Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

# Scalable Data-Flow Analysis through Sparsification and Precise Call Graphs

Kadiray Karakaya

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

*Doktor der Naturwissenschaften (Dr. rer. nat.)*

*Advisor*

Prof. Dr. Eric Bodden

Paderborn, September 3, 2025

**Kadiray Karakaya**
*Scalable Data-Flow Analysis through Sparsification and Precise Call Graphs*
Dissertation, September 3, 2025
Advisor: Prof. Dr. Eric Bodden

**Paderborn University**
*Research Group Secure Software Engineering*
Department of Computer Science
Faculty of Computer Science, Electrical Engineering and Mathematics
Warburger Straße 100
33098 Paderborn

# Abstract

Static data-flow analysis aims to ensure bug-free, secure, and quality software by accounting for all possible executions of a target program. Due to scalability constraints, this often entails sound over-approximations that compromise analysis precision. On the other hand, *sparsification*, an optimization technique that restricts data-flow analyses to analysis-relevant program statements, improves scalability while at the same time maintaining precision.

This thesis presents SPARSEIDE, a novel framework that realizes (data-flow) *fact-specific* sparsification for any data-flow analysis that fits the IDE (Interprocedural Distributive Environments) framework. Although IDE analyses can only be sparsified with respect to static symbols and not dynamic values, SPARSEIDE yields significantly lower runtimes and memory consumptions than the original IDE framework.

Scalability-improving approaches from the literature, including SPARSEIDE, use a fixed call-graph algorithm, without considering its impact on the downstream data-flow analysis. Through extensive empirical evaluation, this thesis shows how precise context-sensitive call graphs significantly reduce data-flow analysis runtimes.

Precise data-flow analyses reason about the heap through pointer analyses, which are also hard to scale. This thesis also presents SPARSEBOOMERANG as an application of fact-specific sparsification to demand-driven pointer analysis. SPARSEBOOMERANG realizes two different sparsification strategies that exploit the characteristics of the pointer analysis domain: a type-aware sparsification and an alias-aware sparsification.

Interprocedural data-flow analyses comprise a data-flow solver, a call graph, and a pointer analysis. This thesis shows how to scale precise data-flow analyses by considering all three components from the perspective of sparsification. Fact-specific sparsification reduces the data-flow solver's workload. As an orthogonal component, the choice of call graph significantly influences data-flow analysis scalability. Pointer analysis, which is known to be a non-distributive problem, also benefits from fact-specific sparsification when formulated within a distributive data-flow analysis framework.

# Zusammenfassung

Die statische Datenflussanalyse zielt darauf ab, fehlerfreie, sichere und hochwertige Software zu gewährleisten, indem sie alle möglichen Ausführungen eines Zielprogramms berücksichtigt. Aufgrund von Skalierbarkeitsbeschränkungen führt dies häufig zu starken Überapproximationen, die die Präzision der Analyse beeinträchtigen. Andererseits verbessert *Sparsifizierung*, eine Optimierungstechnik, die Datenflussanalysen auf analysenrelevante Programmstatements beschränkt, die Skalierbarkeit der Datenflussanalyse und bewahrt gleichzeitig deren Präzision.

Diese Arbeit stellt SPARSEIDE vor, ein neuartiges Framework, das eine (Datenfluss-) Fakt-spezifische Sparsifizierung für jede Datenflussanalyse realisiert, die im IDE-Framework (Interprocedural Distributive Environments) realisiert wird. Obwohl IDE-Analysen nur in Bezug auf Symbole und nicht auf Werte sparsifiziert werden können, erzielt SPARSEIDE deutlich geringere Laufzeiten und einen geringeren Speicherverbrauch als das ursprüngliche IDE.

Ansätze zur Verbesserung der Skalierbarkeit aus der Literatur, darunter SPARSEIDE, verwenden einen festen Call-Graph-Algorithmus, ohne dessen Auswirkungen auf die nachgelagerte Datenflussanalyse zu berücksichtigen. Anhand einer umfassenden empirischen Bewertung zeigt diese Arbeit, wie präzise kontextsensitive Call-Graphen die Laufzeiten der Datenflussanalyse erheblich reduzieren.

Präzise Datenflussanalysen analysieren den Heap anhand von Pointer-Analysen, die ebenfalls schwer zu skalieren sind. Diese Arbeit stellt SPARSEBOOMERANG als Anwendung der Fakt-spezifischen Sparsifizierung auf die bedarfsorientierte Pointersanalyse vor. SPARSEBOOMERANG realisiert zwei verschiedene Sparsifizierung-Strategien, die die Eigenschaften der Pointeranalyse-Domäne nutzen: eine type-aware Sparsifizierung und eine alias-aware Sparsifizierung.

Interprocedural Datenflussanalysen umfassen einen Datenfluss-Solver, einen Call-Graph und eine Pointer-Analyse. Diese Arbeit zeigt, wie präzise Datenflussanalysen skaliert werden können, indem alle drei Komponenten unter dem Gesichtspunkt der Sparsifizierung betrachtet werden. Eine Fakt-spezifische Sparsifizierung reduziert die Arbeitslast des Datenfluss-Solvers. Als orthogonale Komponente hat die Wahl des Call-Graphen einen erheblichen Einfluss auf die Skalierbarkeit der Datenflussanalyse. Die Pointer-Analyse, die bekanntermaßen ein nicht-distributives Problem darstellt, profitiert ebenfalls von einer Fakt-spezifischen Sparsifizierung, wenn sie innerhalb eines distributiven Datenflussanalyse-Frameworks formuliert wird.

# Acknowledgments

I want to thank my advisor, Eric Bodden, who has continuously supported me throughout my doctoral studies. Eric was always reachable to provide his invaluable expertise and guided me through the countless issues I faced in this process. I am especially grateful to him for supporting my career growth; advising me to prioritize my research, discussing with me future career options, and letting me apply my research to practice through an internship.

This work would not have been possible without the previous works of many other researchers. The main contributions of this work are motivated by Dongjie He's sparse IFDS. I am grateful, in particular, to the researchers who have open-sourced their implementations, which enabled this work. Thanks a lot to Eric Bodden for opensourcing HEROS, Johannes Späth for BOOMERANG, Steven Arzt for FLOWDROID, Dongjie He for QILIN, Linghui Luo and Felix Pauck for TAINTBENCH.

I want to thank my colleagues from Paderborn University for supporting this work. I thank Linghui Luo for her advice on managing my research and for her encouragement to implement and try out research ideas early on. I thank Martin Mory and Marcus Hüwe for their support in the SPARSEIDE work. Martin encouraged me to conclude it, and Marcus shared his expertise on the formal notation used in it. I thank Palaniappan Muthuraman for the extended discussions and for enduring many feedback loops during our collaboration on this work's empirical study on call graphs.

Besides conducting the research presented in this work, I also had the chance to lead the development of the SOOTUP framework, which helped me broaden my program analysis knowledge. I thank the SOOTUP team for their incredible support to help me finalize this project. Thanks a lot to Markus Schmidt for sharing his technical expertise, Jonas Klauke and Stefan Schott for managing its various modules and organizing the SOOTUP hackathons and tutorials.

Many of my colleagues were also supportive during this process. Thanks a lot to Mugdha Khedkar for the extended discussions and the collaboration in our first year, Ashwin Prasad for maintaining a productive shared office, and Michael Schlichtig for always sharing his advice on various topics. I also thank Enes Yigitbas for the initial hands-on experience conducting and publishing a research paper.

During my doctoral studies, I also interned at Amazon Web Services in New York. Thanks a lot to Goran Piskachev for introducing me to the researchers at AWS. During this internship, I had the chance to work with incredible researchers. Michael Emmi taught me how to implement research ideas robustly and ship them at scale. Subarno Banerjee and Liana Hadarean mentored me and helped me understand complex research concepts.

The majority of the contributions presented in this work were previously published at international conferences. I thank the anonymous conference reviewers who helped improve the contributions presented in this work. I would also like to thank Ben Hermann, Dongjie He, Mohamed Soliman, and Juraj Somorovsky for joining my defense committee.

I am grateful to my parents for teaching me how to find joy in curiosity and challenge, and for prioritizing my education under all circumstances. Finally, I thank my wife, Merve, for sharing this journey with me and her continuous support and encouragement.

# Publications

This dissertation presents original contributions, many of which have previously been published at software engineering conferences or workshops. Specifically, this dissertation contains content from the following publications, where the author of this dissertation is also the lead author:

- Kadiray Karakaya and Eric Bodden. “Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, 2024. DOI: 10.1145/3597503.3639092
  Parts of Chapter 3 were previously published in this paper.
- Kadiray Karakaya, Palaniappan Muthuraman, and Eric Bodden. “Pick Your Call Graphs Well: On Scaling IFDS-Based Data-Flow Analyses”. In: SOAP ’25. Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 43–50. DOI: 10.1145/3735544.3735587
  Parts of Chapter 4 were previously published in this paper.
- Kadiray Karakaya and Eric Bodden. “Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis”. In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2023, pp. 305–316. DOI: 10.1109/ICST57152.2023.00036
  Parts of Chapter 5 were previously published in this paper.

Chapter *Artifacts* gives a complete overview of the artifacts created during this work, including the implementations of the presented techniques and their evaluation pipelines.

# Contents

# 1 Introduction

Static program analysis techniques enable one to reason about interesting properties of computer programs. Applications of static program analysis have proven successful for diverse purposes, including compiler optimization [Kil73], program comprehension [EKS01], and developer assistance [Vas+20]. It is now an essential component of modern software engineering for assuring bug-free [Aye+08], secure [LL05], and quality software [FHP07].

Static data-flow analysis [KU76], a powerful static program analysis technique, makes it possible to detect information leaks and security vulnerabilities in a target program. It does so by aiming to account for all possible executions of the target program without executing it. This ambitious goal is theoretically unachievable [Ric53]. Fortunately, it can still be approximated within the boundaries set in two dimensions: the level of analysis result details and a runtime budget — or, in static analysis terms, precision and scalability.

Interprocedural data-flow analyses reason about data flows in the presence of method calls. They aim to produce as precise results as possible and are typically characterized by some sensitivity criteria. *Flow-sensitive* analyses keep track of the order of statements that appear inside a method body. *Field-sensitive* analyses distinguish different fields of a base object. *Context-sensitive* analyses distinguish different call sites that invoke a callee method from different calling contexts. An increased level of precision, obtained through these sensitivity criteria, typically results in scalability penalties. Such penalties are expected, as each sensitivity criterion requires the data-flow analysis solver to maintain program representations at a finer level of granularity compared to its insensitive counterpart, and later to reason about (often) exponentially more data. For instance, field sensitivity entails modeling field accesses through access paths, which could be of infinite length, e.g., due to loops [JM79], and context sensitivity entails modeling call chains, e.g., through call strings [SP+78], which can also be of infinite length due to recursion.

To adhere to scalability constraints, data-flow analyses resort to design decisions and optimization techniques that necessitate sound over-approximations. To terminate within a sensible time budget, they sacrifice on precision aspects. For instance, *flow-insensitive* analyses ignore control-flow ordering [SH97], *field-insensitive* analyses

over-approximate field accesses [YHR99], and *context-insensitive* analyses confuse different calling contexts [Ruf95]. On the other hand, interestingly, *sparsification* improves a data-flow analysis' scalability while at the same time *maintaining* its level of precision [Oh+12].

Sparsification is a well-established optimization technique in the literature for improving the scalability of data-flow analyses [CCF91; Oh+12; HL11; SX16]. The outcome of a target program's data-flow analysis depends on how it generates and transforms data of interest. Real-world programs contain many statements that have no impact on data of interest, i.e., are irrelevant to the data-flow analysis problem at hand. The crux of sparsification is restricting the analysis to the analysis-relevant program statements. Relevance of statements is typically computed through a cheaper pre-analysis phase [Shi+18; SX16; HL11], and then provided to the downstream exhaustive data-flow analysis. Initial works in this domain include SSA-based (static single assignment) sparse data-flow evaluation graphs [CCF91], and abstract interpretation-based frameworks for designing generic sparse analyses with precision-preserving guarantees [Oh+12]. Recent work on sparse IFDS [He+19] has demonstrated that further sparsification is possible by exploiting the data-flow facts that become available during the analysis runtime.

Sparse IFDS [He+19] sparsifies the data-flow analyses formulated within the IFDS (Interprocedural, Finite, Distributive, Subset) framework [RHS95]. It has shown that one can often greatly speed up data-flow analysis by computing data flows not for every edge in the program's control-flow graph but instead only along the definition-use chains of the specific data-flow facts of interest. This yields a so-called (data-flow) *fact-specific* sparsification. Fact-specific sparsification [He+19] has been shown to "sparsify" IFDS-based taint analyses. It is extraordinarily effective because the taint-state of one variable does not depend on those of others. This allows one to soundly omit many flow-function computations.

## 1.1 Contributions

Figure 1.1 shows the essential components of precise interprocedural data-flow analyses: ① a data-flow solver to reason about interesting properties of a target program, ② a call graph to guide the solver through possible method calls, and ③ a pointer analysis to reason about heap allocations. This thesis shows how to improve the scalability of precise interprocedural data-flow analysis by considering all three components from the perspective of sparsification.

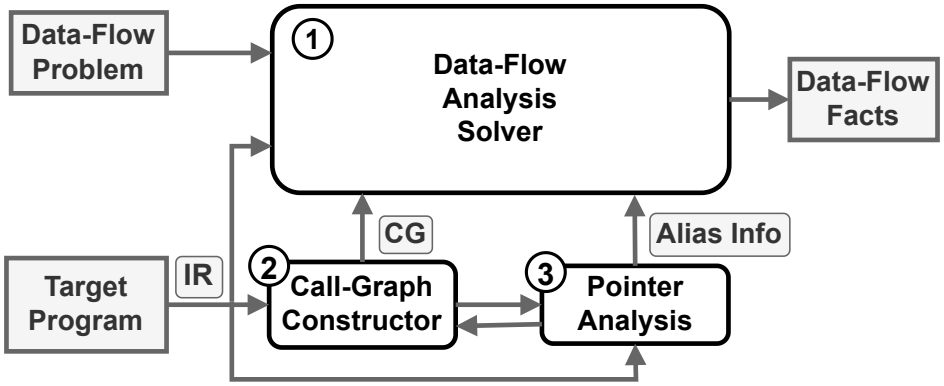



**Figure 1.1:** Components of Interprocedural Data-Flow Analysis

Interprocedural data-flow analysis frameworks like IFDS enable defining a *data-flow problem* depending on the property that one wants to extract from a *target program*. A target program is represented with an intermediate representation (*IR*), typically in the form of a control-flow graph (CFG). The IR is generated by a program analysis framework, which we omit in this diagram. All system components use the same IR. The *call-graph constructor* builds a call graph (*CG*) to represent the calls between different methods of the target program. The *pointer analysis* provides *alias information* to the data-flow analysis solver, so it can reason about variables that may point to the same memory location. Depending on the algorithm, the call-graph constructor may also use the pointer information, and the pointer analysis utilizes the call graph. The *data-flow analysis solver* utilizes all of this information to solve the given data-flow problem in the form of data-flow facts.

**Contribution 1: The SparseIDE Framework** As the first contribution, this thesis presents SparseIDE, a novel framework that realizes *fact-specific* sparsification for any data-flow analysis that fits the IDE [SRH96] (Interprocedural Distributive Environments) framework. Unlike IFDS, IDE comprises distributive problems with large or even *infinitely broad* domains, such as typestate analysis [Fin+08; Li+22] or linear constant propagation [Cal+86; Oct+16]. IDE models data-flow facts as environments [SRH96], which are mappings from symbols (often program variables) to domain values. The IDE framework fits the data-flow analysis problems that go beyond mere symbol reachability, i.e., when the values associated with symbols are also interesting. For instance, a linear constant propagation analysis is interested in finding the constant integer values program variables might hold, where the symbol to value mapping at the statement `x = 42` can be modelled with the mapping, $x \mapsto 42$. SparseIDE generalizes the recent work on sparse IFDS. It enables efficient sparsification, even in the presence of arbitrarily large value domains. We also show

the limits of sparsification in IDE: while one can effectively sparsify with respect to static symbols, such sparsification cannot be performed with respect to dynamic values. We formalize SPARSEIDE and show how this formalization also subsumes IFDS-based data-flow analysis problems as a special case. Although IDE analyses can only be sparsified with respect to symbols, SPARSEIDE yields significantly lower runtimes and often also lower memory consumption compared to the original IDE.

**Contribution 2: Call-Graph Precision's Impact on Data-Flow Analysis Scalability** The IDE framework reduces data-flow analysis problems to graph reachability on an exploded supergraph. The exploded supergraph is a data-flow graph induced by the interprocedural control-flow graph (ICFG) for the whole program. Its nodes are pairs of program statements and data-flow facts, and its edges correspond to statements' effects on the data-flow facts. Data-flow facts are considered to hold at statements, if and only if they are reachable at those statements [SRH96]. In the exploded supergraph, *intraprocedural edges* model symbol reachability within method boundaries, whereas *interprocedural edges* model symbol reachability across methods. Fact-specific sparsification, including SPARSEIDE and sparse IFDS[He+19], only reasons about the intraprocedural edges. The interprocedural edges, on the other hand, are obtained from a fixed call-graph algorithm, whose impact on the downstream data-flow analysis is surprisingly neglected in the literature. As the second contribution, this thesis presents an extensive empirical evaluation of call graphs' impact on data-flow analysis scalability. We assess the impact of 31 different call graphs on IFDS-based data-flow analyses. We show that although precise context-sensitive call graphs can be expensive to build, they significantly reduce the total data-flow analysis runtimes and, in some cases, improve data-flow analysis precision.

**Contribution 3: Query-Specific Sparsification for Demand-Driven Pointer Analysis** To produce precise results, data-flow analyses must account for aliasing between the symbols in a target program. To resolve aliasing, precise data-flow analyses rely on pointer analyses, which are also hard to scale. Compared to exhaustive ones, demand-driven pointer analyses promise improved scalability because they compute alias information only when client data-flow analyses raise a demand. As the third contribution, this thesis shows how fact-specific sparsification can be applied to demand-driven pointer analysis. To this end, we introduce SPARSEBOOMERANG as an extension to the BOOMERANG [Spä+16] pointer analysis framework. SPARSEBOOMERANG provides two sparsification strategies that utilizes information specific

to the pointer analysis domain: a type-aware sparsification (TAS) and an alias-aware sparsification (AAS). We show that SPARSEBOOMERANG, with both strategies, maintains the precision of BOOMERANG while improving its scalability.

## 1.2 Overview

In this thesis, we first show how fact-specific sparsification improves data-flow analysis scalability. Second, we show how the choice of call graph, as an orthogonal component, significantly influences data-flow scalability. Third, we show how pointer analysis, which is known to be a non-distributive problem, also benefits from fact-specific sparsification when formulated as a distributive data-flow analysis problem.

The remainder of this thesis is structured as follows. Chapter 2 provides the background information upon which this thesis builds. It explains IFDS [RHS95] and IDE [SRH96], the two popular data-flow analysis frameworks that solve a set of (distributive) data-flow analysis problems. Moreover, it briefly introduces the concepts of call-graph construction, demand-driven pointer analysis using BOOMERANG, and sparse data-flow analysis. Chapter 3 introduces the SPARSEIDE framework (*Contribution 1*), and explains how it is implemented, extending the HEROS IDE solver [Bod12]. Chapter 4 presents the empirical evaluation of call-graph precision impact on the data-flow analysis scalability (*Contribution 2*). Chapter 5 presents SPARSEBOOMERANG (*Contribution 3*), and its two sparsification strategies that realize fact-specific sparsification for pointer analysis at two granularity levels. Chapter 6 concludes this thesis.

# 2 Background

This chapter presents the key concepts on which this thesis is based. This thesis proposes novel techniques for improving the scalability of interprocedural data-flow analyses. We first introduce data-flow analysis and its key concepts in Section 2.1. We then explain interprocedural data-flow analysis and two state-of-the-art interprocedural data-flow analysis frameworks, IFDS [RHS95] and IDE [SRH96], in Section 2.2. Interprocedural data-flow analysis requires call graphs to reason about method invocations. Section 2.3 explains call-graph construction and its relevance to data-flow analysis. Further, data-flow analyses require pointer information to reason about the heap. Section 2.4 explains how pointer analysis can aid data-flow analyses in producing precise results. Finally, Section 2.5 introduces the concept of sparse data-flow analysis.

## 2.1 Data-Flow Analysis

Static data-flow analysis aims to understand the flow of *certain data* throughout *all possible executions* of a program on *all possible inputs* [KSS09]. It does so without executing the target program. Therefore, many concrete runtime values are unknown during static data-flow analysis, e.g., due to incomputable expressions or the nonexistence of program inputs and configuration options. By using a simple analysis scenario as a motivating example, this section presents how static data-flow analysis overcomes key challenges to producing useful information.

```
1  void main(String[] args){
2    String s = secret();
3    String p = s;
4    if(args[0]!=null){
5      s = "***";
6    }
7    log(s);
8  }
```

**Figure 2.1:** A simple case of information leak

**A simple case of information leak.** Consider the example program in Figure 2.1. The `main` method is provided with a list of arguments, `args`. A variable `s` is assigned sensitive information returned by `secret()`. `s` then passes this information to `p` with the assignment `p = s`. When the `args[0]` is set, `s` no longer contains the secret. In the end, the program logs, i.e., prints to an external file, the content of `s`. Assume we want to detect whether the secret information might be leaked when this program is executed.

**Which data should be tracked?** In this example, the data of interest is the return value of `secret()` and the statement that can potentially leak data is `log(s)`. Detection of such leaks is formulated as a specific kind of data-flow analysis problem, known as *taint analysis* [EL02]. Taint analysis needs to know about a set of *source* and *sink* statements, where sources *taint* program variables, and a leak is detected when tainted variables reach sinks [HCF05].

During program execution, concrete runtime values are computed, whereas during data-flow analysis, it suffices only to compute the necessary information regarding the data of interest. From the perspective of taint analysis, the concrete value returned by `secret()` is irrelevant. The analysis only needs to know whether it is sensitive information, i.e., a taint source. In data-flow analysis, this information is typically encoded within an abstract *domain* [CC77], consisting of a partially ordered set (lattice) of possible domain elements. A taint analysis's abstract domain can be defined as $L = \{\top, \bot\}$, where $\top$ denotes *untainted* and $\bot$ *tainted*[1]. Data-flow analysis associates program variables, i.e., the set of symbols $D$, with the abstract domain values $L$. Such a mapping, i.e., $d \in D$ and $l \in L$, where $[d \to l]$ constitutes a *data-flow fact*. Since the value domain of taint analysis is a binary lattice, a more compact representation only keeps track of the tainted symbols, e.g., the existence of a symbol in the end signifies that it is tainted. This enables the data-flow facts to be reduced to only $D$.

**How to keep track of data?** The definition of an abstract domain clarifies what kind of data-flow facts an analysis can yield; for instance, a taint analysis concludes that a subset of symbols is tainted. Next, the data-flow analysis needs to know about the impact of program statements on data-flow facts. Impacts of statements are represented with so-called *data-flow functions*. In taint analysis, flow functions can

[1]The reverse of this notation is also used in the literature, where $\top$ denotes tainted and $\bot$ denotes untainted. In this thesis, we prefer this notation after Sagiv, Reps, and Horwitz [SRH96]. They use $\top$ for an unknown value and $\bot$ for a non-constant, i.e., the most sound state in integer constant propagation analysis.

*generate*, *kill*, or *propagate* a taint. For instance, the source statement `s = secret()` generates the data-flow fact $s$, meaning after executing this statement, the symbol `s` is tainted. Assume $S$ is the set of data-flow facts before each statement, the corresponding flow function is:

$$f(S) = S \cup \{s\}$$

Similarly, the statement `s = "***"` kills the fact $\{s\}$, i.e., removes $\{s\}$ from the set, and can be shown with the flow function:

$$f(S) = S \setminus \{s\}$$

The statement `p = s` propagates the taint from `s` to `p` if `s` is tainted. Therefore, its corresponding flow function is:

$$f(S) = \begin{cases} S \cup \{p\} & \text{if } s \in S \\ S \setminus \{p\} \end{cases}$$

**How to account for all possible execution paths?** Figure 2.2 shows the control flow graph (CFG) of the example program in Figure 2.1. The expression `args[0]!=null` has two successors, depending on the outcome of this expression; during program execution, only one branch can be taken. On the other hand, during program analysis, this expression cannot be computed because the input, i.e., `args`, simply cannot be interpreted. Therefore, regardless of the outcome of the conditional expression, data-flow analysis has to account for both cases. This necessitates a design decision: should the analysis assume `s` *may* contain the secret, or only warn if and only if `s` *must* contain the secret?

To err on the safe side, data-flow analyses often opt for a sound approximation. Figure 2.2 contains an additional node $\sqcap$ representing a merge point. To achieve a sound result, the merge operation must be *set union* over the set of symbols $D$. As a result, after the merge point, information from both branches can be combined, i.e., $\{p, s\} \cup \{p\}$, which yields $\{p, s\}$, and thus, the analysis detects a potential leak at `log(s)`.

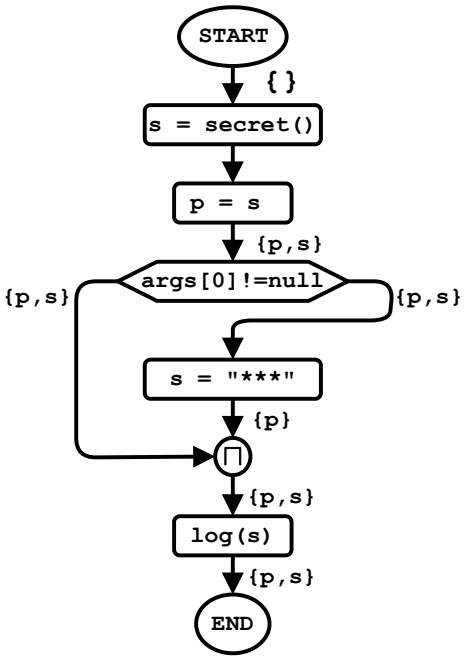



**Figure 2.2:** Control flow graph (CFG) of the code in 2.1

**When to terminate the analysis?** During program execution, infinite loops may occur; they are sometimes even required, for instance, for operating systems, which should run indefinitely. On the other hand, data-flow analyses are expected to *terminate*. Assume replacing `if(args[0]!=null)` with `while(args[0]!=null)` in the example program in Figure 2.1, causing an infinite loop while the condition holds. Since the data-flow analysis is only interested in data-flow fact computations in the abstract domain, it can safely ignore redundant iterations that do not alter the analysis state regarding this domain. The maximal fixed-point (MFP) algorithm in the monotone framework [KU77] proposes a solution for such computations. MFP solution is guaranteed to be a sound approximation of the ideal solution, which is known as the *meet over all paths* (MOP) solution [KU77]. MOP describes a solution where data-flow equations for all the different paths in a target program are solved individually, which is incomputable for monotone frameworks [KU77].

Assuming $n_0$ is the entry node of a method's CFG, $f_n$ is the flow function corresponding to node $n$, and $pred(n)$ returns the predecessors of $n$ in the CFG. The maximal fixed-point solution is defined as:

$$\begin{aligned} \mathrm{MFP}(n_0) &= \text{initial value} \\ \mathrm{MFP}(n) &= \sqcap \{ f_p(\mathrm{MFP}(p)) \mid p \in \mathrm{pred}(n) \} \end{aligned}$$

Monotone data-flow frameworks are defined as $M = (L, \sqcap, F)$, where $L$ is a finite-height semi-lattice with meet operator $\sqcap$, $F$ is the set of monotone flow functions associated with $L$, that satisfies the *monotonicity* condition [KU77]:

$$\forall x, y \in L.\, \forall f \in F :\ f(x) \sqcap f(y) \sqsubset f(x \sqcap y) \tag{2.1}$$

This condition ensures that merging data-flow facts from different branches and then applying the following flow function leads to a safe approximation of applying the flow functions and then merging the resulting facts. For instance, in the taint analysis domain, $L = \{\top, \bot\}$, where $\top$ corresponds to untainted and $\bot$ corresponds to tainted, a safer approximation is to prefer $\bot$ over $\top$, i.e., the least element in this lattice. This ensures *soundly* maintaining a possible taint under uncertainty. In this case, being sound means assuming a data-flow fact *is* tainted if it *may be* tainted. Although such a sound approximation leads to precision loss, it enables computability by allowing one to merge data-flow facts as soon as possible instead of maintaining different sets of data-flow facts for each possible path in a CFG.

## 2.2 Interprocedural Data-Flow Analysis

The monotone framework we explained in the previous section has historically been applied to intraprocedural data-flow analysis problems, typically for compiler optimization [Kil73]. While intraprocedural data-flow analysis focuses on obtaining data-flow information from individual procedures, interprocedural data-flow analysis connects such information to reason about the whole program. It does so by summarizing procedure-level information, and plugging them in their respective call sites [Bar78]. Interprocedural data-flow analysis requires reasoning about the data flows in the presence of method invocations. An inherent challenge of handling method invocations is distinguishing different calling contexts, i.e., context-sensitivity. This section explains IFDS (Interprocedural, Finite, Distributive, Subset) [RHS95] and IDE (Interprocedural Distributive Environments) [SRH96] frameworks for interprocedural flow- and context-sensitive data-flow analysis.

### 2.2.1 IFDS

IFDS [RHS95] represents data-flow analysis problems as graph reachability on an exploded supergraph, whose nodes are pairs of program statements and data-flow facts. The individual edges in the exploded supergraph constitute *flow functions*; they show each statement's effect on each data-flow fact's reachability. A flow function determines whether a data-flow fact is being generated, propagates to the next statement, spawns another fact, or gets killed.

Both IFDS and IDE exploit the *distributivity* property of certain types of data-flow analysis problems. Such problems are a subset of those that can be defined with the monotone framework. Following the Equation 2.1, distributivity property is defined as:

$$\forall x, y \in L.\, \forall f \in F :\ f(x) \sqcap f(y) = f(x \sqcap y) \tag{2.2}$$

This condition ensures that the MFP solution, i.e., merging data-flow facts from different branches and then applying the following flow function leads to the same result as the MOP solution, i.e., applying the flow functions and then merging the resulting facts. This important property allows one to generate compact procedure summaries, which speeds up the analysis.

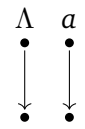
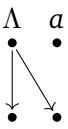
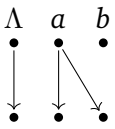
| $f_{id}$: $\lambda S.S$ | $f_{gen}$: $\lambda S.(S \cup \{a\})$ | $f_{as}$: $\lambda S.$if $a \in S$: $(S \cup \{b\})$ else $(S \setminus \{b\})$ |
|---|---|---|
|  |  |  |



**Figure 2.3:** Flow functions (reproduced from [RHS95]).

**Flow function syntax.** Figure 2.3 shows how the flow functions are represented as edges in the exploded supergraph. To define the flow functions we use the following notation: $f_n : \lambda S.S'$, where $f_n$ is the name of the flow fuction for the statement $n$, $S$ is the set of data-flow facts before $n$, and $S'$ shows the the impact of $n$ on $S$.

In Figure 2.3, the data-flow fact above the edge means that it holds before applying the function; the fact below means that it holds after. A special fact, $\Lambda$, holds always. Facts connected to it are newly generated. The identity function, $f_{id}$, leaves data-

flow facts unchanged. The function $f_{gen}$ shows the case where data-flow fact *a* is being generated. The function $f_{as}$ shows how the existing fact, *a* creates another fact, *b*, e.g., at an assignment, `b = a`.

### 2.2.2 IDE

The IDE [SRH96] framework generalizes the IFDS [RHS95] framework by computing *environments*, i.e., mappings from program symbols to data-flow analysis domain values. It does so in two phases: first, it determines whether symbols are reachable, just like IFDS, and then computes their values. IDE achieves this by annotating the individual exploded supergraph edges with so-called *edge functions*, which constitute environment transformers.

