. Figure 4.1 shows the result of the analysis of the initial example from §1.1. Both information leaks in the example are identified correctly and shown in the GUI. The *"Information Flow Control"* view at the bottom contains the description of the detected leaks and allows the user to highlight the statements relevant to the selected leak — currently the statements relevant to the direct leak from line 11 to line 15 are highlighted.

Before the analysis can be triggered, the user has to provide annotations for the secret input and public output operations. Joanas annotations build upon the annotation mechanism integrated into Java > 1.6. In the example shown in Figure 4.1 this was done by annotating method `inputPIN()` as a secret `@Source` and method `print()` as public

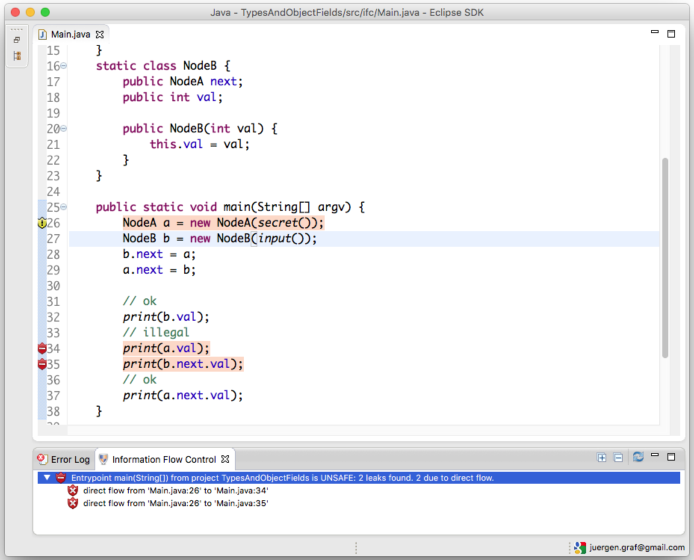


```
 1 package ifc;
 2
 3 import edu.kit.joana.ui.annotations.Sink;
 5
 6 public class InformationFlowLeaks {
 7
 8     static int l, h;
 9
10     public static void main(String[] argv) {
11         h = inputPIN();
12         if (h < 1234)
13             print(0);
14         l = h;
15         print(l);
16     }
17
18     @Source
19     public static int inputPIN() { return 42; }
20     @Sink
21     public static void print(int s) {}
22     public static int input() { return 13; }
23
24 }
```

```
Entrypoint main(String[]) from project InformationFlowLeaks is UNSAFE: 2 leaks found. 1 due to direct flow.
    direct flow from 'InformationFlowLeaks.java:11' to 'InformationFlowLeaks.java:15'
    direct and indirect flow from 'InformationFlowLeaks.java:11' to 'InformationFlowLeaks.java:13'
```

**Figure 4.1:** Joana GUI detecting direct and indirect information leaks (§1.1).

`@Sink`. Then the analysis detects all information leaks and automatically categorizes them with the algorithm presented in Algorithm 4.1. This algorithm applies a series of analysis runs with different configurations to detect the cause of information leaks: Leaks detected when all control dependencies are ignored are categorized as *direct leaks*. Then control dependencies are turned on while exceptions are ignored. Any additionally detected leaks must therefore be caused by indirect flow and are marked as *indirect leaks*. In a third step exceptions are no longer ignored and the newly discovered leaks therefore must be caused by *indirect flow through exceptions*. The whole process can be repeated with increased points-to analysis precision until either all leaks can be discarded as false alarms or the most precise points-to option has been applied.



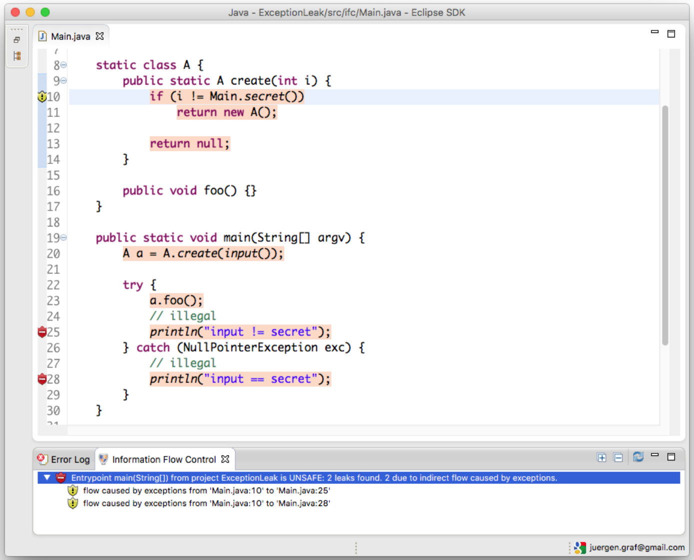
```
15      }
16      static class NodeB {
17          public NodeA next;
18          public int val;
19
20          public NodeB(int val) {
21              this.val = val;
22          }
23      }
24
25      public static void main(String[] argv) {
26          NodeA a = new NodeA(secret());
27          NodeB b = new NodeB(input());
28          b.next = a;
29          a.next = b;
30
31          // ok
32          print(b.val);
33          // illegal
34          print(a.val);
35          print(b.next.val);
36          // ok
37          print(a.next.val);
38      }
```

```
Entrypoint main(String[]) from project TypesAndObjectFields is UNSAFE: 2 leaks found. 2 due to direct flow.
    direct flow from 'Main.java:26' to 'Main.java:34'
    direct flow from 'Main.java:26' to 'Main.java:35'
```

**Figure 4.2:** Joana GUI detecting illegal flow through object fields (§2.1.5).

## **Algorithm 4.1** (Automatic information leak categorization).

```
FOR ($pts$ ∈ [type-based, instance-based, object-sensitive]) DO
  set points-to precision to $pts$
  $result_{dd}$ = compute ifc with data dependencies only
  FORALL ($vio$ ∈ $result_{dd}$) DO
    mark $vio$ as caused by "direct flow"
  DONE
  $result_{no-exc}$ = compute ifc with data + control dependencies ignore exceptions
  FORALL ($vio$ ∈ $result_{no-exc} \setminus result_{dd}$) DO
    mark $vio$ as caused by "indirect flow"
  DONE
  $result_{full}$ = compute ifc with data + control dependencies and exceptions
  FORALL ($vio$ ∈ $result_{full} \setminus result_{no-exc}$) DO
    mark $vio$ as caused by "indirect flow through exceptions"
  DONE

  IF ($result_{full}$ is empty) THEN stop analysis
DONE
```



```
 8     static class A {
 9         public static A create(int i) {
10             if (i != Main.secret())
11                 return new A();
12 
13             return null;
14         }
15 
16         public void foo() {}
17     }
18 
19     public static void main(String[] argv) {
20         A a = A.create(input());
21 
22         try {
23             a.foo();
24             // illegal
25             println("input != secret");
26         } catch (NullPointerException exc) {
27             // illegal
28             println("input == secret");
29         }
30     }
```

```
Entrypoint main(String[]) from project ExceptionLeak is UNSAFE: 2 leaks found. 2 due to indirect flow caused by exceptions.
    flow caused by exceptions from 'Main.java:10' to 'Main.java:25'
    flow caused by exceptions from 'Main.java:10' to 'Main.java:28'
```

**Figure 4.3:** Joana GUI detecting an information leak caused by an exception (§2.1.8).

Figure 4.2 shows another example of the Joana Eclipse plug-in in action. Similar to the previous example only methods `print` and `inputPIN` are annotated. This example is taken from §2.1.8 and illustrates the object-, type- and field-sensitivity of our analysis. Joana correctly identifies two information leaks caused by direct information flow. It is able to detect that the first `print()` statement in line 32 as well as the last print statement in line 37 access an object-field that contains no secret information. Therefore it needs to distinguish the different objects created and the fields within those objects. For example `b.val` does not contain secret information while `b.next.val` does.

Finally the third example in Figure 4.3 shows how Joana can automatically detect information flow caused by exceptions. We analyze the example from §2.1.8 that contains two information leaks caused by a

`NullPointerException` triggered depending on the secret value returned by `Main.secret()`. Joana alerts the user that the secret value used in the conditional clause in line 10 can cause illegal information flow to lines 25 and 28. It also identifies the information flow caused by exceptions. As —depending on the secret value— `A.create()` can return **`null`**, it may cause the subsequent call to `a.foo()` in line 23 to throw an exception. Hence either `println` in line 25 or 28 is executed.

All these examples can be edited and reanalyzed with minimal effort. In previous versions of Joana security annotations had to be placed at manually selected dependence graph nodes. So after a minor change to the source code, or even a change in the analysis options, the annotations had to be redone for the specific graph. The new annotations at source code level are more robust and enable us to combine the results of multiple runs —as in Algorithm 4.1. They are also more user friendly, as they are directly visible in the source code and can be easily modified by a user without detailed knowledge of the underlying system dependence graph. In our experience the new annotation mechanism was an important improvement that raised the popularity of Joana as an information flow control tool amongst other research groups.

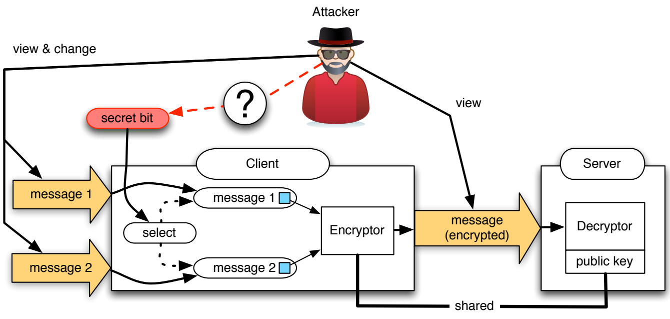


**Figure 4.4:** A simple setup for encrypted communication between a client and server.

## 4.2 Proving cryptographic indistinguishabiliy with Joana

In this case study we show how a noninterference check with Joana can be used to proof *cryptographic indistinguishability* properties. This result is joint work with Ralf Küsters and Tomasz Truderung from the University of Trier [76,75]. We introduce a new approach called *ideal functionality* that allows us to replace real cryptographic operations with an idealized version that is easier to analyze, while the analysis result remains valid in the original setting. We apply this new approach to proof cryptographic indistinguishability for a small case study. In order to achieve this result we have to proof noninterference for an open system with an unknown initial state. Therefore we additionally show how Joana can be used to argue about a whole family of programs that vary in initialization. We provide a list of rules that restrict the allowed variations and proof that when these restrictions are met, the information flow properties detected by Joana are the same for each variant. Thus it suffices to analyze only a single member of the program family.

Figure 4.4 contains an overview of our case study. It consists of a client and a server that exchange an encrypted message through a network connection. The attacker provides two messages to the client. The client then chooses —depending on a secret bit— one of the two messages. Then it uses public-key encryption and sends the encrypted message through an untrusted network to the server. The server receives the message and decrypts it afterwards. The attacker is allowed to observe the network and still should not be able to decide which of the two messages were sent. This property is an instance of *cryptographic indistinguishablility*, a fundamental security property for encryption that is relevant in many applications such as secure message transmission, key exchange and e-voting [75]. The setup of the case study is quite simple from a cryptographic point of view, yet the actual implementation could contain bugs that leak additional information. Therefore we use Joana to verify that the implementation does not contain any intentional or unintentional information leaks.

The given example is clearly not noninterferent, because the value of the encrypted message that is transferred from the client to the server depends on the secret bit. Thus a standard noninterference check with Joana will rightfully discover a security violation. However, we want to show that the program fulfills the weaker security property of cryptographic indistinguishability. So while the original program is not noninterferent it may very well be cryptographically indistinguishable: Secret information may be leaked, but the attacker is not able to reveal it in polynomially bounded time. We are now going to describe how we exchange parts of the functionality in the original program $P_{real}$ with ideal versions. Then we use the program with ideal functionality $P_{ideal}$ to proof basic noninterference which automatically[21] implies cryptographic indistinguishability for $P_{real}$.

$$P_{ideal} \text{ is noninterferent } \implies P_{real} \text{ is cryptographic indistinguishable}$$

This implication has been fully formalized and proven for a large subset of the Java language called Jinja+. The details are out of the scope for this work. They can be found in the corresponding publication [75] and in more detail in its accompanying tech report [76].

We replace the *Encryptor* and *Decryptor* parts of the original program with their ideal counterparts. The ideal version of the *Encryptor* does not encrypt the secret messages —instead it generates a random string that is sent over the network. On the server side the ideal version of the *Decryptor* then replaces the random string with the secret message. This setup ensures that —as long as the program does not contain any other

[21]Through a proof provided by Küsters and Truderung

security violations— the transferred message does not contain any secret information. Thus a standard noninterference check should succeed. At the implementation level we need of course some way to transfer the actual message from the client to the server. This is realized through a map that is shared between the *Encryptor* and *Decryptor*. This map stores the mapping between the random string and the secret message. This way the *Decryptor* can lookup the actual message after it received the random string. So the functionality of the ideal version of the program remains intact, while the message transferred over the network does not contain secret information.

The source code of the ideal version is available in Appendix A. The encryption and decryption functionalities can be found in the `Encryptor` and `Decryptor` classes, respectively. The `main` method in class `Setup` simulates the system. It initially creates a server and then enters a loop where it creates a new client if the attacker provides two new messages to encrypt. It selects one of the two messages depending on the secret value in static variable `Setup.secret` and creates a client that encrypts the message and sends it to the server where it gets decrypted. The input and output of the program is controlled by the class `Environment`. Every operation that reveals information to a public visible channel –e.g. the network– is modeled to change the value of the static field `Environment.result`. To proof noninterference we essentially have to show that the value of `Setup.secret` is never going to influence the value of `Environment.result`.

For the verification process with Joana we only need to annotate two points in the program: We mark the initialization of variable `Setup.secret` as *high* input and modifications to the variable `result` in class `Environment` as *low* output. Using the standard security lattice $low \leq high$ for noninterference, Joana then builds the PDG model of the program and propagates the security labels for all other statements automatically. If a situation occurs where *high* input may flow to a *low* output channel, Joana issues a security warning and computes the set of possible paths in the program along which the information may have been leaked. In the ideal version of the current example Joana did not detect any leaks, therefore we can conclude that $P_{ideal}$ is noninterferent. The analysis took about 11 seconds on a standard PC (Core i5 2.3GHz, 8GB RAM) for the example program with a size of 387LoC — not counting the size of the parts of the JRE 1.4 library that are included automatically. The computation of the PDG model —including interprocedural propagation and summary edge computation— was finished in 10 seconds and only 1 second was needed to detect the absence of illegal information flow in the PDG. The example specific code that sets up the automatic noninterference analysis only takes about 80LoC and can be found in §4.2.2. It is used to configure Joana specifically for this setting.

**Analysis features needed to show noninterference** Even this small client-server example contains some challenges for static analysis tools. We are going to discuss at which points in the example the various features and precision enhancements of Joana were needed to show its noninterference.

**Instance-based** The analysis needs to distinguish different instances of a class. In contrast to very basic type-based security systems, Joana does not expect each instance of the same type to hold the same level of secret or public information. It is a crucial option in this example as `byte`-arrays are used to hold the secret message as well as the encrypted public cipher that is transferred over the network. A simple type-based analysis would assume that the contents of all `byte`-arrays are secret information and issue a false alarm. This precision feature is in general very important as normal programs often reuse the same data-types and -structures –like lists, maps or arrays– to store all kind of different information.

**Field-sensitivity** We need to distinguish different fields inside a single object. §2.2.3 provides more details on field-sensitive analysis. In our example, the class `MessagePair` contains the two fields `ciphertext` and `plaintext`, where only the latter holds secret information. An analysis without field-sensitivity would merge the security label of both fields and thus would assume that `ciphertext` also contains secret information which subsequently leads to a false alarm.

**Context-sensitivity** We also need to distinguish the effects of a method call depending on the point in the program (call site) it has been called from. §2.2.1 provides more details on context-sensitive analysis. In our example the method `copyOf` in class `MessageTools` is used to copy the contents of the parameter to a new array instance. This method is used to copy secret as well as public information. For example, l. 34 in class `Decryptor` copies the contents of the secret plaintext array, while l. 32 in class `Encryptor` copies the public contents of the random cipher array. So depending on the context `copyOf` is called in, it returns either public or secret information. A context-insensitive analysis would assume that `copyOf` always returns secret information and thus the result of the encryption would also be secret — which would lead to a false alarm.

**Parameterized initialization** The program under analysis uses parameterized initialization to simulate the different input it may receive. The input is modeled as a list of integer values in the class `Environment`. Its values are set in method `initialValue` in l. 34. Our analysis has to respect that the analysis result needs to hold independently of the values initially created. So changes to l. 34 should not impact the noninterference property of the rest of the program. This cannot be guaranteed in general, as a single line may contain arbitrary large portions of code that may lead to side-effects in other parts of the program. However if we carefully restrict the allowed changes to this line, we are able to proof that the noninterference property is unchanged, while the initial values can be set arbitrarily. The crafted restrictions as well as the sketch of the proof —changes that adhere to these restrictions do not change our information flow result— is shown in the following section.

### 4.2.1 Parameterized initialization and IFC

This section contains the modified proof sketch from the technical report [76]. It has been adapted to match the common notation of this work. Figure 4.5 and 4.6 contain the model for unknown input in the previous client-server example together with a simple main method. As before the singly linked list of input values is initialized in method `initialValue`. The code shows two variants $V1$, $V2$ of the potentially infinite number of initializations in l. 13 and l. 12. We are now going to show how Joana internally represents this program in PDG form and argue that variants of the initialization do not affect the noninterference property of the whole program. We briefly inspect the corresponding PDG in Figure 4.7. As usual it contains nodes for

```
class Node {
  int value;
  Node next;
  Node(int v, Node n) {
    value = v;  next = n;
  }
}
private static Node list = null;
private static boolean listInitialized = false;
private static Node initialValue() {
  /* WILL BE MODIFIED */
  return new Node(1, new Node(2, null)); // V2
//return new Node(1, null);             // V1
}
static public int untrustedInput() {
  if (!listInitialized) {
    list = initialValue();
    listInitialized = true;
  }
  if (list==null) return 0;
  int tmp = list.value;
  list = list.next;
  return tmp;
}
```

**Figure 4.5:** Code that models unknown input in form of a linked list similar to the client-server example in Appendix A.

```
public class P {
  public static void main(String[] argv) {
    int untrusted1 = untrustedInput();
    int secret = 42;
    int unstrusted2 = untrustedInput();
    low = unstrusted1;
    secret += 21;
    low += unstrusted2;
  }
}
```

**Figure 4.6:** An example of a program without illegal information flow using untrusted input from Figure 4.5.

each statement of the program. For brevity we summarize input and output parameters for each call and method entry in a single *IN* and *OUT* field. Nodes of the same method are enclosed in an outlined rectangle. So edges between nodes that cross those rectangles represent interprocedural dependencies between methods while edges inside are purely intraprocedural dependencies. In the following we are going to

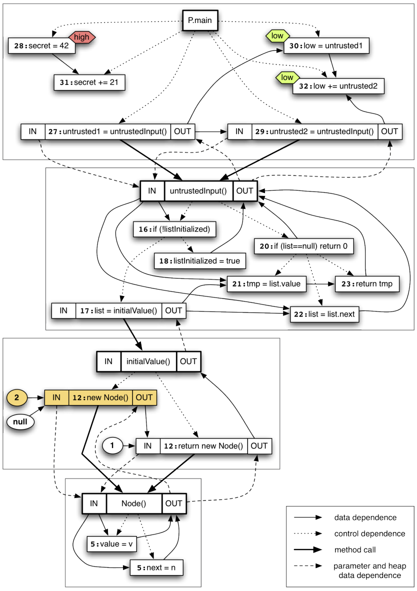


**Figure 4.7:** A PDG of the example program from Figure 4.5 and 4.6. The colored nodes are the additional nodes that occur if version $V2$ of `initialValue` is chosen instead of $V1$.

show that variants of method `initialValue`, as long as they obey certain restrictions, will only lead to additional intraprocedural dependencies that do not interfere with the rest of the program.

**Problem statement** In general we consider programs $P[\vec{o}]$ that consist of two parts: $P_{main}$ and $P_{\vec{o}}$, where $P_{main}$ is a system that meets the conditions given below and $P_{\vec{o}}$ is the part that models the unknown input $\vec{o}$ as in Figure 4.5. We write $m_{\vec{o}}$ for the variant of method `initialInput` that creates an sequence of input values as specified by vector $\vec{o}$.

We require that the main part $P_{main}$ of the program *does not*

1. reference nor modify the variable `list`.

2. create an instance of class `Node`.

Note that when $P_{main}$ meets these conditions, the following statements are true as long as only the initialization of list elements inside $m_{\vec{o}}$ of $P_{\vec{o}}$ is modified:

1. Only method `untrustedInput` references and modifies variable `list`.
2. A single static method `initialValue` ($m_{\vec{o}}$) creates all list elements for variable `list`.
3. The constructor `Node` for elements of `list` is only called inside $m_{\vec{o}}$.
4. Method $m_{\vec{o}}$ has no parameters and does not reference or modify any static variable.
5. Neither `untrustedInput` nor $m_{\vec{o}}$ contain statements that reference high (secret) or low (public) variables.

Using the properties above we can prove the following result.

**Proposition 4.1.** *For all $P[\vec{o}] = P_{main} \cdot P_{\vec{o}}$ that comply with the requirements stated above, it holds that if $P[\vec{o}]$ does not contain illegal information flow, for some sequence $\vec{o}$, then $\forall \vec{u} : P[\vec{u}]$ does not contain illegal information flow.*

Proposition 4.1 enables us to reason about a family of programs by analyzing only a singe instance of it. We take a single specification $\vec{o}$ and analyze $P[\vec{o}]$. If the analysis guarantees security in this case, we know due to Proposition 4.1 that the program is secure for all versions of $m_{\vec{u}}$ that obey the restrictions.

**Example** We use the program from Figure 4.6 as $P_{main}$ and Figure 4.5 as $P_{\vec{o}}$. Indeed all the above requirements are met for $P[\vec{o}] = P_{main} \cdot P_{\vec{o}}$. The main part $P_{main}$ includes reading and writing of untrusted values as well as a computation on a secret value, but not $P_{\vec{o}}$. Figure 4.7 contains the corresponding PDG for variant $V2$ with $\vec{o} = (1, 2)$. This program is considered noninterferent, because the PDG contains no connection from the secret value in line 28 to a statement that leaks to untrusted output (line 30 and 32). Therefore also variant $V1$ and all other variants that match the restrictions are noninterferent.

**Proof sketch of Proposition 4.1** Let us assume that the program $P[\vec{o}] = P_{main} \cdot P_{\vec{o}}$, for some $\vec{o}$ is secure. Additionally we have a set of statements $S_{secret} \subseteq P_{main}$ that are sources of secret values in $P[\vec{o}]$ and a set of statements $S_{out} \subseteq P_{main}$ that refer to untrusted output (e.g. changing the value of a low variable). As $P[\vec{o}]$ is guaranteed to be secure, we also know that there is no path in the corresponding $PDG_{\vec{o}}$ from secret sources to untrusted output:

$$\forall l \in S_{out} : \text{slice}_{PDG_{\vec{o}}}(l) \cap S_{secret} = \{\}$$

If we use $Path(PDG_{\vec{o}})$ as the set of all possible paths in $PDG_{\vec{o}}$ we get:

$$\forall h \in S_{secret}\ \forall l \in S_{out}\ \forall p \in Path(PDG_{\vec{o}}) : h \in p \implies l \notin p_{|h}$$

with

$$p_{|h} = (p_1 \to \ldots \to h \to p_i \to \ldots \to p_n)_{|h} = (h \to p_i \to \ldots \to p_n)$$

With these definitions in place we can continue the proof sketch with the following, auxiliary lemma:

**Lemma 4.1.** *Changes in the PDG local to $m_{\vec{o}}$ are not affecting the security guarantee.*

*Proof.* If the PDG local to $m_{\vec{o}}$ has changed, either previously existing nodes or edges were removed or new ones were introduced. Removing nodes or edges has no effect on the security guarantee as it only reduces the number of possible paths in the whole PDG. So no additional flow can be introduced. Additional nodes or edges inside the PDG part of $m_{\vec{o}}$ does increase the number of possible paths inside the method, but not the number of paths leading to or leaving from it. Because $m_{\vec{o}} \cap S_{secret} = \{\}$ and $m_{\vec{o}} \cap S_{out} = \{\}$ we get that no illegal flow can start or end inside of $m_{\vec{o}}$. Due to the restrictions, we know that $m_{\vec{o}}$ does not take any parameters as arguments and that it does not reference any static variables. So no parameter or heap dependence edges are connected to the entry node of $m_{\vec{o}}$. The only possible paths from a secret source to an untrusted output through $m_{\vec{o}}$ have to enter through the call edge and to leave through the parameter and heap dependence of the *OUT* field. Because any output of a method is dependent on the method execution, there is always a path from the entry node to its *OUT* field. So additional nodes or edges inside $m_{\vec{o}}$ do not introduce new flow from $S_{secret}$ to $S_{out}$. □

Now we change $m_{\vec{o}}$ of $P[\vec{o}]$ to $m_{\vec{u}}$, given by some $\vec{u}$, and assume that the resulting program $P[\vec{u}] = P_{main} \cdot P_{\vec{u}}$ contains an illegal flow. Then we show by contradiction that this is not possible and thus $P[\vec{u}]$ must be secure.

If $P_{\vec{u}}$ contains illegal flow we get

$$\exists h \in S_{secret}\ \exists l \in S_{out}\ : h \in \text{slice}_{PDG_{\vec{u}}}(l)$$

So there must be a path $p$ from $h$ to $l$ in the changed PDG that was not part of the original.

$$\exists p \in Path(PDG_{\vec{u}}) : h \in p \wedge l \in p_{|h} \wedge p \notin Path(PDG_{\vec{o}})$$

Path $p$ is a list of instructions that are connected through dependencies. We are now going to show that any of these dependencies must already have been part of $PDG_{\vec{o}}$ and thus cannot exists if $PDG_{\vec{o}}$ has no illegal flow.

If $p \notin Path(PDG_{\vec{o}})$ then $p$ must contain at least one edge $n_1 \to n_2$ that is not in $PDG_{\vec{o}}$.

$$p \notin Path(PDG_{\vec{o}}) \implies \exists n_1 \to n_2 \in p_{|h} : n_1 \to n_2 \notin PDG_{\vec{o}}$$

Now we consider four different cases.

**Case** $n_1 \notin PDG_{\vec{o}}, n_2 \notin PDG_{\vec{o}}$. Both nodes have not been part of $PDG_{\vec{o}}$. Because $m_{\vec{u}}$ is the only changed part of $P[\vec{u}]$ we know that $n_1, n_2 \in m_{\vec{u}}$. Due to Lemma 4.1 this edge can not introduce new illegal flow.

**Case** $n_1 \notin PDG_{\vec{o}}, n_2 \in PDG_{\vec{o}}$. Because $m_{\vec{u}}$ is the only changed part of $P[\vec{u}]$ we know that $n_1 \in m_{\vec{u}}$. So due to Lemma 4.1 this edge cannot introduce new illegal flow.

**Case** $n_1 \in PDG_{\vec{o}}, n_2 \notin PDG_{\vec{o}}$. $n_2 \in m_{\vec{u}}$ so no new illegal flow is introduced with this edge.

**Case** $n_1, n_2 \in PDG_{\vec{o}}$. Only the edge between $n_1$ and $n_2$ is new in $PDG_{\vec{u}}$. Because of Lemma 4.1 we only have to consider the case $n_1 \to n_2 \notin m_{\vec{u}}$. The new edge may be of four different kinds:

**control dependence** Control dependencies are computed per method using the control flow graph. No method in $P[\vec{o}] \setminus m_{\vec{o}}$ (that is no method of $P[\vec{o}]$ except for $m_{\vec{o}}$) has been changed, so the control flow is unchanged and thus no control dependencies have changed. So $n_1 \rightarrow n_2$ cannot be a control dependency.

**data dependence** Data dependencies are either caused through use and definition of variables or references and modification to heap locations. $P[\vec{u}] \setminus m_{\vec{u}}$ contains no new statements and thus no additional uses and definitions of variables. So the additional data dependence cannot stem from them and has to be introduced through access to heap locations. The only heap locations that have changed are the elements of `list`. Our restrictions forbid references to this list in $P_{main}$, so $n_1 \rightarrow n_2$ cannot be a data dependency.

**call** $P[\vec{u}] \setminus m_{\vec{u}}$ contains no new statements and thus no new call statements. So $n_1 \rightarrow n_2$ cannot be a call dependency.

**parameter and heap dependence** These edges are caused by passing parameters into methods and returning method results. As $P[\vec{u}] \setminus m_{\vec{u}}$ is not changed, no method signatures and calls are altered. So $n_1 \rightarrow n_2$ cannot be caused by additionally passed parameter or returned results. The last remaining possibility is that the edge is caused by a change in the referenced or modified heap locations. Due to the restrictions we know that only heap locations referring to elements of `list` may have changed. As $P_{main}$ does not contain references to `list`, $n_1 \rightarrow n_2$ is no parameter or heap dependency.

We argued that $n_1 \rightarrow n_2$ cannot exists in $PDG_{\vec{u}}$ if $PDG_{\vec{o}}$ contains no illegal flow, thus path $p$ cannot exist and so no illegal information flow can be introduced by changing $m_{\vec{o}}$ of $m_{\vec{u}}$.

### 4.2.2 Example specific analysis code

This section contains the code needed to automatically check the ideal version of the client-server example for noninterference with the help of Joana. It consists of two parts: the annotation of the program specific sources and sinks in class `RunClientServerIFC` and the setup of the analysis with the required precision options in class `IFC`. Generic parts of class `IFC` are left out for brevity. The whole code is accessible at [77] or GitHub[22].