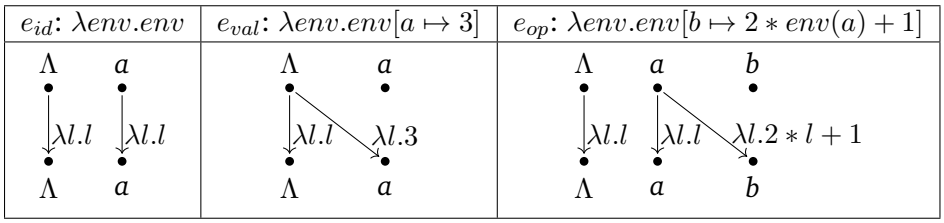


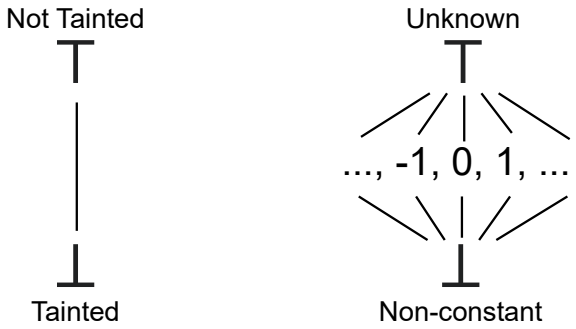

**Figure 2.4:** Edge functions (reproduced from [SRH96]).

**Edge function syntax.** Figure 2.4 shows how the edge functions annotate the edges in the exploded supergraph. To define the edge functions we use the following notation: $e_n : \lambda env.env'$, where $e_n$ is the name of the edge fuction for the statement $n$, $env$ is the environment, i.e., a set of mappings from data-flow facts to domain values, before $n$, and $env'$ shows the the impact of $n$ on $env$.

In Figure 2.4, The environment transformer $e_{id}$ keeps the values as they are. $e_{val}$ shows the case where data-flow fact, *a* is mapped to a domain value, e.g., through a constant assignment, `a = 3`. $e_{op}$ shows how the value of *b* is calculated depending on the value of *a*, e.g., through a linear arithmetic operation, `b = 2*a + 1`. IDE can only compute *linear* equations precisely.

IFDS and IDE apply to a wide class of data-flow analysis problems. IFDS requires data-flow problems to be defined with distributive flow functions over the meet operator. Many reachability problems, such as taint, reaching definitions, or live variables analysis, fall into this category. IDE, on the other hand, also requires data-flow problems to be expressed with distributive environment transformers. IFDS

suits better the problems with a binary value domain, e.g., taint analysis where the domain simply consists of two values, *tainted* or *not tainted* [Arz+14]. It has been applied to more complex domains, e.g., for typestate analysis where the domain contains arbitrary object states [NL08]. The drawback of IFDS is that it forces one to represent non-trivial data-flow facts as symbol-value pairs. This blows up the data-flow fact space with the increasing domain size. Because of this representation, IFDS's runtime performance not only depends on the size of the set of symbols, but also on the size of the value domain. Figure 2.5 shows a comparison of the binary domain for taint analysis and the integer domain for constant propagation analysis. In the case of taint analysis it suffices for IFDS to associate each corresponding symbol with the taint state ($\bot$). However, in the case of integer constant propagation analysis, it needs to associate each corresponding symbol with all the possible integer values it might be take.




**(a)** Binary Domain **(b)** Integer Domain

**Figure 2.5:** Comparison of Lattices for the Domains of Taint Analysis and Integer Constant Propagation Analysis

Figure 2.6 shows a comparison of the data-flow graphs that IFDS and IDE would generate during the integer constant propagation analysis of a simple loop that increments an integer. Theoretically IFDS may not terminate when the value domain is infinitely broad because with each iteration it would generate a new data-flow fact as a symbol-value pair. For instane, in Figure 2.6, the symbol `i` is being associated with a different integer value with each iteration. IDE, on the other hand, restricts data-flow facts to static symbols and computes their (approximated) runtime values using the edge functions along the path where the symbols are reachable in the exploded supergraph. IDE's representation is more efficient than that of the IFDS. Therefore, although IDE-based analysis may also apply the edge functions multiple times, e.g., to merge to some larger interval, it does not lead to generating new data-

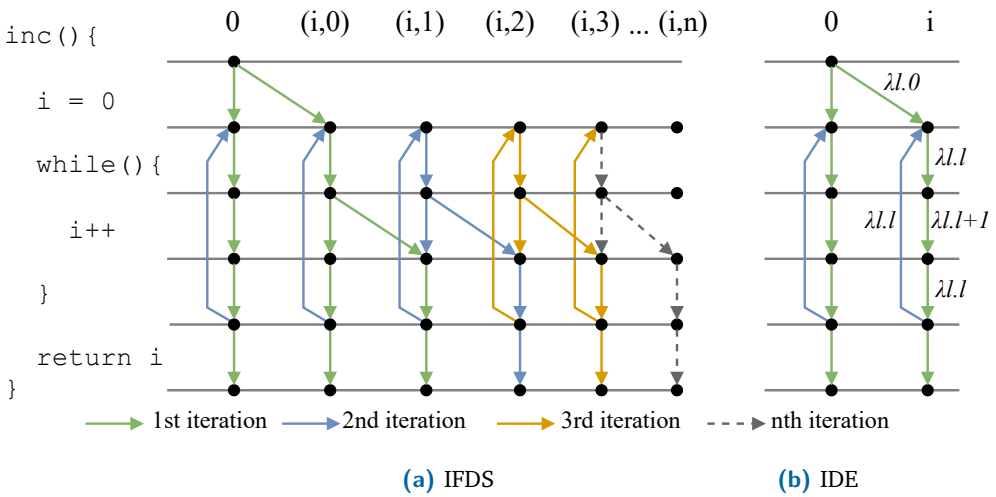



**Figure 2.6:** Comparison of Data-Flow Graphs for Integer Constant Propagation Analysis using IFDS and IDE

flow facts with each iteration. The advantage of IDE over IFDS is that it can terminate efficiently even with infinitely broad value domains—only the set of symbols must be finite[2].

## 2.3 Call-Graph Construction

A call graph is a special kind of data structure that represents method calls that might occur during the execution of a target program. Its nodes model the methods, and the edges between the methods model the possible invocations between them. As shown in Figure 1.1, call-graph construction is an integral component of interprocedural data-flow analysis.

Due to language features and limitations of statically computable information, static call-graph construction has to approximate when resolving potential callees at a call site. For instance, object-oriented languages like Java allow polymorphic calls on interface instances, whose exact receiver object is only known at runtime.

Figure 2.7 shows the inference rules for generic call-graph construction. $R$ is the set of reachable methods. Rule ENTRY states that entry methods are reachable. Rule CALL states that if a method $m$ is reachable, contains a call site $s$ in the form $b.foo(...)$, and resolvePotentialCallee($foo$) can resolve a method $m'$, there exists an

[2]Further information on this can be found at `https://www.youtube.com/watch?v=0uMHX3UY9bg`

$$\frac{m \in E, E \text{ is the set of entry methods}}{m \in R}\ \textsc{Entry}$$

$$\frac{m \in R \quad \text{call site } s : b.foo(\ldots) \text{ in method } m \quad \text{resolvePotentialCallee}(foo) = m'}{s \to m'}\ \textsc{Call} \qquad \frac{s \to m}{m \in R}\ \textsc{Reachable}$$

**Figure 2.7:** Inference rules for generic call-graph construction (adapted from [Ali+14])

edge from $s$ to $m'$. $m'$ is a static approximation of $foo$ depending on the underlying call-graph algorithm used in resolvePotentialCallee(). Finally, rule REACHABLE states that if there is an edge from a call site $s$ to a method $m$, $m$ is reachable.

A target program's call graph essentially steers the data-flow analysis solver through different calling contexts, i.e., the snapshot of the call stack at run time [KSS09]. Therefore, the call graph potentially also impacts the performance of the data-flow analysis. In this thesis, we empirically investigate the impact of call-graph precision on interprocedural data-flow analysis.

## 2.4 Demand-Driven Pointer Analysis

Pointer analysis determines which program variables can point to which objects at runtime. It is required in real-world analysis problems where multiple program variables frequently point to the same object. Two variables that point to the same object are called *aliases*. Such alias information is crucial for a precise data-flow analysis to track indirect data-flows through the aliases. Pointer analysis is not distributive [Ram94] at an assignment `x.f = t`; one must assign aliases of `t` to the `f`-fields of *all* the aliases of `x`. For this reason, one cannot usually soundly handle all aliases independently, and distributive frameworks like IFDS and IDE are not applicable by default.

As opposed to exhaustive pointer analysis, demand-driven pointer analysis [HT01] is performed only for variables on which a demand, e.g., a pointer or alias query, is raised. It computes just enough information to satisfy the query. Interestingly, as Späth et al. showed [Spä+16], one can decompose a flow-sensitive pointer analysis such that when queries raise sub-queries at "points of indirection" (POI), e.g., at reads and writes to/from the heap, the evaluation of those sub-queries *does* become a distributive and thus distributively solvable analysis problem.

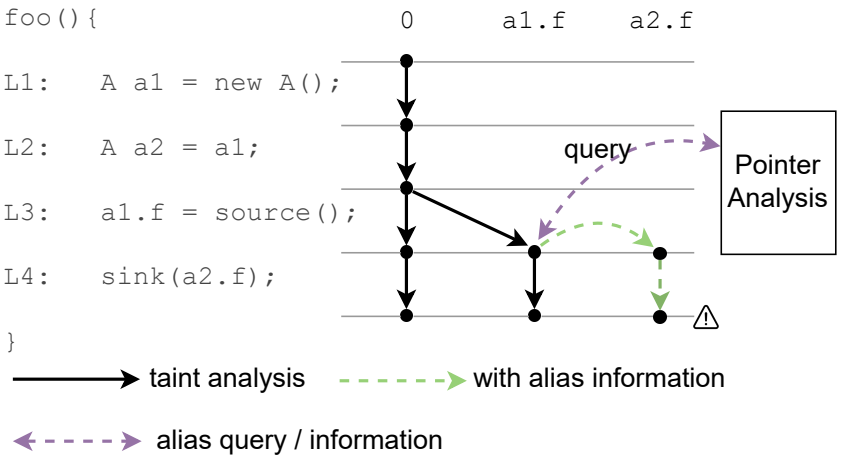



**Figure 2.8:** Data-Flow Graphs for Taint Analysis with Alias Information (Simplified from [Spä+16])

Figure 2.8 shows the data-flow graphs that a taint analysis would produce with alias information. At line `L1`, `a1` is being assigned with an instance of `A`. At `L2`, `a2` and `a1` alias through an assignment. At `L3`, `a1.f` points to sensitive information, i.e., tainted by a taint source, which implicitly causes `a2.f` to hold the same information at runtime. Therefore, to be sound, the static data-flow analysis has to taint both `a1.f` and `a2.f`. To know that the analysis must taint `a2.f` at `L3`, it must know that `a1` and `a2` alias at that point. Pointer analyses provide such information.

BOOMERANG [Spä+16] is a state-of-the-art context-, flow-, and field-sensitive stand alone demand-driven pointer analysis. It allows data-flow analyses to query alias information at program locations where the aliases of variables need to be considered, for instance, at field store and load statements. In this thesis, we use BOOMERANG to resolve aliases during data-flow analysis, and also improve BOOMERANG's scalability through sparsification.

## 2.5 Sparse Data-Flow Analysis

Data-flow analyses can be instantiated to solve a wide range of analysis problems. Depending on the problem, flow functions, which model the effects of the program statements, differ. For instance, the effects of arithmetic operations are important for constant propagation analysis, but not for taint analysis. Sparsification approaches can speed up the analysis by instructing it to ignore statements that have no effect *in the context of that particular analysis problem*.

The sparsification techniques presented in this thesis are based on the state-of-the-art fact-specific sparsification by He et al. [He+19]. In data-flow analysis, flow functions that do not affect *any* facts at a given statement are known as *id* functions. During data-flow analysis, many non-*id* flow functions, in fact, behave as *fact-specific id* functions: while they may affect some data-flow facts, they are irrelevant to many others. Because IFDS, due to its distributivity, evaluates data-flow facts independently of each other, one can, during the evaluation of a data-flow fact $d$, safely disregard a flow function $f$ if it is a $d$-specific identity function. Using this observation, He et al. [He+19] introduced the sparse IFDS algorithm, which includes fact-specific sparsification: it creates on-demand SCFGs *specific* to each data-flow fact that is being propagated. Facts are propagated to their next use point within their individual SCFGs. The original IFDS algorithm [RHS95] instead propagates data-flow facts to *all* reachable program points. Fact-specific sparsification is explained in detail in Section 3.3.

Although the main idea of sparsification is widely accepted in the literature, there is no agreement on a common term that defines *the sparse graph*. In previous works, it is referred to as *Sparse Evaluation Graph (SEG)* [CCF91], *Compact Evaluation Graph* [Ram02], *Sparse Value-Flow Graph (SVFG)* [SYX12], and *Sparse Control-Flow Graph (SCFG)* [He+19]. This thesis uses the term "SCFG" to refer to such sparse graphs.

# 3 SPARSEIDE: Symbol-Specific Sparsification of IDE Problems

This chapter introduces the first major contribution (*Contribution 1*) of this thesis, SPARSEIDE, a novel framework that realizes fact-specific sparsification for data-flow analysis problems that can be modeled within the IDE (Interprocedural Distributive Environments) framework [SRH96]. This also subsumes all the problems that fit the IFDS (Interprocedural, Finite, Distributive, Subset) framework [RHS95], as a special case. In the context of IDE, instead of the term *fact-specific*, we use *symbol-specific*, which better represents the approach, as the sparsification can only be done with respect to symbols, and not to their associated values.

IDE (Interprocedural Distributive Environment) [SRH96], with its extensions [NLR10; AB14; SAB17], is a state-of-the-art, precise interprocedural static analysis framework. It covers a wide class of data-flow problems ranging from variations of classical taint analysis [JKK06] to typestate [Fin+08; Li+22] and constant propagation [Oct+16] analyses. IDE represents data-flow analysis problems on an *exploded supergraph* and models data-flow facts as environments. Environments are mappings from symbols (often program variables) to domain values. The exploded supergraph is a data-flow graph induced by the interprocedural control-flow graph (ICFG) for the whole program. Its nodes are pairs $(s, d)$ of program statements and data-flow facts. A data-flow fact $d$ holds at a statement $s$ if in the exploded supergraph the corresponding node $(s, d)$ is reachable from the start node. The edges of the exploded supergraph represent the effects of program statements on a data-flow fact. IDE computes over the exploded supergraph by tracking all data-flow facts *densely* across all program points. As previous work [He+19; Arz21; Li+21; Yu+20] has shown, this approach does not scale well for large-scale real-world programs. A key observation is, however, that in practice, many program statements do not affect the analysis result. Such statements thus can be safely ignored, e.g., by *sparsifying* the exploded supergraph.

Sparsification is a well-known technique for scaling data-flow analyses [Shi+18; SX16; HL11; Oh+12; Sui+11; HL09] while still maintaining their precision. Sparsification approaches create sparse versions of the original CFGs of a target program by removing statements irrelevant to the analysis and then computing over the sparse CFGs. Recent work on on-demand sparsification [He+19] improves data-flow analysis performance further by utilizing the information available during the analysis. It accelerates taint analysis by computing over sparse CFGs specialized to individual data-flow facts. Yet, it demonstrates sparsification on IFDS-based problems that focus on mere symbol reachability, without considering value computation.

## 3.1 Motivation

The IFDS [RHS95] framework is the "small brother" of IDE. It reduces the data-flow analysis problems to a pure graph reachability problem. Yet, IFDS is limited to data-flow problems with finite domains: all IFDS problems can be encoded as IDE problems, but only a subset of IDE problems can be encoded as IFDS problems [SRH96]. Consider the statement `a = a + 1` as an example. Here, using IFDS, one can encode a simple taint analysis inferring that `a` is tainted/reachable after the statement if and only if it was previously tainted/reachable. Efficient computation of `a`'s numeric value, however, requires one to *compute values* within the infinitely broad domain of integers, going beyond pure reachability.

As we show, this has implications for sparsification: while the statement `a = a + 1` can be safely considered irrelevant w.r.t. `a`'s reachability, and will be disregarded in sparsification approaches for IFDS [He+19], it *is* a relevant statement when constant propagation is considered: it changes `a`'s value. This observation is not limited to constant propagation analysis; it applies to other data-flow analysis problems that require value mappings. For instance, a sparse typestate analysis must retain statements that alter a symbol's associated state value. The recent work on SPARSEDROID, i.e., on sparse IFDS [He+19], does not *generalize* to handle such cases. This motivated us to investigate whether fact-specific sparsification can be generalized to the IDE framework, enabling efficient sparsification, even in arbitrarily large value domains.

## 3.2 Contributions

In this chapter, we formalize SparseIDE, and show how this formalization also covers IFDS data-flow analysis problems as a special case. We implement SparseIDE in a tool called SparseHeros, extending the popular Heros IDE solver [Bod12]. We compare both implementations in terms of performance and show that sparsification maintains correctness. To this end, we implement a linear constant propagation analysis client that uses both implementations. To validate SparseHeros's correctness, we run both on ConstantBench, a novel microbenchmark suite for integer linear constant propagation analysis. To evaluate its performance impact, we run the analysis client on real-world Java libraries using both Heros and SparseHeros. The analysis client produces the same results in both cases while terminating significantly faster when using SparseHeros.

To summarize, this chapter presents the following original contributions:

- A formalization of SparseIDE and its implementation in SparseHeros on top of Heros and Soot [Val+99],
- its correctness evaluation on the ConstantBench micro-benchmark suite for linear constant propagation analysis, and
- its performance evaluation on real-world Java libraries.

## 3.3 Fact-Specific On-Demand Sparsification

Data-flow analysis techniques aim to produce precise results while remaining scalable within a reasonable time budget. Techniques that prioritize scalability often resort to sacrificing precision aspects: flow-insensitive analyses ignore control-flow ordering [YSX14], field-insensitive analyses approximate field accesses [Deu94], and context-insensitive analyses do not distinguish different calling contexts [Li+20]. Sparse data-flow analyses, on the other hand, often improve a *dense* data-flow analysis's scalability while *maintaining* its precision. They sparsify a target program's control-flow graph by removing program statements that *provably* do not affect the analysis result. Sparsification often uses a cheaper pre-analysis stage to aid a more expensive analysis [Shi+18; SX16; HL11]. Recent *on-demand* sparse data-flow analyses sparsify further by exploiting the information that is only available during analysis runtime [He+19]. In this context, there exist no explicit demand in the

sense of demand-driven program analysis. The technique is called on-demand because the control-flow graphs are sparsified only when a data-flow fact needs to be propagated, therefore, the demand implicitly originates from the data-flow analysis itself.

When IFDS and IDE compute a data-flow fact's reachability, starting from the statement that generates the data-flow fact, they propagate it along all statements as long as it is not killed. At each statement, they check whether the statement is relevant for all the data-flow facts that have reached it. Figure 3.1 shows how the reachability is computed for an example constant-propagation analysis setting. The *fact-specific id edges* and *non-id edges* show the edges that IFDS and IDE create when propagating data-flow facts. The data-flow facts actually only need to be propagated to the *required nodes*. For instance, data-flow fact **a** only needs to propagate to the statement `b = a;`, all other statements are irrelevant to the taint status of **a**. Similarly, **b** only needs to propagate to the statement, `c = b + 1`. Based on this observation, He et al. [He+19] introduced the sparse IFDS algorithm in their implementation SPARSEDROID. Instead of propagating all the data-flow facts to the next statement, it propagates them simply to the next statement that uses the respective fact. Sparse IFDS keeps all *non-id edges* and replaces the *fact-specific id edges* with *sparse id edges*, effectively keeping all *required nodes* and skipping over all *redundant nodes*.

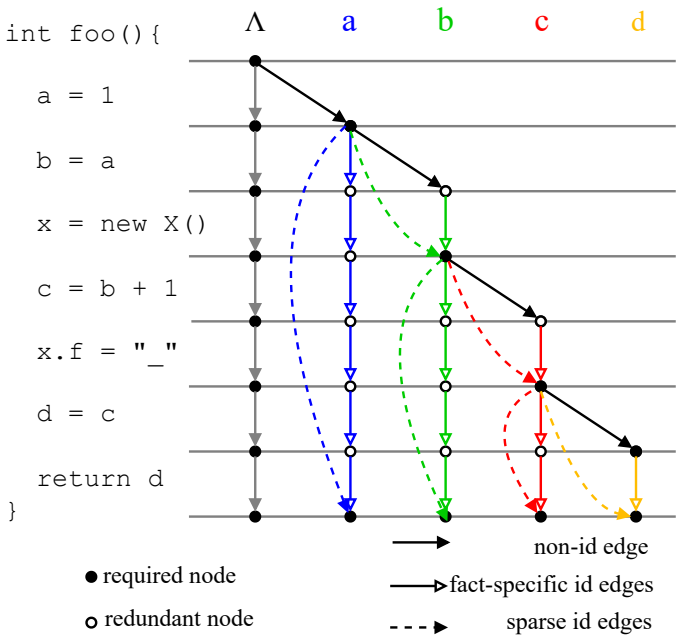



**Figure 3.1:** Original and sparse propagations after applying fact-specific on-demand sparsification.

Fact-specific on-demand sparsification allows effective propagation of the data-flow facts along the sparse CFGs specific to them. This is not limited to data-flow analysis. In Chapter 5, we also apply it to pointer analysis, where the variable in alias queries is treated as the initial data-flow fact and propagated along its query-specific sparse CFGs. Fact-specific on-demand sparsification has only been applied to the analysis problems that deal with fact reachability. With SPARSEIDE, we expand the scope of fact-specific on-demand sparsification to include the data-flow analyses that compute over an additional value domain, specifically IDE.

## 3.4 Symbol-specific On-Demand Sparsificiation with SPARSEIDE

In this section, we first explain the original IDE algorithm [SRH96] in detail. We then introduce the SPARSEIDE algorithm by highlighting the modifications to the original IDE algorithm.

### 3.4.1 The Original IDE Algorithm

Sagiv et al. [SRH96] define an IDE problem instance formally as $IP = (G^*, D, L, M)$, where

- $G^*$ is the program supergraph (ICFG), which consists of control flow graphs (CFG), $G_p$ of individual procedures,
- $D$ is a finite set of program symbols,
- $L$ is a finite-height lattice (which can be infinitely broad), and
- $M : E^* \xrightarrow{d} (Env(D, L) \rightarrow Env(D, L))$ is an assignment of distributive environment transformers to the edges of $G^*$.

The original IDE algorithm [SRH96] solves such an IDE problem, $IP$, in two phases. In Phase I, it creates the jump functions that show the reachability of each $d \in D$, by assuming that their initial mappings to $L$ are always $\lambda l.\top$. In Phase II, it computes each $d$'s actual value mapping to $L$ by evaluating the edge functions defined in $M$.

```
1  Function ForwardComputeJumpFunctionsSLRPs():
2    for ⟨s_p, d'⟩, ⟨m, d⟩ s.t. m occurs in proc. p and d', d ∈ D ∪ {Λ} do
3      JumpFn(⟨s_p, d'⟩ → ⟨m, d⟩) = λl.⊤
4    for corresponding call-return pairs (c, r) and d', d ∈ D ∪ {Λ} do
5      SummaryFn(⟨c, d'⟩ → ⟨r, d⟩) = λl.⊤
6    PathWorkList := {⟨s_main, Λ⟩ → ⟨s_main, Λ⟩}
7    JumpFn(⟨s_main, Λ⟩ → ⟨s_main, Λ⟩) := id
8    while PathWorkList ≠ ∅ do
9      Select and remove an item ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from PathWorkList
10     let f = JumpFn(⟨s_p, d_1⟩ → ⟨n, d_2⟩)
11     switch (n) do
12       case n is a call node in p, calling a procedure q do
13         for d_3 s.t. ⟨n, d_2⟩ → ⟨s_q, d_3⟩ ∈ E# do
14           Propagate(⟨s_q, d_3⟩ → ⟨s_q, d_3⟩, id)
15         let r be the return-site node that corresponds to n
16         for d_3 s.t. e = ⟨n, d_2⟩ → ⟨r, d_3⟩ ∈ E# do
17           Propagate(⟨s_p, d_1⟩ → ⟨r, d_3⟩, EdgeFn(e) ∘ f)
18         for d_3 s.t. f_3 = SummaryFn(⟨n, d_2⟩ → ⟨r, d_3⟩) ≠ λl.⊤ do
19           Propagate(⟨s_p, d_1⟩ → ⟨r, d_3⟩, f_3 ∘ f)
20       case n is the exit node of p do
21         for call node c that calls p with corresponding return-site node r do
22           for d_4, d_5 s.t. ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E# and ⟨e_p, d_2⟩ → ⟨r, d_5⟩ ∈ E# do
23             let f_4 = EdgeFn(⟨c, d_4⟩ → ⟨s_p, d_1⟩) and
24             f_5 = EdgeFn(⟨e_p, d_2⟩ → ⟨r, d_5⟩) and
25             f' = (f_5 ∘ f ∘ f_4) ⊓ SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩)
26           if f' ≠ SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩) then
27             SummaryFn(⟨c, d_4⟩ → ⟨r, d_5⟩) := f'
28             let s_q be the start node of c's procedure
29             for d_3 s.t. f_3 = JumpFn(⟨s_q, d_3⟩ → ⟨c, d_4⟩) ≠ λl.⊤ do
30               Propagate(⟨s_q, d_3⟩ → ⟨r, d_5⟩, f' ∘ f_3)
31       case n is an intraprocedural node in p do
32         for ⟨m, d_3⟩ s.t. ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
33           Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩,
34                     EdgeFn(⟨n, d_2⟩ → ⟨m, d_3⟩) ∘ f)
35
36 Function Propagate(e, f):
37   let f' = f ⊓ JumpFn(e)
38   if f' ≠ JumpFn(e) then
39     JumpFn(e) := f'
40     Insert e into PathWorkList
```

**Figure 3.2:** The original IDE algorithm for Phase I (reproduced from [SRH96]).

According to Sagiv et al. [SRH96], the total cost of the IDE algorithm is bounded by $O(|E||D|^3)$, which is the cost of Phase I. Since $D$ is the set of symbols that belong to the target program, it is out of scope for sparsification. We, therefore, apply our sparsification approach in Phase I, where the jump functions are created by reducing $E$, the set of edges. Phase II is oblivious to how the jump functions are created—it automatically benefits from the sparsification of Phase I.

Figure 3.2 shows the algorithm for Phase I. Each procedure $p$'s CFG, $G_p$ consists of a *start* node $s_p$, an *exit* node $e_p$, and *normal* (non-call) nodes $m$ or $n$. Procedure calls are represented with two nodes: the *call-site* node $c$ denotes the point right before the procedure call, and the *return-site* node $r$ denotes the point right after. Program symbols, e.g., variables, access paths, etc., are denoted with $d', d \in D \cup \{\Lambda\}$ including the special symbol $\Lambda$. $\Lambda$ is required for generating new symbols at arbitrary program points.

**Initialization.** In lines 2–5, jump and summary functions are initialized. Jump functions, denoted by $JumpFn$, correspond to the *same-level realizable paths* (SLRPs) from the start node $s_p$ of a procedure $p$ to a node $m$ in $p$. Summary functions, denoted by $SummaryFn$, summarize the effect of a procedure call through same-level realizable paths from the call-site $c$ to return-site $r$. In line 3, $JumpFn(\langle s_p, d'\rangle \rightarrow \langle m, d\rangle) = \lambda l.\top$ states that the jump function from the node $\langle s_p, d'\rangle$ to each $\langle m, d\rangle$ is initialized to $\lambda l.\top$. In line 5, $SummaryFn(\langle c, d'\rangle \rightarrow \langle r, d\rangle) = \lambda l.\top$ states that the summary function from each call-site node $\langle c, d'\rangle$ to its corresponding return-site $\langle r, d\rangle$ is initialized to $\lambda l.\top$. Line 6 initializes the *PathWorkList* to $\{\langle s_{main}, \Lambda\rangle \rightarrow \langle s_{main}, \Lambda\rangle\}$ representing a self-loop edge on the start node of the *main* procedure whose jump function is the identity function, *id*. The jump function from the start node $s_p$ until the current statement $n$ is denoted with $f$.

**Call nodes.** Lines 12-19 handle the case where $n$ is a call-site node in $p$, calling a procedure $q$. In line 14, the self-loop edge on the start node of the callee procedure $q$ is initialized with *id*. In line 17, the edge from $s_p$ the corresponding return-site $r$ is computed by composing the $f$, the jump function until $n$ and the edge function from $n$ to $r$. In line 19, the edge from $s_p$ the corresponding return-site $r$ is computed by composing $f$ and $f_3$, the corresponding summary function when it is not mapping to $\top$.

**Exit nodes.** Lines 20-30 handle the case where $n$ is the exit node of $p$. Edges from each call-site node $c$ to the start node $s_p$ (shown with $f_4$) and from the exit node, $e_p$ to each caller's return-site $r$ (shown with $f_5$) must be computed. In line 25, a new summary function $f'$ is computed by composing $f_5$, $f$, and $f_4$ and merging the existing summary function for the same $c$ and $r$. When it is a new summary, a new jump function is computed from the caller procedure's start node $s_q$ to the node return-site node $r$ by composing the $f'$ with the existing jump function $f_3$ from $s_q$ to call-site node $c$.

**Normal nodes.** Lines 31-33 handle the case where $n$ is a non-call or intraprocedural node. Edges from the start node $s_p$ to each node $m$, which is the statement that appears directly after $n$ in procedure $p$, are computed by composing the edges from $s_p$ to $n$ (shown with $f$) and the edges from $n$ to $m$.

### 3.4.2 The SparseIDE Algorithm

In the original IDE algorithm, each symbol $d \in D \cup \{\Lambda\}$ at a statement $n$ is propagated to its direct successor statement $m$. As also pointed out in previous work [He+19], this behavior is desired when $n$ is a call and exit node. For these nodes, the reachability of each $d$ in different contexts is left to the data-flow function definition. *call-flow functions* propagate each $d$ into the context of the callee procedure. *return-flow functions* propagate each $d$ back to the context of the caller procedure. *call-to-return-flow functions* propagate each $d$ from before a procedure is called to after the procedure is called. However, when $n$ is a non-call node, each $d$ can safely be propagated to $d$'s next use statement.

Figure 3.3 shows the modifications for the SparseIDE algorithm for Phase I. We replace line 17 from the original IDE algorithm with lines 17-19 in the SparseIDE algorithm. Instead of propagating $d_3$ to the direct return site node $r$, we obtain $r'$, which is the next use statement of $d_3$ in its *symbol-specific* sparse control flow graph. Similarly, we replace line 33 with lines 33-35, to propagate $d_3$ to its next use statement $m'$, its sparse control flow graph. Our sparsification approach mirrors that of sparse IFDS algorithm [He+19], however, since we generalize it to IDE, we also account for edge function composition.