[22] `https://github.com/jgf/crypto-client-ifc`

## Class RunClientServerIFC

```java
public class RunClientServerIFC {

  public static void main(final String[] args) throws ClassHierarchyException,
      IOException, UnsoundGraphException, CancelException {
    final IFCConfig configClientServer = IFCConfig.create("./example/bin",
      "Lde/uni/trier/infsec/protocol/Setup", SecurityPolicy.CONFIDENTIALITY);

    // annotate input of method untrusted output as leaked to low output
    configClientServer.addAnnotation(Annotation.create(
      "de.uni.trier.infsec.environment.Environment.untrustedOutput(I)V",
      BytecodeLocation.ROOT_PARAM_PREFIX + "0",
      SDGNode.Kind.FORMAL_IN,
      SecurityLabel.LOW
    ));

    // annotate references to static variable Setup.secret as high input
    configClientServer.addAnnotation(Annotation.create(
      "de.uni.trier.infsec.protocol.Setup.main([Ljava/lang/String;)V",
      "de.uni.trier.infsec.protocol.Setup.secret",
      SDGNode.Kind.EXPRESSION,
      SDGNode.Operation.REFERENCE,
      SecurityLabel.HIGH
    ));

    IFC.run(configClientServer);
  }
}
```

## Class IFC

```java
class IFC {

  /**
   * Runs an information flow check using the given configuration. Returns
   * <tt>true</tt> if the program is considered save, returns <tt>false</tt> if a
   * leak has been found.
   * @param config The information flow configuration.
   * @return <tt>true</tt> if the program is considered save, or <tt>false</tt> if a
   * leak has been found.
   */
  public static boolean run(final IFCConfig config) throws ClassHierarchyException,
      IOException, UnsoundGraphException, CancelException {
    final Collection<Violation> vios = computeIFC(config);

    if (vios.size() == 0) {
      System.out.println("OK:␣The␣program␣is␣noninterferent.");
    } else {
      switch (config.policy) {
      case CONFIDENTIALITY: {
        System.out.println("WARNING:␣Program␣MAY␣leak␣high␣information.");
      } break;
      case INTEGRITY: {
        System.out.println("WARNING:␣Public␣(low)␣input␣MAY␣influence␣"
          + "confidential␣(high)␣information.");
      } break;
      default:
        throw new IllegalStateException("unknown␣security␣policy:␣" + config.policy);
      }
    }

    return vios.isEmpty();
  }

  private static void buildSDG(final IFCConfig config, final String mainClassSimpleName,
      final String outputSDGfile) throws ClassHierarchyException, IOException,
      UnsoundGraphException, CancelException {
    Log.setMinLogLevel(LogLevel.WARN);
    System.out.println("Analyzing␣class␣files␣from␣'"
      + new File(config.classpath).getAbsolutePath() + "'");
    final Main.Config cfg = new Main.Config(mainClassSimpleName,
      config.mainClassSignature.substring(1) + ".main([Ljava/lang/String;)V",
      config.classpath,
      PointsToPrecision.OBJECT_SENSITIVE,
      ExceptionAnalysis.INTRAPROC, false, Main.STD_EXCLUSION_REG_EXP, null,
      "./lib/jSDG-stubs-jre1.4.jar", null, ".",
      FieldPropagation.OBJ_GRAPH);

    final joana.sdg.SDG sdg = Main.compute(System.out, cfg);

    System.out.print("Saving␣SDG␣to␣" + outputSDGfile + "...␣");
    final BufferedOutputStream bOut =
      new BufferedOutputStream(new FileOutputStream(outputSDGfile));
    SDGSerializer.toPDGFormat(sdg, bOut);
    System.out.println("done.");
  }

  public static Collection<Violation> computeIFC(final IFCConfig config) throws
      ClassHierarchyException, IOException, UnsoundGraphException, CancelException {
    final String mainClassSimpleName =
      WriteGraphToDot.sanitizeFileName(config.mainClassSignature.substring(1));
    final String outputSDGfile = mainClassSimpleName + "-main.ifc.pdg";

    buildSDG(config, mainClassSimpleName, outputSDGfile);

    System.out.println("Checking␣" + outputSDGfile);
    final joana.sdg.SDG sdgSec =
      joana.sdg.SDG.readFrom(outputSDGfile, new SecurityNodeFactory());
    System.out.print("Annotating␣SDG␣nodes...␣");
    final IEditableLattice<String> lat = createLattice("low<=high");
    final int matches = matchAnnotationsWithNodes(sdgSec, config.getAnnotations());
    System.out.print("(" + matches + "␣nodes)␣");

    // set required and provided
    annotateSecurityNodes(config.getAnnotations(), config.policy);
    System.out.println("done.");
    System.out.print("Checking␣information␣flow...␣");
    final PossibilisticNIChecker ifc = new PossibilisticNIChecker(sdgSec, lat);
    final Collection<Violation> vios = ifc.checkIFlow();

    System.out.println("(" + vios.size() + "␣violations)␣done.");

    return Collections.unmodifiableCollection(vios);
  }
}
```

```java
1  public class Example {
2
3    private static int a;
4    public static int result = 0;
5
6    public void main(int secret) {
7      a = 42;
8      bar(secret);
9      int b = foo(secret);
10     result = b;
11   }
13   static int foo(int secret) {
14     int b = a;
15     if (secret == 0)
16       b += secret;
17     return b;
18   }
19
20   static void bar(int secret) {
21     int[] arr = new int[100];
22     for(int i = 0; i < 100; ++i)
23       arr[i] = secret + i + a;
24     result += a - 1;
25   }
26 }
```

**Figure 4.8:** A noninterferent program that cannot be verified with Joana.

## 4.3 Combining analysis tools for hybrid verification – An example

This section introduces a novel hybrid approach to noninterference analysis, that combines Joana with other tools in order to increase analysis precision. Joana is a very precise automatic static analysis tool, but it still contains several approximations that may introduce false alarms in favor of significantly enhanced analysis runtime when compared to other tools like the interactive theorem prover KeY [2]. Therefore certain programs that are in fact noninterferent cannot be verified with Joana.

Figure 4.8 shows such an example. If we want to verify that the value of parameter secret in l. 6 can never influence the value of static variable result, Joana issues a false alarm — even though the program never leaks information about parameter secret. A closer inspection of the source code reveals that the secret value is used during computations in method foo and method bar, but it never has any effect on the value of result. Method bar uses secret in some method local computations in l. 23. As variable arr never leaves the method, information about secret cannot be revealed. Joana can detect that bar does not leak information, but it fails to detect noninterference of method foo. In l. 16 the value of secret is added to the return value of foo and later on in l. 10 result is set to the return value. So at first glance there is an illegal information flow, but going back to l. 16 we see that the value of secret is only added

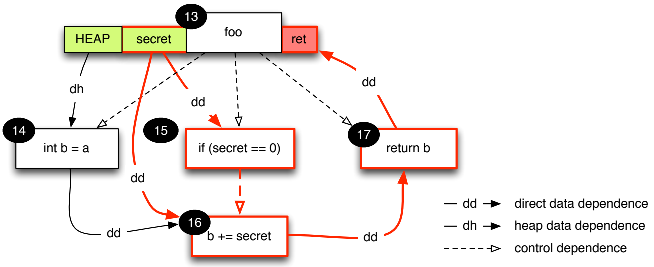


**Figure 4.9:** The PDG of method foo from Figure 4.8. Bold red outlined nodes and edges show statements influenced by parameter `secret`.

to the return value, if it equals zero. Consequently the operation does not affect the return value.

The PDG of foo in Figure 4.9 shows how Joana traces information flow for parameter secret. The edges and nodes marked in bold red represent all statements and dependencies that are potentially influenced by secret. Joana detects potentially influenced statements through an advanced reachability analysis —called slicing— on the PDG. The PDG contains data dependency between parameter secret and statement 16, because technically statement 16 reads the value of secret and adds it to variable b. The data dependency does not cover the fact that the value of secret is only used when it is 0 and therefore has no effect on the value of b. An additional analysis that applies path conditions may remove this imprecision in the future, but currently there is a path (13 → 16 → 17) from secret to the return value and a potential security violation is detected.

However we are able to show that only method foo poses a potential security problem and the rest of the program is safe. Thus we only need to apply a more precise analysis —in our case KeY— to method foo, but we don't need to consider the other parts of the program. This reduces the amount of manual work that is required for a non-automatic tool like KeY. With the help of KeY we are able to prove that the value of b at l. 14 is always the same as its value in l. 17. This enables us to create a conservative extension of foo that behaves similar to the original version, but makes the independence of b from secret more explicit, so that an

```
1  class Extension {
2    static int tmp;
3  }
4
5  class Example {
6    ...
7    static int foo(int secret) {
8      int b = a;
9      Extension.tmp = b;
10     if (secret == 0)
11       b += secret;
12     b = Extension.tmp;
13     return b;
14   }
15   ...
16 }
```

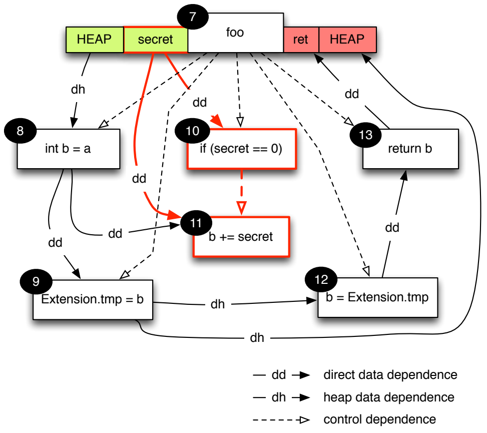


**Figure 4.10:** A conservative extension of method foo for the program in Figure 4.8 that can be verified with Joana and its matching PDG.

automated tool like Joana can verify noninterference.

Figure 4.10 shows the so-called *conservative extension* of method foo and its matching PDG. As we could prove with KeY that the value of b is not changed, we can add two additional statements to the code without changing its behavior. We save the value of b in l. 9 before the part that uses parameter secret and load the original value back into b at l. 12. Hence we create a new class Extension with a static field tmp that temporarily stores the value of b. Note that we did not change any existing code —like completely removing the conditional add instruction— as in general it may have additional effects on the behavior of the program, aside from changing the value of b. This way we can restrict the amount of work that needs to be done with the KeY tool, as we only need to focus on the value of b and can disregard any other effects of the code.

The right side of Figure 4.10 shows the effects the conservative extension of foo has on the PDG and thus the result of the noninterference analysis. Joana is now able to detect that the return value is not influenced by the value of secret, because variable b is explicitly overwritten before its value is returned. However the conservative extension introduced a new side-effect: The static variable `tmp` still contains the value of `b` at the end of method `foo`. A heap data dependency from node 9 to the heap parameter-out node captures this effect. This static variable could potentially be read at another point in the program and introduce additional unwanted information flow. To prevent this we carefully considered what a conservative extension is allowed to do and proved that if the extensions obeys those rules, no additional flow will be introduced. The definition of a conservative extension and the accompanying proof are part of a joint publication [74] in cooperation with Ralf Küsters and Tomasz Truderung from the University of Trier and the members of the KeY group from the Karlsruhe Institute of Technology. A short summary of the results can be found in [73]. As a detailed discussion is out of scope for this work, we only show the definition of a conservative extension as a result from this collaboration.

**Definition 4.1.** *Let $P$ be a program. An* extension *of $P$ is a program $P'$ obtained from $P$ in the following way. First, a new class $M$ with the following properties is added to $P$:*

*(i) the methods and fields of $M$ are static,*
*(ii) the methods of $M$ do not call any methods defined in $P$ (that is $\emptyset \vdash M$),*
*(iii) the arguments and the results of the methods of $M$ are of primitive types,*
*(iv) all potential exceptions are caught inside $M$.*

*Second, $P$ is extended by adding statements of the following form in arbitrary places within methods of $P$:*

*(a) (*output to *$M$)*

$$M.f(e_1, \dots, e_n), \tag{4.1}$$

*where $f$ is a (static) method of $M$ and $e_1, \dots, e_n$ are expressions without side-effects and of the types corresponding to the types of the arguments of $M.f$.*

*(b) (input from $M$)*

$$r = M.f(e_1, \dots, e_n), \tag{4.2}$$

*where $f$ is a method of $M$ with some (primitive) return type $t$, expressions $e_1, \dots, e_n$ are as above, and $r$ is an expression that evaluates without side-effects to a reference of type $t$. (Such an expression can for example be a variable or an expression of the form $o.x$, where $o$ is an object with field $x$.)*

*Such an extension $P'$ of $P$ is called a* conservative extension *of $P$, if, additionally, the following is true. For a run of the program $P'$, whenever a statement of the* *form* (4.2) *is executed, it does not change the value of* $r$*. That is, the value of* $r$ *right* before *the execution of the assignment coincides with the value returned by the method call* $M.f(e_1, \ldots, e_n)$*. As such, statement* (4.2) *is redundant.*

In general the above definition aims to ensure that the state —the accessed memory— of extension $M$ is strictly separated from the rest of the program $P$. Hence the behavior of $P$ is not influenced at all by $M$. This guarantees that as long as our extension follows above rules, we can infer noninterference of $P$ from the noninterference of $P'$.

$$\text{conservative extension } P' \text{ is noninterferent} \implies P \text{ is noninterferent}$$

In this section we introduced the notion of hybrid analysis, where we use so-called conservative extensions to combine different analysis techniques in order to reduce false alarms. Through this approach we can benefit from the advantages of the different techniques. We use the fully automated Joana noninterference analysis for large parts of the program and apply the more precise KeY framework that requires more manual effort to only a small part. In the following chapter we show how the hybrid approach is applied in a more realistic setting to prove cryptographic privacy properties in the context of an e-voting application.

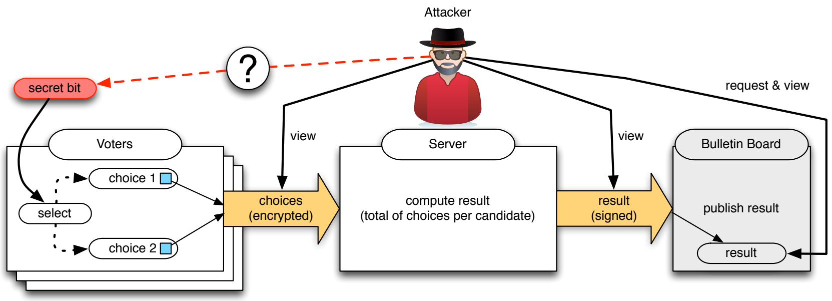


**Figure 4.11:** A schematic view of the simple e-voting system.

## 4.4 Verifying a simple e-voting example with a novel hybrid approach

This case study shows how Joana can be combined with other verification tools to proof advanced cryptographic security properties. In addition to the ideal functionality [75,76] approach introduced in §4.2 we also combine our analysis with the KeY verification tool [2,9] as presented in §4.3. KeY is an interactive theorem proofer for first-order dynamic logic (JavaDL) [55,50,8]. In contrast to well known general purpose theorem provers like Isabelle [99] or Coq [12], KeY specializes in verification of Java programs. It can be used to verify arbitrary advanced properties of sequential Java programs. KeY is more precise then Joana, but it is not an automated tool. It requires significant user effort and has difficulties to scale to large programs. Hence we reduce the parts of the program that need to be formalized with KeY through an automated analysis with Joana. This approach reduces the amount of manual verification required, while it still benefits from KeY's enhanced precision.