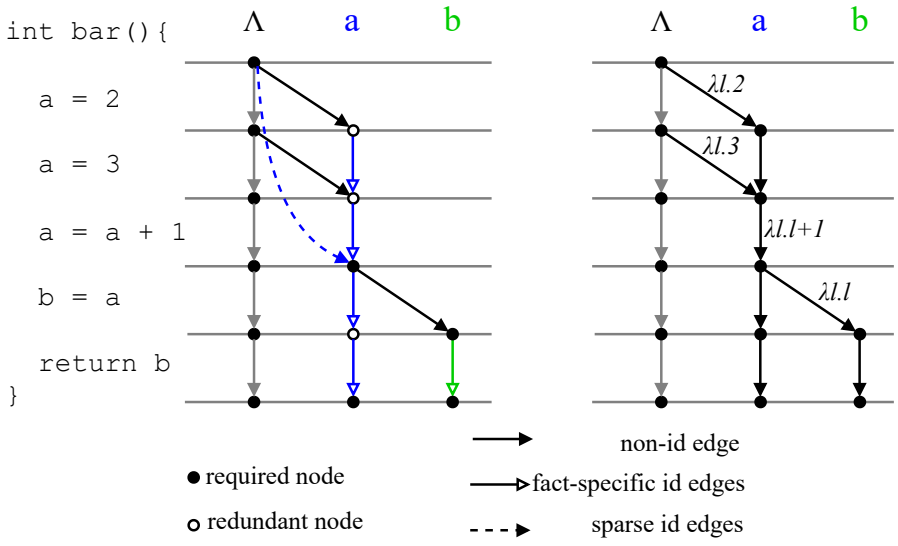
```
1  Function ForwardComputeSparseJumpFunctionsSLRPs():
2      ...
8      while PathWorkList ≠ ∅ do
9          Select and remove an item ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from PathWorkList
10         let f = JumpFn(⟨s_p, d_1⟩ → ⟨n, d_2⟩)
11         switch (n) do
12             case n is a call node in p, calling a procedure q do
13                 ...
15                 let r be the return-site node that corresponds to n
16                 for d_3 s.t. e = ⟨n, d_2⟩ → ⟨r, d_3⟩ ∈ E# do
17                     let r' = NextUse(p, d_3, r)
18                     Propagate(⟨s_p, d_1⟩ → ⟨r', d_3⟩,
19                               EdgeFn(⟨n, d_2⟩ → ⟨r, d_3⟩) ∘ f)
20             ...
31             case n is an intraprocedural node in p do
32                 for ⟨m, d_3⟩ s.t. ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
33                     let m' = NextUse(p, d_3, n)
34                     Propagate(⟨s_p, d_1⟩ → ⟨m', d_3⟩,
35                               EdgeFn(⟨n, d_2⟩ → ⟨m, d_3⟩) ∘ f)
36
41 Function NextUse(p, d, n):
42     let G_{p,d} be the sparse CFG of d in procedure p
43     let C be the sparse CFG cache with (p, d) typed keys and G_{p,d} as values
44     if G_{p,d} ∉ C then
45         construct G_{p,d} and add to C
46     return the next statement after n from G_{p,d}
```

**Figure 3.3:** Modifications for SPARSEIDE algorithm for Phase I (mirrors the design from [He+19]).

### 3.4.3 Sparse IFDS Revisited

As shown in Figure 3.1, a statement can behave as *identity function*, meaning it does not affect any data-flow fact, $d \in D$. However, as shown by He et al. [He+19], many statements only affect a few data-flow facts, often even just a single fact. Their flow functions can be considered *fact-specific identity functions* for the facts that they do not affect. Sparse IFDS defines fact-specific identity functions as follows [He+19]:

Given a symbol, $d \in D$ and a flow function, $f \in 2^D \to 2^D$, $f$ is a *d-specific identity function* if the following conditions hold:




**(a)** Sparse IFDS **(b)** SPARSEIDE

**Figure 3.4:** Comparison of the Sparsification Approaches of Sparse IFDS and SPARSEIDE

$$\forall X \in 2^D : d \in X \Rightarrow d \in f(X) \tag{1.1}$$

$$\forall X \in 2^{D \setminus \{d\}} : f(X) \setminus \{d\} = f(X \cup \{d\}) \setminus \{d\} \tag{1.2}$$

Condition 1.1 states that $d$ is not affected by other facts when applying $f$, and 1.2 states that $d$ does not affect the other facts when applying $f$. However, these conditions only apply to symbols from $D$ and ignore mappings from $D$ to the value domain $L$, and, if applied to IDE problems, one would wrongly treat such flow functions that are annotated with non-identity edge functions as $d$-specific identity functions as well.

Figure 3.4 shows two important cases where sparse IFDS would sparsify incorrectly. First, reassignments: `a = 3` reassigns `a`, but sparse IFDS recognizes that $a$ already exists (is "tainted"), and therefore it treats this statement as $a$-specific identity. Second, value updates: `a = a + 1` updates $a$'s value, but sparse IFDS has no notion of values; therefore, from its perspective, this statement is "identity" as well. SPARSEIDE, on the other hand, is aware of the effects on the value domain and retains both statements.

### 3.4.4 Fact-Specific Identity Transformers

To generalize fact-specific sparsification to the IDE framework, we define symbol-specific identity transformers that take into account the environments that map the symbols from domain $D$ to the values from domain $L$. Given a symbol $d \in D$ and a value $l \in L$, $env = [d \mapsto l]$ is an environment $env$ mapping from $d$ to $l$, i.e., $env(d) = l$. Then $env$ is an element of the set of environments $Env(D, L)$. An environment transformer, $t \in Env(D, L) \rightarrow Env(D, L)$ is a *d-specific identity transformer*, denoted by $t \equiv t^d$, if the following holds: First, the transformer $t$ keeps all $d$-specific mappings intact:

$$\text{given } d \in D : \forall env \in Env(D, L) : \\ env(d) = t(env)(d) \tag{2.1}$$

Second, for all other mappings, $t$ produces identical results no matter whether or not $d$-specific mappings are present:

$$\text{given } d \in D : \forall env \in Env(D, L).\ \forall d' \in D \setminus \{d\}.\ \forall l \in L : \\ t(env)(d') = t(env[d \mapsto l])(d') \tag{2.2}$$

We test the edge functions from Figure 2.4 under these conditions. $e_{id}$ is an $a$-specific identity transformer ($e_{id} \equiv e^a_{id}$), because applying $\lambda env.env$ does not change a's previous mapping. $e_{val}$ is not an $a$-specific identity transformer ($e_{val} \not\equiv e^a_{val}$), because applying $\lambda env.env[a \mapsto 3]$ changes a's previous mapping. $e_{op}$ is also not an $a$-specific identity transformer ($e_{op} \not\equiv e^a_{op}$) because applying $\lambda env.env[b \mapsto 2 * env(a) + 1]$ changes another value's mapping (for $b$) depending on what $a$ maps to, and because it changes b's value $e_{op}$ is not a $b$-specific identity transformer either ($e_{op} \not\equiv e^b_{op}$). Note that, importantly, a transformer can only be considered a $d$-specific identity transformer if the above restrictions hold *irrespective* of any concrete $l \in L$ that might be associated with $b$: (2.2) quantifies over all $l \in L$. This is necessary because IDE produces procedure summaries that must be sound with respect to *all* $l$ and thus their creation must not be made dependent on $l$.

As explained in Section 3.4.1, the IDE algorithm consists of two phases; the reachability of each symbol $d$ is computed in Phase I and symbol to value mappings, $[d \mapsto l]$, are calculated in Phase II. Since the values are not computed until Phase II, the values are not yet known at the time the sparsification takes place, i.e., in Phase I. In other words, IDE can support symbol-specific but not value-specific sparsification!

### 3.4.5 Determining Symbol-Specific Identity

When propagating fact $d$, we consider only those statements as irrelevant statements for $d$ that fulfil conditions (2.1) and (2.2). But since these conditions are value-agnostic—they quantify over all $l \in L$, this allows one to determine *ahead of time* the statements whose environment transformers adhere to both conditions, structurally. First, by Condition 2.1, a statement's corresponding environment transformer $t$ is *not* a $d$-specific identity transformer if $t$ affects $d$'s value mapping in any way, i.e., $t = \lambda env.env[\boldsymbol{d} \mapsto \_]$. Second, by Condition 2.2, $t$ is *not* a $d$-specific identity transformer either, if $t$ uses $d$'s value mapping $env(d)$ to compute another fact's value, i.e., $t = \lambda env.env[\_ \mapsto \ldots env(\boldsymbol{d}) \ldots]$.

Naturally, sparsification effectiveness is closely tied to the analysis-specific environment-transformer definitions. The environment transformer for the statement `a = a + 1` is $t \equiv t^a$ for taint analysis, where $t = \lambda env.env$. For constant propagation analysis, however, $t \not\equiv t^a$, where $t = \lambda env.env[env(a) + 1]$.

SPARSEIDE strictly generalizes Sparse IFDS as presented in SPARSEDROID [He+19]. One can easily define sparse IFDS as an instantiation of SPARSEIDE by restricting the value domain $L$ to $\{\bot, \top\}$, where symbols that map to $\bot$ are considered reachable. In this setting, our definitions (2.1) and (2.2) become equivalent to (1.1) and (1.2).

## 3.5 Application to Linear Constant Propagation

As Sagiv, Reps, and Horwitz explain in their seminal work [SRH96], constant propagation analysis is the perfect problem setting where IDE outperforms IFDS [RHS95]. This is not only because the problem's lattice is larger than the binary domain, but also because it is infinitely broad, where IFDS cannot terminate. We are, therefore, motivated to apply the SPARSEIDE framework to linear constant propagation analysis. HEROS, and thus SPARSEHEROS, are generic tools and they are independent of the target language and their intermediate representations (IRs). In this work, we use SOOT [Val+99] static program analysis framework for Java and its intermediate representation JIMPLE. Therefore, in the following, we explain our implementation based on the JIMPLE IR.

**Table 3.1:** Statements for Linear Constant Propagation Analysis with Corresponding IRs and Flow/Edge Functions.

| Statement | IR | Flow Function | Edge Function |
|---|---|---|---|
| constant | $a \leftarrow Const$ | $\lambda S.\{S \cup \{a\}\}$ | $\lambda env.env[a \mapsto Const]$ |
| binop | $a \leftarrow b \odot Const$ | $\lambda S. \begin{cases} S \cup \{a\} & \text{if } b \in S \\ S \setminus \{a\} \end{cases}$ | $\lambda env.env[a \mapsto env(b) \hat{\odot} Const]$ |
| local | $a \leftarrow b$ | $\lambda S. \begin{cases} S \cup \{a\} & \text{if } b \in S \\ S \setminus \{a\} \end{cases}$ | $\lambda env.env[a \mapsto env(b)]$ |
| field load | $a \leftarrow b.f$ | $\lambda S. \begin{cases} S \cup \{a\} & \text{if } b.f \in S \\ S \setminus \{a\} \end{cases}$ | $\lambda env.env[a \mapsto env(b.f)]$ |
| field store | $a.f \leftarrow b$ | $\lambda S. \begin{cases} S \cup \{p.f \mid p \in aliases(a)\} & \text{if } b \in S \\ S \setminus \{p.f \mid p \in aliases(a)\} \end{cases}$ | $\lambda env.env[p.f \mapsto env(b)]$ |
| static field load | $a \leftarrow T.f$ | $\lambda S. \begin{cases} S \cup \{a\} & \text{if } T.f \in S \\ S \setminus \{a\} \end{cases}$ | $\lambda env.env[a \mapsto env(T.f)]$ |
| static field store | $T.f \leftarrow b$ | $\lambda S. \begin{cases} S \cup \{p.f \mid p \in aliases(T)\} & \text{if } b \in S \\ S \setminus \{p.f \mid p \in aliases(T)\} \end{cases}$ | $\lambda env.env[p.f \mapsto env(b)]$ |
| array load | $a \leftarrow A[i]$ | $\lambda S. \begin{cases} S \cup \{a\} & \text{if } A[i] \in S \\ S \setminus \{a\} \end{cases}$ | $\lambda env.env[a \mapsto env(A[i])]$ |
| array store | $A[i] \leftarrow b$ | $\lambda S. \begin{cases} S \cup \{p[i] \mid p \in aliases(A)\} & \text{if } b \in S \\ S \setminus \{p[i] \mid p \in aliases(A)\} \end{cases}$ | $\lambda env.env[p[i] \mapsto env(b)]$ |
| call | $r \leftarrow b.m(a_i)$ | $\lambda S. \begin{cases} S \cup \{p_i\} & \text{if } a_i \in S \wedge a_i \mapsto p_i \text{ in } m \\ S \setminus \{p_i\} \end{cases}$ | $\lambda env.env[p_i \mapsto env(a_i)]$ |
| return | $r \leftarrow b.m(a_i)$ | $\lambda S. \begin{cases} S \cup \{r\} & \text{if } r' \in S \wedge m \text{ returns } r' \\ S \setminus \{r\} \end{cases}$ | $\lambda env.env[r \mapsto env(r')]$ |
| call-to-return | $r \leftarrow b.m(a_i)$ | $\lambda S. \begin{cases} S \setminus \{a_i\} & \text{if } a_i \in S \wedge a_i \mapsto p_i \text{ in } m \\ S \end{cases}$ | $\lambda env.env$ |

### 3.5.1 Analysis Definition

Linear constant propagation analysis handles the linear expressions that generate a new data-flow fact by using just a single other fact, e.g., `a = b` or `a = 2*b + 1`. Full constant propagation analysis involves statements such as `a = b + c`. Such a statement's flow function is not distributive; it cannot be precisely computed within the IDE framework. Our linear constant propagation analysis implementation handles the assignment statements shown in Table 3.1.

**IR.** The IR always ensures binary operation (*binop*) representation by reducing more complex operations to binary operations. For instance, `a = 2*b + 1` would be reduced to `s1 = 2 * b` and `a = s1 + 1`. The IR also reduces longer access paths to multiple assignments with a single access path (*n=1*). For instance, a statement such as `a = b.f1.f2` would be reduced to `s1 = b.f1`, `s2 = s1.f2`, and `a = s2`. The same reduction applies to procedure invocations as well.

**Flow functions.** We *generate* a symbol when it is assigned with an *integer constant*. As discussed, we handle the binary operations in the linear form. We distinguish between the assignments that require alias handling and the ones that do not. The assignments such as *local, field load, static field load,* and *array load*, overwrite the local variable, $a$, on their left-hand side and therefore do not need to know $a$'s aliases. The assignments such as *field store, static field store,* and *array store*, on the other hand, require handling the aliases of the base variables or the array references. To handle aliasing, we use the BOOMERANG [Spä+16] demand-driven pointer analysis framework. When necessary, we query the aliases of the base variables and add them to the set of propagated symbols. Note that in Table 3.1, the alias sets also contain the query variable. The IDE framework requires three flow function types to model the effects of invoke statements. The *call* flow function propagates the symbol for the actual parameter to the context of the callee procedure, by mapping it to the procedure's corresponding formal parameter. The *return* flow function propagates the symbol for the returned variable to the context of the caller procedure, by mapping it to the symbol on the left-hand side of the invoke expression. The *call-to-return* flow function propagates the symbols that are not passed to the context of the callee procedure, to the next statement after the invoke statement.

**Edge functions.** For most statements, the edge functions map the target symbol to the value of the source symbol, acting as *identity transformers*. The *constant* and *binop* statements are the only exceptions. The constant statement maps the target symbol, $a$ to the given constant value, $Const$. The binop statement maps the target symbol, $a$ to a new value. The value is computed by simulating the operation $\odot$ using the source symbol's value, $env(b)$ and the constant operand, $Const$. Edge functions must be composed and reduced to a simple value mapping when computing the actual values. Given $f_1, f_2 \in Env(D,\ L)$ and $f_1$ appears before $f_2$ as an edge in the exploded supergraph, we compose the edge functions as follows:

$$f_2 \circ f_1 := \begin{cases} f_2 & \text{if } f_1 = \lambda env.env \\ f_1 & \text{if } f_2 = \lambda env.env \\ f_2 & \text{if } f_2 = \lambda env.env[a \mapsto Const] \\ f_2(f_1) & \text{if } f_2 = \lambda env.env[a \mapsto env(b) \mathbin{\hat{\odot}} Const] \end{cases}$$

If an edge function is the identity transformer, we always apply the other function by the first two conditions. We always apply the subsequent edge function if it is a constant assignment, by the third condition. If the subsequent edge is a binary operation, we compute its value immediately in place by applying the preceding edge first, as suggested in previous work [Bod12].

**Lattice.** We perform the linear constant propagation on integers. Therefore, the lattice is $\mathbb{Z}_\bot^\top$. Given $l_1, l_2 \in \mathbb{Z}_\bot^\top$, we define the meet operator as follows:

$$l_1 \sqcap l_2 = \begin{cases} l_1 & \text{if } l_2 = \top \\ l_2 & \text{if } l_1 = \top \\ \bot & \text{if } l_1 = \bot \ \vee \ l_2 = \bot \\ \top & \text{if } l_1 = \top \ \wedge \ l_2 = \top \end{cases}$$

If a value is $\top$, the meet operator yields the other value by the first two conditions. If either of the values is $\bot$, the meet yields $\bot$, and if both values are $\top$ it yields $\top$ by the third and fourth conditions respectively.

### 3.5.2 Sparsification for Constant Propagation

Our sparsification approach has much in common with the one proposed by He et al. [He+19], though modifications were necessary. We build the sparse control flow graphs (CFGs) by ignoring symbol-specific identity functions. Given a procedure, $p$, $G_p$ is its original *dense* CFG. We build sparse CFGs specific to each symbol, $d$ in $p$, denoted as $G_{p,d}$, and propagate $d$ across its own sparse CFG. As shown with the IR in Table 3.1, $d$ can be a local, an instance field, a static field, or an array access. $G_{p,d}$ is constructed by determining whether each statement's corresponding flow function in $G_p$ is a $d$-specific identity function.

As a significant modification, and most importantly, we account for a statement's effect on the value domain. In addition to determining whether each statement's corresponding flow function is a *d-specific identity function*, we determine whether its edge function is a *d-specific identity transformer* with the assumptions explained

in Section 3.4.3. Further, we propagate the *tautological* fact, $\Lambda$, (sparsely) to the statements that can generate new data-flow facts, e.g., $a \leftarrow Const$. Otherwise, it is impossible to generate new facts at arbitrary program points. Finally, we soundly retain all branching statements to keep the original CFGs' control flow as it is.

## 3.6 Evaluation

Next, we explain the research questions guiding our evaluation and its experimental setup, then discuss the evaluation results. Sparse data-flow analyses promise extensive performance improvements while maintaining their non-sparse counterparts' precision. Therefore, first, we compare the sparse analysis results against the non-sparse analysis results. Second, we measure whether the sparse analysis produces the promised performance benefits. Third, we investigate the factors contributing to the performance impact. Therefore, we focus on the following research questions:

- RQ1: Does Sparse IDE produce the same results as the original IDE?
- RQ2: How does the sparsification impact the performance in terms of runtime and memory?
- RQ3: To what extent does the number of propagations correlate with the performance impact?

### 3.6.1 Experimental Setup

We implement the proposed approach in SPARSEHEROS, by extending the open source HEROS IDE solver's latest version, at the time of writing (`e7e4a85`) [soo12]. We implement a linear constant propagation analysis using SPARSEHEROS and the SOOT static analysis framework [Val+99]. To handle aliasing, we integrate our client analysis with the BOOMERANG [Spä+16] demand-driven pointer analysis, using the version (`1179227`) [Cod19].

As benchmark subjects, we use:

- **CONSTANTBENCH**: A benchmark suite for constant propagation analysis targeting Java, did not previously exist. We, therefore, created CONSTANTBENCH as a micro-benchmark suite for integer linear constant propagation analysis. We run both HEROS and SPARSEHEROS on this benchmark suite and compare the analysis results that they produce.

- **Real-world Libraries**: We include real-world Java libraries to investigate the performance of our approach under the workload of large-scale and complex programs. Unlike applications, libraries do not have a specific *entry* method. We follow the *closed package assumption* [Rei+16] for analyzing library code, and treat public methods of the libraries as entry methods. We consider a method as an entry method if it adheres to the following entry method selection criteria:

  - **c1:** The method is a public instance method that is not abstract, native, or a constructor,

  - **c2:** The method contains an integer assignment statement.

  We selected the most downloaded Java libraries, with more than 5000 downloads, from the Maven repository [Rep]. We discarded the libraries that do not contain any entry methods according to the selection criteria, and the ones that caused an error in the underlying static analysis tool, SOOT [Val+99]. In the end, we retained 30 libraries.

- **Replication Package**: We set up a replication package, available at `https://zenodo.org/records/10461449`

We have performed the evaluations on an Intel i7 Quad-Core at 2,3 GHz with 32GB of memory. We configured the JVM with 25GB maximum heap size (`-Xmx25g`) and 1GB stack size (`-Xss1g`).

### 3.6.2 RQ1: Does Sparse IDE produce the same results as the original IDE?

CONSTANTBENCH consists of 40 target programs with various program properties and sensitivity-testing edge cases, as listed in Table 3.2. *Assignment* cases test possible flow and edge functions, as well as flow sensitivity. *Branching* and *Loops* cases test the meet operation. *Field sensitivity* cases test field sensitivity and aliasing scenarios. *Context sensitivity* cases test various calling contexts. *Array* cases test array handling and *NonLinear* cases test analysis' behavior under unanticipated non-linear operations. The results validate the correctness of SPARSEIDE by showing that SPARSEHEROS produces the same outputs as the non-sparse HEROS.

**Table 3.2:** ConstantBench Test Cases

| **Assignment** | **Field Sensitivity** |
|---|---|
| Constant | LoadConstant |
| ConstantBinop | StoreConstant |
| LocalBinop | StoreViaAlias |
| LocalMultipleBinop | StoreBinop |
| Overwrite | FieldToField |
| Increment | StoreBinopViaAlias |
| Operators | StoreLocalViaAlias |
| AssignmentChain | **Context Sensitivity** |
| Static | Id |
| **Branching** | Increment |
| SameValueMergedAndUsed | Add |
| SameValueMergedNotUsed | Nested |
| SameValueMergedAndUsedInBinop | AssignFieldInCallee |
| DiffValuesMergedAndUsed | AssignStaticInCallee |
| DiffValuesMergedNotUsed | **Array** |
| DiffValuesMergedAndUsedInBinop | LoadConstant |
| **Loops** | StoreConstant |
| ForLoopFixedBound | ArrayToArray |
| ForLoopUnkownBound | AliasedArrays |
| WhileTrue | LargeIndex |
| WhileUnknown | **Non-Linear** |
| NestedLoops | Binop |
| | HashCode |

### 3.6.3 RQ2: How does the sparsification impact the performance in terms of runtime and memory?

Figure 3.5 shows the relative analysis runtime spent by SPARSEIDE in comparison to the runtime of the baseline original IDE algorithm. We sorted the results for each library by the time spent by the original IDE algorithm. Note that we keep the same sorting for the rest of the paper. This sorting highlights the fact that our SPARSEIDE approach pays off better for the cases where the original IDE's runtime is relatively larger. SPARSEIDE, compared to the original IDE algorithm, performs up to 30.7x faster. We measure the mean speedup as 7.9x, and the median speedup as 6.7x. The concrete measurements are presented in Table 3.3. Results show that, in terms of runtime, SPARSEIDE outperforms the original IDE in each run, except for the libraries #1-#3 (jcl-over-slf4j, slf4j-api, lombok), which have the shortest analysis time. In each run, the Sparse CFG construction overhead is lower than 1% of the SPARSEIDE total analysis runtime, which is substantially smaller than the achieved speedups.

Figure 3.6 shows the relative memory consumption of SPARSEIDE in comparison to the memory consumption of the original IDE algorithm. We have measured up to 94% reduction in memory consumption in the best case, and up to a 19% increase in the worst. The SPARSEIDE algorithm, compared to the original IDE, associates data-flow facts with fewer statements; therefore, we anticipated memory improvements. On the other hand, because we cache sparse CFGs ($G_{d,p}$) per each symbol and procedure pair $(d, p)$, for some input programs, memory consumption increases. However, as shown in Figure 3.6, these cases are limited to a few outliers. Moreover, the mean and median impacts on memory consumption are 51% and 63% reduction, respectively.

We statistically assess the significance of the SPARSEIDE algorithm's impact on runtime and memory improvements. According to the Wilcoxon signed-rank test [Wil92] at a 0.05 significance level, SPARSEIDE significantly improves both the runtime ($p = 6.1\text{e}{-08}$) and memory consumption ($p = 5.7\text{e}{-07}$) of the original IDE algorithm.

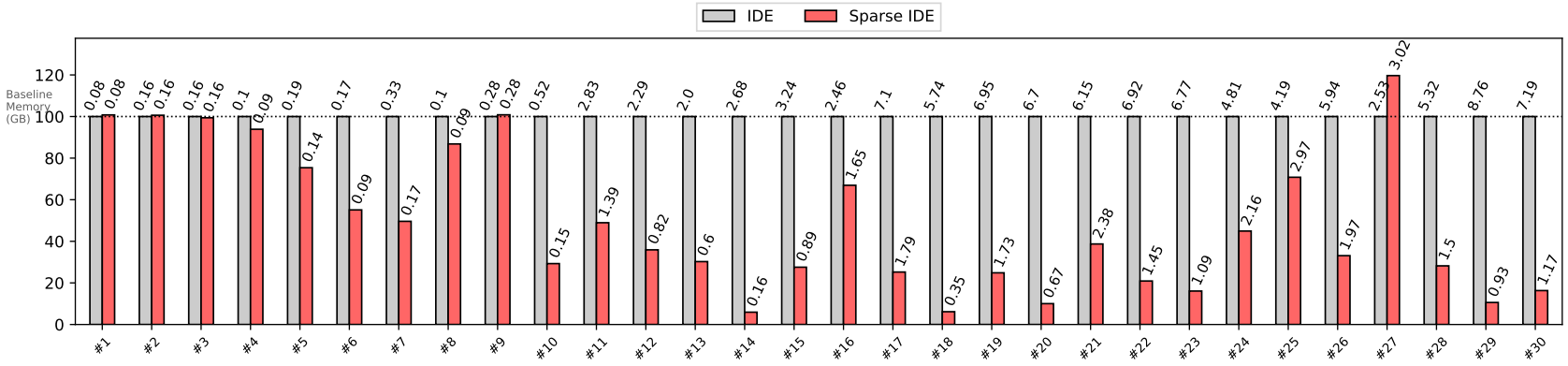



**Figure 3.6:** Memory consumption of SPARSEIDE compared to the baseline original IDE in %, annotated with exact memory consumptions in GB, using the same sorting as Figure 3.5

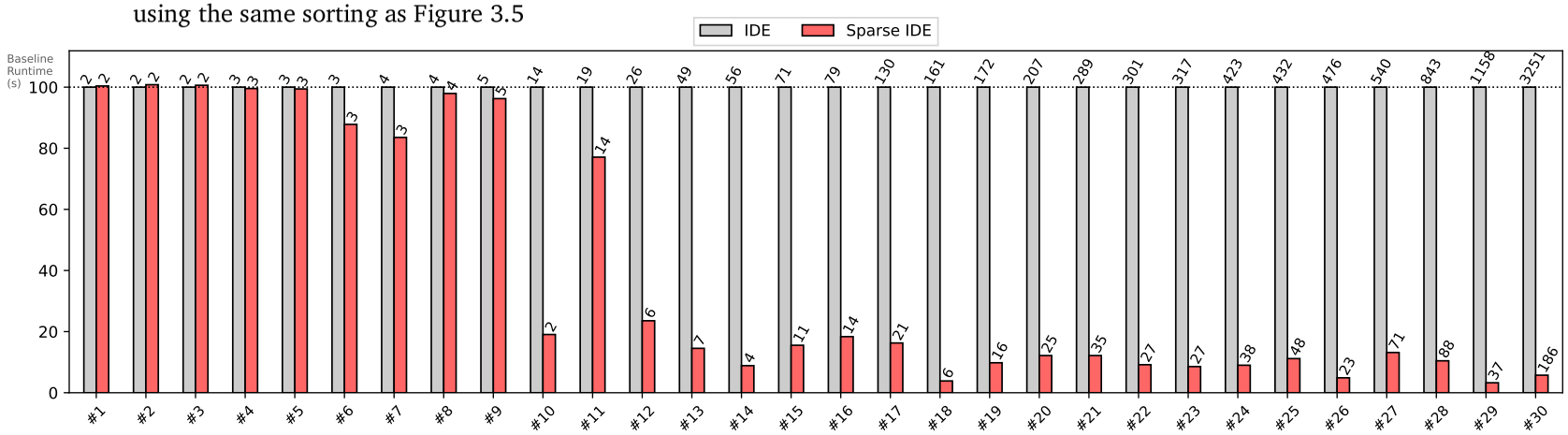



**Figure 3.5:** Relative runtime of SPARSEIDE compared to the baseline original IDE in %, annotated with exact runtimes in seconds, sorted by original IDE's runtime

**Table 3.3:** Performance of Sparse IDE compared to the baseline original IDE algorithm

| # | Library | Version | #Entry Methods | Runtime (s) | | | Memory (GB) | | | #Propagations | | | Sparse CFG | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | IDE | SP | IDE/SP | IDE | SP | SP/IDE (%) | IDE | SP | IDE/SP | Count | Const. (ms) | %Runtime |
| 1 | jcl-over-slf4j | 2.0.7 | 1 | 2 | 2 | 1.00 | 0.08 | 0.08 | 100.78 | 48 | 34 | 1.41 | 2 | 0 | 0.01 |
| 2 | slf4j-api | 2.0.7 | 7 | 2 | 2 | 0.99 | 0.16 | 0.16 | 100.62 | 104 | 94 | 1.11 | 13 | 0 | 0.00 |
| 3 | lombok | 1.18.26 | 5 | 2 | 2 | 0.99 | 0.16 | 0.16 | 99.40 | 894 | 227 | 3.94 | 13 | 0 | 0.02 |
| 4 | commons-logging | 1.2 | 14 | 3 | 3 | 1.00 | 0.10 | 0.09 | 93.87 | 1,509 | 917 | 1.65 | 41 | 0 | 0.00 |
| 5 | junit-jupiter-api | 5.9.2 | 10 | 3 | 3 | 1.01 | 0.19 | 0.14 | 75.39 | 182 | 158 | 1.15 | 20 | 0 | 0.00 |
| 6 | jackson-annotations | 2.14.2 | 79 | 3 | 3 | 1.14 | 0.17 | 0.09 | 55.10 | 13,115 | 6,511 | 2.01 | 190 | 0 | 0.00 |
| 7 | maven-plugin-api | 3.9.1 | 13 | 4 | 3 | 1.20 | 0.33 | 0.17 | 49.61 | 17,353 | 4,780 | 3.63 | 294 | 4 | 0.14 |
| 8 | junit-jupiter-engine | 5.9.2 | 23 | 4 | 4 | 1.02 | 0.10 | 0.09 | 86.81 | 3,204 | 1,181 | 2.71 | 105 | 0 | 0.02 |
| 9 | osgi.core | 8.0.0 | 124 | 5 | 5 | 1.04 | 0.28 | 0.28 | 100.83 | 58,675 | 28,247 | 2.08 | 664 | 7 | 0.15 |
| 10 | jakarta.servlet-api | 6.0.0 | 12 | 14 | 2 | 5.25 | 0.52 | 0.15 | 29.28 | 126,656 | 341 | 371.43 | 33 | 0 | 0.00 |
| 11 | commons-io | 2.11.0 | 178 | 19 | 14 | 1.30 | 2.83 | 1.39 | 48.94 | 156,595 | 15,290 | 10.24 | 1,279 | 116 | 0.78 |
| 12 | commons-codec | 1.15 | 77 | 26 | 6 | 4.25 | 2.29 | 0.82 | 35.90 | 652,560 | 100,866 | 6.47 | 532 | 13 | 0.21 |
| 13 | json | 20230227 | 33 | 49 | 7 | 6.88 | 2.00 | 0.60 | 30.24 | 1,071,045 | 10,846 | 98.75 | 407 | 0 | 0.00 |
| 14 | logback-classic | 1.4.7 | 93 | 56 | 4 | 11.28 | 2.68 | 0.16 | 5.92 | 1,286,543 | 8,027 | 160.28 | 372 | 12 | 0.24 |
| 15 | logback-core | 1.4.7 | 218 | 71 | 11 | 6.44 | 3.24 | 0.89 | 27.55 | 1,739,303 | 14,767 | 117.78 | 925 | 0 | 0.00 |
| 16 | gson | 2.10.1 | 147 | 79 | 14 | 5.45 | 2.46 | 1.65 | 66.93 | 2,009,909 | 29,391 | 68.39 | 1,586 | 54 | 0.37 |
| 17 | commons-lang3 | 3.12.0 | 318 | 130 | 21 | 6.14 | 7.10 | 1.79 | 25.22 | 3,418,491 | 31,856 | 107.31 | 1,144 | 0 | 0.00 |
| 18 | commons-beanutils | 1.9.4 | 109 | 161 | 6 | 25.97 | 5.74 | 0.35 | 6.15 | 5,855,012 | 20,640 | 283.67 | 648 | 2 | 0.04 |
| 19 | mockito-core | 5.3.1 | 235 | 172 | 16 | 10.20 | 6.95 | 1.73 | 24.85 | 5,025,407 | 51,374 | 97.82 | 1,663 | 119 | 0.71 |
| 20 | junit-jupiter-params | 5.9.2 | 293 | 207 | 25 | 8.22 | 6.70 | 0.67 | 10.03 | 6,266,620 | 99,285 | 63.12 | 1,506 | 109 | 0.43 |
| 21 | assertj-core | 3.24.2 | 334 | 289 | 35 | 8.22 | 6.15 | 2.38 | 38.71 | 10,033,236 | 45,563 | 220.21 | 2,418 | 37 | 0.11 |
| 22 | commons-collections4 | 4.4 | 620 | 301 | 27 | 10.90 | 6.92 | 1.45 | 20.91 | 9,140,963 | 42,741 | 213.87 | 1,796 | 1 | 0.01 |
| 23 | testng | 7.7.1 | 246 | 317 | 27 | 11.68 | 6.77 | 1.09 | 16.08 | 9,329,214 | 116,084 | 80.37 | 2,910 | 15 | 0.06 |
| 24 | joda-time | 2.12.5 | 375 | 423 | 38 | 11.11 | 4.81 | 2.16 | 44.93 | 15,151,487 | 137,705 | 110.03 | 3,227 | 69 | 0.18 |
| 25 | guice | 5.1.0 | 336 | 432 | 48 | 8.95 | 4.19 | 2.97 | 70.80 | 15,141,525 | 390,634 | 38.76 | 3,918 | 58 | 0.12 |
| 26 | hamcrest-all | 1.3 | 290 | 476 | 23 | 20.48 | 5.94 | 1.97 | 33.10 | 17,953,051 | 71,200 | 252.15 | 1,105 | 28 | 0.12 |
| 27 | log4j-core | 2.20.0 | 512 | 540 | 71 | 7.60 | 2.53 | 3.02 | 119.69 | 18,746,154 | 1,218,580 | 15.38 | 4,666 | 64 | 0.09 |
| 28 | jackson-databind | 2.14.2 | 844 | 843 | 88 | 9.57 | 5.32 | 1.50 | 28.20 | 35,842,682 | 166,906 | 214.75 | 7,884 | 5 | 0.01 |
| 29 | okhttp | 4.10.0 | 717 | 1,158 | 37 | 30.69 | 8.76 | 0.93 | 10.58 | 37,431,312 | 581,852 | 64.33 | 5,928 | 69 | 0.18 |
| 30 | guava-31.1 | jre | 1,332 | 3,251 | 186 | 17.43 | 7.19 | 1.17 | 16.31 | 131,993,565 | 239,589 | 550.92 | 12,200 | 4 | 0.00 |

### 3.6.4 RQ3: To what extent does the number of propagations correlate with the performance impact?

The essence of the SparseIDE approach is that, compared to the original IDE algorithm, it propagates data-flow facts to fewer statements. We investigate the extent to which this contributes to improving the scalability of the original IDE algorithm. Figure 3.7, shows how the ratio of data-flow fact propagations in IDE and SparseIDE correlate with the ratio of runtime speedups. We observe that reducing the number of propagations is an effective approach to improving IDE's scalability in terms of runtime.