Figure 4.11 shows an overview of the example. It consists of a set of voters that communicate their choice to a central server over a secure (encrypted) channel. The server collects all choices and computes the number of total votes for each candidate. Then it transfers the result over an authenticated channel to a bulletin board. We assume that everything that is sent over the network can be read by an attacker, as well as every information posted on the bulletin board.

**Advanced cryptographic security property** Our goal is to proof the privacy of individual votes. The attacker should not gain any additional information that helps to deduce the choice of a voter. As he knows the number of total votes, he can deduce some information. For example if candidate $A$ gets 0 votes, he knows that no single voter chose to vote for $A$. But in case A has a single vote, he has no means to decide which voter chose A, because of equal probability. Hence we want to make sure that the attacker cannot gain any information that improves chances to deduce the choice of a single voter compared to a random guess.

This security property differs significantly from the very strict noninterference property that forbids leakage of any secure information at all. Therefore we set our system up in a way that allows us to infer the privacy of individual votes through showing cryptographic indistinguishability for it. Then we use ideal functionality —as presented in §4.2— to transfer the cryptographic indistinguishability property to classical noninterference. As Joana is not able to automatically show noninterference for this system, we introduce an additional step. We modify the system to enable automatic analysis and use another tool —such as KeY— to verify that these modifications did not change the behavior of the system —as introduced in §4.3. In the following paragraphs we start with a description of the system, then we explain how cryptographic indistinguishability shows privacy of individual votes in this setup. Finally we discuss the necessary modifications that enable automatic analysis with Joana.

We want to allow the attacker to see the result of the vote, but not individual votes. Our system uses two arrays $\vec{c_1}, \vec{c_2}$ as the potential voter choices. Each array entry represents the choice of a single voter. We have two possible choices for each voter $v_n$: One is stored in $\vec{c_1}[n]$ and the other is stored in $\vec{c_2}[n]$. The program then checks if the choices in $\vec{c_1}$ result in the same total of votes for each candidate as the choices in $\vec{c_2}$. This technique guarantees that knowledge of the voting result does not help to decide whether $\vec{c_1}$ or $\vec{c_2}$ represent the individual choices. The system then uses a secret boolean value to decide if $\vec{c_1}$ or $\vec{c_2}$ are used as voter choices. Then the selected choices are encrypted and transfered to the server, allowing the attacker to inspect the encrypted individual choices. The server proceeds with computing the total of all votes and eventually sends them to the bulletin board.

Given this setup we can now use the same cryptographic indistinguishability property we introduced in §4.2 to proof privacy of individual votes in the system. If the attacker is unable to infer any information about the value of the secret boolean, he is also not able to decide whether $\vec{c}_1$ or $\vec{c}_2$ are the actual voter choices. If we show the behavior of the system is cryptographically indistinguishable for both values of the secret boolean, we can also proof that an attacker cannot gain additional information about individual voters choices.

As we want to use Joana to proof this property we have to take some additional steps to model the problem accordingly. We transfer the original program $P$ to a version $P_{ideal}$ that uses ideal functionality for encryption of the individual votes sent to the server, so we can use classical noninterference to show cryptographic indistinguishability. Then we add additional code at certain parts that enable Joana to detect noninterference of the modified version $P_{hybrid}$. We use KeY to proof that the additional code does not change the behavior of the program. This way we can show cryptographic indistinguishability of $P$ by proving noninterference for $P_{hybrid}$.

$$\begin{aligned} P_{hybrid} \text{ noninterferent} \quad &\Longrightarrow \quad P_{ideal} \text{ noninterferent} \\ &\Longrightarrow \quad P \text{ cryptographic indistinguishable} \end{aligned}$$

Joana is not able to show noninterference for $P_{ideal}$ without additional modifications. The programs noninterference depends on the property that the value of total votes is the same for $\vec{c}_1$ and $\vec{c}_2$ and therefore it is impossible to decide which option has been chosen from the number of total votes alone. Joana searches for violating paths in the PDG model of the program. It detects which statements depend on the outcome of other statements, without considering the actual values computed. Thus it detects that the number of total votes is computed by adding up the individual votes and the individual votes are selected through the secret boolean. Therefore Joana concludes that the result depends on the secret boolean. This approximation improves the performance of the analysis, but in this case it also reduces precision.

Our solution is a hybrid approach where we use an additional tool — the KeY tool — to guarantee that the value of the result is independent of the choice between $\vec{c}_1$ and $\vec{c}_2$. This is true, because otherwise the program aborts before selecting individual votes and it never sends them to the server. Based on this guarantee we can modify the program at two specific points to enable a noninterference analysis with Joana without changing its behavior:

- We store the number of total votes for $\vec{c}_1$ and $\vec{c}_2$ a separate variable *total*, before we choose through the secret value of $s$ which votes are sent to the server.
- After the server receives either $\vec{c}_1$ or $\vec{c}_2$ and computes the number of total votes, we replace the results of the computation with the value in variable *total*.

As the analysis approach used by Joana has no way to verify that the computation in fact computes the total of all votes, this work-around allows Joana to establish that the result of the computation is the same as the number of total votes computed at the beginning. Hence information flow from single individual voter choices is prevented. With the help of KeY we are then able to show that these modifications do not change the behavior of the program and thus its noninterference property is unchanged. Therefore if we can show noninterference for the modified version $P_{hybrid}$ we can automatically infer noninterference of the original version $P_{ideal}$.

We provide the full source code of version $P_{hybrid}$ in Appendix B. The vectors $\vec{c}_1$ and $\vec{c}_2$ are modeled as **byte**`[]` arrays `voterChoices1`, `voterChoices2` whose values are read in from the environment in l. 231 in the main method of class `HonestVotersSetup`. Input and output from the environment as well as network communication visible to the attacker is modeled by the classes in package `environment.*`. Any public visible output results in a modification of attribute `result` defined in l. 13 of class `Environment`. Unknown and untrusted input is simulated by a list of integer values initialized in l. 18. Class `HonestVotersSetup` contains the setup and main loop of the program. It stores the copy of the precomputed results in the attributes of inner class `CorrectResult` in l. 51 and computes them in method `computeCorrectResult` in l. 116. The loop starting at l. 183 triggers the various possible events like selecting votes, sending votes to the server, or posting results on the bulletin board. It simulates the life cycle of the e-voting system. The secret bit that decides which voter choices are selected is stored in attribute `secret` in l. 57 of `HonestVoterSetup`. In order to show noninterference in this example we have to proof that no information about the value of

`HonestVoterSetup.secret` can flow to the environment and thus to the value of `Environment.result`.

We are able to analyze the modified version $P_{hybrid}$ with a size of 950 LoC on a standard PC (Core i5 2.3GHz, 8GB RAM) in about 17 seconds. Computation of the PDG model takes about 16 seconds and running the slicing-based IFC check is performed in under 1 second. The computation time is relatively large compared to the size of the program. This has two major reasons: (1) We need to analyze the program code in combination with the parts of the JRE standard library that it uses. This significantly increases the size of the analyzed code. (2) We need to apply a very precise but slow object-sensitive points-to analysis. Less precise and faster analyses yield false positives.

**Relevant analysis features for noninterference check** Although the program is quite small, it still poses a challenge for a static analysis to verify its noninterference. The following precision enhancing features of Joana were required for a successful noninterference check:

**Object-, context- and field-sensitivity** We need to distinguish methods and object instances based on the context in which they are used at several points in the program. See sections §2.2.1, §2.2.3 and §2.2.2 for more details on these sensitivity features. §2.5.2 explains the details of the object-sensitive points-to analysis.

- Secret as well as public information are both stored in **`byte`**`[]` arrays. In many points in the program the content of these arrays is leaked to a publicly visible channel. We need to verify that only array instances containing public information are leaked.
- The method `copyOf` in l. 5 of class `MessageTools` is used to copy secret as well as public information to a new array. The security level of its return value depends on the context it is called in. Furthermore we need to distinguish the contents of arrays from their metadata —such as their address or length.

**Flow-sensitivity and local killing definitions** §2.2.4 contains a detailed description of flow-sensitivity and killing definitions. These features are needed at several points in the program to increase the precision of the analysis. The most important part is in method `getResult` at l. 95 in class `Server`:

```
private byte[] getResult() {
  if (!resultReady()) return null;

  votesForA = HonestVotersSetup.CorrectResult.votesForA;
  votesForB = HonestVotersSetup.CorrectResult.votesForB;

  return formatResult(votesForA, votesForB);
}
```

After the server computes the number of total votes from the received voters choices and before it formats and returns the result, it overwrites the computed value with the values stored in `CorrectResult`. We have to detect that the assignment definitely kills the computed value and afterwards all accesses to the returned result only depend on the values in `CorrectResult`.

**Enhanced exception analysis and fine-grained field accesses**
Sections §2.1.8, §2.4.1 and §2.4.3 describe in detail how we use exception analysis and fine-grained fields to increase precision. In the current example fine-grained field accesses allow us to detect that the secret value that influences certain array-set instructions cannot trigger an exception, as it is used to select the field value and not the base pointer. This is crucial for the part that copies the selected votes to the message that is transfered to the server in class `HonestVoterSetup` l. 130:

```
private static byte[] chooseVoterChoices(byte[] voterChoices1, byte[] voterChoices2) {
  byte[] voterChoices = new byte[Server.NumberOfVoters];
  for (int i=0; i<Server.NumberOfVoters; ++i) {
    final byte data1 = voterChoices1[i];
    final byte data2 = voterChoices2[i];
    voterChoices[i] = (secret ? data1 : data2);
  }
  return voterChoices;
}
```

**Applying IFC analysis to the case study** The code that performs the IFC analysis on the case study using the Joana API is shown below. We mark the value of the boolean in `HonestVotersSetup.secret` as source of secret information. All output to the public visible environment, like data sent over the network, is modeled in our example as a modification to the static variable `Environment.result`. Therefore we annotate each modification of this variable as a sink of public information. Then we build the PDG model for the program with object-sensitive points-to precision and enabled exception optimization. Finally we mark the information sources and sinks accordingly and perform the IFC analysis on the model. If no violations are found, we know for sure, that the program is noninterferent.

```java
public final class AnalyzeHonestVoterSetup {

  private AnalyzeHonestVoterSetup() {
    throw new IllegalStateException();
  }

  private final static String CLASSPATH = "./build/";

  public static void main(String[] args) throws ClassHierarchyException, IOException,
      UnsoundGraphException, CancelException {
    final String[] secretSource = new String[] {
      "de.uni.trier.infsec.protocols.smt_voting.HonestVotersSetup.secret"
    };
    final String[] publicOut = new String[] {
      "de.uni.trier.infsec.environment.Environment.result"
    };
    final IFCAnalysis ana = buildAndAnnotate(
      "de.uni.trier.infsec.protocols.smt_voting.HonestVotersSetup",
      PointsToPrecision.OBJECT_SENSITIVE, secretSource, publicOut);
    final Collection<IllicitFlow> leaks = ana.doIFC();

    if (leaks.isEmpty()) {
      System.out.println("Program␣is␣noninterferent:␣No␣leaks␣found.");
    } else {
      System.out.println("Leaks␣found:␣" + leaks.size());
    }

    final SDG sdg = ana.getProgram().getSDG();
    SDGSerializer.toPDGFormat(sdg, new FileOutputStream("HonestVotersSetup.pdg"));
    for (final IllicitFlow leak : leaks) {
      System.out.println(leak);
      final Violation vio = leak.getViolation();
      System.out.println("Violation:␣" + vio);
    }
  }

  private static IFCAnalysis buildAndAnnotate(final String className,
      final PointsToPrecision pts, final String[] secretSource,
      final String[] publicOut) throws ClassHierarchyException, IOException,
      UnsoundGraphException, CancelException {
    final JavaMethodSignature mainMethod =
      JavaMethodSignature.mainMethodOfClass(className);
    final SDGConfig config =
      new SDGConfig(CLASSPATH, mainMethod.toBCString(), Stubs.JRE_14);
    config.setComputeInterferences(false);
    config.setExceptionAnalysis(ExceptionAnalysis.INTRAPROC);
    config.setFieldPropagation(FieldPropagation.OBJ_GRAPH);
    config.setPointsToPrecision(pts);
    config.setMhpType(MHPType.NONE);
    final SDGProgram prog = SDGProgram.createSDGProgram(config, System.out,
       NullProgressMonitor.INSTANCE);

    final IFCAnalysis ana = new IFCAnalysis(prog);
    for (final String strSec : secretSource) {
      final SDGProgramPart secret = ana.getProgramPart(strSec);
      assert (secret != null);
      ana.addSourceAnnotation(secret, BuiltinLattices.STD_SECLEVEL_HIGH);
    }

    for (final String strPub : publicOut) {
      final SDGProgramPart output = ana.getProgramPart(strPub);
      assert (output != null);
      ana.addSinkAnnotation(output, BuiltinLattices.STD_SECLEVEL_LOW);
    }

    return ana;
  }
}
```

*We can only see a short distance ahead, but we can see plenty there that needs to be done.*

Alan Touring

# 5 Conclusion

We presented numerous improvements to dependence graph computation and slicing-based information flow control analysis in Chapter 2. In Chapter 3 we discussed new solutions for the modularization of SDG computation and information flow of components in unknown context. Finally Chapter 4 demonstrated the usage of our IFC tool on several applications.

Our work has been incorporated into the Joana information flow control framework and successfully applied to real world programs [85] as well as research prototypes. As of today, Joana is the most advanced IFC tool for Java that can guarantee noninterference and comes with support for multithreaded programs. Joana needs only few annotations and is easy to use. It scales for programs up to 100kLoC and comes with a wide variety of options that allow the user to hand pick the best combination of precision and scalability options for the given problem. We also included a simple configuration and categorization inference algorithm (§4.1) that detects the best configuration out of a set of predefined options. In addition it categorizes the discovered information leaks in direct, indirect and exception induced leaks — helping the user to decide on the severity of each leak. Future work in this area aims to automatically compute a custom generated points-to analysis that only applies a precise analysis where needed, hence keeping the computation overhead minimal.

In Chapter 2 we discussed the challenges of analyzing a realistic object-oriented language and showed current state-of-the-art methods to tackle them. We introduced in detail several improvements to SDG computation including

- a precise analysis of the effects of potential exceptions (§2.4.1) that enhances the precision of the analysis result by detecting impossible and always occurring exceptions. Previously, exceptions significantly harmed the precision of the SDG, but when ignored the result can no longer guarantee security properties such as noninterference. Hence they need to be included. With the presented analysis it is possible to achieve precise results that include the effect of exceptions and thus can still guarantee noninterference. Further extensions of this analysis are currently developed. They will enable us to detect additional exceptions such as array out of bounds errors.

- a fine-grained model of field access operations (§2.4.3) that allows a more detailed tracking of information flow, enhances precision and enables the computation of access paths used in modular SDGs.

- termination sensitive control dependencies (§2.4.2) that can optionally be activated to detect information leaks through nontermination. Several tests have shown that termination sensitivity severely harms analysis precision. Thus, while it is a theoretical interesting property, in practice too many termination related dependencies are introduced that prevent a meaningful result. In our view the large number of false alarms is likely the reason why most information flow control related analyses ignore termination sensitivity.

- a new parameter model (§2.6) for interprocedural side-effects. We studied the phases of SDG computation and identified the interprocedural side-effect computation —also known as parameter model— as a performance critical part. We discovered a scalability bottleneck in the old object-tree model and removed the weakness with our new parameter model using object-graphs. Compared to the old model it performs consistently better and in most cases even enhances precision, as shown in the extensive evaluation of the performance of different parameter models with varying points-to analyses.

Furthermore, we developed a flexible algorithm (Algorithm 2.3 and Algorithm 2.5) that can be used for either object-tree or object-graph computation with only minor adjustments. The intermediate representation of the algorithm based on field-candidates allows easy modifications like hand-tuned variants of the parameter computation. For example we experimented with a maximal node count threshold, where whenever the number of parameter nodes for a method exceeds the threshold, parameter candidates are merged until the resulting nodes are below the threshold. This modification guaranteed fast summary edge computation times, but often at severe precision penalties. However, we expect that future work in this area can achieve performance gains with only a minor precision penalty, when the "correct" candidates are merged.

Chapter 3 contains a new approach for precise information flow control with SDGs in unknown context — a major step towards modularization of SDG computation. We discovered that most points-to analysis currently used in analysis frameworks have in common that they do not support strong updates, which in turn leads to a monotonicity property. This property allows us to approximate the outcome of the analysis of a component in different contexts, without explicitly performing the analysis. We provide a proof that these approximations are conservative and therefore can be used to give guarantees in our security related setting. Based on this property, we develop algorithms that —when given certain context properties— can compute minimal and maximal points-to sets and generate context stubs for the component. These context stubs can be used to analyze the component with any whole program analysis meeting the prerequisites of the monotonicity property without any change to the analysis itself. Hence we are able to use our standard IFC analysis to compute information flow for components in unknown contexts. Additionally, we invented the modular SDG that —once computed— contains a summary of the component in any potential context and thus allows a fast adaption to a concrete context when needed. While the computation of the modular SDG is theoretically possible, we observed that in practice it is often not feasible, due to the huge amount of different context configurations possible. Therefore, we introduced a way to approximate the modular SDG with the help of so-called access paths. Our evaluation showed that the computation of the modular SDG is very complex and hence future work needs to discover additional optimizations in order to work with larger components. Nevertheless the theoretical concept is sound —as shown in our sketched proof— and can be used as foundation for future work.

In general, we noticed that scalability and precision can still be improved in slicing based IFC analyses, but in our opinion the point of diminishing returns is reached. It may prove beneficial to relax the fixation on soundness and focus research on theoretical possible information leaks that are in practice a real threat. We think that proven soundness guarantees are important, but allowing the user to optionally ignore some leaks can be very useful. For example if a program cannot be proven secure, it is relevant if it contains obvious leaks —such as direct information flow through local variables— or if only more intricate leaks are found. Complex languages such as Java can be analyzed, but secure programs are hard to write and even harder to automatically proof secure. Hence, we suggest that either soundness has to be sacrificed at some point or we need to restrict security critical program parts to a subset of Java. Research in the direction of domain specific IFC friendly languages may also be beneficial. Still the problem remains that currently many security critical programs written in Java already exist. These programs most likely cannot be proven secure with 100% soundness, as they either contain potential information leaks or the conservative approximations of the IFC analysis produce a false alarm. Therefore, future work should also focus on quantitative IFC and tools that support a manual decision on the severity of a leak. The current state of the Joana IFC tool should provide a solid foundation for future improvements in these directions.

# A
# Sourcecode of Client-Server Example

This section contains the sourcecode of ideal version $P_{ideal}$ of the client-server example as shown in Figure 4.4 of §4.2.

## Package protocol

### Class Setup

```
package de.uni.trier.infsec.protocol;

import de.uni.trier.infsec.environment.Environment;
import de.uni.trier.infsec.network.Network;
import de.uni.trier.infsec.network.NetworkError;
import de.uni.trier.infsec.pkenc.Decryptor;
import de.uni.trier.infsec.pkenc.Encryptor;
import de.uni.trier.infsec.protocol.Client;
import de.uni.trier.infsec.protocol.Server;

/**
 *  Setup for the simple protocol: it creates the server and then,
 *  depending on the input from the untrusted network, creates some
 *  (potentially unbounded) number of clients and makes them send
 *  their messages.
 *
 *  In case of each client, two messages are determined by the
 *  environment; one of them is picked and sent by the client,
 *  depending on the value of a secret bit. The adversary is not
 *  supposed to learn the value of this bit.
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class Setup {

```

```java
  static private boolean secret = false; // SECRET -- an arbitrary value put here

  public static void main(String[] args) throws NetworkError {

    // Public-key encryption functionality for Server
    Decryptor serverDec = new Decryptor();
    Encryptor serverEnc = serverDec.getEncryptor();
    Network.networkOut(serverEnc.getPublicKey()); // public key of Bob is published

    // Creating the server
    Server server = new Server(serverDec);

    // The adversary decides how many clients we create:
    while( Network.networkIn() != null )  {
      // determine the value the client encrypts:
      // the adversary gives two values
      byte s1 = Network.networkIn()[0];
      byte s2 = Network.networkIn()[0];
      // and one of them is picked depending on the value of the secret bit
      byte s = secret ? s1 : s2;
      Client client = new Client(serverEnc, s);

      // initialize the client protocol
      // (Alice sends out an encrypted value s to the network)
      client.onInit();
      // read a message from the network...
      byte[] message = Network.networkIn();
      // ... and deliver it to the server (server will decrypt it)
      server.onReceive(message);
    }
  }
}
```

## Class Server

```java
package de.uni.trier.infsec.protocol;

import de.uni.trier.infsec.pkenc.Decryptor;

/**
 *  Server of a simple protocol that simply decrypts received message.
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
final public class Server {
  private Decryptor BobPKE;
  private byte[] receivedMessage = null;

  public Server(Decryptor BobPKE) {
    this.BobPKE = BobPKE;
  }

  public void onReceive(byte[] message) {
    receivedMessage = BobPKE.decrypt(message);
  }
}
```

### Class Client

```
package de.uni.trier.infsec.protocol;

import de.uni.trier.infsec.network.Network;
import de.uni.trier.infsec.network.NetworkError;
// import de.uni.trier.infsec.pkenc.PKEnc;
import de.uni.trier.infsec.pkenc.Encryptor;

/**
 *  Client of a simple protocol: it encrypts a given message and sends
 *  it over the network.
 *
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
final public class Client {
  private Encryptor BobPKE;
  private byte[] message;

  public Client(Encryptor BobPKE, byte message) {
    this.BobPKE = BobPKE;
    this.message = new byte[] {message};
  }

  public void onInit() throws NetworkError {
    byte[] encMessage = BobPKE.encrypt(message);
    Network.networkOut(encMessage);
  }
}
```

## Package environment

### Class Environment

```
package de.uni.trier.infsec.environment;

/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
class Node {
  int value;
  Node next;
  Node(int v, Node n) {
    value = v; next = n;
  }
}

/**
 *  Generic environment for verifying noninterference in an open
 *  systems (systems interacting with untrusted
 *  environment/libraries).
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class Environment {

  private static boolean result; // the LOW variable

  private static Node list = null;
  private static boolean listInitialized = false;

  private static Node initialValue() {
    // Unknown specification of the following form:
    // return new Node(U1, new Node(U2, ...));
    // where U1, U2, ...Un are constant integers.
    return new Node(1, new Node(7,null));  // just an example
  }

  public static int untrustedInput() {
    if (!listInitialized) {
      list = initialValue();
      listInitialized = true;
    }
    if (list==null) return 0;
    int tmp = list.value;
    list = list.next;
    return tmp;
  }

  public static void untrustedOutput(int x) {
    if (untrustedInput()==0) {
      result = (x==untrustedInput());
      throw new Error();  // abort
    }
  }
}
```

## Package pkenc

### Class Decryptor

```
package de.uni.trier.infsec.pkenc;

import de.uni.trier.infsec.crypto.CryptoLib;
import de.uni.trier.infsec.crypto.KeyPair;

/**
 * Ideal functionality for public-key encryption: Decryptor
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public final class Decryptor {

  private byte[] privKey;
  private byte[] publKey;
  private MessagePairList log = new MessagePairList();

  public Decryptor() {
    KeyPair keypair = CryptoLib.generateKeyPair();
    publKey = MessageTools.copyOf(keypair.publicKey);
    privKey = MessageTools.copyOf(keypair.privateKey);
  }

  public Encryptor getEncryptor() {
    return new Encryptor(log, publKey);
  }

  public byte[] decrypt(byte[] message) {
    byte[] messageCopy = MessageTools.copyOf(message);
    if (!log.contains(messageCopy)) {
      return MessageTools.copyOf(
        CryptoLib.decrypt(MessageTools.copyOf(privKey), messageCopy) );
    } else {
      return MessageTools.copyOf( log.lookup(messageCopy) );
    }
  }
}
```

## Class Encryptor

```
package de.uni.trier.infsec.pkenc;

import de.uni.trier.infsec.crypto.CryptoLib;

/**
 * Ideal functionality for public-key encryption: Encryptor
 *
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public final class Encryptor {

  private MessagePairList log;
  private byte[] publKey;

  Encryptor(MessagePairList mpl, byte[] publicKey) {
    log = mpl;
    publKey = publicKey;
  }

  public byte[] getPublicKey() {
    return MessageTools.copyOf(publKey);
  }

  public byte[] encrypt(byte[] message) {
    byte[] messageCopy = MessageTools.copyOf(message);
    // Note the fixed size (1) of a message
    byte[] randomCipher = MessageTools.copyOf(
    CryptoLib.encrypt(MessageTools.getZeroMessage(1),
    MessageTools.copyOf(publKey)));
    if( randomCipher == null ) return null;
    log.add(messageCopy, randomCipher);
    return MessageTools.copyOf(randomCipher);
  }

}
```

## Class MessagePairList

```java
package de.uni.trier.infsec.pkenc;

/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class MessagePairList {

  static class MessagePair {
    byte[] ciphertext;
    byte[] plaintext;
    MessagePair next;

    public MessagePair(byte[] ciphertext, byte[] plaintext, MessagePair next) {
      this.ciphertext = ciphertext;
      this.plaintext = plaintext;
      this.next = next;
    }
  }

  private MessagePair first = null;

  public void add(byte[] pTxt, byte[] cTxt) {
    first = new MessagePair(cTxt, pTxt, first);
  }

  byte[] lookup(byte[] ciphertext) {
    MessagePair tmp = first;
    while( tmp != null ) {
      if ( MessageTools.equal(tmp.ciphertext, ciphertext) )
        return tmp.plaintext;
      tmp = tmp.next;
    }
    return null;
  }

  boolean contains(byte[] ciphertext) {
    MessagePair tmp = first;
    while( tmp != null ) {
      if ( MessageTools.equal(tmp.ciphertext, ciphertext) )
        return true;
      tmp = tmp.next;
    }
    return false;
  }

}
```

## Class MessageTools

```java
package de.uni.trier.infsec.pkenc;

/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class MessageTools {

  public static byte[] copyOf(byte[] message) {
    if (message == null) return null;
    byte[] copy = new byte[message.length];
    for (int i = 0; i < message.length; i++) {
      copy[i] = message[i];
    }
    return copy;
  }

  public static boolean equal(byte[] a, byte[] b) {
    if ( a.length != b.length ) return false;
    for(int i = 0; i < a.length; ++i)
      if ( a[i] != b[i] ) return false;
    return true;
  }

  public static byte[] getZeroMessage(int messageSize) {
    byte[] zeroVector = new byte[messageSize];
    for (int i = 0; i < zeroVector.length; i++) {
      zeroVector[i] = 0x00;
    }
    return zeroVector;
  }
}
```

## Package crypto

### Class CryptoLib

```
package de.uni.trier.infsec.crypto;

import de.uni.trier.infsec.environment.Environment;

/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class CryptoLib {

  public static byte[] encrypt(byte[] in, byte[] publKey) {
    // input
    Environment.untrustedOutput(0x66); // Function code for encryption
    Environment.untrustedOutput(in.length);
    for (int i = 0; i < in.length; i++) {
      byte b = in[i];
      Environment.untrustedOutput(b);
    }
    Environment.untrustedOutput(publKey.length);
    for (int i = 0; i < publKey.length; i++) {
      byte b = publKey[i];
      Environment.untrustedOutput(b);
    }

    // output
    int len = Environment.untrustedInput();
    if (len < 0) return null;
    byte[] returnval = new byte[len];
    for (int i = 0; i < len; i++) {
      returnval[i] = (byte) Environment.untrustedInput();
    }
    return returnval;
  }

  public static byte[] decrypt(byte[] message, byte[] privKey) {
    // input
    Environment.untrustedOutput(0x77); // Function code for decryption
    Environment.untrustedOutput(message.length);
    for (int i = 0; i < message.length; i++) {
      byte b = message[i];
      Environment.untrustedOutput(b);
    }
    Environment.untrustedOutput(privKey.length);
    for (int i = 0; i < privKey.length; i++) {
      byte b = privKey[i];
      Environment.untrustedOutput(b);
    }

    // output
    int len = Environment.untrustedInput();
    if (len < 0) return null;
    byte[] returnval = new byte[len];
    for (int i = 0; i < len; i++) {
      returnval[i] = (byte) Environment.untrustedInput();
    }
    return returnval;
  }

  public static KeyPair generateKeyPair() {
    // input
    Environment.untrustedOutput(0x88); // Function code for generateKeyPair

    // ouptut
    KeyPair returnval = new KeyPair();
    returnval.privateKey = null;
    int len = Environment.untrustedInput();
    if (len >= 0) {
      returnval.privateKey = new byte[len];
      for (int i = 0; i < len; i++) {
        returnval.privateKey[i] = (byte) Environment.untrustedInput();
      }
    }
    returnval.publicKey = null;
    len = Environment.untrustedInput();
    if (len >= 0) {
      returnval.publicKey= new byte[len];
      for (int i = 0; i < len; i++) {
        returnval.publicKey[i] = (byte) Environment.untrustedInput();
      }
    }
    return returnval;
  }
}
```

### Class KeyPair

```java
package de.uni.trier.infsec.crypto;

/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class KeyPair {
  public byte[] publicKey;
  public byte[] privateKey;
}
```

## Package network

### Class Network

```java
package de.uni.trier.infsec.network;

import de.uni.trier.infsec.environment.Environment;


/**
 *  @author Andreas Koch (University of Trier)
 *  @author Tomasz Truderung (University of Trier)
 */
public class Network {

  public static void networkOut(byte[] outEnc) throws NetworkError {
    // input
    Environment.untrustedOutput(0x55);
    Environment.untrustedOutput(outEnc.length);
    for (int i = 0; i < outEnc.length; i++) {
      Environment.untrustedOutput(outEnc[i]);
    }
    // output
    if (Environment.untrustedInput()==0) throw new NetworkError();
  }

  public static byte[] networkIn() throws NetworkError {
    // input
    Environment.untrustedOutput(0x56);

    // output
    if (Environment.untrustedInput()==0) throw new NetworkError();
    int len = Environment.untrustedInput();
    if (len < 0) return null;
    byte[] val = new byte[len];
    for (int i = 0; i < len; i++) {
      val[i] = (byte) Environment.untrustedInput();
    }
    return val;
  }
}
```

# B Sourcecode of the Simple e-Voting Example

This section contains the $P_{hybrid}$ version of the sourcecode for the e-voting example discussed in §4.4.