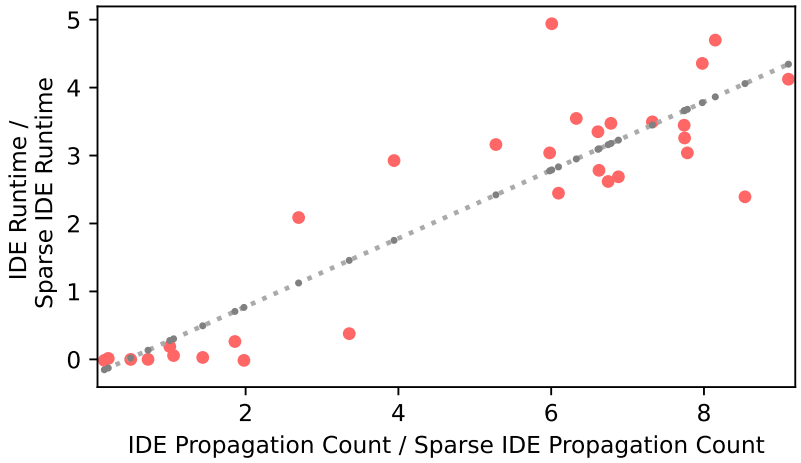



**Figure 3.7:** Ratio of data-flow fact propagations and corresponding speedup ratios, in log scale

We also investigate the impact of sparsification on the memory consumption. Figure 3.8 shows how the ratio of data-flow fact propagations in IDE and SparseIDE correlate with the ratio of memory consumption in IDE and SparseIDE. We observe a comparable trend, but not to the same degree. Given these findings, in the future, one could investigate the potential synergies between our approach and recent approaches that improve scalability, particularly in terms of memory [Arz21; Li+21].

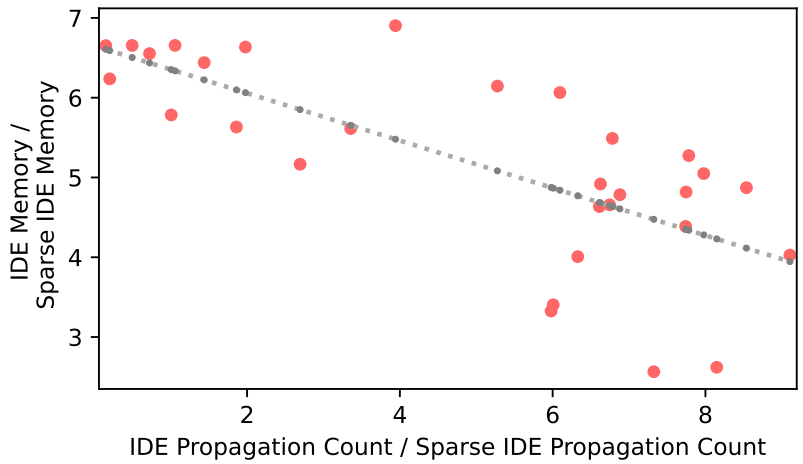



**Figure 3.8:** Ratio of data-flow fact propagations and corresponding memory consumption ratios, in log scale

### 3.6.5 Threats to Validity

By definition, SPARSEIDE can solve the same data-flow problems as the original IDE framework [SRH96]. It requires data-flow analysis problems to be expressible as distributive environment problems. Many popular static analyses, such as taint analysis for vulnerability detection [Arz+14] or typestate analysis for API misuse detection [Emm+21], are expressible as distributive environment problems. Just like the sparse IFDS [He+19], SPARSEIDE also exploits analysis domain knowledge. Domain-specific analysis semantics must be correctly encoded with flow and edge function definitions within the IDE framework.

SPARSEIDE should theoretically lead to a similar performance impact on other data-flow analysis problems where IDE is applicable. For instance, when performing a typestate analysis, SPARSEIDE would safely omit the statements that have no impact on the tracked state. However, due to space constraints, we were not able to empirically show whether our evaluation results carry over to other analysis problems.

The reported evaluation results might depend on the selected set of Java libraries, and entry-method selection criteria. Nevertheless, for real-world library selection, we followed the systematic procedure described in Section 3.6.1. To account for variations in runtime and memory measurements, we conducted three runs and presented the average across these runs.

A direct comparison to SPARSEDROID [He+19] was not possible for many reasons. It extends an existing taint analysis client (FLOWDROID [Arz+14]) that has a basic integrated alias analysis, whereas our analysis client utilizes a sophisticated *external* demand-driven pointer analysis [Spä+16]. Therefore, SPARSEDROID's performance gains benefit from sparsifying the integrated alias analysis. Moreover, SPARSEDROID also benefits from FLOWDROID's multi-threadedness. HEROS, and thus SPARSEHEROS, also support multi-threading. Yet, because BOOMERANG is single-threaded, our client analysis uses a single thread. Therefore, our evaluation results present single-thread performance. Finally, SPARSEDROID's implementation is not publicly available, and most importantly, IFDS may not terminate when the value domain is infinitely broad.

## 3.7 Related Work

The IFDS [RHS95] and IDE [SRH96] frameworks enabled precise interprocedural data-flow analyses that are flow- and context-sensitive. Previous works have extended these frameworks with diverse goals. Naeem et al. [NLR10] proposed four extensions to the IFDS framework, to improve its scalability and precision under certain practical analysis conditions. HEROS [Bod12] introduced a Java-based generic IFDS and IDE solver. REVISER [AB14] proposed an algorithm to adapt IFDS and IDE to incremental program updates. CLEANDROID [Arz21] introduced a technique for reducing the memory footprint of IFDS-based data-flow analyses. DISKDROID [Li+21] applied a disk-assisted computing approach for improving the scalability of IFDS-based taint analysis.

Sparsification has been applied to improve the scalability of static analyses. Choi et al. [CCF91] introduced sparse data-flow evaluation graphs based on SSA (static-single assignment). Oh et al. [Oh+12] presented an abstract interpretation-based framework for designing generic sparse analyses, which guarantees to preserve the precision of the non-sparse analysis through data dependencies. PINPOINT [Shi+18], SVF [SX16] and SFS [HL11] utilize cheaper pre-analyses to sparsify pointer analyses. Recent on-demand sparsification approaches exploit the data-flow facts that become available during the analysis runtime for further sparsification. SPARSEBOOMERANG, as we show in this thesis, exploits the variables in alias queries during demand-driven pointer analysis to create query-specific sparse CFGs. The sparse IFDS algorithm [He+19] exploits data-flow facts to create fact-specific sparse

CFGs and propagate each fact on its own sparse CFG. In this work, we present the more generic SPARSEIDE algorithm that efficiently solves not just IFDS-based reachability problems, but also IDE problems that require value computation.

## 3.8 Conclusion

In this chapter, we presented the first major contribution of this thesis, the SPARSEIDE framework (*Contribution 1*) as a scalable alternative to the original IDE framework. SPARSEIDE is the first fact-specific sparsification approach that allows for computations on infinitely broad domains. SPARSEIDE produces equally precise results as the original IDE, while significantly improving its scalability. We also explicitly discuss the limits of sparsification for IDE: while symbol-specific sparsification is possible and useful, one cannot sparsify with respect to the (typically numeric and infinite) value domain.

The essence of SPARSEIDE is creating symbol-specific sparse control flow graphs on demand, and propagating data-flow facts sparsely through these graphs. Therefore, sparsification is applied only intraprocedurally, i.e., within individual methods obtained from a *call graph* at call sites. In the next chapter (Chapter 4), we investigate the extent to which call-graph precision impacts the scalability of data-flow analyses. To this end present and discuss the findings from an extensive empricial evaluation using 31 different call-graph algorithms (*Contribution 2*).

# 4 An Empirical Study on the Impact of Call-Graph Precision on the Scalability of Data-Flow Analysis

In this thesis, we present novel techniques to improve the scalability of IFDS- and IDE-based interprocedural data-flow analysis, mainly through sophisticated fact-specific sparsification. As explained in Section 2.2, the call graph is an integral component of interprocedural data-flow analysis.

Surprisingly, the literature neglects the call graphs' impact on the scalability of data-flow analysis. The large body of work on call-graph analysis [EGH94; DGC95; BS96; TP00; Sun+00; GC01; Lho03] *only* assess call-graph quality, typically in interms of precision and recall of the call graphs themselves, but without looking at the client analyses that rely on these call graphs.

Although there are many recent works [Arz21; He+23; Li+21; Wan+23; He+19; Sch+24] on scaling IFDS and IDE-based analyses, none of them target optimizing the call graph. They propose sophisticated techniques ranging from sparsification and disk-assisted computing to intelligent garbage collection. Yet, they choose a fixed call graph, thereby disregarding its implications on scalability.

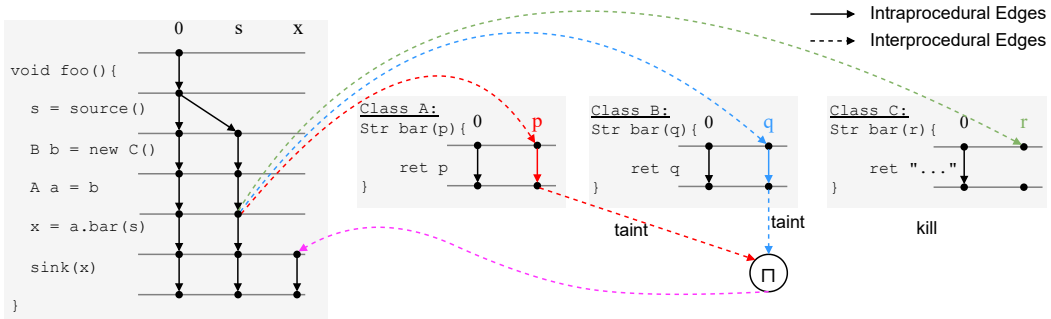
In this chapter, we present the second major contribution of this thesis, an empirical evaluation of the impact of call graph precision on the precision and scalability of the IFDS framework (*Contribution 2*). To this end, we build QCG, a call graph generation tool for Android that extends the QILIN pointer analysis framework, and integrate it with FLOWDROID, a state-of-the-art IFDS-based taint analysis solver. We assess the precision of 27 call graphs built with QCG and 4 default call graphs in FLOWDROID, on the TAINTBENCH benchmark of Android malware. We then evaluate how increasing the call-graph precision impacts FLOWDROID's runtime performance and memory consumption on real-world apps.

As we report, the time invested in building precise context-sensitive call graphs pays off: They significantly reduce IFDS analyses' runtimes while also improving their precision. However, there appears to be a sweet spot in the trade-off between the call graph construction time and the reduction in total analysis runtime.

## 4.1 Motivation

As previous works [Avd+15; Hua+15] have shown, IFDS-based analyses can easily become unscalable when analyzing complex real-world programs. The scalability of the IFDS framework is bounded by (1) the runtime cost of propagating individual path edges and (2) the memory requirements to accommodate these edges for later reuse. Scalable IFDS extensions aim to either improve how these edges are computed [He+19], or how they are stored in memory [Arz21; He+23; Li+21; Wan+23; Sch+24]. They, however, often choose a fixed call-graph algorithm without considering its implications on how these edges are computed, and thus, how they are stored later on.

Precision and scalability are historically known to be competing objectives. Previous works on call graphs [Gro+97; Mur+98; TP00; GC01] have shown that increased call-graph precision comes with an increased runtime cost during callgraph construction. Others [Bod18] have seen a trend where cheaper-to-compute call graphs waste time and precision in the later phases of the analysis, but so far there has been *no empirical evidence* that confirms this observation.




**(a) caller** **(b) callee in A** **(c) callee in B** **(d) callee in C**

**Figure 4.1:** Call-graph precision's theoretical implications on the computation of the IFDS algorithm.[1]

Figure 4.1 shows a motivating example of an IFDS-based taint analysis on an exploded supergraph. Here, having a precise call graph not only reduces the workload of the IFDS algorithm but also improves its precision. Given the method `foo()` in Figure 4.1.a, `s` is being tainted by `source()` and passed to method `bar()`. Assuming the class hierarchy, where `C <: B <: A`, the method `bar()` in class `C` should be executed, which kills the taint, and therefore, there should not be any leaks. Using an imprecise call-graph algorithm, for instance, CHA (Class Hierarchy Analysis) [DGC95], would resolve to method `bar()` in all three classes. Consequently, the IFDS algorithm will map the data-flow fact `s` into all three methods and union their effects on `s`. This results in both an increased number of methods to process and an imprecise finding, as the taint analysis would *falsely* report a leak at `sink(x)`. We are therefore motivated to assess whether this theoretical implication holds in practice.

## 4.2 Contributions

This chapter presents an empirical evaluation of the impact of call-graph precision on the precision and scalability of the IFDS framework. To perform the experiments, we use FlowDroid [Arz+14], a state-of-the-art Android taint analysis, as a reference for the IFDS implementation. To obtain call graphs with varying degrees of precision, we extend Qilin [HLX22], a state-of-the-art Java pointer analysis framework, in a tool called Qcg. Qcg enables leveraging Qilin's pointer analysis to obtain Android-compatible call graphs. First, we sort call graphs by their degrees of precision. To do so, we assess how they impact the precision and recall of FlowDroid on the TaintBench [Luo+22] benchmark of Android malware. Second, we evaluate how increasing the call-graph precision impacts FlowDroid's runtime performance and memory on a set of popular real-world apps.

To summarize, this chapter presents the following contributions:

- Qcg, a Qilin-based call-graph generation tool for Android, which is open-sourced at Github,
- precision and recall of FlowDroid when using its 4 call graphs and 27 call graphs obtained by Qcg, and
- an evaluation of the call graphs' impact on FlowDroid's scalability.

## 4.3 Foundations

This section briefly introduces the foundations that our empirical study builds on. We first explain how and why call graphs may affect data-flow analysis performance. Then we discuss the relationship between call-graph generation and pointer analysis. We present the QILIN pointer analysis framework, how it fits into this work, and why we had to extend it in QCG.

### 4.3.1 Call Graph's Implications on IFDS

Real-world programs written in object-oriented languages often contain many polymorphic call sites, whose actual runtime call targets can only be approximated by static call-graph algorithms. To be efficient, they compromise on many precision aspects (e.g., on flow-, field-, or context-sensitivities), which results in an imprecise call graph with a coarser set of potential call targets for each call site. This has direct implications for the IFDS algorithm regarding both precision and runtime.

**Precision.** The IFDS framework [RHS95] requires data-flow analysis problems to define distributive flow functions over the meet operator. In IFDS, the meet operator is set union.[2] This property allows propagating data-flow facts along diverging execution paths, e.g., for conditional branches or potential call targets, and *unioning* them when the paths meet without compromising soundness. Compared to a precise one, an imprecise call graph contains more potential call targets. This requires propagating data-flow facts into more call targets, which increases the probability of losing precision. Precise static call-graph generation techniques rely on pointer information to either prune existing imprecise call graphs [Sun+00] or create them on-the-fly [AL12].

**Runtime.** The complexity of the IFDS algorithm is in $O(|E||D|^3)$, and is determined by the number of edges $|E|$ and the number of data-flow facts $|D|$ [RHS95]. $|D|$ depends on how the data-flow analysis is formulated; in problems with a binary lattice (e.g., in taint analysis), $D$ is the set of program variables. It can, however, grow larger with a larger lattice (e.g., in type-state analysis, where variables are also associated with a state). $|E|$ depends on the control flow of the program under analysis, i.e., on how many edges need to be created within each method, and

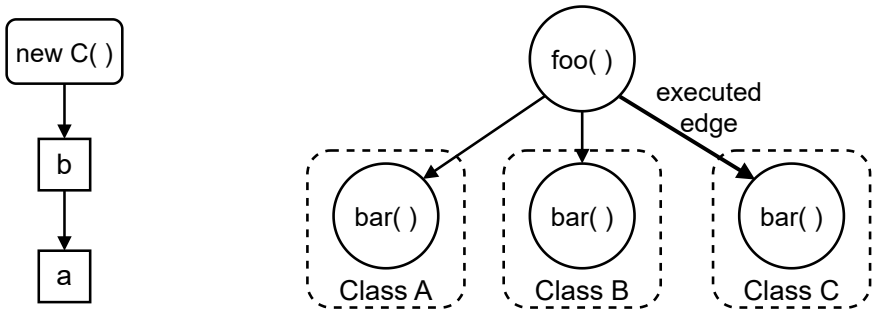
[2] Intersection problems need to be transformed to a complementary union problem [RHS95].

across method boundaries. As shown in Figure 4.1, the *intraprocedural edges* connect data-flow facts per statements within individual method boundaries, whereas the *interprocedural edges* map the data-flow facts at call sites to corresponding callee methods and back to their return sites in the caller methods. The number of interprocedural edges at each call site increases linearly with the number of potential call targets. This also implies more intraprocedural edges to process in each call target, leading to an increased runtime.

### 4.3.2 Call-Graph Generation using Pointer Information

Pointer information associates program variables with sets of objects they might refer to at runtime. Static pointer analyses approximate this information by choosing an appropriate level of abstraction [LH06], usually by trading off between precision and scalability. Pointer information is crucial for call-graph generation because (virtual) method resolution depends on the runtime types of receiver objects [Sun+00].

Recall the motivating example in Figure 4.1, where the statements `b = new C()` and `a = b` result in the pointer assignment graph (PAG) in Figure 4.2a. By looking at the PAG, one can see that the receiver object of the method `bar()` is of type `C`. Disregarding this information would lead to the call graph in Figure 4.2b, where the call `b.bar()` resolves to all possible implementation of method `bar()`. By utilizing the pointer information, on the other hand, one can obtain an induced subset of the same call graph that contains the actual executed edge.




**(a)** PAG **(b)** Call graph

**Figure 4.2:** Pointer assignment graph (PAG) and call graph of the code in Figure 4.1

Precise pointer analysis is undecidable [Ram94]. Real-world programs require handling various scenarios, where static pointer analyses cannot preserve precision [Hin01]. They often maintain a set of feasible sensitivity criteria that can be optimized depending on the analysis requirements. Pointer analysis frameworks [Lho03; LH08; BS09; LH22; HLX22] enable testing variations of such sensitivity criteria.

### 4.3.3 QILIN

QILIN [HLX22] is a pointer analysis framework that implements many state-of-the-art, and also advanced, pointer analysis techniques. QILIN modularly separates analysis logic from its implementation to support easy extension. It builds on top of a parameterized pointer analysis kernel that can be configured with context-sensitivity parameters to obtain different pointer analysis flavors.

Table 4.1 shows the pointer analysis techniques in QILIN. The level of context-sensitivities ranges from Andersen-style *insensitive* and traditional *method-level* $k$-limiting analyses to more recent variable-level *fine-grained* context-sensitive analyses. QILIN also enables parametric instantiations of existing techniques by specifying a *context constructor* (e.g., insensitive, callsite-, object-, type-sensitive, or hybrid (callsite- and object- sensitive) ), a *context selector* (e.g., uniform, heuristic, selective, or partial), and a *heap abstractor* (e.g., allocation-site, heuristic, or type-consistency). By doing so, for instance, one can instantiate $k$-object-sensitive analyses based on ZIPPER (Z-$k$OBJ), EAGLE (E-$k$OBJ) or TURNER (T-$k$OBJ).

**Table 4.1:** Pointer Analysis Techniques in QILIN.

| Context-Sensitivity | Techniques |
|---|---|
| Method-Level | $k$-callsite-sensitivity ($k$CFA) [SP+78],<br>$k$-object-sensitivity ($k$OBJ) [MRR02],<br>$k$-type-sensitivity ($k$TYPE) [SBL11],<br>hybrid $k$-object-sensitivity H-$k$OBJ [KS13] |
| Fine-Grained | BEAN [TLX16], MAHJONG [TLX17], ZIPPER [Li+18],<br>EAGLE [LX19], TURNER [He+21], CONCH [HLX21],<br>DATA-DRIVEN [Jeo+17]<br>CONTEXT-TUNNELING [JJO18] |

QILIN is built on top of the SOOT [Val+99] static analysis framework for Java and therefore it also contains SOOT's default call-graph algorithms. However, QILIN can only construct call graphs for Java code; to conduct our evaluations on Android applications, we have extended QILIN in QCG.

### 4.3.4 QCG

QILIN itself expects a `main()` method to be present in a given target program. `main()` is the default entry method for Java applications, which is used as the starting point of the pointer analyses in QILIN. QILIN also models complex language features of

Java by introducing a `FakeMain()` method. It contains method calls for the cases that JVM handles implicitly, for instance, system/main thread groups, class initialization, static initializers [HLX22].

Unlike Java applications, which typically have a single entry point, Android applications feature multiple entry points. In Android, many methods are invoked implicitly by the Android framework based on the application's state defined in the Android life cycle. The Android life cycle manages various components within an Android application, such as activities, services, broadcast receivers, and content providers. Previous work by Arzt et al. [Arz+14] shows how these implicit methods must be handled to obtain sound call graphs for Android applications. In QCG, we mirror the same approach. We extend QILIN to model the implicit calls of the Android framework. By doing so, we unleash QCG's capabilities for Android. QCG can be used as a standalone tool to obtain sound call graphs for Android applications powered by QILIN's pointer analyses.

## 4.4 Study Design

This section presents the empirical study design. We begin with the research questions and explain how we designed the experiments to address these questions. Then, we give an overview of the experimental setup and explain each component in our setup.

### 4.4.1 Research Questions

This chapter primarily focuses on measuring the impact of call-graph precision on the scalability of the IFDS-based data-flow analyses. As secondary goals, we also investigate the call graph's impact on analysis precision and the influence of call-graph properties on analysis performance. To summarize, our empirical study focuses on the following concrete research questions:

- RQ1: How does call graph precision impact the precision of IFDS analyses?
- RQ2: How does call graph precision impact the performance of IFDS analyses in terms of runtime and memory?
- RQ3: How does the number of interprocedural edges correlate with the analysis runtime and memory consumption?

In section 4.3.1, we discussed the theoretical implications of call-graph precision on the precision of the IFDS-based analyses. With *RQ1*, we investigate whether these implications hold when performing IFDS analyses on real-world applications. Similarly, in section 4.3.1, we discussed the theoretical implications of call-graph precision on the runtime of the IFDS-based analyses. With *RQ2*, we investigate the extent of these implications on practical IFDS analysis runs. The main advantage of a precise call graph is that it contains fewer potential call targets, which then constitute the interprocedural edges in the IFDS framework. Intuitively, one expects a correlation between the number of interprocedural edges and the analysis performance. With *RQ3*, we aim to find out whether such a correlation exists when analyzing real-world applications.

### 4.4.2 Experimental Setup

We have designed our experiments to be conducted in two phases. In *Phase I*, we aim to answer *RQ1* and in *phase II*, we aim to answer *RQ2* and *RQ3*. In the following, we introduce the components in our setup and explain the workings of the phases in detail.

**IFDS Analysis Client.** In both phases, we use FLOWDROID as the reference IFDS-based taint analysis implementation. FLOWDROID is one of the most mature and widely used taint analyses. It soundly handles many Android-specific language features, as well as many precision dimensions. Flow- and context-sensitivities are inherent properties of the IFDS framework. Further, its solver is field-sensitive (by using access paths as data-flow fact abstraction) and it contains an integrated on-demand alias analysis.

**Call-Graph Framework.** In both phases, we use QCG as the call-graph framework. QCG extends QILIN with Android-specific call-graph-building capabilities. QCG essentially provides the connection between FLOWDROID and QILIN, by also ensuring that QILIN-generated call graphs contain necessary methods for FLOWDROID to perform the taint analysis. We configure QILIN in each run with the parameters for a specific pointer analysis flavor. In addition, in some runs, for comparison, we utilize the call-graph algorithms available in FLOWDROID by default. In those cases, FLOWDROID handles the call-graph generation internally. At the end, we obtain 31 different call graphs.

**Replication Package.** We have prepared an artifact to run the complete evaluation pipeline, which is available at `https://zenodo.org/records/17041537`.

### Phase I

Figure 4.3 shows the setup for the first phase of our experimental study. In this phase, we use TAINTBENCH as the analysis target. TAINTBENCH contains 39 real-world Android malware applications with custom *sources & sinks* and well-documented *ground truth*. In taint analysis, sources are a set of tainting methods, and sinks are a set of methods that taints should not reach. The Ground truth defines expected and unexpected *taint-flows* that might occur between the sources and the sinks.

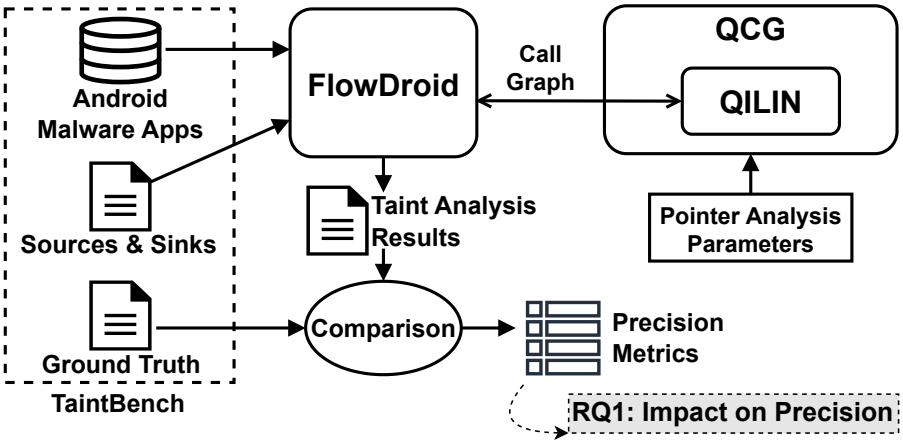



**Figure 4.3:** Experimental Setup Phase I: Collecting precision metrics

As an alternative to TAINTBENCH, one could also use other popular benchmarks such as DROIDBENCH [Arz+14] or ICC-BENCH [Wei+18]. They do not fit our study well because the apps in those benchmarks are designed to test the handling of individual Android-specific features. Their call graphs do not contain more than a few methods, which would not show the differences between different call-graph algorithms.

In this phase, we evaluate the precision of FLOWDROID under different call graphs, each with varying degrees of precision. We run FLOWDROID with each of the 31 call graphs on each of the 39 apps from TAINTBENCH, in total we conduct $31 \cdot 39 = 1,209$ runs. We compare the taint analysis results against the ground truth from TAINTBENCH and compute the precision of FLOWDROID with each call graph. In the end, we select a set of call-graph algorithms that lead to the most precise taint analysis results.

### Phase II

Figure 4.4 shows the setup for the second phase of our experimental study. In this phase, we use 20 popular Android applications from the Play Store (for the names, see Table 4.4) and FLOWDROID's default sources & sinks. We chose to use apps from the Play Store to conduct the performance evaluations because, during our preliminary evaluations on TAINTBENCH apps, we discovered that it is rather cheap to analyze them. We believe analysis performance on TAINTBENCH may not be *representative* of analysis performance on large-scale code bases.

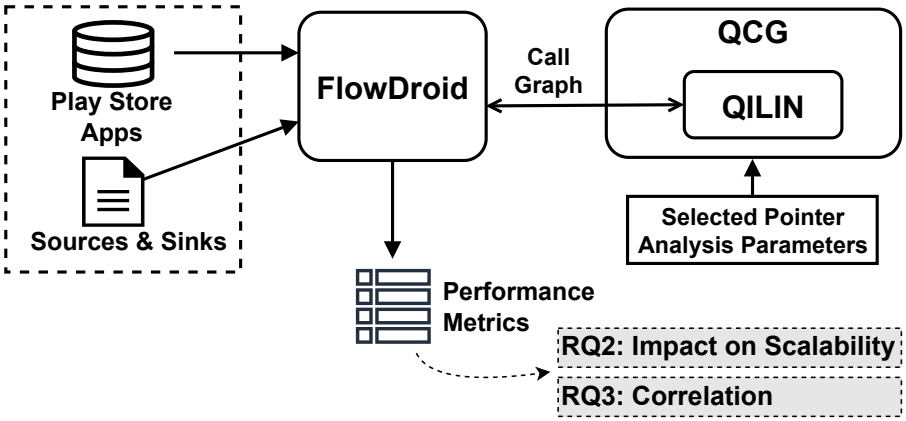



**Figure 4.4:** Experimental Setup Phase II: Collecting performance metrics

In this phase, we evaluate the call graph's impact on the scalability of FLOWDROID in terms of runtime and memory consumption. We run FLOWDROID with 8 different call-graph algorithms, where 4 of them are the default algorithms in FLOWDROID and 4 of them are selected as the most precise algorithms in *Phase I*. The default call-graph options in FLOWDROID represent a set of most commonly used (cheap and rather imprecise) call graphs in literature, namely CHA (class hierarchy analysis) [DGC95], RTA (rapid type analysis) [BS96], VTA (variable type analysis) [Sun+00], and SPARK [Lho03]. The selected algorithms from QILIN represent a set of state-of-the-art call graphs that rely on fine-grained context-sensitive pointer analyses. In *Phase 2*, in total, we conduct $20x8$ distinct runs and repeat each distinct run 3 times to account for the noise on runtime and memory measurements. We perform the evaluations on a Linux virtual machine with 22 CPUs and 256 GB of memory. Each run was given a 5-hour time budget and 220 GB memory budget by configuring the JVM's maximum heap size (`-Xmx220g`).

## 4.5 Experimental Results

We next present the experimental results. We start by measuring the precision (RQ1) of the call graphs and selecting the most precise ones to compare against a set of commonly used call graphs. Then we measure the call graphs' impact on the scalability of the IFDS analysis (RQ2). Finally, we investigate the correlation (RQ3) between the performance impact and the number of interprocedural edges that the data-flow analysis obtains from the call graphs.

### 4.5.1 RQ1: How does call graph precision impact the precision of IFDS analyses?

Table 4.2 shows the precision measurements of the call graphs and taint analysis measurements when using each call graph. Column 1 shows the call graph algorithms. Column 2 shows the total number of call edges in each call graph created for all the apps in TAINTBENCH. Column 3 shows the percentage of call edges in each call graph compared to those of the CHA-based, the least precise, call graph. Columns 4 and 5 show taint analysis findings as true positives and false positives, respectively. Columns 6 and 7 are the taint analysis precision and recall, respectively, and Column 8 is the F1 score.

**Call-Graph Precision.** We use the prefix $k$ to refer to all the parameterized instantiations of a call graph. We use the number of call edges as the precision metric for the call graphs. Our findings align with the experiments presented in previous work [LX19; Li+18; He+21; LHX21; HLX22]. EAGLE-based [LX19] call graphs (E-$k$OBJ) preserve the number of call-graph edges, compared to those of corresponding $k$OBJ-based call graphs. MAHJONG [TLX17] (M-$k$OBJ and M-$k$CFA) and ZIPPER [Li+18] (Z-$k$OBJ and Z-$k$CFA) sometimes lose precision, i.e., contain more call-graph edges than other $k$OBJ- and $k$CFA-based call graphs. TURNER-based call graphs (T-$k$OBJ) [He+21] contain either the same or slightly less number of call edges, compared to the Z-$k$OBJ-based call graphs. Selective context sensitivity-based call graphs (s-$k$CFA) [LHX21] always contain either less or the same number of call edges, compared to the Z-$k$CFA-based call graphs.