## Package `functionalities.*`

### Class `AMT`

```
package de.uni.trier.infsec.functionalities.amt.ideal;

import de.uni.trier.infsec.utils.MessageTools;
import de.uni.trier.infsec.functionalities.pki.ideal.PKIError;
import de.uni.trier.infsec.environment.network.NetworkClient;
import de.uni.trier.infsec.environment.network.NetworkError;
import de.uni.trier.infsec.environment.amt.AMTEnv;

/**
 * Ideal functionality for AMT (Authenticated Message Transmission).
 *
 * Every party who wants to use this functionality should first register itself:
 *
 *     AgentProxy a = AMT.register(ID_OF_A);
 *
 * Then, to send messages to a party with identifier ID_OF_B:
 *
 *     Channel channel_to_b = a.channelTo(ID_OF_B);
 *     channel_to_b.send( message1 );
 *     channel_to_b.send( message2 );
 *
 * To read messages sent to the agent a:
 *
 *     AuthenticatedMessage msg = a.getMessage();
 *     // msg.message contains the received message
 *     // msg.sender_id contains the id of the sender
 */
public class AMT {

  //// The public interface ////

  static public class AMTError extends Exception {}

  /**
   * Pair (message, sender_id).
   *
   * Objects of this class are returned when an agent reads a message from its queue.
   */
  static public class AuthenticatedMessage {
    public byte[] message;
    public int sender_id;

    private AuthenticatedMessage(byte[] message, int sender) {
      this.sender_id = sender;  this.message = message;
    }
  }

  /**
   * Objects representing agents' restricted (private) data that can be used
   * to securely send and receive authenticated message.
   *
   * Such an object allows one to
   *   - get messages from the queue or this agent (method getMessage),
   *     where the environment decides which message is to be delivered,
   *   - create a channel to another agent (channelTo and channelToAgent); such
   *     a channel can be used to securely send authenticated messages to the
   *     chosen agent.
   */
  static public class AgentProxy
  {
    public final int ID;
    private final MessageQueue queue;  // messages sent to this agent

    private AgentProxy(int id) {
      this.ID = id;
      this.queue = new MessageQueue();
    }

    /**
     * Returns next message sent to the agent. It return null, if there is no such
     * a message.
     *
     * In this ideal implementation the environment decides which message is to be
     * delivered.
     * The same message may be delivered several times or not delivered at all.
     */
    public AuthenticatedMessage getMessage(int port) throws AMTError {
      if (registrationInProgress) throw new AMTError();
      int index = AMTEnv.getMessage(this.ID, port);
      if (index < 0) return null;
      return queue.get(index);
    }

    public Channel channelTo(int recipient_id, String server, int port)
        throws AMTError, PKIError, NetworkError {
```

```
      if (registrationInProgress) throw new AMTError();
      boolean network_ok = AMTEnv.channelTo(ID, recipient_id, server, port);
      if (!network_ok) throw new NetworkError();
      // get the answer from PKI
      AgentProxy recipient = registeredAgents.fetch(recipient_id);
      if (recipient == null) throw new PKIError();
      // create and return the channel
      return new Channel(this, recipient, server, port);
    }
  }

  /**
   * Objects representing secure and authenticated channel from sender to recipient.
   *
   * Such objects allow one to securely send a message to the recipient, where the
   * sender is authenticated to the recipient.
   */
  static public class Channel
  {
    private final AgentProxy sender;
    private final AgentProxy recipient;
    private final String server;
    private final int port;

    private Channel(AgentProxy from, AgentProxy to, String server, int port) {
      this.sender = from;
      this.recipient = to;
      this.server = server;
      this.port = port;
    }

    public void send(byte[] message) {
      byte[] output_message =
        AMTEnv.send(message, sender.ID, recipient.ID, server, port);
      recipient.queue.add(MessageTools.copyOf(message), sender.ID);
      try {
        NetworkClient.send(output_message, server, port);
      } catch (NetworkError e) {}
    }
  }

  /**
   * Registering an agent with the given id. If this id has been already used,
   * registration fails (the method returns null).
   */
  public static AgentProxy register(int id) throws AMTError, PKIError {
    if (registrationInProgress) throw new AMTError();
    registrationInProgress = true;
    // call the environment/simulator
    AMTEnv.register(id);
    // check whether the id has not been claimed
    if( registeredAgents.fetch(id) != null ) {
      registrationInProgress = false;
      throw new PKIError();
    }
    // create a new agent, add it to the list of registered agents, and return it
    AgentProxy agent = new AgentProxy(id);
    registeredAgents.add(agent);
    registrationInProgress = false;
```

```
      return agent;
   }

   private static boolean registrationInProgress = false;


   //// Implementation ////

   //
   // MessageQueue -- a queue of messages (along with the id of the sender).
   // Such a queue is kept by an agent and represents the messages that has been
   // sent to this agent.
   //
   private static class MessageQueue
   {
      private static class Node {
         final byte[] message;
         final int sender_id;
         final Node next;
         Node(byte[] message, int sender_id, Node next) {
            this.message = message;
            this.sender_id = sender_id;
            this.next = next;
         }
      }
      private Node first = null;

      void add(byte[] message, int sender_id) {
         first = new Node(message, sender_id, first);
      }

      AuthenticatedMessage get(int index) {
         if (index<0) return null;
         Node node = first;
         for( int i=0;  i<index && node!=null;  ++i )
            node = node.next;
         return  (node != null
            ? new AuthenticatedMessage(MessageTools.copyOf(node.message), node.sender_id)
            : null);
      }
   }

   //
   // AgentsQueue -- a collection of registered agents.
   //
   private static class AgentsQueue
   {
      private static class Node {
         final AgentProxy agent;
         final Node  next;
         Node(AgentProxy agent, Node next) {
            this.agent = agent;
            this.next = next;
         }
      }
      private Node first = null;

      public void add(AgentProxy agent) {
         first = new Node(agent, first);
```

```
    }

    AgentProxy fetch(int id) {
      for( Node node = first;  node != null;  node = node.next )
        if( id == node.agent.ID )
          return node.agent;
      return null;
    }
  }

  // static list of registered agents:
  private static AgentsQueue registeredAgents = new AgentsQueue();
}
```

## Class SMT

```
package de.uni.trier.infsec.functionalities.smt.ideal;

import de.uni.trier.infsec.utils.MessageTools;
import de.uni.trier.infsec.functionalities.pki.ideal.PKIError;
import de.uni.trier.infsec.environment.network.NetworkClient;
import de.uni.trier.infsec.environment.network.NetworkError;
import de.uni.trier.infsec.environment.smt.SMTEnv;

/**
 * Ideal functionality for SAMT (Secure Authenticated Message Transmission).
 *
 * Every party who wants to use this functionality should first register itself:
 *
 *     AgentProxy a = SAMT.register(ID_OF_A);
 *
 * Then, to send messages to a party with identifier ID_OF_B:
 *
 *     Channel channel_to_b = a.channelTo(ID_OF_B);
 *     channel_to_b.send( message1 );
 *     channel_to_b.send( message2 );
 *
 * (It is also possible to create a channel to b by calling a.channelToAgent(b).)
 *
 * To read messages sent to the agent a:
 *
 *     AuthenticatedMessage msg = a.getMessage();
 *     // msg.message contains the received message
 *     // msg.sender_id contains the id of the sender
 */
public class SMT {

  //// The public interface ////

  static public class SMTError extends Exception {}

  /**
   * Pair (message, sender_id).
   *
   * Objects of this class are returned when an agent reads a message from its queue.
   */
  static public class AuthenticatedMessage {
    public final byte[] message;
    public final int sender_id;
    public AuthenticatedMessage(byte[] message, int sender) {
      this.sender_id = sender;  this.message = message;
    }
  }

  /**
   * Objects representing agents' restricted (private) data that can be used
   * to securely send and receive authenticated message.
   *
   * Such an object allows one to
   *  - get messages from the queue or this agent (method getMessage),
   *    where the environment decides which message is to be delivered,
   *  - create a channel to another agent (channelTo and channelToAgent); such
   *    a channel can be used to securely send authenticated messages to the
   *    chosen agent.
   */
  static public class AgentProxy
  {
    public final int ID;
    private final MessageQueue queue;  // messages sent to this agent

    private AgentProxy(int id) {
      this.ID = id;
      this.queue = new MessageQueue();
    }

    /**
     * Returns next message sent to the agent. It return null, if there is no such
     * a message.
     *
     * In this ideal implementation the environment decides which message is to be
     * delivered.
     * The same message may be delivered several times or not delivered at all.
     */
    public AuthenticatedMessage getMessage(int port) throws SMTError {
      if (registrationInProgress) throw new SMTError();
      int index = SMTEnv.getMessage(this.ID, port);
      if (index < 0) return null;
      return queue.get(index);
    }

    public Channel channelTo(int recipient_id, String server, int port)
        throws SMTError, PKIError, NetworkError {
      if (registrationInProgress) throw new SMTError();
      boolean network_ok = SMTEnv.channelTo(ID, recipient_id, server, port);
      if (!network_ok) throw new NetworkError();
      // get the answer from PKI
      AgentProxy recipient = registeredAgents.fetch(recipient_id);
      if (recipient == null) throw new PKIError();
      // create and return the channel
      return new Channel(this, recipient, server, port);
    }
  }

  /**
   * Objects representing secure and authenticated channel from sender to recipient.
   *
   * Such objects allow one to securely send a message to the recipient, where the
   * sender is authenticated to the recipient.
   */
  static public class Channel
  {
    private final AgentProxy sender;
    private final AgentProxy recipient;
    private final String server;
    private final int port;

    private Channel(AgentProxy from, AgentProxy to, String server, int port) {
      this.sender = from;
      this.recipient = to;
      this.server = server;
      this.port = port;
    }

    public void send(byte[] message) {
      byte[] output_message =
        SMTEnv.send(message.length, sender.ID, recipient.ID, server, port);
      recipient.queue.add(MessageTools.copyOf(message), sender.ID);
      try {
        NetworkClient.send(output_message, server, port);
      } catch (NetworkError e) {}
    }
  }

  /**
   * Registering an agent with the given id. If this id has been already used,
   * registration fails (the method returns null).
   */
  public static AgentProxy register(int id) throws SMTError, PKIError {
    if (registrationInProgress) throw new SMTError();
    registrationInProgress = true;
    // call the environment/simulator
    SMTEnv.register(id);
    // check whether the id has not been claimed
    if( registeredAgents.fetch(id) != null ) {
      registrationInProgress = false;
      throw new PKIError();
    }
    // create a new agent, add it to the list of registered agents, and return it
    AgentProxy agent = new AgentProxy(id);
    registeredAgents.add(agent);
    registrationInProgress = false;
    return agent;
  }

  private static boolean registrationInProgress = false;


  //// Implementation ////

  //
  // MessageQueue -- a queue of messages (along with the id of the sender).
  // Such a queue is kept by an agent and represents the messages that has been
  // sent to this agent.
  //
  private static class MessageQueue
  {
    private static class Node {
      final byte[] message;
      final int sender_id;
      final Node next;
      Node(byte[] message, int sender_id, Node next) {
        this.message = message;
        this.sender_id = sender_id;
        this.next = next;
      }
    }
    private Node first = null;

    void add(byte[] message, int sender_id) {
      first = new Node(message, sender_id, first);
    }

    AuthenticatedMessage get(int index) {
      if (index<0) return null;
      Node node = first;
      for( int i=0;  i<index && node!=null;  ++i )
        node = node.next;
      return (node != null
        ? new AuthenticatedMessage(MessageTools.copyOf(node.message), node.sender_id)
        : null);
    }
  }

  //
  // AgentsQueue -- a collection of registered agents.
  //
  private static class AgentsQueue
  {
    private static class Node {
      final AgentProxy agent;
      final Node  next;
      Node(AgentProxy agent, Node next) {
        this.agent = agent;
        this.next = next;
      }
    }
    private Node first = null;

    public void add(AgentProxy agent) {
      first = new Node(agent, first);
    }

    AgentProxy fetch(int id) {
      for( Node node = first;  node != null;  node = node.next )
        if( id == node.agent.ID )
          return node.agent;
      return null;
    }
  }

  // static list of registered agents:
  private static AgentsQueue registeredAgents = new AgentsQueue();
}
```

## Package utils.*

### Class MessageTools

```
package de.uni.trier.infsec.utils;

public class MessageTools {

  public static byte[] copyOf(byte[] message) {
    if (message==null) return null;
    byte[] copy = new byte[message.length];
    for (int i = 0; i < message.length; i++) {
      copy[i] = message[i];
    }
    return copy;
  }

  public static byte[] concatenate(byte[] m1, byte[] m2) {
    // we allocate 4 additional bytes for the length of m1
    byte[] out = new byte[m1.length + m2.length + 4];
    byte[] len = intToByteArray(m1.length);

    // copy all bytes to the output array
    int j = 0;
    for( int i=0; i<len.length; ++i ) out[j++] = len[i]; // the length of m1
    for( int i=0; i<m1.length;  ++i ) out[j++] = m1[i];  // m1
    for( int i=0; i<m2.length;  ++i ) out[j++] = m2[i];  // m2

    return out;
  }

  public static final byte[] intToByteArray(int value) {
    return new byte[] {
      (byte)(value >>> 24),
      (byte)(value >>> 16),
      (byte)(value >>> 8),
      (byte)value};
  }
}
```

## Package environment.*

### Class NetworkServer

```
package de.uni.trier.infsec.environment.network;

import de.uni.trier.infsec.environment.Environment;

public class NetworkServer {

  public static void listenForRequests(int port) throws NetworkError {
    // input
    Environment.untrustedOutput(0x2400);
    Environment.untrustedOutput(port);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
  }

  public static byte[] nextRequest(int port) throws NetworkError {
    // input
    Environment.untrustedOutput(0x2401);
    Environment.untrustedOutput(port);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
    return Environment.untrustedInputMessage();
  }

  public static void response(byte[] message) throws NetworkError {
    // input
    Environment.untrustedOutput(0x2402);
    Environment.untrustedOutputMessage(message);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
  }

  public static byte[] read(int port) throws NetworkError {
    // input
    Environment.untrustedOutput(0x2403);
    Environment.untrustedOutput(port);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
    return Environment.untrustedInputMessage();
  }
}
```

## Class NetworkClient

```
package de.uni.trier.infsec.environment.network;

import de.uni.trier.infsec.environment.Environment;

public class NetworkClient {

  public static void send(byte[] message, String server, int port) throws
      NetworkError {
    // input
    Environment.untrustedOutput(0x2301);
    Environment.untrustedOutputMessage(message);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
  }

  public static byte[] sendRequest(byte[] message, String server, int port)
      throws NetworkError {
    // input
    Environment.untrustedOutput(0x2302);
    Environment.untrustedOutputMessage(message);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    // output
    if ( Environment.untrustedInput()==0 ) throw new NetworkError();
    return Environment.untrustedInputMessage();
  }
}
```

## Class AMTEnv

```
package de.uni.trier.infsec.environment.amt;

import de.uni.trier.infsec.environment.Environment;

public class AMTEnv {
  public static void register(int id) {
    Environment.untrustedOutput(7801);
    Environment.untrustedOutput(id);
  }

  public static boolean channelTo(int sender_id, int recipient_id, String server,
      int port) {
    Environment.untrustedOutput(7802);
    Environment.untrustedOutput(sender_id);
    Environment.untrustedOutput(recipient_id);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    return Environment.untrustedInput()==0;
  }

  public static byte[] send(byte[] message, int sender_id, int recipient_id,
      String server, int port) {
    Environment.untrustedOutput(7803);
    Environment.untrustedOutputMessage(message);
    Environment.untrustedOutput(sender_id);
    Environment.untrustedOutput(recipient_id);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    return Environment.untrustedInputMessage();
  }

  public static int getMessage(int id, int port) {
    Environment.untrustedOutput(7804);
    Environment.untrustedOutput(id);
    Environment.untrustedOutput(port);
    return Environment.untrustedInput();
  }
}
```

## Class SMTEnv

```
package de.uni.trier.infsec.environment.smt;


import de.uni.trier.infsec.environment.Environment;


public class SMTEnv {

  public static void register(int id) {
    Environment.untrustedOutput(7801);
    Environment.untrustedOutput(id);
  }

  public static boolean channelTo(int sender_id, int recipient_id, String server,
      int port) {
    Environment.untrustedOutput(7802);
    Environment.untrustedOutput(sender_id);
    Environment.untrustedOutput(recipient_id);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    return Environment.untrustedInput()==0;
  }

  public static byte[] send(int message_length, int sender_id, int recipient_id,
      String server, int port) {
    Environment.untrustedOutput(7803);
    Environment.untrustedOutput(message_length);
    Environment.untrustedOutput(sender_id);
    Environment.untrustedOutput(recipient_id);
    Environment.untrustedOutputString(server);
    Environment.untrustedOutput(port);
    return Environment.untrustedInputMessage();
  }

  public static int getMessage(int id, int port) {
    Environment.untrustedOutput(7804);
    Environment.untrustedOutput(id);
    Environment.untrustedOutput(port);
    return Environment.untrustedInput();
  }
}
```

## Class Environment

```
package de.uni.trier.infsec.environment;

class Node {
  int value;
  Node next;
  Node(int v, Node n) {
    value = v; next = n;
  }
}

public class Environment {

  private static boolean result; // the LOW variable

  private static Node list = null;
  private static boolean listInitialized = false;

  private static Node initialValue() {
    // Unknown specification of the following form:
    // return new Node(U1, new Node(U2, ...));
    // where U1, U2, ...Un are constant integers.
    return new Node(1, new Node(7,null));  // just an example
  }

  public static int untrustedInput() {
```

```java
    if (!listInitialized) {
      list = initialValue();
      listInitialized = true;
    }
    if (list==null) return 0;
    int tmp = list.value;
    list = list.next;
    return tmp;
  }

  public static void untrustedOutput(int x) {
    if (untrustedInput()==0) {
      result = (x==untrustedInput());
      throw new Error();  // abort
    }
  }

  public static byte[] untrustedInputMessage() {
    int len = untrustedInput();
    if (len<0) return null;
    byte[] returnval = new byte[len];
    for (int i = 0; i < len; i++) {
      returnval[i] = (byte) Environment.untrustedInput();
    }
    return returnval;
  }

  public static void untrustedOutputMessage(byte[] t) {
    untrustedOutput(t.length);
    for (int i = 0; i < t.length; i++) {
      untrustedOutput(t[i]);
    }
  }

  public static void untrustedOutputString(String s) {
    untrustedOutput(s.length());
    for (int i = 0; i < s.length(); i++) {
      untrustedOutput((int)s.charAt(i));
    }
  }
}
```

## Package protocols.*

### Class BulletinBoard

```java
package de.uni.trier.infsec.protocols.smt_voting;

import de.uni.trier.infsec.functionalities.amt.ideal.AMT;
import de.uni.trier.infsec.functionalities.amt.ideal.AMT.AMTError;
import de.uni.trier.infsec.utils.MessageTools;

/*
 * Bulletin board on which the server can post messages (the result) and
 * everybody can retrieve the posted messages.
 */
public class BulletinBoard {

  public BulletinBoard(AMT.AgentProxy proxy) {
    content = new MessageList();
    amt_proxy = proxy;
  }

  /*
   * Reads a message, checks if it comes from the server, and, if this is the
   * case, adds it to the maintained list of messages.
   */
  public void onPost() throws AMTError {
    AMT.AuthenticatedMessage am =
      amt_proxy.getMessage(Parameters.DEFAULT_LISTEN_PORT_BBOARD_AMT);
    if (am == null) return;
    if (am.sender_id != Identifiers.SERVER_ID) return;

    byte[] message = am.message;
    content.add(message);
  }

  /*
   * Sends its content (that is the concatenation of all the message in the maintained
   * list of messages) over the network.
   */
  public byte[] onRequestContent() {
    byte[] contentMessage = {};
    for( MessageList.Node node = content.first;  node!=null;  node = node.next ) {
      contentMessage = MessageTools.concatenate(contentMessage, node.message);
    }
    return contentMessage;
  }

  /// implementation ///

  class MessageList {
    class Node {
      byte[] message;
      Node next;
      Node(byte[] message, Node next) { this.message = message; this.next = next; }
    }

    Node first = null;

    void add(byte[] message) {
      first = new Node(message, first);
    }
  }

  private MessageList content;
  private AMT.AgentProxy amt_proxy;
}
```

## Class HonestVotersSetup

```
package de.uni.trier.infsec.protocols.smt_voting;

import de.uni.trier.infsec.environment.network.NetworkError;
import de.uni.trier.infsec.environment.Environment;
import de.uni.trier.infsec.functionalities.pki.ideal.PKIError;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT.SMTError;
import de.uni.trier.infsec.functionalities.amt.ideal.AMT;
import de.uni.trier.infsec.functionalities.amt.ideal.AMT.AMTError;


/*
 * A setup for a server and (multiple) honest clients using secure authenticated
 * channel (secure, authenticated message transmission functionality) to send their
 * choices to the server.
 *
 * The adversary determines two variants of voters'choices, one of which is picked,
 * based on the value of the secret bit. During the voting process the adversary
 * determines actions to be taken.
 */
public class HonestVotersSetup {

  static class Adversary {
    public final SMT.Channel channel_to_server;
    public final AMT.Channel channel_to_BB;

    public Adversary() throws SMTError, PKIError, NetworkError, AMTError {
      SMT.AgentProxy adversary_samt_proxy = SMT.register(Identifiers.ADVERSARY_ID);
      channel_to_server = adversary_samt_proxy.channelTo(Identifiers.SERVER_ID,
        "www.server.com", 89);
      AMT.AgentProxy adversary_amt_proxy = AMT.register(Identifiers.ADVERSARY_ID);
      channel_to_BB = adversary_amt_proxy.channelTo(Identifiers.BULLETIN_BOARD_ID,
        "www.bulletinboard.com", 89);
    }
  }

  /*
   * Objects representing a result of the e-voting process. For now, two candidates
   * only.
   */
  static class Result {
    public int votesForA = 0;
    public int votesForB = 0;
  }

  /*
   * Class with static fields to store the correct result computed from the votes
   * actually used by the voters. This class plays the role of the class M from the
   * hybrid approach, as described in the paper.
   */
  static class CorrectResult {
    static public int votesForA = 0;
    static public int votesForB = 0;
  }


  static private boolean secret;  // SECRET INPUT

  static private Voter[] voters;
  static private Server server;
  static private BulletinBoard BB;


  /**
   * Compute the correct result from a vector of voters' choices
   */
  private static Result result(byte[] choices) {
    Result result = new Result();
    for( int i=0; i<choices.length; ++i ) {
      int candidate = choices[i];
      if (candidate==0) result.votesForA++;
      if (candidate==1) result.votesForB++;
      // all the remaining values as considered as invalid
    }
    return result;
  }

  /**
   * Check whether two results are the same.
   */
  private static boolean sameResults(Result res1, Result res2 ) {
    return res1.votesForA==res2.votesForA  &&  res1.votesForB==res2.votesForB;
  }


  /**
   * Computes the correct result, as determined by the vectors voters' choices given
   * as parameters, checks if these two vectors yield the same result. If not, false
   * is returned. Otherwise, voters are registered and created.
   */
  private static boolean select_voters_choices_and_create_voters(byte[] voterChoices1,
       byte[] voterChoices2) throws SMTError, PKIError, NetworkError {
    // we check whether voterChoices1 and voterChoices2 yield the same
    // results:
    boolean status = computeCorrectResult(voterChoices1, voterChoices2);
    if (!status) return false;

    // now, one of the vectors of voters' choices given by the adversary is chosen
    // to be used by the voters, depending on the value of the secret bit:
    byte[] voterChoices = chooseVoterChoices(voterChoices1, voterChoices2);

    // Register and create the voters
    registerAndCreateVoters(voterChoices);

    return true;
  }

  /**
   * Checks whether voterChoices1 and voterChoices2 yield the same results
   */
  private static boolean computeCorrectResult(byte[] voterChoices1,
      byte[] voterChoices2) {
    Result result1 = result(voterChoices1);
    Result result2 = result(voterChoices2);
    if( !sameResults(result1,result2) ) return false;
    CorrectResult.votesForA = result1.votesForA; // hybrid approach extension
    CorrectResult.votesForB = result1.votesForB; // hybrid approach extension
    return true;
  }

  /**
   * One of the vectors of voters' choices given by the adversary is chosen
```

```java
   * to be used by the voters, depending on the value of the secret bit:
   */
  private static byte[] chooseVoterChoices(byte[] voterChoices1, byte[] voterChoices2) {
    byte[] voterChoices = new byte[Server.NumberOfVoters];
    for (int i=0; i<Server.NumberOfVoters; ++i) {
      final byte data1 = voterChoices1[i];
      final byte data2 = voterChoices2[i];
      voterChoices[i] = (secret ? data1 : data2);
    }
    return voterChoices;
  }

  /**
   * Register and create the voters.
   */
  private static void registerAndCreateVoters(byte[] voterChoices) throws SMTError,
      PKIError, NetworkError {
    voters = new Voter[Server.NumberOfVoters];
    for( int i=0; i<Server.NumberOfVoters; ++i ) {
      SMT.AgentProxy voter_proxy = SMT.register(i);
      voters[i] = new Voter(voterChoices[i], voter_proxy);
    }
  }

  /**
   * Register and create the server.
   */
  private static void create_server() throws SMTError, PKIError, AMTError,
      NetworkError {
    SMT.AgentProxy server_samt_proxy = SMT.register(Identifiers.SERVER_ID);
    AMT.AgentProxy server_amt_proxy = AMT.register(Identifiers.SERVER_ID);
    server = new Server(server_samt_proxy, server_amt_proxy);
  }

  /**
   * Register and create the bulletin board.
   */
  private static void create_bulletin_board() throws AMTError, PKIError {
    // Register and create the bulletin board:
    AMT.AgentProxy BB_proxy = AMT.register(Identifiers.BULLETIN_BOARD_ID);
    BB = new BulletinBoard(BB_proxy);
  }

  private static void onVote() throws SMTError {
    int voter_id = Environment.untrustedInput();
    if (voter_id>=0 && voter_id<Server.NumberOfVoters) {
      voters[voter_id].onSendBallot();
    }
  }

  /**
   * Run the main loop of the setup.
   *
   * First, the adversary registers his SAMT and AMT functionalities. Then, in a loop,
   * the adversary decides which actions are taken.
   */
  private static void run() throws SMTError, PKIError, NetworkError, AMTError {
    Adversary adversary = new Adversary();
    // Main loop -- the adversary decides how many times it runs and what to do in
    // each step:
    while( Environment.untrustedInput() != 0 )  {
      byte[] message;
      int decision = Environment.untrustedInput();
      switch (decision) {
      case 0: // a voter (determined by the adversary) votes according to voterChoices
          onVote();
          break;

      case 1: // server reads a message (possibly a ballot) from a secure channel
          server.onCollectBallot();
          break;

      case 2: // server sends the result of the election (if ready) over the network
          try {
            server.onSendResult("", 1);
          }
          catch (NetworkError err) {}
          break;

      case 3: // server posts the result (if ready) on the bulletin board
          server.onPostResult();
          break;

      case 4: // the bulletin board reads a message:
          BB.onPost();
          break;

      case 5: // the bulletin board sends its content (over the network):
          byte[] content = BB.onRequestContent();
          Environment.untrustedOutputMessage(content);
          break;

      case 6: // the adversary sends a message using its channel to the server
          message = Environment.untrustedInputMessage();
          adversary.channel_to_server.send(message);
          break;

      case 7: // the adversary sends a message using its channel to the bulletin board
          message = Environment.untrustedInputMessage();
          adversary.channel_to_BB.send(message);
          break;
      }
    }
  }

  public static void main(String[] args) throws SMTError, PKIError, NetworkError,
      AMTError {
    // the adversary determines two possible ways the voters vote:
    byte[] voterChoices1 = new byte[Server.NumberOfVoters];
    byte[] voterChoices2 = new byte[Server.NumberOfVoters];
    for( int i=0; i<Server.NumberOfVoters; ++i ) {
      voterChoices1[i] = (byte)Environment.untrustedInput();
      voterChoices2[i] = (byte)Environment.untrustedInput();
    }

    boolean status =
      select_voters_choices_and_create_voters(voterChoices1, voterChoices2);
    if (!status) return;
    create_server();
    create_bulletin_board();
    run();
  }
}
```

## Class Identifiers

```
package de.uni.trier.infsec.protocols.smt_voting;

/*
 * Agent identifiers.
 */
public class Identifiers {
  static final int SERVER_ID = -1;
  static final int BULLETIN_BOARD_ID = -2;
  static final int ADVERSARY_ID = -3;
  // eligible voters get the identifiers in the range 0..Server.NumberOfVoters
}
```

## Class Parameters

```
package de.uni.trier.infsec.protocols.smt_voting;

public class Parameters {
  // Listen port for Voter requests
  static final int DEFAULT_LISTEN_PORT_SERVER_AMT = 4088;
  // Listen port for Voter requests
  static final int DEFAULT_LISTEN_PORT_SERVER_SMT = 4089;

  // Listen port for Server requests
  static final int DEFAULT_LISTEN_PORT_BBOARD_AMT = 4090;
  // Listen port for Server requests
  static final int DEFAULT_LISTEN_PORT_BBOARD_SMT = 4091;
  // Listen port for result requests
  static final int DEFAULT_LISTEN_PORT_BBOARD_REQUEST = 4092;

  static final String DEFAULT_HOST_SERVER = "localhost";
  static final String DEFAULT_HOST_BBOARD = "localhost";
}
```