**Taint Analysis Precision.** Considering the precision of the taint analysis, we observe that, generally, a more precise call graph *does* lead to a more precise taint analysis. Table 4.2 is first sorted by *precision* and then by *recall* of the taint analysis. The most precise call graphs by the number of call edges are underlined. Each of them helps

taint analysis to yield more precise results. However, when considering the recall of the taint analysis, 2CFA- and M-2CFA-based call graphs cause a slight drop in recall.

**Table 4.2:** Call graphs and their impact on FLOWDROID's findings, sorted by descending precision and recall. The most precise call graphs are underlined. The call graphs that are selected for the evaluation in phase II are indicated in **bold**.

| Call Graph | #Call Edges | Comp. to CHA | Taint Analysis TP | FP | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|
| **s-2CFA** | 36 543 | 7.47% | 45 | 8 | 0.85 | 0.22 | 0.35 |
| **Z-2OBJ** | 36 621 | 7.49% | 45 | 8 | 0.85 | 0.22 | 0.35 |
| **M-2OBJ** | 36 660 | 7.50% | 44 | 8 | 0.85 | 0.22 | 0.35 |
| **T-1OBJ** | 36 660 | 7.50% | 43 | 8 | 0.84 | 0.21 | 0.34 |
| 2HYB | 36 621 | 7.49% | 43 | 8 | 0.84 | 0.21 | 0.34 |
| 2OBJ | 36 621 | 7.49% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 2TYPE | 36 621 | 7.49% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 1CFA | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 1HYB | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 1HYB-TYPE | 36 699 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 1OBJ | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 1TYPE | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 2CFA | 36 543 | 7.47% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| 2HYB-TYPE | 36 699 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| D-2CFA | 36 699 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| D-2OBJ | 36 699 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| E-1OBJ | 36 816 | 7.53% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| E-2OBJ | 36 738 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| M-1CFA | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| M-1OBJ | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| M-2CFA | 36 543 | 7.47% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| B-2OBJ | 36 621 | 7.49% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| Z-1CFA | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| Z-1OBJ | 36 699 | 7.51% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| Z-2CFA | 36 582 | 7.48% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| T-2OBJ | 36 621 | 7.49% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| s-1CFA | 36 660 | 7.50% | 41 | 8 | 0.84 | 0.2 | 0.32 |
| **SPARK** | 66 729 | 13.65% | 42 | 10 | 0.81 | 0.21 | 0.33 |
| **VTA** | 137 904 | 28.20% | 42 | 10 | 0.81 | 0.21 | 0.33 |
| **RTA** | 271 245 | 55.48% | 42 | 10 | 0.81 | 0.21 | 0.33 |
| **CHA** | 488 943 | 100% | 43 | 10 | 0.81 | 0.21 | 0.33 |

It appears that a more precise call graph does not always lead to more effective taint analysis results; one must also consider its impact on recall. Based on these results, we select the following QILIN-based call graphs, s-2CFA, Z-2OBJ, M-2OBJ, T-1OBJ, to be compared against the default call graphs in FLOWDROID, i.e., CHA, RTA, VTA, SPARK in phase II.

*A more precise call graph leads to a more precise taint analysis, but it does not guarantee better recall.*

### 4.5.2 RQ2: How does call graph precision impact the performance of IFDS analyses in terms of runtime and memory?

Figure 4.5 shows average analysis runtime and memory consumption when running FLOWDROID with each call graph on 20 real-world apps.

**Runtime.** We observe that FLOWDROID's runtime tends to *decrease* with increased call-graph precision. The least precise call-graph algorithm, CHA, leads to an average runtime of 3410 seconds, whereas the most precise algorithm, s-2CFA, leads to an average runtime of 2106 seconds (1.6x speedup). We observe, in particular, that all the algorithms with fine-grained context sensitivity, compared to FLOWDROID's default call-graph algorithms, lead to a smaller analysis runtime. Yet, a more precise call graph does not always reduce the total analysis runtime: Figure 4.5 shows that although RTA, VTA, and SPARK are more precise than CHA, they lead to a longer runtime than CHA. Similarly, M-2OBJ and Z-2OBJ lead to a slightly longer runtime than T-1OBJ.

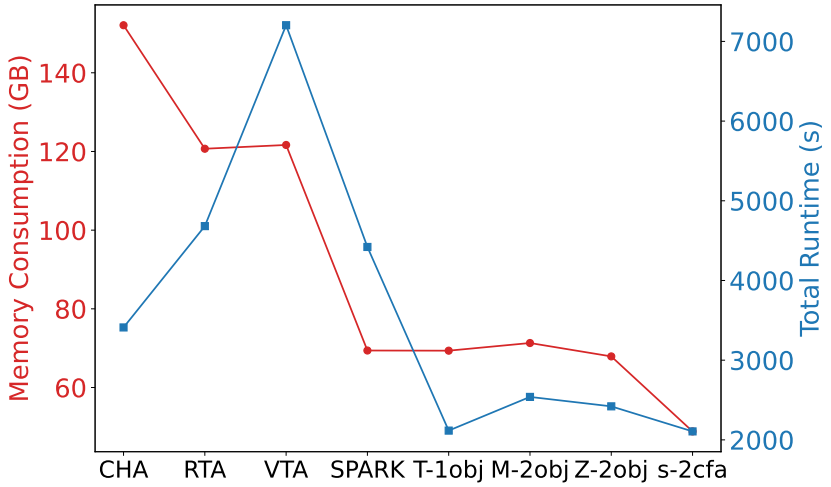



**Figure 4.5:** Average analysis runtime and memory consumption

**Memory.** FLOWDROID's memory consumption tends to *decrease* with increased call-graph precision. The least precise call-graph algorithm, CHA, leads to an average memory consumption of 152 GB, whereas the most precise algorithm, s-2CFA, leads to an average of 48 GB (69% reduction). We observe a slight increase in memory consumption when switching from context-insensitive SPARK to algorithms with

fine-grained context sensitivity. Call graphs with context sensitivity are expected to increase memory consumption because they store each method's context per a context identifier. However, surprisingly, overall memory consumption when using context-sensitive call graphs decreases substantially. We attribute this to the reduction in the number of call-edges, which reduces IFDS computation and the space required to store the method summaries.

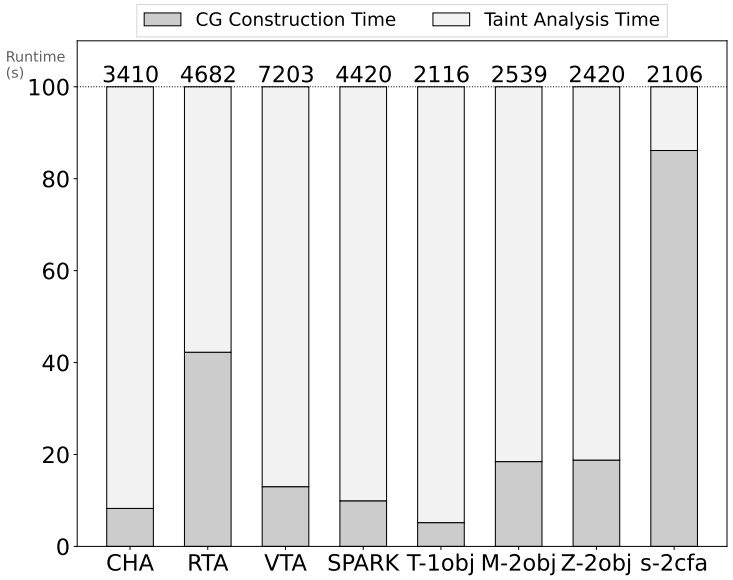



**Figure 4.6:** Relative time spent on taint analysis and CG construction (in %)

**Trade-off.** We observe a trade-off between the time invested in constructing a more precise call graph and the time saved during the downstream taint analysis. Figure 4.6 shows the relative time spent on constructing call graphs and on taint analysis. We see that precise call graphs are expensive to compute. For instance, the relative times spent on call-graph construction by RTA, VTA and SPARK are larger than that of CHA. Similarly, the relative times spent on call-graph construction by M-2OBJ, Z-2OBJ and s-2CFA are larger than those of CHA, VTA and SPARK, and T-1OBJ has the least relative call-graph construction time. Among FLOWDROID's context-insensitive call-graph algorithms, CHA has the least precision. Despite this, counterintuitively, a large number of interprocedural edges does not result in a blow-up in the IFDS analysis runtime. Among the call graphs based on fine-grained context-sensitive pointer analyses, T-1OBJ is the cheapest to compute but still precise enough that the total analysis time when using T-1OBJ is still less than the total analysis times when using M-2OBJ and Z-2OBJ, respectively. Figure 4.6 clearly shows that this is because of the larger portion of total time being spent on call-graph construction by M-2OBJ and Z-2OBJ. s-2CFA, on the other hand, shows the exemplary case,

where despite a larger portion of time spent on call-graph construction, the speedup in the IFDS analysis runtime is so much that it pays off for the total analysis duration.

> *Time invested in building precise context-sensitive call graphs pays off, but a more precise call graph does not always improve scalability, as it might take longer to build.*

### 4.5.3 RQ3: How does the number of interprocedural edges correlate with the analysis runtime and memory consumption?

Section 4.5.2 showed the call-graph precision's impact on the scalability of the IFDS analyses. This section investigates the reasons behind this impact. As explained in Section 4.3.1, a call graph can impact the scalability of the IFDS framework primarily through the number of interprocedural edges it provides. By definition, an increased number of interprocedural edges also leads to an increased number of intraprocedural edges, which increases the total number of data-flow facts that are being propagated by the IFDS solver. We, therefore, investigate the call-graph precision's impact primarily by looking at how the number of interprocedural edges correlates with runtime and memory. Moreover, we also investigate how the number of interprocedural edges correlates with the number of data-flow fact propagations.

Figure 4.7 shows the correlation between FLOWDROID's *runtime* and the number of *interprocedural edges* that FLOWDROID obtains through each call-graph algorithm. Each subplot shows a trend where an increasing number of interprocedural edges positively correlates with an increasing runtime. This correlation seems less apparent for the analysis run that uses the CHA algorithm, where the slope of the trend is the smallest among all subplots.

Figure 4.8 shows the correlation between FLOWDROID's *memory consumption* and the number of *interprocedural edges* in FLOWDROID. Each subplot shows a trend where an increasing number of interprocedural edges positively correlates with an increasing memory consumption. This correlation seems more apparent for the analysis runs that use the CHA, the RTA, and the VTA algorithms, where the slopes of the trends are the largest.

Both Figure 4.7 and 4.8 show how the number of interprocedural edges obtained through each call-graph implementation directly correlates with the scalability of the IFDS-based analyses in terms of runtime and memory. Figure 4.9, on the other

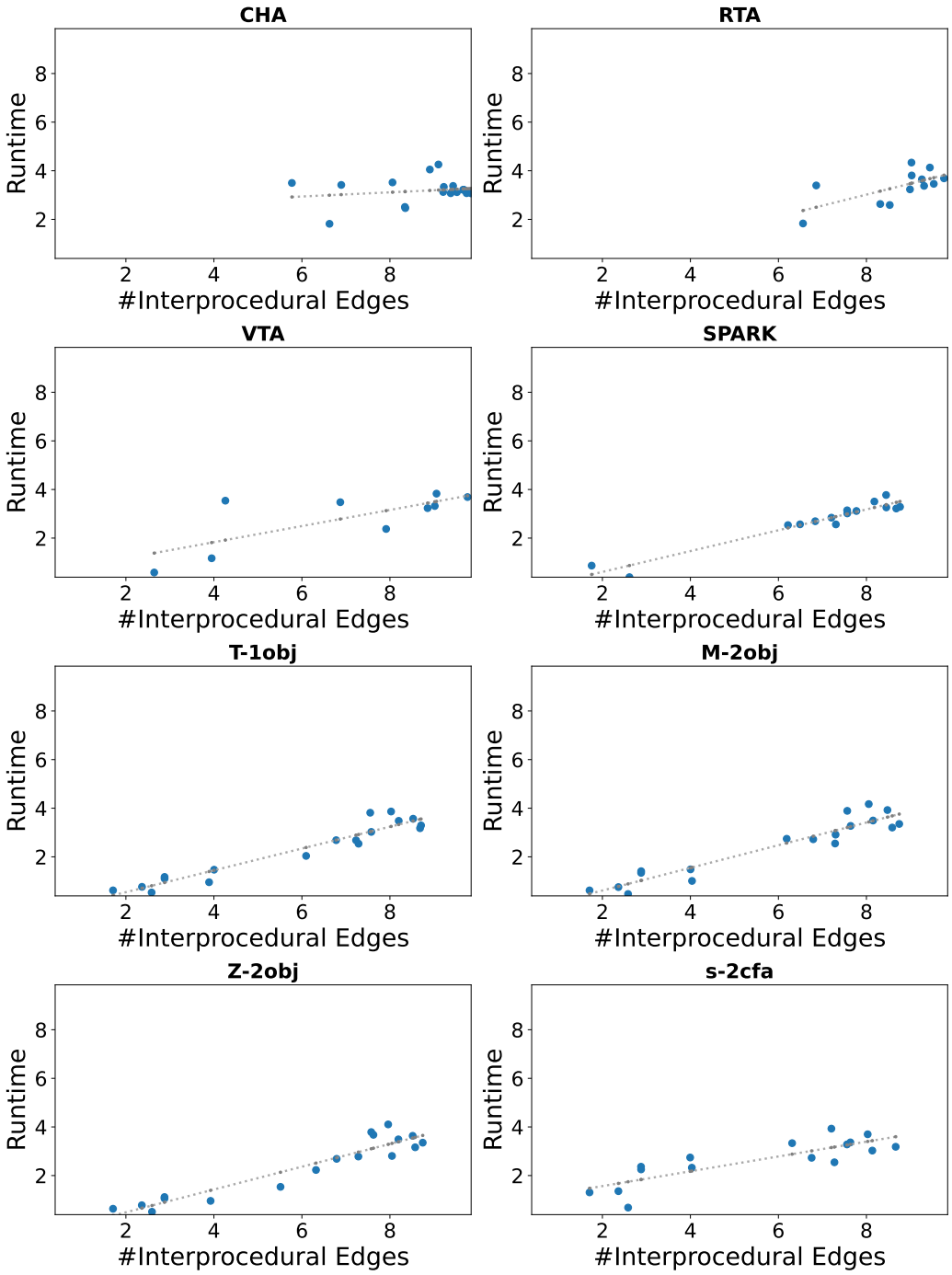



**Figure 4.7:** Number of interprocedural edges obtained from each call graph and corresponding total analysis runtime on each app (in log scale)

hand, shows the correlation between the number of data-flow fact *propagations* that FLOWDROID performs and the number of *interprocedural edges* in FLOWDROID

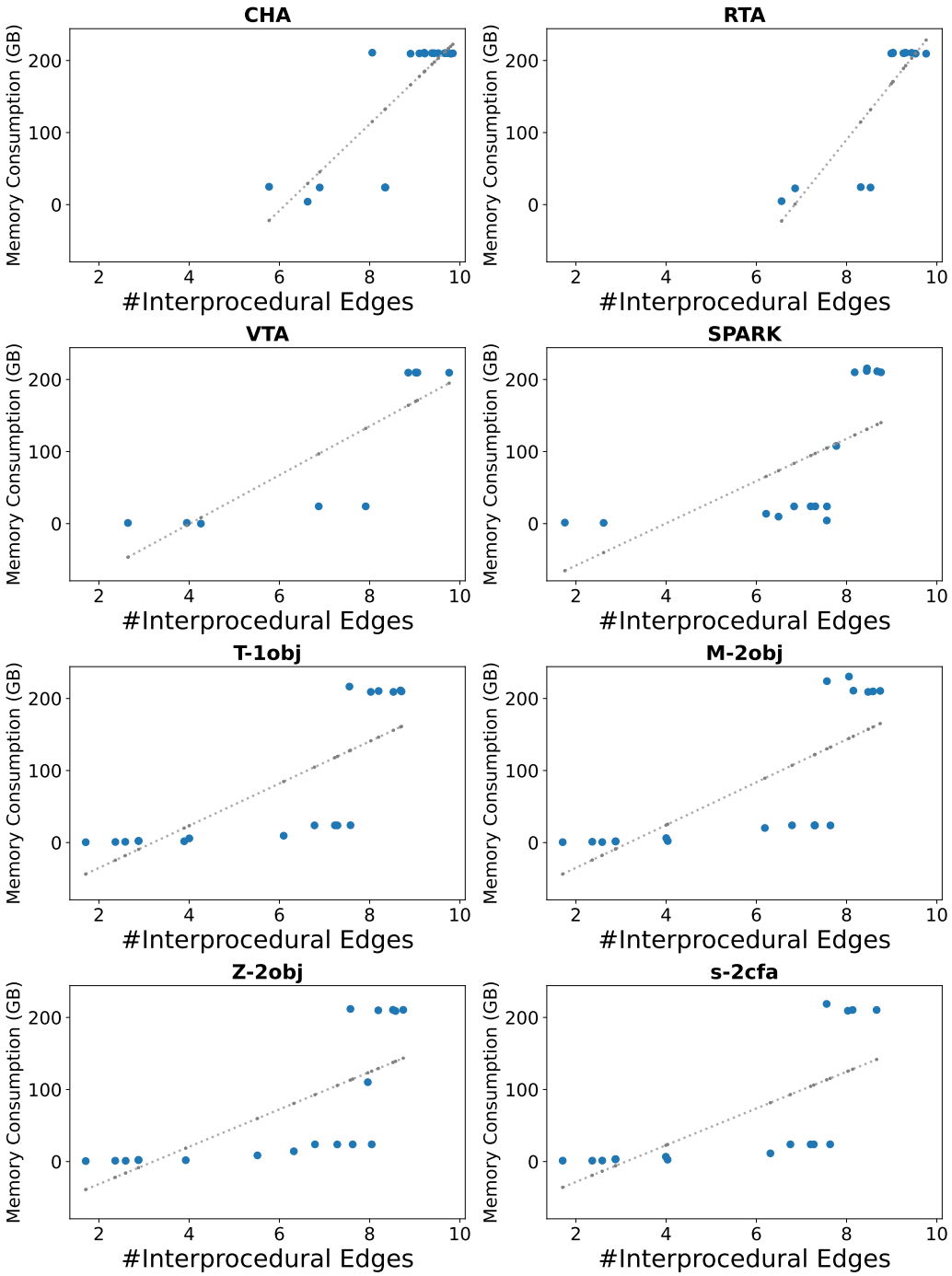



**Figure 4.8:** Number of interprocedural edges obtained from each call graph and corresponding memory consumption by the taint analysis on each app (in log scale)

obtains through each call graph. The number of data-flow fact propagations, i.e., propagations along both intraprocedural and interprocedural edges, increases when the number of interprocedural edges increases.

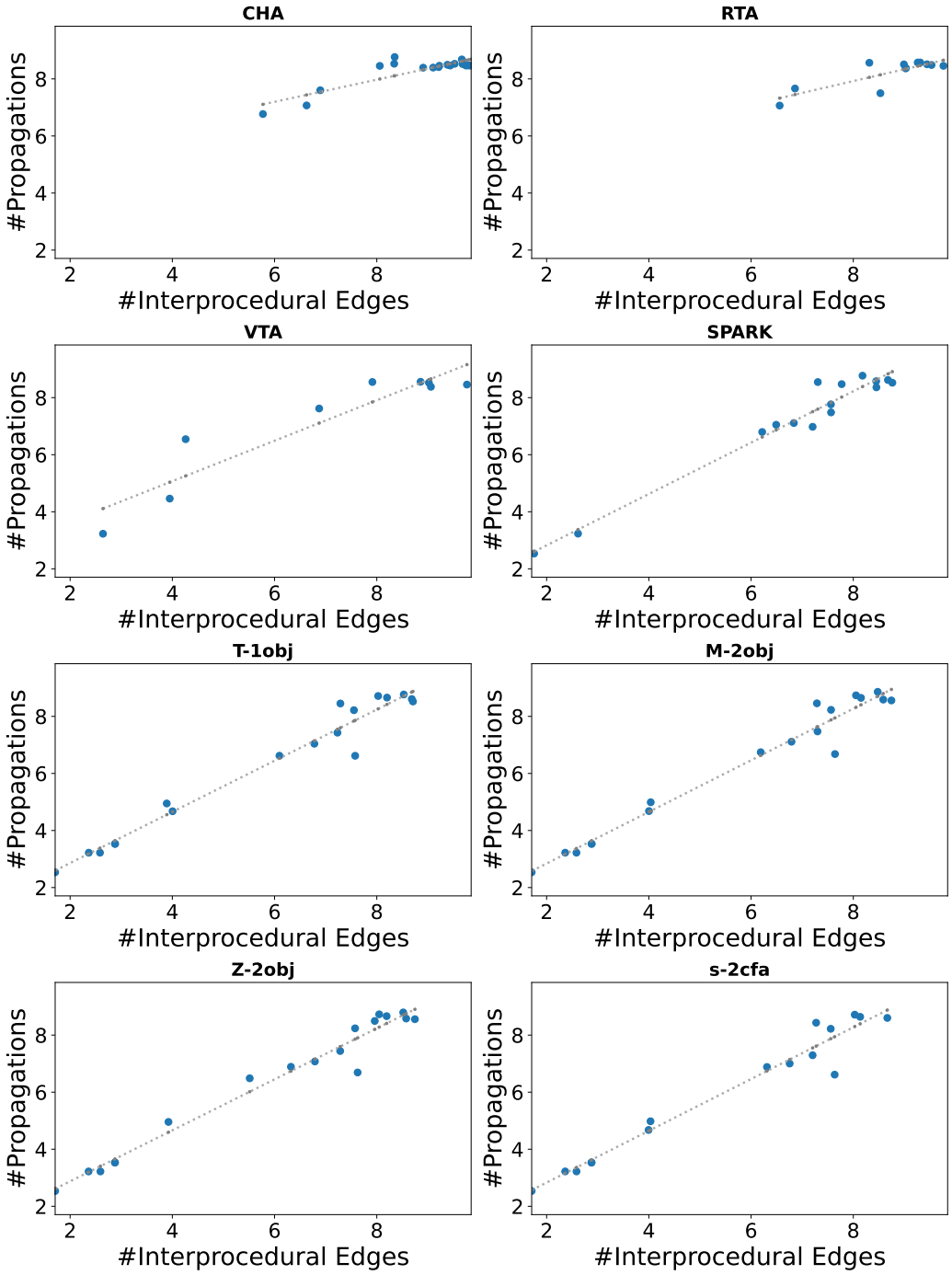



**Figure 4.9:** Number of interprocedural edges obtained from each call graph and corresponding number of data-flow fact propagations by the taint analysis on each app (in log scale)

Table 4.3 shows the Pearson coefficients and $p$-values for the correlations between the number of interprocedural edges and the performance metrics: runtime, memory consumption, and the number of data-flow fact propagations. Positive coefficients

mean a positive correlation between the number of interprocedural edges and the performance metrics, where a value between 0.5 and 1.0 indicates a strong correlation. We, therefore, report a strong correlation between all the metrics and the number of interprocedural edges for each call graph, except for CHA, where the runtime has a weak correlation. We also report that all the correlations are significant at a 0.05 significance level, except for CHA's runtime. We attribute this finding to CHA's cheap construction time, compensating for the subsequent expensive IFDS taint analysis time. We generally observe a trend where fewer interprocedural edges, i.e., fewer methods contained in call graphs, cause fewer data-flow fact propagations. This means the data-flow solver must propagate fewer data-flow facts through fewer method contexts, leading to a lower runtime and memory consumption.

> *The main promise of a precise call graph is to reduce the number of methods analyzed by the data-flow analysis. This rule holds in general: fewer methods in the call graph mean fewer data-flow fact propagations, which in turn means a lower runtime and memory consumption.*

### 4.5.4 Discussion

Table 4.4 presents the full set of absolute numbers for runtime and propagation metrics collected in Phase II. It shows the number of data-flow fact propagations (*#P*), the number of interprocedural edges (*#IE*), and the analysis runtimes (as seconds) with each call-graph algorithm and on each app. Note that, due to time (5 hours) and memory (220GB) budget, some runs resulted in time-out exceptions (TOE) or out-of-memory errors (OOM). The runtimes of the runs with TOE are still included in the final results (as 5 hours), but the ones with OOM errors are not included in the final results, to prevent rewarding the early terminations with a shorter runtime. Also on some apps, analyses terminated prematurely with a runtime exception (RE). We observe that although CHA is imprecise, the taint analysis runs that use CHA almost always terminate, except for the app Viber. RTA terminates with a runtime exception when analyzing 4 apps, and VTA terminates with a runtime exception on 5 apps. We find this behavior surprising because, in theory, they should prune an initial CHA-based call graph. These exceptions appear to be implementation bugs in the underlying SOOT framework. Therefore, when using SOOT-based call-graphs, the least precise CHA still seems to be a sound option. The analysis runs that use context-sensitive call-graph algorithms, i.e., T-1OBJ, M-2OBJ, Z-2OBJ, and s-2CFA, never terminated due to OOM, while only in a few cases they terminated due to TOE.

**Table 4.3:** Statistical significance of the correlation between #Interprocedural edges and performance metrics

| Call Graph | Metric | Coefficient | $p$-value |
|---|---|---|---|
| CHA | Runtime | 0.19 | 4.3e−1 |
| | Memory | 0.83 | 9.9e−6 |
| | #Propagations | 0.87 | 1.5e−6 |
| RTA | Runtime | 0.66 | 1.9e−2 |
| | Memory | 0.85 | 4.7e−4 |
| | #Propagations | 0.83 | 8.3e−4 |
| VTA | Runtime | 0.75 | 2.1e−2 |
| | Memory | 0.85 | 3.7e−3 |
| | #Propagations | 0.93 | 2.2e−4 |
| SPARK | Runtime | 0.96 | 7.79e−9 |
| | Memory | 0.64 | 1.0e−2 |
| | #Propagations | 0.98 | 1.72e−10 |
| T-1OBJ | Runtime | 0.96 | 3.53e−10 |
| | Memory | 0.76 | 2.68e−4 |
| | #Propagations | 0.98 | 3.63e−13 |
| M-2OBJ | Runtime | 0.95 | 1.94e−9 |
| | Memory | 0.75 | 3.34e−4 |
| | #Propagations | 0.98 | 3.02e−13 |
| Z-2OBJ | Runtime | 0.94 | 3.89e−9 |
| | Memory | 0.71 | 9.4e−4 |
| | #Propagations | 0.99 | 1.06e−13 |
| s-2CFA | Runtime | 0.83 | 6.47e−5 |
| | Memory | 0.69 | 2.84e−3 |
| | #Propagations | 0.98 | 3.34e−11 |

A Call graph can theoretically impact the scalability of the IFDS framework primarily through the number of interprocedural edges it provides. Table 4.4 shows that, in general, when the number of interprocedural edges (*#IE*) increases, analysis runtimes increase as well. In the IFDS framework, by definition, an increased number of interprocedural edges will lead to an increased number of data-flow fact propagations, i.e., the tainted variables will be propagated through an increased number of method contexts. We observe that when the number of interprocedural edges (*#IE*) increases (i.e., when using a less precise call graph), the number of data-flow fact propagations (*#P*) increases as well. However, it is hard to conclude a general rule for the correlation between the number of interprocedural edges and propagations. The number of program statements, through which the data-flow facts are propagated, is unlikely to be the same across all different methods. Although we observe a clear correlation, it is hard to predict a call graph's exact impact on the scalability solely based on the number of its call edges.

**Table 4.4:** Runtime and propagation metrics of FLOWDROID on 20 real-world apps, with each call graph

| | CHA | | | RTA | | | VTA | | | SPARK | | | T-1OBJ | | | M-2OBJ | | | Z-2OBJ | | | s-2CFA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APK | #P | #IE | time | #P | #IE | time | #P | #IE | time | #P | #IE | time | #P | #IE | time | #P | #IE | time | #P | #IE | time | #P | #IE | time |
| gkeyboard | 5.0M | 592K | 3149 | 0 | 0 | 7957 | 3M | 1.8K | 3464 | 9M | 16M | 693 | 0 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 5 |
| ucmobile | 294M | 2.7B | 2372 | - | - | 17957 | - | - | 17957 | 58M | 36M | 1390 | 518M | 106M | 7315 | 546M | 113M | 14801 | 535M | 111M | 640 | 519M | 106M | 4993 |
| gclock | 340M | 219M | 324 | 367M | 206M | 428 | 353M | 82M | 237 | 351M | 20M | 366 | 282M | 19M | 346 | 286M | 19M | 353 | 312M | 91M | 12706 | 271M | 18M | 350 |
| whatsapp | 313M | 2.4B | 1189 | 322M | 977M | 1715 | - | - | RE | 12M | 6M | 494 | 11M | 6M | 483 | 12M | 6M | 528 | 11M | 6M | 487 | 10M | 5M | 538 |
| garena | 338M | 3.3M | 1317 | 376M | 2B | 2386 | - | - | 17957 | - | - | OOM | 88K | 7.7K | 9 | 97K | 11K | 10 | 90K | 8.4K | 9 | 95K | 11K | 210 |
| shareme | 293M | 7B | 1174 | 287M | 5.8B | 4871 | 287M | 5.8B | 4871 | 297M | 59M | 1305 | 47K | 10K | 29 | 47K | 10K | 30 | 3M | 326K | 34 | 47K | 9.8K | 553 |
| mxplayer | 289M | 1.7B | 2167 | - | - | RE | - | - | RE | 588M | 151M | 3192 | 455M | 158M | 3019 | 445M | 141M | 3118 | 461M | 155M | 3082 | 437M | 136M | 1064 |
| shareit | 286M | 114M | 3307 | - | - | RE | - | - | OOM | - | - | 17947 | 165M | 35M | 6522 | 169M | 36M | 7796 | 172M | 37M | 6142 | 166M | 36M | 1906 |
| msword | 294M | 6.3B | 1379 | - | - | RE | - | - | RE | - | - | 17949 | 3K | 757 | 14 | 3K | 760 | 25 | 3K | 754 | 13 | 3K | 754 | 231 |
| tiktok | 247M | 808M | 11242 | - | - | OOM | - | - | OOM | 377M | 280M | 5928 | 584M | 337M | 3689 | 724M | 300M | 8408 | 623M | 329M | 4309 | - | - | 17940 |
| viber | - | - | 17940 | - | - | OOM | - | - | ER | 333M | 580M | 1930 | 332M | 510M | 1963 | 361M | 556M | 2255 | 361M | 556M | 2255 | - | - | 17940 |
| gfiles | 249M | 1.3B | 17937 | 234M | 1B | 17937 | - | - | 17939 | - | - | 17941 | - | - | 17941 | - | - | 17941 | - | - | 17939 | - | - | 17940 |
| webview | 481M | 4.6B | 1644 | 31M | 338M | 389 | 28K | 8.9K | 14 | 343 | 57 | 7 | 343 | 51 | 4 | 343 | 51 | 4 | 343 | 51 | 4 | 343 | 51 | 20 |
| netflix | 333M | 4.8B | 1661 | 326M | 2.8B | 13540 | 359M | 718M | 1708 | 11M | 3M | 368 | 4M | 1M | 109 | 5M | 1M | 552 | 7M | 2M | 169 | 7M | 2M | 2152 |
| mivideo | 40M | 7M | 2608 | 46M | 7M | 2487 | 41M | 7M | 2993 | 30M | 36M | 1037 | 26M | 17M | 479 | 29M | 19M | 817 | 27M | 19M | 606 | 19M | 16M | 8509 |
| zoho-show | 585M | 223M | 290 | 375M | 1.8B | 4439 | 333M | 1B | 2089 | 415M | 476M | 1649 | 405M | 484M | 1500 | 387M | 385M | 1601 | 381M | 375M | 1456 | 400M | 462M | 1528 |
| excel | 299M | 5.7B | 1387 | - | - | RE | - | - | RE | - | - | 17941 | 3K | 757 | 12 | 3K | 758 | 21 | 3K | 754 | 11 | 3K | 757 | 179 |
| fblite | 11M | 4M | 65 | 11M | 3M | 67 | 1K | 440 | 3 | 1K | 412 | 2 | 1K | 384 | 3 | 1K | 383 | 2 | 1K | 392 | 3 | 1K | 384 | 4 |
| candycrush | 295M | 5.5B | 1193 | 307M | 3.4B | 2892 | - | - | 17940 | 6M | 1M | 342 | 1K | 232 | 5 | 1K | 231 | 5 | 1K | 230 | 6 | 1K | 231 | 22 |
| gsearchlite | 259M | 1.6B | 1337 | 239M | 1B | 6397 | 240M | 1.1M | 6752 | 229M | 283M | 1827 | 4M | 37M | 1058 | 4M | 43M | 1871 | 4M | 42M | 4746 | 4M | 43M | 2314 |

Out-of-memory errors are indicated with an OOM, runtime exceptions are indicated with an RE and timeout-errors are indicated in blue. For brevity, we use K for thousand, M for million and B for billion.