## Class Server

```
package de.uni.trier.infsec.protocols.smt_voting;

import de.uni.trier.infsec.environment.network.NetworkClient;
import de.uni.trier.infsec.environment.network.NetworkError;
import de.uni.trier.infsec.functionalities.pki.ideal.PKIError;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT.SMTError;
import de.uni.trier.infsec.functionalities.amt.ideal.AMT;
import de.uni.trier.infsec.functionalities.amt.ideal.AMT.AMTError;

/*
 * The server of TrivVoting. Collects votes send to it directly (via method call).
 *
 * Two-candidates case only (for now).
 */
public class Server {

  public static final int NumberOfVoters = 50;
  // ballotCast[i]==true iff the i-th voter has already cast her ballot
  private final boolean[] ballotCast;
  private int votesForA;
  private int votesForB;
  private final SMT.AgentProxy samt_proxy;
  private final AMT.Channel channel_to_BB;

  public Server(SMT.AgentProxy samt_proxy, AMT.AgentProxy amt_proxy) throws AMTError,
      PKIError, NetworkError {
    votesForA = 0;
    votesForB = 0;
    this.samt_proxy = samt_proxy;
    channel_to_BB = amt_proxy.channelTo(Identifiers.BULLETIN_BOARD_ID,
    Parameters.DEFAULT_HOST_BBOARD, Parameters.DEFAULT_LISTEN_PORT_BBOARD_AMT);
    // initially no voter has cast her ballot
    ballotCast = new boolean[NumberOfVoters];
  }

  /*
   * Collect one ballot (read from a secure channel)
   */
  public void onCollectBallot() throws SMTError {
    SMT.AuthenticatedMessage am =
    samt_proxy.getMessage(Parameters.DEFAULT_LISTEN_PORT_SERVER_SMT);
    if (am==null) return;
    int voterID = am.sender_id;
    byte[] ballot = am.message;

    if( voterID<0 || voterID>=NumberOfVoters ) return;  // invalid  voter ID
    if( ballotCast[voterID] ) return;  // the voter has already voted
    ballotCast[voterID] = true;
    if( ballot==null || ballot.length!=1 ) return;  // malformed ballot
    int candidate = ballot[0];
    if (candidate==0) ++votesForA;
    if (candidate==1) ++votesForB;
    // all the remaining values are consider invalid
  }

  /*
   * Returns true if the result is ready, that is if all the eligible voters have
   * already voted.
   */
  public boolean resultReady() {
    for( int i=0; i<NumberOfVoters; ++i ) {
      if( !ballotCast[i] )
        return false;
    }
    return true;
  }

  /*
   * Send the result (if ready) of the election over the network.
   */
  public void onSendResult(String addr, int port) throws NetworkError {
    byte[] result = getResult();
    if (result != null)
      NetworkClient.send(result, addr, port);
  }

  /*
   * Post the result (if ready) on the bulletin board.
   */
  public void onPostResult() throws AMTError {
    byte[] result = getResult();
    if (result != null)
      channel_to_BB.send(result);
  }

  private byte[] getResult() {
  // the result is only returned when all the voters have voted
    if (!resultReady()) return null;

    // PROVE THAT
    //    votesForA == HonestVotersSetup.CorrectResult.votesForA
    //    votesForB == HonestVotersSetup.CorrectResult.votesForB
    // (this shows that the extension is conservative)
    votesForA = HonestVotersSetup.CorrectResult.votesForA; // hybrid approach extension
    votesForB = HonestVotersSetup.CorrectResult.votesForB; // hybrid approach extension

    return formatResult(votesForA, votesForB);
  }

  /*
   * Format the result of the election.
   */
  private static byte[] formatResult(int a, int b) {
    String s = "Result␣of␣the␣election:";
    s += "␣␣Number␣of␣voters␣=␣" + NumberOfVoters + "\n";
    s += "␣␣Number␣of␣votes␣for␣candidate␣1␣=" + a + "\n";
    s += "␣␣Number␣of␣votes␣for␣candidate␣2␣=" + b + "\n";
    return s.getBytes();
  }
}
```

## Class Voter

```
package de.uni.trier.infsec.protocols.smt_voting;

import de.uni.trier.infsec.environment.network.NetworkError;
import de.uni.trier.infsec.functionalities.pki.ideal.PKIError;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT;
import de.uni.trier.infsec.functionalities.smt.ideal.SMT.SMTError;

/*
 * Voter client for TrivVoting.
 */
public class Voter {
  private final byte vote;
  private final SMT.Channel channel_to_server;

  public Voter(byte vote, SMT.AgentProxy voter_proxy) throws SMTError, PKIError,
      NetworkError  {
```

```
    this.vote = vote;
    // create secure channel to the server
    this.channel_to_server = voter_proxy.channelTo(Identifiers.SERVER_ID,
    Parameters.DEFAULT_HOST_SERVER, Parameters.DEFAULT_LISTEN_PORT_SERVER_SMT);
  }

  /*
   * Prepare the ballot containing the vote and send it using the secure channel to
   * the server.
   */
  public void onSendBallot() throws SMTError {
    byte [] ballot = new byte[] {vote};
    channel_to_server.send(ballot);
  }
}
```

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Software and System Modeling*, 4:32–54, 2005.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[4] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, University of Copenhagen, 1994.

[5] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and Ponnuswamy Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer Berlin / Heidelberg, 2006.

[6] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. objectives and progress report. In *TGC '06*, LNCS. Springer-Verlag, 2006.

[7] B. Beckert, D. Bruns, R. Küsters, C. Scheben, P. H. Schmitt, and T. Truderung. The KeY approach for the cryptographic verification of Java programs: A case study. Technical Report 2012-8,

Department of Informatics, Karlsruhe Institute of Technology, 2012.

[8] Bernhard Beckert. A dynamic logic for Java Card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000.

[9] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[10] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[11] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 04 1977.

[14] David Binkley, Susan Horwitz, and Thomas Reps. The multi-procedure equivalence theorem. Technical report, University of Wisconsin-Madison, 1989.

[15] Eric Bodden. Position paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, New York, NY, USA, 2012. ACM.

[16] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[17] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[18] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. On improvements of low-deterministic security. In *Principles of Security and Trust, POST 2016, Part of ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016*, volume 9635 of *Lecture Notes in Computer Science*, pages 68–88. Springer Berlin Heidelberg, 2016.

[19] L. Burdy, Y. Cheon, D. Cok, et al. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 2005.

[20] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 463 –475, dec. 2007.

[21] Han Chen and Pasquale Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[22] S. Chong, K. Vikram, and A. Myers. Sif: enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[23] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electron. Notes Theor. Comput. Sci.*, 112:149–166, January 2005.

[24] 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.

[25] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[26] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[27] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language Systems*, 9(3):319–349, 1987.

[28] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 346–362, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[30] Dennis Giffhorn and Gregor Snelting. Probabilistic noninterference based on program dependence graphs. Technical Report 6, Karlsruhe Institute of Technology, April 2012.

[31] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, 2015.

[32] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[33] Joseph A. Goguen and José Meseguer. Interference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

[34] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[35] Jürgen Graf. Improving and evaluating the scalability of precise system dependence graphs for objectoriented languages. Technical report, Universität Karlsruhe (TH), Fak. f. Informatik, 2009.

[36] Jürgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 105–114, September 2010.

[37] Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs - a practical guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215, pages 123–138. Springer Berlin / Heidelberg, February 2013.

[38] Jürgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive interference analysis for java: Combining program dependence graphs with dynamic pushdown networks. In *International Workshop on Interference and Dependence*, 2013.

[39] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Checking applications using security apis with joana. July 2015. 8th International Workshop on Analysis of Security APIs.

[40] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Sicherheitsanalyse mit joana, 2016. to appear at GI Sicherheit 2016.

[41] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Tool demonstration: Joana. In *Principles of Security and Trust, POST 2016, Part of ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016*, volume 9635 of *Lecture Notes in Computer Science*, pages 89–93. Springer Berlin Heidelberg, 2016.

[42] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[43] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, New York, NY, USA, 2011. ACM.

[44] Christian Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3.

[45] Christian Hammer. Efficient algorithms for control closures. In *International Workshop on Interference and Dependence*, 2013.

[46] Christian Hammer and Gregor Snelting. An improved slicer for java. In *PASTE '04*, pages 17–22, New York, NY, USA, 2004. ACM.

[47] 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, December 2009.

[48] Ben Hardekopf and Calvin Lin. Semi-sparse flow sensitive pointer analysis. In *POPL '09*, pages 226–238, New York, NY, USA, 2009. ACM.

[49] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[50] David Harel. Dynamic logic. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Co., Dordrecht, 1984.

[51] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *In Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, pages 1–18, 1999.

[52] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. *SIGPLAN Not.*, 36(5):24–34, May 2001.

[53] Markus Herhoffer. Algorithmik- und logikbasierte points-to analysen im vergleich. Master's thesis, Karlsruher Institut für Technologie, December 2012.

[54] Markus Herhoffer. Präziser Kontrollfluss für Exceptions durch interprozedurale Datenflussanalyse. Studienarbeit, Juni 2012.

[55] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[56] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[57] 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.

[58] Daniel Jackson. Hazards of verification. In *Proc. Haifa Verification Conference*, volume 5394 of *LNCS*, pages 1–1, 2008.

[59] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM.

[60] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317, September 1977.

[61] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Bug localization by algorithmic debugging and program slicing. In *PLILP '90: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, pages 60–74, London, UK, 1990. Springer-Verlag.

[62] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34nd ACM SIGPLAN conference on Programming language design and implementation*, 2013.

[63] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. Model-driven development of information flow-secure systems with IFlow. *ASE Science Journal*, 2(2), 2013.

[64] Kuzman Katkalov, Peter Fischer, Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Evaluation of jif and joana as information flow analyzers in a model-driven approach. In Roberto Di Pietro, Javier Herranz, Ernesto Damiani, and Radu State, editors, *Data Privacy Management and Autonomous Spontaneous Security*, volume 7731 of *Lecture Notes in Computer Science*, pages 174–186. Springer Berlin Heidelberg, 2013.

[65] Bastian Katz, Marcus Krug, Andreas Lochbihler, Ignaz Rutter, Gregor Snelting, and Dorothea Wagner. Gateway decompositions for constrained reachability problems. In *Proceedings of the 9th International Symposium on Experimental Algorithms*, volume 6049 of *LNCS*, pages 449–461. Springer, May 2010.

[66] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[67] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[68] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *In Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[69] Jens Krinke. Static slicing of threaded programs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, Montreal, Canada, June 1998.

[70] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, April 2003.

[71] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. (accompanying technical report) extending and applying a framework for the cryptographic verification of java programs. Cryptology ePrint Archive, Report 2014/038, 2014. Extended version of the POST 2014 paper with additional appendix.

[72] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and applying a framework for the cryptographic verification of java programs. In *Principles of Security and Trust, POST 2014, Part of ETAPS 2014, Grenoble, France, April 5-13, 2014*, volume 8424 of *Lecture Notes in Computer Science*, pages 220–239. Springer, 2014.

[73] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of java programs, April 2013. Grande Region Security and Reliability Day (GRSRD) 2013.

[74] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving noninterference of java programs. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*. IEEE Computer Society, 2015. to appear.

[75] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, June 2012.

[76] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of java-like programs. Technical report, University of Trier, March 2012.

[77] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. Source Code for Simple Protocol Case Study with Joana, 2012. Available at `http://infsec.uni-trier.de/publications/software/simplProtJoana.zip`.

[78] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[79] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.

[80] Ondrej Lhoták and Laurie J. Hendren. Context-sensitive points-to analysis: Is it worth it? In Alan Mycroft and Andreas Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 47–64. Springer, 2006.

[81] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[82] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[83] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[84] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012.

[85] Enrico Lovat, Alexander Fromm, Martin Mohr, and Alexander Pretschner. Shrift system-wide hybrid information flow tracking. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 371–385. Springer International Publishing, 2015.

[86] Heiko Mantel. On the Composition of Secure Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, Oakland, CA, USA, May 12–15 2002. IEEE Computer Society.

[87] Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 141–156, Berlin, Heidelberg, 2007. Springer-Verlag.

[88] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, Tucson, AZ, USA, June 9–11, 2008.

[89] John Mclean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1:37–58, 1992.

[90] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 305–315, New York, NY, USA, 2010. ACM.

[91] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[92] Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding android support to a static information flow control tool. In *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2015.

[93] 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. ACM, 1999.

[94] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow, July 2001.

[95] Mangala Gowri Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay, 2001.

[96] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 180–190, New York, NY, USA, 2000. ACM.

[97] Mangala Gowri Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Transactions on Programming Language Systems (TOPLAS)*, 28(6):1088–1144, 2006.

[98] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.

[99] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[100] Phil Pfeiffer and RebeccaParsons Selke. On the adequacy of dependence-based representations for programs with heaps. In Takayasu Ito and AlbertR. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 365–386. Springer Berlin Heidelberg, 1991.

[101] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, September 1990.

[102] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[103] V. Ranganath and J. Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 2007.

[104] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

[105] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, 1994.

[106] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 360–374. Springer Berlin Heidelberg, 1989.

[107] J.M Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[108] Neil Walkinshaw Marc Roper. The java system dependence graph. In *SCAM '03*, pages 5–5, 2003.

[109] A. W. Roscoe. Csp and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, SP '95, pages 114–, Washington, DC, USA, 1995. IEEE Computer Society.

[110] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[111] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI International, Computer Science Laboratory, 1992.

[112] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, CSFW '05, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[113] Jason Sawin and Atanas Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. *Automated Software Engg.*, 16(2):357–381, June 2009.

[114] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[115] G. Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *J. Comput. Secur.*, 14(6):591–623, 2006.

[116] Scott F. Smith and Mark Thober. Improving usability of information flow security in java. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 11–20, New York, NY, USA, 2007. ACM.

[117] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. *Proceedings of the Third International Symposium on Static Analysis*, pages 332–348, September 1996.

[118] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, November 2014.

[119] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering Methodology*, 15(4):410–457, 2006.

[120] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 112–122. ACM, 2007.

[121] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[122] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[123] Kurt Stenzel, Kuzman Katkalov, Marian Borek, and Wolfgang Reif. A model-driven approach to noninterference. *JoWUA*, pages 30–43, 2014.

[124] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, 2009.

[125] Jan Vitek, R.Nigel Horspool, and JamesS. Uhl. Compile-time analysis of object-oriented programs. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 1992.

[126] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[127] Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

[128] Daniel Wasserrab and Denis Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop - VERIFY-2010*, 2010.

[129] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On pdg-based noninterference and its modular proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*, pages 31–44. ACM, June 2009.

[130] 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.

[131] Mark Weiser. Program slicing. In *IEEE Transactions in Software Engineering*, pages 352–357, 1984.

[132] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 185–195, New York, NY, USA, 2007. ACM.

[133] Reishi Yokomori, Fumiaki Ohata, Yoshiaki Takata, Hiroyuki Seki, and Katsuro Inoue. An information-leak analysis system based on program slicing. *Information and Software Technology*, 44(15):903 – 910, 2002.

[134] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–. IEEE Computer Society, 2003.

[135] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pages 312–320. IEEE Computer Society Press, 1996.

[136] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. *SIGPLAN Not.*, 39(6):145–157, June 2004.