### 4.5.5 Threats to Validity

Similarly to previous work [LX19; Li+18; He+21; LHX21; HLX22], we use the number of call-graph edges discovered to measure the precision of call-graph algorithms. Previous work also often uses the number of polymorphic calls discovered as a metric, in this study, however, we instead use the number of interprocedural edges. We prefer this because, although interprocedural edges are obtained through polymorphic calls, only a subset of the polymorphic calls end up being interprocedural edges in the IFDS framework. The IFDS framework creates interprocedural edges if and only if data-flow facts need to be propagated into callee methods.

We perform the precision measurements on TAINTBENCH in phase I and performance metrics on a set of popular apps from the Play Store in phase II. Doing both measurements on the same set of apps was not possible, because the apps from taint analysis benchmarks [Arz+14; Wei+18; Luo+22] contain ground truth to asses analysis precision but they are not complex enough to test analysis scalability. Similarly, the apps from the Play Store are complex enough to test analysis scalability but they do not contain ground truth to asses analysis precision. Therefore, the reported evaluation results might depend on the target apps used in each phase. We, nevertheless, selected a set of the most popular apps to obtain representative results. The set of most precise call-graph algorithms that we selected after obtaining the precision results in phase I, might differ when using different benchmarks. However, to the best of our knowledge, TAINTBENCH is the only benchmark of real-world Android malware applications with ground truth.

Our work measures the call-graph precision's impact on the scalability of IFDS-based data-flow analyses. We employ FLOWDROID as the reference IFDS analysis implementation because of its maturity and wide acceptance in the research community. The results presented in this work should theoretically generalize to other IFDS-based [RHS95] analysis implementations as well as to IDE-based [SRH96] analyses.

## 4.6 Related Work

This section presents the related work on call graphs and approaches that aim to improve the scalability of the IFDS-based data-flow analyses.

### 4.6.1 Call Graphs

Call graphs are indispensable data structures for interprocedural program analyses and have been studied intensively in the literature [TP00; GC01]. CHA [DGC95] and RTA [BS96] are classical examples of call-graph algorithms that do not rely on pointer information. Modern call-graph techniques rely on pointer analyses to improve their precision. Pointer analyses aim to compute a set of memory locations that each variable in a program might point to. Andersen's pointer analysis [And94] is one of the earlier works in this domain that is widely used as a baseline technique. Sundaresan et al. [Sun+00] show how pointer information can aid with virtual method resolution at polymorphic call sites. SPARK pointer analysis framework [Lho03] of Soot [Val+99] enables one to obtain cheap context-insensitive pointer information, which is the default choice in many Soot-based analyses, such as FLOWDROID.

Handling contexts in pointer analysis is key to achieving increased precision. Emami et al. [EGH94] introduced a context-sensitive pointer analysis that simply models the entire heap as a single concrete location. Milanova et al. [MRR02] proposed an object-sensitive pointer analysis technique that analyzes methods individually for each receiving object. Smaragdakis et al. [SBL11] introduced type-sensitive pointer analysis that utilizes types as context. Kastrinis et al. [KS13] combined both call-site- and object-sensitivity in pointer analysis, formulating a hybrid sensitivity approach.

Modern pointer analysis techniques aim to increase scalability by intelligently deciding when to sacrifice precision. MAHJONG [TLX17] is a technique that aims to serve a specific set of type-dependent analysis clients, such as call graphs. EAGLE [LX19] and ZIPPER [Li+18] are partial context-sensitivity approaches that perform context-sensitive analysis only for selected allocation sites or methods. QILIN [HLX22] is a pointer analysis framework that implements many of the modern pointer analysis techniques.

Sui et al. [Sui+20] measured the recall of call-graph algorithms in practice and showed that handling dynamic language features can significantly improve recall. Reif et al. [Rei+19] performed an extensive study to find sources of unsoundness for Java call graphs. Neupane and Thakur [NT23] measured the effect of FLOWDROID's default call-graph algorithms on its precision. In this work, we present an empirical study on the impact of call-graph precision on the scalability of the IFDS-based data-flow analyses, by using an extensive set of call graphs that use state-of-the-art context-sensitive pointer analyses.

### 4.6.2 Scalable IFDS Extensions

As discussed by related work [Avd+15; Hua+15; Sch+24], scaling the IFDS- and IDE-based data-flow analyses is an open challenge. Recently, many new techniques have been introduced to address this challenge. SPARSEDROID [He+19] speeds up IFDS analyses through sparsification, where intraprocedural edges are created not for every statement in a method but instead only for statements where data-flow facts are being used. DISKDROID [Li+21] improves the memory consumption of IFDS analyses by storing analysis data on disk when memory consumption reaches a threshold. DSTREAM [Wan+23] scales IFDS analyses through a fine-grained and highly parallel streaming-based computation model. CLEANDROID [Arz21] improves the memory footprint of IFDS analyses through an intelligent garbage collection mechanism that safely removes the intraprocedural edges that the IFDS solver no longer needs. FPC [He+23] improves this approach using a fine-grained garbage collection at the data-flow fact level. Schiebel et al. [Sch+24] propose two optimizations for IDE-based data-flow analyses, and implement these optimizations on top of the IDE solver in PhASAR [SHB19], an LLVM-based static analysis framework. They first present an efficient layout for storing the jump-functions in memory, and then they leverage the intelligent garbage collection approach of CLEANDROID [Arz21].

Interestingly, none of the scalable IFDS extensions mention which call graphs they use under the hood, except for CLEANDROID, which reports using FLOWDROID's default context-insensitive call-graph algorithm SPARK [Lho03]. We argue that all of these techniques would benefit from employing a more precise call graph because they will essentially have fewer interprocedural edges to compute.

## 4.7 Conclusion

This chapter presented the second major contribution of this thesis, an empirical study on the impact of call-graph precision on the precision and scalability of IFDS-based data-flow analyses (*Contribution 2*). Interprocedural data-flow analysis frameworks like IFDS rely on call graphs to propagate data-flow facts across methods. We show, for the specific case of IFDS, that fewer call edges lead to an increase in data-flow analysis precision, but they may also lead to a slight decrease in analysis recall. We also show that, in general, an increased call-graph precision pays off for the IFDS analyses in terms of memory consumption. In terms of runtime, there appears to be a sweet spot in the trade-off between the precision, and thus the

construction time, of a call graph and the total analysis runtime. Nonetheless, we observe that fine-grained context-sensitive call graphs lead to significantly better IFDS analysis runtimes while improving the analysis precision. Since IDE only differentiates from IFDS in annotating the intraprocedural edges with the mappings on the value domain, we expect the results to carry over to IDE-based data-flow analyses.

As explained in Section 2.4, pointer analyses are essential for improving the precision of data-flow analyses. They help data-flow analyses reason about aliasing relationships between program variables, which may point to the same objects during program execution. In the next chapter (Chapter 5), we show precise pointer analyses can also benefit from fact-specific sparsification. To this end, we implement SPARSE-BOOMERANG (*Contribution 3*), a sparse alternative to the BOOMERANG [Spä+16] demand-driven pointer analysis.

# 5 SparseBoomerang: Query-Specific Sparsification for Demand-Driven Pointer Analysis

As explained in Section 2.4, precise data-flow analyses rely on pointer analyses to resolve aliasing. Exhaustive pointer analyses compute pointer information for the whole program upfront, which is argued to be inefficient [Spä+16]. Client data-flow analyses need pointer information only at statements that read from and write to the heap. Demand-driven pointer analysis seeks to be efficient by computing pointer information only for variables on which a demand is raised, through a points-to or alias query. Yet, research has shown that when applied to large-scale programs, even demand-driven analyses can become expensive in terms of memory and runtime.

In Chapter 3, we have shown how fact-specific sparsification can improve the scalability of whole-program data-flow analysis (*Contribution 1*). In this chapter, we present the third major contribution of this thesis (*Contribution 3*) by investigating to what extent demand-driven pointer analysis can be accelerated further if being executed over a sparse control-flow graph (CFG), specialized to those queries. In the context of demand-driven pointer analysis, instead of the term *fact-specific*, we use *query-specific*. An alias query typically contains more information than an individual data-flow fact.

Static program analysis clients, independent of their domain, require pointer information [Hin01]. For that purpose, they sometimes implement a pointer analysis as part of the client analysis [Arz+14], but more frequently use a pre-existing pointer analysis [LBS19]. Fast and precise pointer analysis is still an open challenge for large-scale programs. To be precise, pointer analyses track calling contexts, fields, and statements, but that can hinder scalability. To be more scalable, many current pointer analyses are performed in a demand-driven manner [HT01], as opposed to conducting an exhaustive whole-program analysis [Lho03]. They benefit from the fact that client analyses frequently require pointer information only for certain variables at certain program points. For instance, assume a direct assignment in a

taint analysis, e.g., `x.f = t` where `t` is tainted. Here, aliases of `x` need to be known to the taint analysis so that this analysis can taint the `f` fields of `x`'s aliases, too. Demand-driven pointer analyses exploit just that: They compute alias information only for variables on which clients raise a demand through a query. Yet, previous work has shown that even demand-driven analyses can be expensive when run on large-scale programs [Spä+16].

BOOMERANG [Spä+16] is a state-of-the-art demand-driven pointer analysis framework that uses synchronized pushdown systems (SPDS) [SAB19]. Pushdown systems (PDS) [Rep+05] apply to context-free language reachability problems, with which context- and field-sensitivity can be modeled [SAB19]. BOOMERANG intersects two PDS that model context- and field-sensitivity, respectively. Both PDS depend on *rules* that correspond to data-flow functions. In this work, we exploit that many of these rules are *redundant* as they only affect data-flow facts that do not matter to the end result. Data-flow facts in pointer analysis correspond to the variables and their aliases. Redundant rules exist because control flow graphs (CFG) not only contain statements that affect the alias relationships, but also many other statements that do not. The beauty of demand-driven pointer analysis is that one knows the exact *query variable* (i.e., the variable, whose aliases are being queried), ahead of the analysis time. Therefore, when answering a raised demand, one can sparsify the CFG by removing the statements that are irrelevant to the result *for the particular query variable*, and thus omit the redundant rules during the construction of the SPDS.

Although our query-specific sparsification is inspired by the sparse IFDS work presented by He et al. [He+19], this work presents insights from two novel aspects when applying such sparsifications. First, we demonstrate fact-specific sparsification on the domain of pointer analysis, which enable us to propose two different sparsification strategies that exploit characteristics of this domain. Second, we demonstrate fact-specific sparsification on top a novel SPDS solver, showing that fact-specific sparsification is not limited to IFDS- and IDE-based solvers.

## 5.1 Motivation

Previous work has successfully applied sparsification to improve the scalability of general data-flow analyses [Shi+18; SX16; Oh+12] and pointer analysis in particular [Sui+11; HL11; HL09]. All these approaches create sparse versions of the control flow graphs (CFGs) of a target program. These sparse versions are often

referred to as sparse value flow graphs (SVFGs) [SX16] or sparse control flow graphs(SCFGs) [He+19]. Most previous approaches create those SCFGs in a pre-analysis stage, for the whole program, and thus settle for the information available at that stage. Recent work by He et al. [He+19] showed that one can increase sparseness, i.e., omit from the CFG more irrelevant statements, by specializing the SCFGs to the individual data-flow facts. Their work was applied to the IFDS [RHS95] framework, which applies only to distributive analysis problems. Pointer analysis is known to be non-distributive [PK13]. In this work, we thus investigate to what extent one can make use of the idea of fact-specific sparseness, also in pointer analysis. In the proposed framework SPARSEBOOMERANG, the analysis creates a new SCFG specific to any queried value.

The goal of sparsification is to speed up the analysis run by restricting it to fewer program statements, while ideally generating results identical to those of an exhaustive analysis. Yet, the creation of the SCFGs incurs a cost in both memory and runtime. Sparsification pays off when the savings during evaluating the sparse graph, in comparison to the original exhaustive graph, outweigh the construction time. To investigate this performance trade-off, this chapter presents two sparsification strategies with varying degrees of sparsification. Both strategies create on-demand SCFGs specific to each alias query. First, *type-aware sparsification* (TAS), where the resulting CFG only consists of the statements containing variables that are type-compatible with the query variable. Second, *alias-aware sparsification* (AAS), where the resulting CFG consists of the def-use chains of the query variable and all its intra-procedural aliases. Those strategies mirror designs published earlier in the context of virtual call resolution [Sun+00], declared-type analysis (DTA), and variable-type analysis (VTA), respectively. DTA and VTA create assignment chains, where each node represents a variable either as its declared type (in DTA) or as itself (in VTA). Our strategies create def-use chains, where each node represents a statement either with the types (in TAS) or with the variables (in AAS) it contains.

## 5.2 Contributions

In this chapter, we evaluate the applicability of the proposed two sparsification strategies within the SPDS framework. For that, we implement SPARSEBOOMERANG by extending the SPDS-based BOOMERANG. To validate whether the two sparsification strategies maintain the precision of the original exhaustive BOOMERANG, we run all approaches on the POINTERBENCH [Eng19] benchmark suite for alias analysis. To evaluate the performance impact of the strategies, we run both BOOMERANG

and SPARSEBOOMERANG on real-world Android applications. To this end, we extend FLOWDROID, a state-of-the-art taint analysis client for Android applications, so that it creates on-demand alias queries to BOOMERANG and SPARSEBOOMERANG. Evaluation results show that SPARSEBOOMERANG using either of the sparsification strategies solves the alias queries on average twice as fast as BOOMERANG, and while maintaining full precision. The performance gains achieved by the demand-driven pointer analysis are reflected in the taint analysis client, FLOWDROID. To summarize, this paper presents these original contributions:

- Two sparsification strategies: type-aware sparsification and alias-aware sparsification for demand-driven pointer analysis,
- a sparse implementation of BOOMERANG, which we call SPARSEBOOMERANG, which maintains BOOMERANG's precision, and
- a modification of FLOWDROID that uses demand-driven pointer analyses BOOMERANG and SPARSEBOOMERANG, and their performance evaluation on real-world android apps.

## 5.3 Background

In this section, we introduce the key concepts that are required to understand the contributions presented in this chapter. We first explain why pointer analysis is a non-distributive problem. We then explain the concept of demand-driven pointer analysis, and how it can benefit from fact-specific sparsification. Finally, we briefly explain BOOMERANG's approach to pointer analysis.

### 5.3.1 Non-Distributivity of Pointer Analysis

Pointer analysis determines which program variables can point to which objects at runtime. It is essential in real-world analysis settings where multiple program variables frequently point to the same object. Two variables that point to the same object are called aliases. Such alias information is crucial for a precise data-flow analysis, enabling the tracking of indirect data flows through aliases. Pointer analysis is usually not distributive: at an assignment `x.f = t`; one must assign aliases of `t` to the `f`-fields of *all* the aliases of `x`.

```
1  if(...){
2    x = new A();
3  } else {
4    x = new B();
5  }
6
7  if(x instanceof A){
8    x.f1 = 23;
9  } else if(x instanceof B){
10   x.f2 = 42;
11 }
```

**Figure 5.1:** Example code demonstraing non-distributivity of pointer analysis

For instance, assume performing pointer analysis on the example code in Figure 5.1 and recall the distributivity property presented with the equation 2.2 on page 12, where the MOP (meet over all paths) solution equals to the MFP (maximal fixed-point) solution. We first apply the MOP approach, i.e., maintain data-flow facts from all the branches separately and then merge at the end. In this solution, after `l2` (line 2), $x \mapsto A$, and after `l4`, $x \mapsto B$. Since these facts are kept separate, we can conclude that after `l8`, $\{A\}.f1 \mapsto 23$, and after `l10`, $\{B\}.f2 \mapsto 42$. Therefore the MOP solution $S_{MOP} = \{\{A\}.f1 \mapsto 23, \{B\}.f2 \mapsto 42\}$ when the analysis terminates. We then apply the MFP approach, i.e., merge data-flow facts from different branches immediately. In this case, after `l5`, $x \mapsto \{A, B\}$, as we merge the results from `l2` and `l4`. Using this information, we conclude that after `l8`, $\{\{A\}.f1 \mapsto 23, \{B\}.f1 \mapsto 23\}$, and after l10, $\{\{A\}.f2 \mapsto 42, \{B\}.f2 \mapsto 42\}$ as in both branches $x$ may point to an instance of $A$ and of $B$. Therefore the MFP solution is $S_{MFP} = \{\{A\}.f1 \mapsto 23, \{B\}.f1 \mapsto 23, \{A\}.f2 \mapsto 42, \{B\}.f2 \mapsto 42\}$. Since $S_{MOP} \neq S_{MFP}$, we see that the pointer analysis problem violates the distributivity property.

### 5.3.2 Demand-driven Pointer Analysis

Because pointer analysis is not distributive, one cannot usually soundly handle all aliases independently, and the IFDS framework is not applicable by default, thus neither is Sparse IFDS. Yet, as opposed to whole-program pointer analysis, demand-driven pointer analysis [HT01] is performed only for variables on which a demand, e.g., a pointer or alias query, is raised. It computes just enough information to satisfy the query. Interestingly, as Späth et al. showed [Spä+16], one can decompose a flow-sensitive pointer analysis such that when queries raise sub-queries at "points of indirection" (POI), e.g., at reads and writes to/from the heap, the evaluation of those sub-queries *does* become a distributive and thus distributively solvable analysis

problem. Figure 2.8 on page 17 shows the data-flow graphs that a taint analysis would produce with alias information. To know that the analysis must taint `a2.f` at `L3`, it must know that `a1` and `a2` alias at that point. In the case of a context- and flow-sensitive demand-driven pointer analysis, an alias query would look as follows:

$$\textit{mayAliases}(v, s, m)$$

$v$ is the query variable for which alias information is required. $s$ is the query statement and $m$ its surrounding method. Thus, the *query* in Figure 2.8 on page 17 would be instantiated as:

$$\textit{mayAliases}(a1, L3, foo())$$

In demand-driven pointer analysis, the query variable is used as the initial data-flow fact. It is provided explicitly, ahead of the analysis time. This allows one to perform on-demand fact-specific sparsification, building an SCFG specific to each particular query.

### 5.3.3 BOOMERANG

With BOOMERANG, Späth et al. [Spä+16] showed that the pointer analysis problem can be modeled with distributive *sub-queries* that can still be solved with the IFDS algorithm. Such sub-queries are created at POIs (points of indirections). POIs cause the outer IFDS solver to instantiate sub-queries in the opposite direction, which are then again solved by inner IFDS solvers. BOOMERANG handles the following POIs:

- **Allocation Site:** Upon finding an allocation site (new object creation) during a backward analysis, a forward sub-query is created to find out which variables point to this object.
- **Field Write and Read:** Upon finding a field write statement during a forward analysis, a backward sub-query is created to find the aliases of the base variable of the field. Field read statements are handled in a backward analysis similar to the field write statements in the forward analysis.
- **Return and Call:** Return indirections are caused by the context change during the forward analysis, and call indirections are caused during the backward analysis.

Recently, BOOMERANG has been reimplemented with SPDS [SAB19] instead of the IFDS framework. IFDS and PDS are equally expressive and can be used to model the same inter-procedural data-flow problems [Rep+05]. SPDS uses two pushdown

systems, Call-PDS and Field-PDS. The Call-PDS models flow- and context-sensitive data-flow analysis. Its push rules correspond to call-flow functions of the IFDS framework, whereas its pop rules correspond to return-flow functions. Normal rules of the Call-PDS are equivalent to the intra-procedural flow functions of IFDS, i.e., normal-flow functions and call-to-return-flow functions. The Field-PDS models flow- and field-sensitive data-flow analysis. Push and pop rules of the Field-PDS represent field store and field load statements respectively, where its normal rules correspond to assignments. SPDS improves over the IFDS via its compact encoding of field-sensitivity with the Field-PDS [SAB19]. Call-PDS and Field-PDS both benefit from the proposed sparsification strategies because they both process the same CFG for the same query variable. From the sparsification point of view, the underlying solver (IFDS-, or SPDS-based) does not directly matter because, in the end, they both compute the same data flows over the same CFGs. Therefore, although our sparsification technique is inspired by the sparse IFDS work [He+19], we implement it on top of BOOMERANG's new SPDS-based solver.

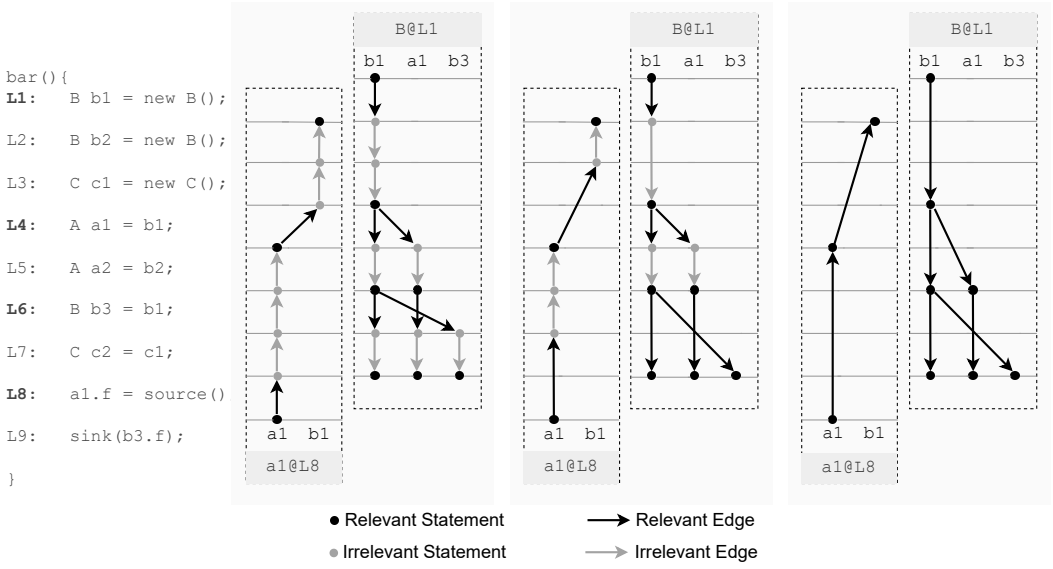
## 5.4 Demand-Driven Sparsification Strategies

In Section 5.3.2, we showed that an alias query consists of a query variable, a statement where its aliases are required, and a method that defines its context. Figure 5.2a shows an input program for pointer analysis. An example alias query looks as follows:

$$\mathit{mayAliases}(a1, L8, bar())$$

We seek to find the aliases of `a1` at line `L8` in method `bar()`. Figure 5.2b shows how BOOMERANG performs this on a non-sparse CFG by default. It first initiates a backward pass (for `a1@L8`) to find the allocation site of the object that `a1` points to. After finding the allocation site at `L1`, a POI, a forward pass for `B@L1` is initiated. The forward pass continues until it reaches the initial query location, yielding all variables that point to the same object (`B@L1`) as the query variable.

In fact, only the statements in lines `L1`, `L4`, `L6` can affect the aliasing relationships of `a1`, where `L8` is the query statement. Therefore, the edges that originate from the other statements are irrelevant. Irrelevant statements and edges that start from them are highlighted in Figure 5.2. Note that after sparsification, *relevant edges* connect to the *relevant statements* that are next in the respective SCFGs. By sparsifying CFGs, i.e., removing redundant transition rules, a significant amount of computation time




**(a)** Input Program **(b)** Non-sparse CFG **(c)** Type-aware SCFG **(d)** Alias-aware SCFG

**Figure 5.2:** Data-flow Graphs of BOOMERANG's Analysis on Non-Sparse CFG and SPARSE-BOOMERANG's Analyses on Type-aware and Alias-aware SCFGs (Sparse Control Flow Graphs) on an Input Program where `B` is a subtype of `A`

can potentially be saved. However, sparsification also consumes computation time, which depends on the degree of sparsification. Therefore, in the remainder of this paper, we will seek to validate or refute the following assumption:

> **Assumption**: *A fine-grained SCFG is cheap to analyze, yet expensive to build, whereas a coarse SCFG is cheap to build, yet more expensive to analyze.*

To investigate whether this assumption holds, we implemented two sparsification strategies with different degrees of sparsification. We will present these next.

Given a query, *mayAliases*$(v, s, m)$, the sparse CFG specific to the query, $SCFG_{v,s,m}$, is obtained from the original CFG of the method $m$, $CFG_m$, by removing those statements that are irrelevant to the aliasing of $v$ at the statement $s$. Below, we explain how the two sparsification strategies identify irrelevant statements.

### 5.4.1 Type-Aware Sparsification

Type-aware sparsification follows a heuristic that is inspired by Declared-Type Analysis [Sun+00], which is based on the following idea. Given a program written in a strongly typed language, a variable can only point to an object compatible with its

declared type. In object-oriented languages such as Java, this definition includes any types that are subtypes or supertypes of the declared type of the variable. Supertypes need to be included to incorporate the possibility of explicit casts. Accordingly, an assignment, leading to aliasing, can only happen between two variables whose types are in a subtype-supertype relationship. With type-aware sparsification, we utilize this information specific to the pointer analysis domain. Therefore, given a query variable, we obtain an SCFG by keeping only the relevant statements according to definition of type-compatibility.

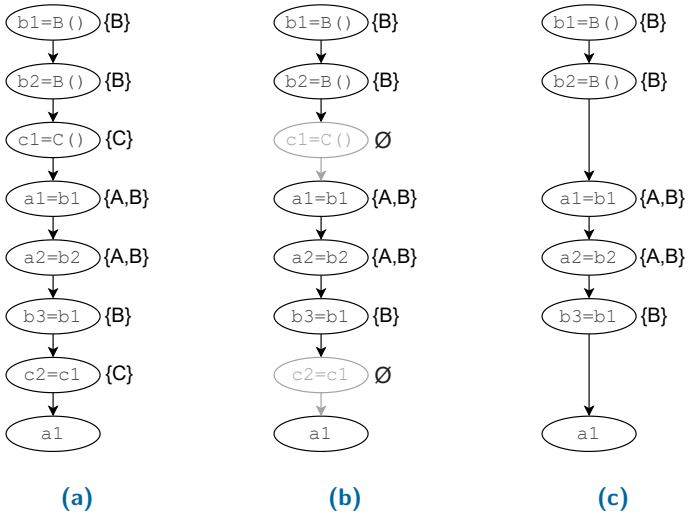
Type-aware sparsification retains the relevant statements as follows. Given a query variable $v$, $hierarchy\_types(v)$ is the set of types in the type hierarchy of $v$'s declared type, i.e., its sub- and supertypes. $var\_types(s)$ is the set of types that the statement $s$ references, e.g., the type of the left-hand side and right-hand side for an assignment, or the argument types and base type for a method call. Then, $s$ is a *relevant statement with respect to* $v$ if and only if:

$$hierarchy\_types(v) \cap var\_types(s) \neq \emptyset$$

Figure 5.3 illustrates how type-aware sparsification works on the example code provided in Figure 5.2a, when an alias query as *mayAliases*$(a1, L8, bar())$ is raised. First, as shown in Figure 5.3a, we find the types of all the variables contained in each statement, e.g., by applying the $var\_types$ function. Then we find the statements that contain a variable whose type is also contained in the type hierarchy of the initial query variable $a1$. Since in this example type `B` is a subtype of `A`, $hierarchy\_types(a1) = \{A, B\}$. Therefore, we mark such statements as relevant statements, and finally remove the other *irrelevant* statements from the initial CFG and obtain a SCFG specific to the given query. Figure 5.2c shows how SPARSE-BOOMERANG solves an alias query over the resulting $SCFG_{a1,L8,bar}$. Note that it still contains irrelevant edges, due to the coarse-grained type-aware approach.

### 5.4.2 Alias-Aware Sparsification

Alias-aware sparsification likewise follows the approach introduced with the Variable-Type Analysis (VTA) algorithm [Sun+00]; it represents variables by themselves, here denoted by variable names. VTA uses def-use chains. Definitions and uses of a variable cause aliasing. Intuitively, one can obtain an SCFG that only consists of the statements that belong to the def-use chain of the query variable. However, it is also necessary to be aware of the def-use chains of all the aliases created in the




(a) Each statement $s_i$ is associated with its $var\_types(s_i)$
(b) After applying $hierarchy\_types(a1) \cap var\_types(s_i)$
(c) The final $SCFG_{a1,L8,bar}$ after removing irrelevant statements

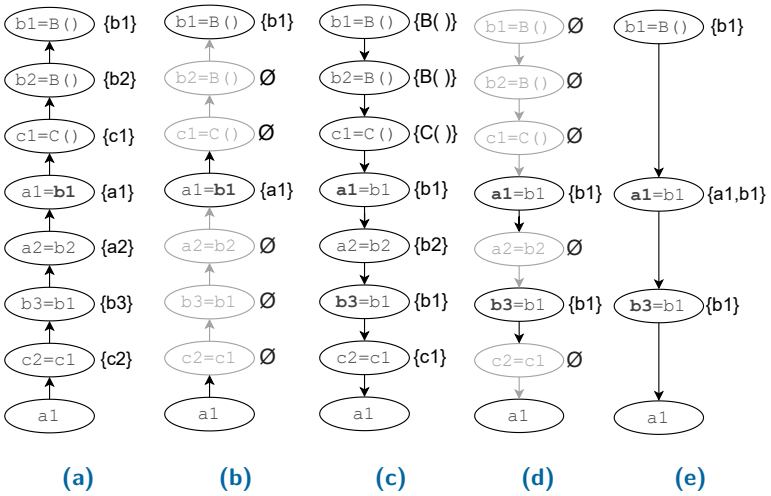
**Figure 5.3:** Running example of applying type-aware sparsification on the code in Figure 5.2a

initial def-use chain, until a fixed point is reached where there are no new aliases to be discovered. To ensure this, alias-aware sparsification works in two passes, similarly to BOOMERANG [Spä+16] but intra-procedurally. First, a backward pass is performed until an allocation site is found, then a forward pass follows until the query statement is reached. There may be multiple such passes, whose details are explained in Section 5.5.2.

Alias-aware sparsification retains the relevant statements as follows. Given a query variable $v$, $intra\_aliases(v)$ is the set of intra-procedural aliases of $v$. $uses(s)$ is the set of variables used in the statement $s$, then $s$ is a *relevant statement with respect to* $v$ if and only if:

$$intra\_aliases(v) \cap uses(s) \neq \emptyset$$

We maintain $intra\_aliases(v)$, which initially only contains the *v* itself. The meaning of *use* depends on the direction of the pass. For instance, in a backward pass, the left-hand side (LHS) of an assignment is in the $uses(s)$, and in a forward pass, the right-hand side (RHS). Accordingly, in a backward pass, the RHS of an assignment is added to the set $intra\_aliases(v)$, and in a forward pass, the LHS.




(a) Backward pass, each $s_i$ is labeled with $uses(s_i)$, i.e., LHS
(b) After applying $intra\_aliases(a1) \cap uses(s_i)$
(c) Forward pass, each $s_i$ is represented by $uses(s_i)$, i.e., RHS
(d) After applying $intra\_aliases(a1) \cap uses(s_i)$
(e) The final $SCFG_{a1,L8,bar}$, union of the relevant statements in all passes

**Figure 5.4:** Running example of applying alias-aware sparsification on the code in Figure 5.2a

Figure 5.4 illustrates how alias-aware sparsification works on the example code provided in Figure 5.2a, when the alias query *mayAliases*$(a1, L8, bar())$ is raised. First, in Figure 5.4a, in a backward pass, we label each statement with its *uses*, i.e., LHS of the assignments. Then, in Figure 5.4b, we find all the statements that contain intraprocedural aliases of the initial query variable $a1$. Since initially $intra\_aliases(a1) = \{a1\}$, we find the statement `a1=b1` as relevant. This statement causes $b1$ to alias with $a1$ through an assignment; therefore, we check the relevance of the next statements (in the backward direction) to $a1$ with $intra\_aliases(a1) = \{a1, b1\}$. In Figure 5.4c, in a forward pass, we again label each statement with its *uses*, i.e., RHS of the assignments. Then in Figure 5.4d, we again find the statements that contain intraprocedural aliases of $a1$. Since we know that $intra\_aliases(a1) = \{a1, b1\}$, we also additionally identify the statement `b3=b1` as relevant, which was initially overlooked, when $b1$ was not identified as an alias of $a1$. At the end of this pass, intraprocedural aliases of $a1$ contain $\{a1, b1, b3\}$. Finally, in Figure 5.4e, we union the relevant statements found in each pass, remove the *irrelevant* statements from the initial CFG and obtain a SCFG specific to the given query. Figure 5.2d shows how SPARSEBOOMERANG solves an alias query over $SCFG_{a1,L8,bar}$ that is obtained via the alias-aware sparsification.

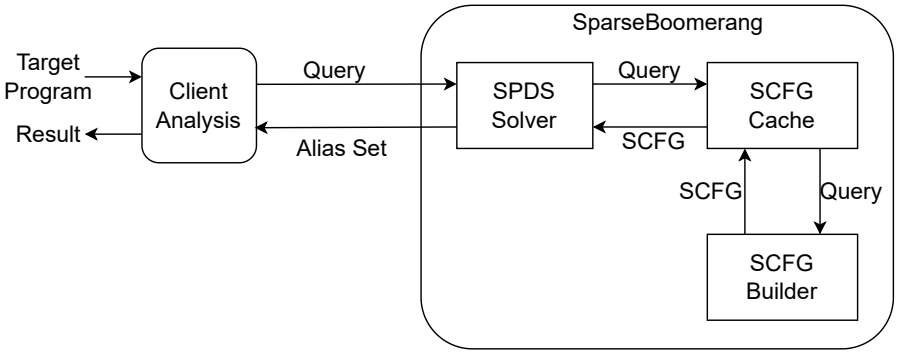



**Figure 5.5:** System Overview of SPARSEBOOMERANG

## 5.5 SparseBoomerang

In this section, we explain the implementation details of our approach. Figure 5.5 shows the system overview of SPARSEBOOMERANG. It applies a caching mechanism similar to that of sparse IFDS [He+19], with a nuance that SCFGs are cached per query instead of per data-flow fact. Queries can be both originating from the client or internal queries that the SPDS solver issues, e.g., on switching contexts. Depending on the configured sparsification strategy (type-aware or alias-aware), the corresponding SCFG builder and the cache are instantiated.

In Section 5.4, we explained how the proposed sparsification strategies find *relevant statements*. We next explain how the statements are handled at the intermediate representation (IR) level and introduce the algorithms for each strategy.

### 5.5.1 Implementation of Type-Aware Sparsification

Table 5.1 shows the statements handled by type-aware sparsification with their IR. *assign* and *cast* statements are handled the same, but we make the distinction to point out that assignments from both supertypes and subtypes exist. *load* and *store* statements concern reading from, and writing to the heap using field references. This makes it necessary to track the aliases of their base variables. To do so, we maintain a worklist of types, *typeWorklist*, where we store the declared types of the base variables of field references and process them in the subsequent iterations. *var_types* correspond to the declared types of the variables involved in a statement. Note that *invoke* statements may have multiple arguments (e.g., $b.m(a_1.f, a_2.f, ...)$), so each of them must be included.

**Table 5.1:** Statements Handled by Type-aware Sparsification.
*To handle aliasing that may be caused because of field load, store, and method invoke statements, types of field bases and invocation receivers are also considered.*

| Statement | IR | var_types | Effect on typeWorklist |
|---|---|---|---|
| assign | $x \leftarrow y$ | $\{t(x), t(y)\}$ | — |
| cast | $x \leftarrow (T)y$ | $\{t(x), t(y)\}$ | — |
| load | $x \leftarrow y.f$ | $\{t(x), t(y.f)\}$ | $add(t(y))$ |
| store | $x.f \leftarrow y$ | $\{t(x.f), t(y)\}$ | $add(t(x))$ |
| invoke | $r \leftarrow b.m(a_i.f)$ | $\{t(r), t(a_i.f)\}$ | $add(t(b), t(a_i))$ |

Figure 5.6 shows the algorithm of type-aware sparsification. It takes as input the variables passed as part of the alias query, *mayAliases*$(v, s, m)$. *relevantStmts* is the set of statements that are relevant to the alias query. *typeWorklist* is initiated with the type of the query variable, *type(v)*. The algorithm works until the *typeWorklist* is empty, e.g., there are no further relevant types to process.

```
1  Function TypeAwareSparsification(v, s, m):
2      relevantStmts← {}
3      typeWorklist← {type(v)}
4      while typeWorklist ≠ {} do
5          Get t from typeWorklist
6          FindRelevantStmts(t, s, m)
7      Sparsify(m, relevantStmts)
8  Function FindRelevantStmts(t, s, m):
9      foreach s_i in pred(s) in m do
10         if var_types(s_i) ∩ hierarchy_types(t) ≠ ∅ then
11             Add s_i to relevantStmts
12             HandleBase(s_i)
13 Function HandleBase(s_i):
14     if s_i contains field then
15         Add type(base(field)) to typeWorklist
```

**Figure 5.6:** The Algorithm of Type-Aware Sparsification

*FindRelevantStmts* iterates over the CFG of the query method *m* until it reaches the query statement *s*. The method *HandleBase* identifies the statements that contain a field reference and populates the *typeWorklist* with the type of their base variables, i.e., *type(base(field))*. To *sparsify* the CFG, it is traversed at the end to retain the *relevantStmts*.

Figure 5.7 shows, on a simple example code, how the type-aware sparsification algorithm works. Given a query variable `q` with the type `A` at statement `q=y.f` the *typeWorkList* is instantiated with `A`. All the statements that contain a type that

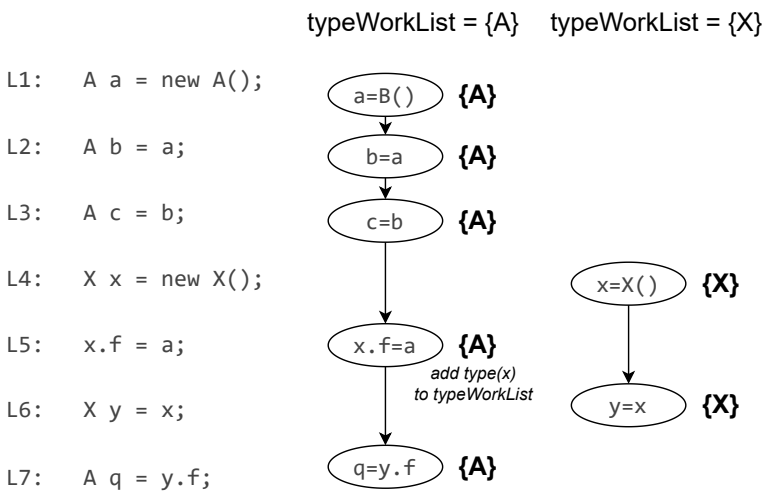



**Figure 5.7:** Visualization of Type-Aware Sparsification Algorithm on an Example

belongs to the same type hierarchy as `A` are marked as relevant statements. As shown in Table 5.1, certain statements require new types to be handled. For instance, at line `L5` due to the field store at `x.f=a`, *type(x)*, `X` is added to the *typeWorklist*. In the next iteration, all the statements that are type compatible with `X` are then also marked as relevant statements, e.g., at `L4`, `x = new X()`, and at `L6`, `y=x`. Ignoring these statements would prevent BOOMERANG's solver from discovering the fact that `x` and `y` alias at line `L6`, and therefore also `x.f`, `y.f`, and `q` alias. The statements that are marked as relevant are then retained during sparsification.

### 5.5.2 Implementation of Alias-aware Sparsification

Table 5.2 shows the statements handled by alias-aware sparsification. In this case, we additionally handle *allocation* and *identity* statements, which were treated as simple assignments by the type-aware variant. These statements originate from the underlying JIMPLE intermediate representation (IR). In JIMPLE, *identity* statements model assignments from method parameters to local variables. *allocation* indicates that an object is instantiated within the current CFG. *identity* and *load* indicate that an object is instantiated elsewhere. *store* signals that the base variable of the field reference must be handled. *invoke* is a special case. In a backward pass, it must be handled similarly to an allocation; in a forward pass, it must be handled similarly to a field store.

**Table 5.2:** Statements Handled by Alias-aware Sparsification.
*To handle aliasing, certain statements (as explained in [Spä+16]) cause switching analysis direction. Impact of such statements, depending on the analysis direction, are shown in **bold**. Other statements are simply iterated in the analysis direction.*

| Statement | IR | During Backward Pass | During Forward Pass |
|---|---|---|---|
| assign | $x \leftarrow y$ | $add(y)$ to *bwWorklist* | $add(x)$ to *fwWorklist* |
| cast | $x \leftarrow (T)y$ | $add(y)$ to *bwWorklist* | $add(x)$ to *fwWorklist* |
| allocation | $x \leftarrow T()$ | $add(x)$ to ***fwWorklist*** | $add(x)$ to *fwWorklist* |
| identity | $x \leftarrow arg$ | $add(x)$ to ***fwWorklist*** | $add(x)$ to *fwWorklist* |
| load | $x \leftarrow y.f$ | $add(y.f)$ to *bwWorklist*<br>$add(x)$ to ***fwWorklist*** | $add(x)$ to *fwWorklist* |
| store | $x.f \leftarrow y$ | $add(y)$ to *bwWorklist* | $add(x.f)$ to *fwWorklist*<br>$add(x)$ to ***bwWorklist*** |
| invoke | $r \leftarrow b.m(a_i.f)$ | $add(r)$ to ***fwWorklist*** | $add(b, a_i)$ to ***bwWorklist*** |

To discover the aliasing relationships caused by different kinds of statements, this strategy maintains two worklists. A backward worklist is used to create the def-use chains in the backward pass, and a forward worklist is used to create them in the forward pass. During a backward pass, the current value is searched in the LHS of the statements. When a matching statement is found, its RHS is added to the backward worklist *bwWorklist*. Similarly, during a forward pass, the current value is searched in the RHS of the statements, and the LHS of a matching statement is added to the forward worklist *fwWorklist*.

Certain statements require special handling to account for statements that do not belong to the discovered def-use chain in the current analysis direction but may cause aliasing relationships. During a backward pass, all the statements that may instantiate an object outside the current method's context are handled similarly to an allocation statement, and their LHSs are added to the *fwWorklist*. These include *identity*, *load*, and *invoke* statements. During a forward pass, base variables of *store* statements and receivers of *invoke* statements are added to the *bwWorklist*.

Figure 5.8, shows the algorithm of alias-aware sparsification. It may perform multiple backward and forward passes depending on the number of statements that cause switching direction. To be brief, we assume *intra_aliases(v)* is maintained implicitly. Both algorithms soundly preserve branching statements and stop processing after reaching the query statement.

```
1  Function AliasAwareSparsification(v, s, m):
2      relevantStmts ← {}
3      bwWorklist ← {v}
4      fwWorklist ← {}
5      while bwWorklist ≠ {} do
6          Get v from bwWorklist
7          s = FindDefBW(v, s, m)
8          while fwWorklist ≠ {} do
9              Get v from fwWorklist
10             s = FindUseFW(v, s, m)
11     Sparsify(m, relevantStmts)
12 Function FindDefBW(v, s, m):
13     foreach s_i in pred(s) in m do
14         if lhs(s_i) ∩ intra_aliases(v) ≠ ∅ then
15             Add s_i to relevantStmts
16             Add rhs(s_i) to bwWorklist
17             if rhs(s_i) is alloc, identity, load, or invoke then
18                 Add lhs(s_i) to fwWorklist
19             return s_i
20     return s
21 Function FindUseFW(v, s, m):
22     foreach s_i in succ(s) in m do
23         if rhs(s_i) ∩ intra_aliases(v) ≠ ∅ then
24             Add s_i to relevantStmts
25             Add lhs(s_i) to fwWorklist
26             if lhs(s_i) contains field then
27                 Add base(field) to bwWorklist
28             if s_i contains invoke then
29                 Add receiver(invoke) to bwWorklist
30             return s_i
31     return s
```

**Figure 5.8:** The Algorithm of Alias-Aware Sparsification

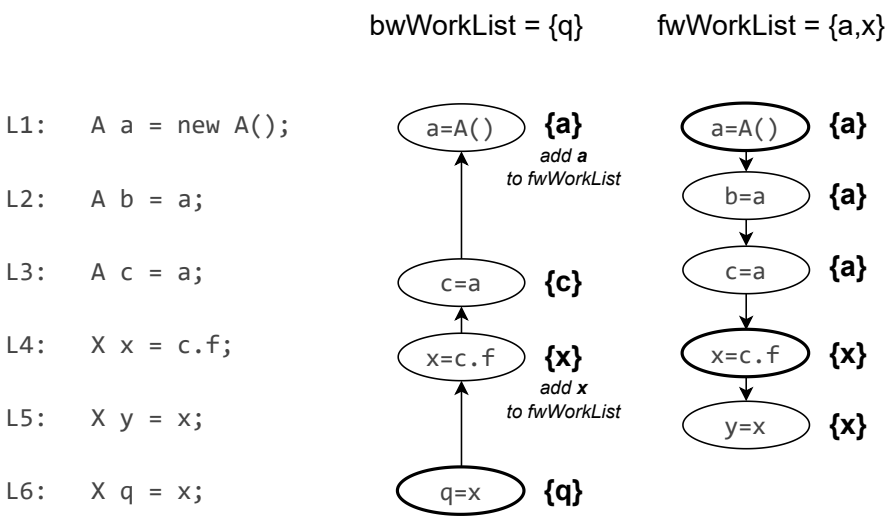



**Figure 5.9:** Visualization of Alias-Aware Sparsification Algorithm on an Example

Figure 5.9 shows, on a simple example code, how alias-aware sparsification is performed. Given a query variable `q` at statement `q=x`, a backward pass is initiated. The backward pass continues until an object instantiation is reached. Iterations continue for the values discovered along the way, e.g., for `x` at `q=x`, `c.f` at `x=c.f`, and `a` at `c=a` until `a=new A()` is found. All the statements that belong to this consecutive def-use chain are marked as relevant statements, i.e., the ones that should not be sparsified away. As shown in Table 5.2, certain statements require changing the analysis direction. For instance, in this example, at `L4`, `x` and at `L1`, `a` are added to the *fwWorklist*. This is required to discover the relevant statements in the forward pass. Otherwise these statements would be *wrongly ignored*, e.g., at `L2`, `b=a` and at `L5`, `y=x`.

## 5.6 Evaluation

Sparsification aims to enhance the performance of analyses while maintaining their precision. Therefore, we evaluate the impact of the proposed sparsification strategies considering two dimensions: Precision and performance impact. To do so, we have formulated the following research questions:

- RQ1: Do the sparsification strategies cause precision loss?
- RQ2: How do the sparsification strategies impact the performance of the demand-driven pointer analysis and its client?
- RQ3: How does the degree of sparsification impact the SCFG construction time and its evaluation time?

### 5.6.1 Experimental Setup

SPARSEBOOMERANG, available at `https://github.com/secure-software-engineering/SparseBoomerang`, extends the latest version of BOOMERANG at the time of writing (`1179227`) [Cod19]. We use FLOWDROID as a taint analysis client with its default source and sink definitions. We also extended the latest version of FLOWDROID (`d97f9d9`) [Eng18] so that it creates on-demand alias queries for BOOMERANG and SPARSEBOOMERANG instead of using its own integrated alias analysis. Both tools are based on the Soot static analysis framework [Val+99]. We use the following benchmarks in our experiments:

- **POINTERBENCH**: POINTERBENCH [Eng19] is a micro-benchmark suite for alias analysis. We use this suite to evaluate the correctness of the sparsification approaches. We check whether we can obtain the same aliases by issuing alias queries to BOOMERANG without sparsification and to SPARSEBOOMERANG with type-aware and alias-aware sparsification strategies.
- **Real-world Apps**: We include real-world Android apps to investigate the performance of our approach under the workload of large-scale and complex programs. For that, we selected the 20 most downloaded Android apps from the Google Play store listed in androidrank.org [Ran], then we downloaded their most recent version from Androzoo [All+16].
- **Replication Package**: We provide a replication package that contains the complete toolchain to reproduce the findings, along with their source codes. The replication package is available at `https://zenodo.org/records/16928052`.

All the experiments were performed on a Quad-Core Intel i7 processor at 2,3 GHz and 32 GB of memory. The JVM was configured with a maximum heap size of 25GB, and a maximum stack size of 1GB. All performance data was generated as the average of five runs of each input app with each alias analysis.

### 5.6.2 RQ1: Do the sparsification strategies cause precision loss?

It is crucial for the sparsification approaches to maintain the precision of their non-sparse counterparts. Sparsification aims to reduce the number of statements that are irrelevant to the particular analysis, but this requires a careful study of the program statements to find out how to handle each one of these. We, therefore, test whether both approaches maintain the level of precision that is obtained by non-sparse

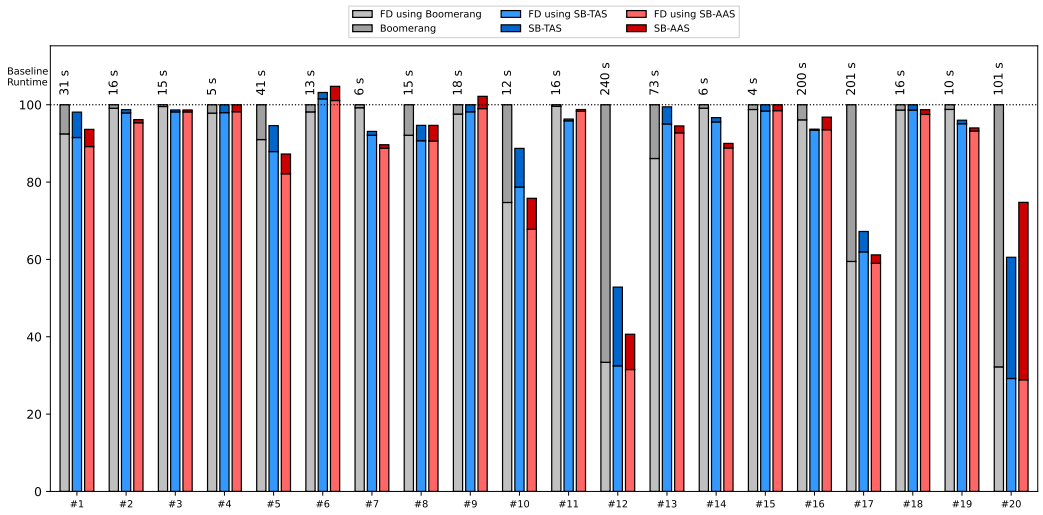



**Figure 5.10:** Relative time spent on taint analysis and solving alias queries by FLOWDROID (FD) using baseline BOOMERANG, SPARSEBOOMERANG (SB) with TAS and with AAS, in %

BOOMERANG, on POINTERBENCH. The micro-benchmark suite contains 35 target programs. Its basic tests include branching, loops, recursion, and inter-procedural aliasing. It also includes corner cases where field-, flow-, and context-sensitivities are tested. The results show that both sparsification approaches yield results identical to those of the non-sparse analysis. Precision, in particular, is therefore maintained.

### 5.6.3 RQ2: How do the sparsification strategies impact the performance of the demand-driven pointer analysis and its client?

As discussed in Section 5.5, both sparsification strategies come at a cost. They need to build on-demand, sparse versions of the original control flow graphs (CFGs) of the input programs. To measure the impact of the sparsification strategies on solving alias queries and on the overall runtime of the taint analysis client, we evaluate the performance of SPARSEBOOMERANG by comparing it to BOOMERANG.

Figure 5.10 shows the relative time spent by FLOWDROID on taint analysis and by BOOMERANG and SPARSEBOOMERANG on solving alias queries on each real-world app from the Google Play store that we have included in our evaluation set. FLOWDROID using BOOMERANG is used as the baseline. Each run by SPARSEBOOMERANG with TAS, and with AAS is normalized against this baseline. It can be observed that, despite their cost in SCFG construction, both sparsification strategies frequently

reduce the time spent by demand-driven pointer analysis on solving alias queries. More specifically, SPARSEBOOMERANG, compared to BOOMERANG, solves the alias queries on average 2.4 times faster with type-aware sparsification, and 2.8 times faster with the alias-aware variant. The maximum speedups achieved by each strategy are 14.6 times and 18.7 times, respectively. The speedups gained during the pointer analysis are also reflected in the client analysis. FLOWDROID using SPARSEBOOMERANG performs the taint analysis on average 1.13 times faster with type-aware sparsification and 1.17 times faster with alias-aware sparsification. The maximum speedups by each strategy are 1.9 times and 2.5 times, respectively. The full set of absolute numbers is contained in Table 5.3.

To investigate the significance of the results, we have also performed the Wilcoxon signed-rank test [Wil92] at a 0.05 significance level. Both TAS ($p=0.0027$) and AAS ($p=0.0094$) improve the performance of the pointer analysis significantly. Similarly, the client's performance also increases significantly when using TAS ($p=0.0012$) and AAS ($p=0.0011$).

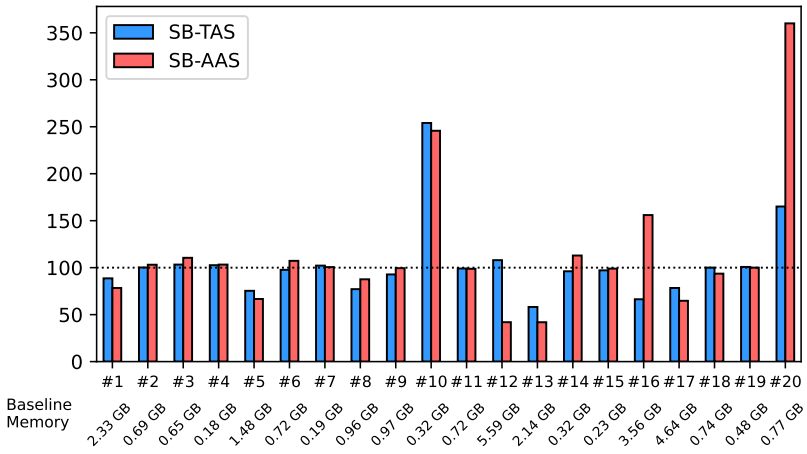



**Figure 5.11:** Relative Memory Consumption of FLOWDROID using SPARSEBOOMERANG with TAS and with AAS compared to the baseline BOOMERANG, in %

Figure 5.11 shows the maximum memory consumption of FLOWDROID using SPARSEBOOMERANG with TAS and AAS compared to the baseline memory consumption of FLOWDROID using BOOMERANG. On average, the maximum memory consumption increases. We have measured an average of 3% increase in memory consumption when using SPARSEBOOMERANG with TAS, and 13% when it's using AAS. However, according to the Wilcoxon signed-rank test, memory increases with TAS ($p=0.24$), and AAS ($p=0.70$) are insignificant. The impact on memory consumption is largest

**Table 5.3:** Performance of FLOWDROID using the baseline BOOMERANG (B) and SPARSEBOOMERANG with TAS and AAS

| # | APK | Runtime (s) | | | | | Memory (GB) | | | Total Query Time (ms) | | | | | Query Solv. (ms) | | | SCFG Const. (ms) | | DoS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | B | TAS | B/TAS | AAS | B/AAS | B | TAS | AAS | B | TAS | B/TAS | AAS | B/AAS | B | TAS | AAS | TAS | AAS | TAS | AAS |
| 1 | candycrush | 31 | 30 | 1.02 | 29 | 1.07 | 2.33 | 2.06 | 1.82 | 2368 | 2062 | 1.15 | 1396 | 1.70 | 2368 | 2029 | 1357 | 32 | 39 | 0.30 | 0.33 |
| 2 | chrome | 15 | 15 | 1.01 | 15 | 1.04 | 0.69 | 0.69 | 0.71 | 141 | 130 | 1.08 | 126 | 1.12 | 141 | 113 | 111 | 17 | 15 | 0.40 | 0.51 |
| 3 | excel | 14 | 14 | 1.01 | 14 | 1.01 | 0.65 | 0.68 | 0.72 | 67 | 74 | 0.91 | 73 | 0.92 | 67 | 64 | 64 | 9 | 9 | 0.75 | 0.77 |
| 4 | fblite | 5 | 5 | 1.00 | 5 | 1.00 | 0.18 | 0.18 | 0.18 | 108 | 102 | 1.06 | 90 | 1.20 | 108 | 81 | 68 | 21 | 22 | 0.47 | 0.49 |
| 5 | garena | 40 | 38 | 1.06 | 35 | 1.15 | 1.48 | 1.12 | 0.99 | 3674 | 2740 | 1.34 | 2081 | 1.77 | 3674 | 2624 | 1885 | 116 | 196 | 0.37 | 0.42 |
| 6 | gclock | 12 | 13 | 0.97 | 13 | 0.95 | 0.72 | 0.70 | 0.77 | 233 | 209 | 1.12 | 463 | 0.50 | 233 | 172 | 439 | 37 | 23 | 0.43 | 0.53 |
| 7 | gfiles | 5 | 5 | 1.07 | 5 | 1.12 | 0.19 | 0.19 | 0.19 | 45 | 57 | 0.78 | 52 | 0.86 | 45 | 40 | 39 | 17 | 12 | 0.33 | 0.55 |
| 8 | gkeyboard | 15 | 14 | 1.06 | 14 | 1.06 | 0.96 | 0.74 | 0.84 | 1183 | 595 | 1.99 | 609 | 1.94 | 1183 | 414 | 446 | 181 | 162 | 0.33 | 0.52 |
| 9 | gsearchlite | 18 | 18 | 1.00 | 18 | 0.98 | 0.97 | 0.90 | 0.97 | 444 | 335 | 1.32 | 584 | 0.76 | 444 | 287 | 550 | 48 | 33 | 0.44 | 0.51 |
| 10 | mivideo | 12 | 11 | 1.13 | 9 | 1.32 | 0.32 | 0.80 | 0.78 | 3134 | 1240 | 2.53 | 992 | 3.16 | 3134 | 1085 | 884 | 155 | 108 | 0.52 | 0.53 |
| 11 | msword | 16 | 15 | 1.04 | 16 | 1.01 | 0.72 | 0.72 | 0.72 | 66 | 72 | 0.91 | 75 | 0.88 | 66 | 62 | 65 | 10 | 9 | 0.75 | 0.77 |
| 12 | mxplayer | 239 | 126 | 1.89 | 97 | 2.46 | 5.59 | 6.04 | 2.35 | 159583 | 48761 | 3.27 | 21874 | 7.30 | 159583 | 45194 | 15304 | 3567 | 6569 | 0.40 | 0.47 |
| 13 | netflix | 73 | 72 | 1.01 | 69 | 1.06 | 2.14 | 1.25 | 0.90 | 10150 | 3228 | 3.14 | 1306 | 7.77 | 10150 | 3125 | 1216 | 102 | 90 | 0.45 | 0.73 |
| 14 | shareit | 6 | 5 | 1.03 | 5 | 1.11 | 0.32 | 0.30 | 0.36 | 53 | 66 | 0.80 | 70 | 0.76 | 53 | 58 | 61 | 8 | 9 | 0.32 | 0.54 |
| 15 | shareme | 4 | 4 | 1.00 | 4 | 1.00 | 0.23 | 0.23 | 0.23 | 49 | 64 | 0.76 | 61 | 0.80 | 49 | 52 | 49 | 12 | 11 | 0.46 | 0.56 |
| 16 | tiktok | 199 | 187 | 1.07 | 193 | 1.03 | 3.56 | 2.36 | 5.56 | 7801 | 534 | 14.59 | 6501 | 1.20 | 7801 | 499 | 4760 | 35 | 1740 | 0.37 | 0.31 |
| 17 | ucmobile | 201 | 135 | 1.49 | 123 | 1.63 | 4.64 | 3.63 | 3.00 | 81620 | 10632 | 7.68 | 4374 | 18.66 | 81620 | 10137 | 3498 | 495 | 875 | 0.47 | 0.49 |
| 18 | viber | 15 | 15 | 1.00 | 15 | 1.01 | 0.74 | 0.74 | 0.70 | 215 | 215 | 1.00 | 190 | 1.13 | 215 | 199 | 175 | 16 | 15 | 0.39 | 0.44 |
| 19 | webview | 10 | 9 | 1.04 | 9 | 1.06 | 0.48 | 0.48 | 0.48 | 121 | 91 | 1.33 | 80 | 1.50 | 121 | 79 | 68 | 12 | 12 | 0.44 | 0.55 |
| 20 | whatsapp | 101 | 61 | 1.65 | 75 | 1.34 | 0.77 | 1.28 | 2.78 | 68743 | 31784 | 2.16 | 46522 | 1.48 | 68743 | 31658 | 46030 | 126 | 491 | 0.42 | 0.36 |

for apps #10 and #20. When observing the same apps in Figure 5.10, it can be observed that these apps also benefited from a large speedup in the analysis runtime.

An increase in memory consumption was expected. This is because, after all, both sparsification strategies make use of caching to reduce the amount of time spent on sparsification in case the same queries are issued. However, surprisingly, we see that for some subject apps, e.g., #5, #13, and #17, sparsification substantially *decreases* memory consumption. We attribute this to savings in the client analysis, which, given the sparsification, requires associating data-flow facts with fewer CFG nodes.

### 5.6.4 RQ3: How does the degree of sparsification impact the SCFG construction time and its evaluation time?

We have already informally used the term *degree of sparsification (DoS)*. We define it formally as follows. Given an input program $p$, $M$ is the set of all the methods of $p$ in which an alias query is issued. Let $m$ a method in $M$, where $CFG_m$ is its original non-sparse CFG, and $SCFG_m$ is its sparse SCFG. $|CFG_m|$ is the number of statements in $CFG_m$ and $|SCFG_m|$ is the number of statements in $SCFG_m$. $DoS_p$ is then calculated as:

$$DoS_p = \frac{\sum_{m \in M} |CFG_m| - |SCFG_m|}{\sum_{m \in M} |CFG_m|}$$

In Section 5.4, we made the assumption that a higher DoS would lead to a larger decrease in runtime when solving alias queries. To investigate this, in Figure 5.12, we show, for each run, the correlation between DoS and the average time taken to solve alias queries in these runs. The trend shows the assumed inverse relation on a small scale. When the DoS increases, i.e., when more irrelevant statements are removed, it takes less time to solve the alias queries. Accordingly, when the DoS decreases, i.e., when it is necessary to retain a large fraction of relevant statements, on average, it takes more time to solve the alias queries.

To further highlight the impact of the DoS, in Figure 5.13 we show the relative time spent by each sparsification strategy on constructing the SCFGs, and then solving the alias queries over them. We observe that the assumption generally holds: in most cases, a higher degree of sparsification shortens the alias-query evaluation time. However, the results on apps #6, #7, #8, #9 contradict the rule: counter-intuitively,

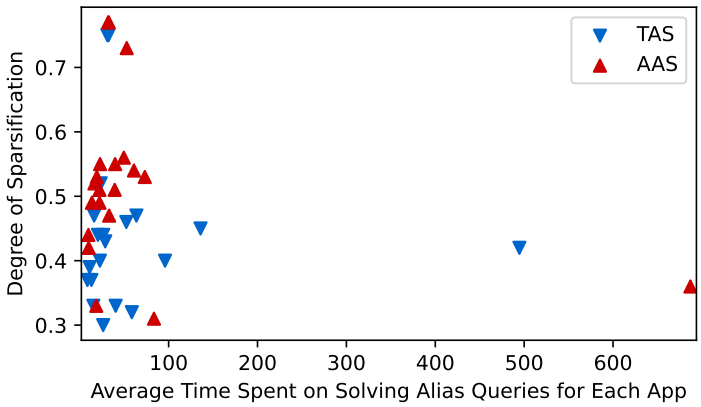



**Figure 5.12:** Degree of sparsification (DoS) and average time spent on solving alias queries

for those apps, we see that the construction of the type-aware SCFGs actually takes longer than the construction of the alias-aware SCFGs, although the latter actually operates on a more detailed data structure, the def-use chains.

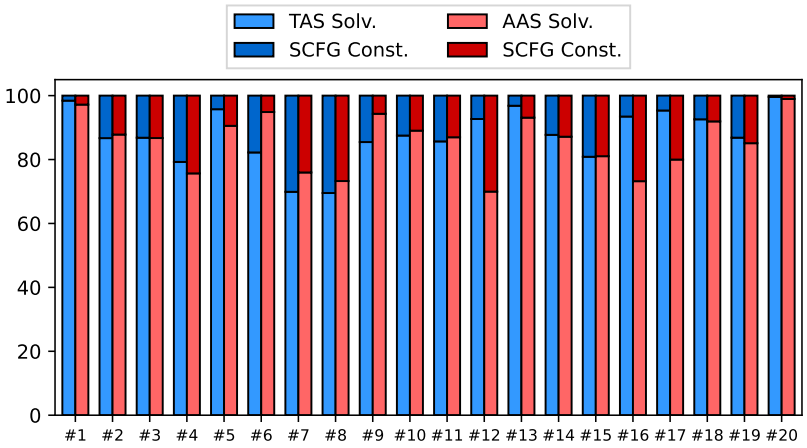



**Figure 5.13:** Relative Time Spent on Solving the Alias Queries and Constructing the SCFGs by each Strategy, in %

We have preferred calculating DoS based on the number of statements, to have a common metric that can be used by both the pointer analysis and the sparsification approaches. While the complexity of pointer analysis depends on the number of edges in the CFG, the complexities of the sparsification strategies depend on the diversity of the base types (for TAS) and the number of POIs (for AAS).

### 5.6.5 Discussion and Threats to Validity

The sparsification techniques we present in this thesis apply at two different abstraction levels. First, at the framework level, we extend an existing *non-sparse* framework with a new interface, so that it can obtain a `SparseCFGBuilder` specific to the concrete data-flow analysis problem (domain). Second, at the client analysis level, we implement concrete data-flow analyses and provide their corresponding `SparseCFGBuilder`, like the constant propagation analysis client in Chapter 3. BOOMERANG poses a special case, where the analysis framework also contains an integrated pointer analysis client. Therefore, in this case, we could focus on multiple sparsification strategies (TAS and AAS) specific to the domain of pointer analysis.

Theoretically, AAS corresponds to the most finely grained, variable-level fact-specific sparsification. TAS, on the other hand, is a more coarse-grained, type-level sparsification. Whether one can apply a similar strategy to TAS for SPARSEIDE depends on the concrete analysis problem at hand. For instance, for integer constant propagation analysis, one can retain all the statements that contain an integer type. Similarly, for a type state analysis, one can retain all the statements containing the types whose state properties are under analysis.

For the evaluation of SPARSEBOOMERANG, we have used the most installed apps on androidrank.org [Ran]. Among them, we have discarded the ones that did not contain any sources or sinks that FLOWDROID could detect with its default configuration. Further, we have ignored the apps that caused an error for the underlying static analysis framework, Soot. This might introduce some selection bias, however, appears hard to avoid.

To even out noise in runtime and memory measurements, we measured five runs and here report the average over these five runs. Because both the pointer and the client analysis are running in the same process, it is very hard to attribute increases or decreases in memory consumption to either of them, let alone individual data structures and algorithms. We thus focus on reporting the overall consumption.

## 5.7 Related Work

Sparsification has been applied in diverse static-analysis settings. PINPOINT [Shi+18] is a staged sparsification approach that uses intraprocedural data dependence to solve only the necessary interprocedural data dependence queries selectively. SVF [SX16] uses as input points-to information that is generated by a cheap, imprecise

analysis, constructs value-flows which are then used for a precise, sparse analysis. Sparsification approaches are typically specific to particular analyses; however, Oh et al. [Oh+12] introduced a general sparsification framework that is theoretically applicable to any analysis. Our work can be seen as an instantiation of their framework.

Applications of sparsification have also been performed for pointer analysis in particular. SFS [HL11] is a flow-sensitive pointer-analysis approach that uses sparse def-use chains created by a flow-insensitive analysis stage. With alias-aware sparsification, we also create def-use chains, albeit in a demand-driven manner, utilizing the query information available at analysis time. Hardekopf and Lin [HL09] introduced a semi-sparse approach, where sparsification is only applied to top-level variables. This approach could be employed as a further, more coarse-grained sparsification strategy. SPAS [Sui+11] is a path-sensitive sparsification approach that is applied in stages to pointer analysis. Handling path-sensitivity is beyond the scope of our study.

Many of the existing approaches sparsify program parts in a pre-analysis stage, where only limited information about the target program is available. The Sparse IFDS algorithm by He et al. [He+19] showed that further sparsification is possible when applying sparsification on-demand and using the information available at the runtime of the analysis. Their approach is also demonstrated by extending FLOWDROID. Yet, a direct comparison with their approach was not possible because they sparsify the FLOWDROID itself, whereas we sparsify BOOMERANG. So the impact of our approach is only indirectly reflected in FLOWDROID. FLOWDROID is multi-threaded, but BOOMERANG currently does not support multi-threading, so neither does SPARSEBOOMERANG.

## 5.8 Conclusion

In this chapter, we presented the third major contribution of this thesis. We proposed two sparsification strategies to accelerate demand-driven pointer analysis, implemented in SPARSEBOOMERANG (*Contribution 3*). Both strategies create query-specific sparse CFGs by utilizing the information available at the analysis runtime. This highligts how sparsification can benefit from domain-specific information, in this case, in the domain of pointer analysis. Moreover, with this work we also show how fact-specific sparsification is not limited to IFDS- and IDE-based data-flow anal-

yses. We demonstrate it in a novel setting, on top of a pushdown systems-based solver. Although sparse CFG construction requires time, we have demonstrated that it is negligible given the achieved speedups.

# 6 Conclusion and Outlook

Scaling precise interprocedural static data-flow analyses on large-scale real-world code bases is challenging. To improve scalability, static analysis designers often need to restrict various analysis sensitivities, which means too many imprecise findings for the analysis clients.

The SPARSEIDE framework (*Contribution 1*), presented in Chapter 3, shows how one can significantly scale up the data-flow analysis problems that fit the classical IDE framework [SRH96]. Importantly, SPARSEIDE maintains the precision of the IDE framework. While SPARSEIDE mirrors the design of previous work on sparse IFDS [He+19], unlike sparse IFDS, SPARSEIDE can realize sparsifications for data-flow analysis problems with large or even infinite domains. SPARSEIDE strictly generalizes sparse IFDS; one can define sparse IFDS as an instantiation of SPARSEIDE. SPARSEIDE's sparsification is limited to program symbols; it cannot sparsify with respect to the value domain.

Sparsification techniques, including sparse IFDS [He+19] and SPARSEIDE, create on-demand sparse control flow graphs (SCFGs) for individual procedures, on which the data-flow solver propagates data-flow facts sparsely. They only optimize *intraprocedural* data-flow edges; on the other hand, *interprocedural* data-flow edges, obtained from call graphs, are required for modelling method invocations. In Chapter 4, we perform an extensive empirical evaluation to investigate the extent to which call graphs impact the scalability of data-flow analyses (*Contribution 2*). We show that although precise call-graphs are expensive to generate, they significantly improve the scalability of downstream data-flow analyses, and in some cases, improve their precision. We hope that these findings will motivate researchers to develop further optimizations targeting the call graph and interprocedural edges when scaling data-flow analyses.

Precise data-flow analyses must account for the aliasing problem, which is often delegated to standalone pointer analyses. Previous works [SAB17; LBS19] have improved data-flow analysis precision by integrating the BOOMERANG [Spä+16] demand-driven pointer analysis into their solvers to obtain alias information. The alias analysis of BOOMERANG is flow-, field-, and context-sensitive, which makes it highly precise, yet hard to scale. In Chapter 5, we show how BOOMERANG can also

benefit from sparsification. To this end, we propose two sparsification strategies: a type-aware sparsification (TAS) and an alias-aware sparsification (AAS), implemented in SPARSEBOOMERANG (*Contribution 3*).

This thesis presents novel tools and techniques for improving the scalability of precise interprocedural data-flow analyses, mainly through sparsification. We show that sparse data-flow analyses pay off when analyzing real-world code bases, and can replace their non-sparse counterparts. Further, we show how call-graph precision directly impacts data-flow analysis scalability, and can complement the performance benefits of sparse data-flow analyses. Finally, we show how pointer analysis can be sparsified through fact-specific sparsification specialized for the pointer analysis domain.

The sparsification strategies presented in this work were implemented manually depending on the requirements of each data-flow analysis problem. For instance, the sparse CFG builder for the integer constant propagation analysis problem needs to recognize integer constant assignments and arithmetic operations, whereas the sparse CFG builder for the pointer analysis problem needs to recognize object initializations. Generating sparse CFG builders for each analysis problem would enable problem-independent sparsification; however, we were not able to feasibly solve this problem within the scope of this thesis. Therefore, problem-independent sparsification remains an open challenge.

The fact-specific sparsification approach presented in this work creates a sparse CFG for each data-flow fact in a method. This causes a runtime overhead, which is often negligible. However, as we have observed, uniformly applying a fixed sparsification strategy may lead to suboptimal performance where the speedups gained during analyzing the sparse CFG do not outweigh the sparsification overhead. A so-called adaptive sparsification may utilize method features to decide which sparsification strategy to apply, e.g., in the presence of multiple strategies, or not to sparsify at all. Application of such an adaptive sparsification can be investigated in future work.

Current fact-specific sparsification techniques operate intraprocedurally, i.e., within method boundaries, and they always soundly retain method invocations that use the data-flow fact being actively propagated. This can be optimized further. A simple heuristic can guide the sparsification strategies to check whether the data-flow fact is actually being used in the callee method context. Employing such a heuristic can potentially prevent doing additional work for method contexts that have no impact on the analysis end result. Future work may investigate the performance benefits of employing sparsifications that go beyond method boundaries.

# Artifacts

In this thesis, we have presented several tools and techniques for improving the scalability of data-flow analyses. To enable other researchers to reuse, validate, and extend these contributions we provide all the implementations as open source artifacts.

## SparseIDE: Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems

The implementation of the SparseIDE framework that we have presented in Chapter 3 is published at `https://github.com/secure-software-engineering/SparseIDE`. This repository contains SparseHeros, the sparse version of the Heros solver, the contant propagation analysis client that we have implemented on top of SparseHeros, and the ConstantBench microbenchmark suite. The repository contains instructions for building the client from the source code and also using the SparseIDE framework as a dependency. Moreover, the complete toolchain of the evaluation presented in Section 3.6 along with all the source code is available at `https://zenodo.org/records/10498325`.

## Qcg Framework for Call-Graph Generation

The implementation of the Qcg framework for call-graph generation that we have used for the emprical evaluation presented in Chapter 4 is published at `https://github.com/secure-software-engineering/QCG`. The complete toolchain of the evalution presented in Section 4.5 along with all the source code is available at `https://zenodo.org/records/17041537`.

## SparseBoomerang: Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis

The implementation of the SparseBoomerang for demand-driven sparse pointer analysis presented in Chapter 5 is published at `https://github.com/secure-software-engineering/SparseBoomerang/`. The complete toolchain of the evalution presented in Section 5.6 along with all the source code is available at `https://zenodo.org/records/16928052`.

# Bibliography


[AL12] Karim Ali and Ondřej Lhoták. “Application-Only call graph construction”. In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP’12. Beijing, China: Springer-Verlag, 2012, pp. 688–712. DOI: 10.1007/978-3-642-31057-7_30 (cit. on p. 48).

[Ali+14] Karim Ali, Marianna Rapoport, Ondrej Lhoták, Julian Dolby, and Frank Tip. “Constructing Call Graphs of Scala Programs”. In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 54–79. DOI: 10.1007/978-3-662-44202-9_3 (cit. on p. 16).

[All+16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. “AndroZoo: Collecting Millions of Android Apps for the Research Community”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR ’16. Austin, Texas: ACM, 2016, pp. 468–471. DOI: 10.1145/2901739.2903508 (cit. on p. 88).

[And94] Lars Ole Andersen. “Program analysis and specialization for the C programming language”. In: (1994) (cit. on p. 67).

[Arz21] Steven Arzt. “Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection”. In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1098–1110. DOI: 10.1109/ICSE43902.2021.00102 (cit. on pp. 19, 40, 42, 45, 46, 68).

[AB14] Steven Arzt and Eric Bodden. “Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 288–298. DOI: 10.1145/2568225.2568243 (cit. on pp. 19, 42).

[Arz+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, et al. “FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 259–269. DOI: 10.1145/2594291.2594299 (cit. on pp. 14, 41, 42, 47, 51, 53, 66, 71).

[Avd+15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, et al. “Mining apps for abnormal usage of sensitive data”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 426–436 (cit. on pp. 46, 68).

[Aye+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William W. Pugh. “Using Static Analysis to Find Bugs”. In: *IEEE Softw.* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130 (cit. on p. 1).

[BS96] David F. Bacon and Peter F. Sweeney. “Fast static analysis of C++ virtual function calls”. In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’96. San Jose, California, USA: Association for Computing Machinery, 1996, pp. 324–341. DOI: 10.1145/236337.236371 (cit. on pp. 45, 54, 67).

[Bar78] Jeffrey M. Barth. “A practical interprocedural data flow analysis algorithm”. In: *Commun. ACM* 21.9 (Sept. 1978), pp. 724–736. DOI: 10.1145/359588.359596 (cit. on p. 11).

[Bod12] Eric Bodden. “Inter-procedural data-flow analysis with IFDS/IDE and Soot”. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*. Ed. by Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman. ACM, 2012, pp. 3–8. DOI: 10.1145/2259051.2259052 (cit. on pp. 5, 21, 33, 42).

[Bod18] Eric Bodden. “The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them)”. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA ’18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 85–93. DOI: 10.1145/3236454.3236500 (cit. on p. 46).

[BS09] Martin Bravenboer and Yannis Smaragdakis. “Strictly declarative specification of sophisticated points-to analyses”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 243–262. DOI: 10.1145/1639949.1640108 (cit. on p. 49).

[Cal+86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. “Interprocedural constant propagation”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’86. Palo Alto, California, USA: Association for Computing Machinery, 1986, pp. 152–161. DOI: 10.1145/12276.13327 (cit. on p. 3).

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. “Automatic Construction of Sparse Data Flow Evaluation Graphs”. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’91. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 55–66. DOI: 10.1145/99583.99594 (cit. on pp. 2, 18, 42).

[Cod19] CodeShield. *CodeShield-Security/SPDS: Efficient and Precise Pointer-Tracking Data-Flow Framework*. https://github.com/CodeShield-Security/SPDS. (Accessed on 03/30/2023). 2019 (cit. on pp. 34, 88).

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cit. on p. 8).

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis”. In: *ECOOP’95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*. Ed. by Walter G. Olthoff. Vol. 952. Lecture Notes in Computer Science. Springer, 1995, pp. 77–101. DOI: 10.1007/3-540-49538-X_5 (cit. on pp. 45, 47, 54, 67).

[Deu94] Alain Deutsch. “Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 230–241. DOI: 10.1145/178243.178263 (cit. on p. 21).

[EKS01] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. “Aiding Program Comprehension by Static and Dynamic Feature Analysis”. In: *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*. IEEE Computer Society, 2001, pp. 602–611. DOI: 10.1109/ICSM.2001.972777 (cit. on p. 1).

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. “Context-sensitive interprocedural points-to analysis in the presence of function pointers”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 242–256. DOI: 10.1145/178243.178264 (cit. on pp. 45, 67).

[Emm+21] Michael Emmi, Liana Hadarean, Ranjit Jhala, et al. “RAPID: checking API usage for the cloud in the cloud”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1416–1426. DOI: 10.1145/3468264.3473934 (cit. on p. 41).

[Eng18] Secure Software Engineering. *secure-software-engineering/FlowDroid: FlowDroid Static Data Flow Tracker*. https://github.com/secure-software-engineering/FlowDroid. (Accessed on 09/22/2022). 2018 (cit. on p. 88).

[Eng19] Secure Software Engineering. *A points-to and alias analysis benchmark suite*. https://github.com/secure-software-engineering/PointerBench. (Accessed on 10/04/2022). 2019 (cit. on pp. 73, 88).

[EL02] D. Evans and D. Larochelle. “Improving security using extensible lightweight static analysis”. In: *IEEE Software* 19.1 (2002), pp. 42–51. DOI: 10.1109/52.976940 (cit. on p. 8).

[Fin+08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. "Effective Typestate Verification in the Presence of Aliasing". In: *ACM Trans. Softw. Eng. Methodol.* 17.2 (May 2008). DOI: 10.1145/1348250.1348255 (cit. on pp. 3, 19).

[FHP07] Jeffrey S. Foster, Michael W. Hicks, and William W. Pugh. "Improving software quality with static analysis". In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*. Ed. by Manuvir Das and Dan Grossman. ACM, 2007, pp. 83–84. DOI: 10.1145/1251535.1251549 (cit. on p. 1).

[GC01] David Grove and Craig Chambers. "A framework for call graph construction algorithms". In: *ACM Trans. Program. Lang. Syst.* 23.6 (2001), pp. 685–746. DOI: 10.1145/506315.506316 (cit. on pp. 45, 46, 67).

[Gro+97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. "Call Graph Construction in Object-Oriented Languages". In: *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*. Ed. by Mary E. S. Loomis, Toby Bloom, and A. Michael Berman. ACM, 1997, pp. 108–124. DOI: 10.1145/263698.264352 (cit. on p. 46).

[HCF05] V. Haldar, D. Chandra, and M. Franz. "Dynamic taint propagation for Java". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. 2005, 9 pp.–311. DOI: 10.1109/CSAC.2005.21 (cit. on p. 8).

[HL09] Ben Hardekopf and Calvin Lin. "Semi-Sparse Flow-Sensitive Pointer Analysis". In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 226–238. DOI: 10.1145/1594834.1480911 (cit. on pp. 20, 72, 95).

[HL11] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code". In: *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE. 2011, pp. 289–298 (cit. on pp. 2, 20, 21, 42, 72, 95).

[He+23] Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. "Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 101–113. DOI: 10.1145/3597926.3598041 (cit. on pp. 45, 46, 68).

[He+19] Dongjie He, Haofeng Li, Lei Wang, et al. "Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 267–279. DOI: 10.1109/ASE.2019.00034 (cit. on pp. 2, 4, 18–22, 26, 27, 30, 33, 41, 42, 45, 46, 68, 72, 73, 77, 82, 95, 97).

[He+21] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. “Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability”. In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 16:1–16:31. DOI: 10.4230/LIPICS.ECOOP.2021.16 (cit. on pp. 50, 55, 66).

[HLX21] Dongjie He, Jingbo Lu, and Jingling Xue. “Context Debloating for Object-Sensitive Pointer Analysis”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 79–91. DOI: 10.1109/ASE51524.2021.9678880 (cit. on p. 50).

[HLX22] Dongjie He, Jingbo Lu, and Jingling Xue. “Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis”. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 30:1–30:29. DOI: 10.4230/LIPIcs.ECOOP.2022.30 (cit. on pp. 47, 49–51, 55, 66, 67).

[HT01] Nevin Heintze and Olivier Tardieu. “Demand-Driven Pointer Analysis”. In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI ’01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 24–34. DOI: 10.1145/378795.378802 (cit. on pp. 16, 71, 75).

[Hin01] Michael Hind. “Pointer analysis: haven’t we solved this problem yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 54–61. DOI: 10.1145/379605.379665 (cit. on pp. 49, 71).

[Hua+15] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. “Scalable and precise taint analysis for Android”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 106–117. DOI: 10.1145/2771783.2771803 (cit. on pp. 46, 68).

[JJO18] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. “Precise and scalable points-to analysis via data-driven context tunneling”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276510 (cit. on p. 50).

[Jeo+17] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. “Data-driven context-sensitivity for points-to analysis”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133924 (cit. on p. 50).

[JM79] Neil D. Jones and Steven S. Muchnick. “Flow analysis and optimization of LISP-like structures”. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’79. San Antonio, Texas: Association for Computing Machinery, 1979, pp. 244–256. DOI: 10.1145/567752.567776 (cit. on p. 1).

[JKK06] N. Jovanovic, C. Kruegel, and E. Kirda. “Pixy: a static analysis tool for detecting Web application vulnerabilities”. In: *2006 IEEE Symposium on Security and Privacy (S&P’06)*. 2006, 6 pp.–263. DOI: 10.1109/SP.2006.29 (cit. on p. 19).

[KU76] John B. Kam and Jeffrey D. Ullman. “Global Data Flow Analysis and Iterative Algorithms”. In: *J. ACM* 23.1 (1976), pp. 158–171. DOI: 10.1145/321921.321938 (cit. on p. 1).

[KU77] John B. Kam and Jeffrey D. Ullman. “Monotone Data Flow Analysis Frameworks”. In: *Acta Informatica* 7 (1977), pp. 305–317. DOI: 10.1007/BF00290339 (cit. on pp. 10, 11).

[KB23] Kadiray Karakaya and Eric Bodden. “Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis”. In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2023, pp. 305–316. DOI: 10.1109/ICST57152.2023.00036 (cit. on p. vii).

[KB24] Kadiray Karakaya and Eric Bodden. “Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, 2024. DOI: 10.1145/3597503.3639092 (cit. on p. vii).

[KMB25] Kadiray Karakaya, Palaniappan Muthuraman, and Eric Bodden. “Pick Your Call Graphs Well: On Scaling IFDS-Based Data-Flow Analyses”. In: SOAP ’25. Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 43–50. DOI: 10.1145/3735544.3735587 (cit. on p. vii).

[KS13] George Kastrinis and Yannis Smaragdakis. “Hybrid context-sensitivity for points-to analysis”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 423–434. DOI: 10.1145/2491956.2462191 (cit. on pp. 50, 67).

[KSS09] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009 (cit. on pp. 7, 16).

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*. Ed. by Patrick C. Fischer and Jeffrey D. Ullman. ACM Press, 1973, pp. 194–206. DOI: 10.1145/512927.512945 (cit. on pp. 1, 11).

[Lho03] Ondrej Lhoták. “Spark: A flexible points-to analysis framework for Java”. In: (2003) (cit. on pp. 45, 49, 54, 67, 68, 71).

[LH06] Ondřej Lhoták and Laurie Hendren. “Context-Sensitive points-to analysis: is it worth it?” In: *Proceedings of the 15th International Conference on Compiler Construction*. CC’06. Vienna, Austria: Springer-Verlag, 2006, pp. 47–64. DOI: 10.1007/11688839_5 (cit. on p. 49).

[LH08] Ondřej Lhoták and Laurie Hendren. “Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation”. In: 18.1 (Oct. 2008). DOI: 10.1145/1391984.1391987 (cit. on p. 49).

[Li+21] Haofeng Li, Haining Meng, Hengjie Zheng, et al. “Scaling Up the IFDS Algorithm with Efficient Disk-Assisted Computing”. In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*. Ed. by Jae W. Lee, Mary Lou Soffa, and Ayal Zaks. IEEE, 2021, pp. 236–247. DOI: 10.1109/CGO51591.2021.9370311 (cit. on pp. 19, 40, 42, 45, 46, 68).

[Li+22] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. “Path-sensitive and alias-aware typestate analysis for detecting OS bugs”. In: *ASPLOS ’22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. Ed. by Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch. ACM, 2022, pp. 859–872. DOI: 10.1145/3503222.3507770 (cit. on pp. 3, 19).

[Li+18] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. “Precision-guided context sensitivity for pointer analysis”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 141:1–141:29. DOI: 10.1145/3276511 (cit. on pp. 50, 55, 66, 67).

[Li+20] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. “A Principled Approach to Selective Context Sensitivity for Pointer Analysis”. In: *ACM Trans. Program. Lang. Syst.* 42.2 (May 2020). DOI: 10.1145/3381915 (cit. on p. 21).

[LH22] Bozhen Liu and Jeff Huang. “SHARP: fast incremental context-sensitive pointer analysis for Java”. In: 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527332 (cit. on p. 49).

[LL05] V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis”. In: *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. Ed. by Patrick D. McDaniel. USENIX Association, 2005 (cit. on p. 1).

[LHX21] Jingbo Lu, Dongjie He, and Jingling Xue. “Selective Context-Sensitivity for k-CFA with CFL-Reachability”. In: *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*. Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Vol. 12913. Lecture Notes in Computer Science. Springer, 2021, pp. 261–285. DOI: 10.1007/978-3-030-88806-0_13 (cit. on pp. 55, 66).

[LX19] Jingbo Lu and Jingling Xue. “Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 148:1–148:29. DOI: 10.1145/3360574 (cit. on pp. 50, 55, 66, 67).

[LBS19] Linghui Luo, Eric Bodden, and Johannes Späth. “A Qualitative Analysis of Android Taint-Analysis Results”. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 102–114. DOI: 10.1109/ASE.2019.00020 (cit. on pp. 71, 97).

[Luo+22] Linghui Luo, Felix Pauck, Goran Piskachev, et al. “TaintBench: Automatic real-world malware benchmarking of Android taint analyses”. In: *Empir. Softw. Eng.* 27.1 (2022), p. 16. DOI: 10.1007/S10664-021-10013-5 (cit. on pp. 47, 66).

[MRR02] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to and side-effect analyses for Java”. In: *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*. Ed. by Phyllis G. Frankl. ACM, 2002, pp. 1–11. DOI: 10.1145/566172.566174 (cit. on pp. 50, 67).

[Mur+98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S.-C. Lan. “An Empirical Study of Static Call Graph Extractors”. In: *ACM Trans. Softw. Eng. Methodol.* 7.2 (1998), pp. 158–191. DOI: 10.1145/279310.279314 (cit. on p. 46).

[NL08] Nomair A. Naeem and Ondrej Lhotak. “Typestate-like Analysis of Multiple Interacting Objects”. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA ’08. Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 347–366. DOI: 10.1145/1449764.1449792 (cit. on p. 14).

[NLR10] Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. “Practical Extensions to the IFDS Algorithm”. In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Rajiv Gupta. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 124–144. DOI: 10.1007/978-3-642-11970-5_8 (cit. on pp. 19, 42).

[NT23] Prakash Neupane and Manas Thakur. “Variational Study of the Impact of Call Graphs on Precision of Android Taint Analysis”. In: *Proceedings of the 16th Innovations in Software Engineering Conference*. ISEC ’23. Allahabad, India: Association for Computing Machinery, 2023. DOI: 10.1145/3578527.3578545 (cit. on p. 67).

[Oct+16] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick D. McDaniel. “Composite Constant Propagation and its Application to Android Program Analysis”. In: *IEEE Trans. Software Eng.* 42.11 (2016), pp. 999–1014. DOI: 10.1109/TSE.2016.2550446 (cit. on pp. 3, 19).

[Oh+12] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. "Design and Implementation of Sparse Global Analyses for C-like Languages". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: Association for Computing Machinery, 2012, pp. 229–238. DOI: 10.1145/2254064.2254092 (cit. on pp. 2, 20, 42, 72, 95).

[PK13] Rohan Padhye and Uday P. Khedker. "Interprocedural Data Flow Analysis in Soot Using Value Contexts". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*. SOAP '13. Seattle, Washington: Association for Computing Machinery, 2013, pp. 31–36. DOI: 10.1145/2487568.2487569 (cit. on p. 73).

[Ram94] G. Ramalingam. "The undecidability of aliasing". In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1994), pp. 1467–1471. DOI: 10.1145/186025.186041 (cit. on pp. 16, 49).

[Ram02] G. Ramalingam. "On sparse evaluation representations". In: *Theoretical Computer Science* 277.1 (2002). Static Analysis, pp. 119–147. DOI: https://doi.org/10.1016/S0304-3975(00)00315-7 (cit. on p. 18).

[Ran] Android Rank. *Free Android Market Data, History, Rankings | since 2011*. https://www.androidrank.org/. (Accessed on 09/29/2022) (cit. on pp. 88, 94).

[Rei+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. "Call Graph Construction for Java Libraries". In: FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 474–486. DOI: 10.1145/2950290.2950312 (cit. on p. 35).

[Rei+19] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. "Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs". In: ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 251–261. DOI: 10.1145/3293882.3330555 (cit. on p. 67).

[Rep] Maven Repository. *Maven Repository: Search/Browse/Explore*. https://mvnrepository.com/. (Accessed on 03/30/2023) (cit. on p. 35).

[Rep+05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. "Weighted pushdown systems and their application to interprocedural dataflow analysis". In: *Science of Computer Programming* 58.1 (2005). Special Issue on the Static Analysis Symposium 2003, pp. 206–263. DOI: https://doi.org/10.1016/j.scico.2005.02.009 (cit. on pp. 72, 76).

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 49–61. DOI: 10.1145/199448.199462 (cit. on pp. 2, 5, 7, 11–13, 18–20, 30, 42, 48, 66, 73).

[Ric53] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on p. 1).

[Ruf95] Erik Ruf. “Context-insensitive alias analysis reconsidered”. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI ’95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 13–22. DOI: 10.1145/207110.207112 (cit. on p. 2).

[SRH96] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. “Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation”. In: *Theor. Comput. Sci.* 167.1&2 (1996), pp. 131–170. DOI: 10.1016/0304-3975(96)00072-2 (cit. on pp. 3–5, 7, 8, 11, 13, 19, 20, 23–25, 30, 41, 42, 66, 97).

[Sch+24] Fabian Schiebel, Florian Sattler, Philipp Dominik Schubert, Sven Apel, and Eric Bodden. “Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications: An Experience Report”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 36:1–36:28. DOI: 10.4230/LIPIcs.ECOOP.2024.36 (cit. on pp. 45, 46, 68).

[SHB19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. “PhASAR: An Interprocedural Static Analysis Framework for C/C++”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 393–410 (cit. on p. 68).

[SH97] Marc Shapiro and Susan Horwitz. “Fast and accurate flow-insensitive points-to analysis”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: Association for Computing Machinery, 1997, pp. 1–14. DOI: 10.1145/263699.263703 (cit. on p. 1).

[SP+78] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . ., 1978 (cit. on pp. 1, 50).

[Shi+18] Qingkai Shi, Xiao Xiao, Rongxin Wu, et al. “Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 693–706. DOI: 10.1145/3192366.3192418 (cit. on pp. 2, 20, 21, 42, 72, 94).

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. “Pick your contexts well: understanding object-sensitivity”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 17–30. DOI: 10.1145/1926385.1926390 (cit. on pp. 50, 67).

[soo12] soot-oss. *soot-oss/heros: IFDS/IDE Solver for Soot and other frameworks*. `https://github.com/soot-oss/heros`. (Accessed on 03/30/2023). 2012 (cit. on p. 34).

[SAB17] Johannes Späth, Karim Ali, and Eric Bodden. "IDE$^{al}$: efficient and precise alias-aware dataflow analysis". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 99:1–99:27. DOI: 10.1145/3133923 (cit. on pp. 19, 97).

[SAB19] Johannes Späth, Karim Ali, and Eric Bodden. "Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems". In: 3.POPL (Jan. 2019). DOI: 10.1145/3290361 (cit. on pp. 72, 76, 77).

[Spä+16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 22:1–22:26. DOI: 10.4230/LIPICS.ECOOP.2016.22 (cit. on pp. 4, 16, 17, 32, 34, 42, 69, 71, 72, 75, 76, 80, 85, 97).

[Sui+20] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. "On the recall of static call graph construction in practice". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1049–1060. DOI: 10.1145/3377811.3380441 (cit. on p. 67).

[SX16] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-Flow Analysis in LLVM". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 265–266. DOI: 10.1145/2892208.2892235 (cit. on pp. 2, 20, 21, 42, 72, 73, 94).

[SYX12] Yulei Sui, Ding Ye, and Jingling Xue. "Static memory leak detection using full-sparse value-flow analysis". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, pp. 254–264. DOI: 10.1145/2338965.2336784 (cit. on p. 18).

[Sui+11] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. "SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA". In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 155–171 (cit. on pp. 20, 72, 95).

[Sun+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, et al. "Practical virtual method call resolution for Java". In: *SIGPLAN Not.* 35.10 (Oct. 2000), pp. 264–280. DOI: 10.1145/354222.353189 (cit. on pp. 45, 48, 49, 54, 67, 73, 78, 79).

[TLX16] Tian Tan, Yue Li, and Jingling Xue. “Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting”. In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 489–510. DOI: 10.1007/978-3-662-53413-7_24 (cit. on p. 50).

[TLX17] Tian Tan, Yue Li, and Jingling Xue. “Efficient and precise points-to analysis: modeling the heap by merging equivalent automata”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 278–291. DOI: 10.1145/3140587.3062360 (cit. on pp. 50, 55, 67).

[TP00] Frank Tip and Jens Palsberg. “Scalable propagation-based call graph construction algorithms”. In: *SIGPLAN Not.* 35.10 (Oct. 2000), pp. 281–293. DOI: 10.1145/354222.353190 (cit. on pp. 45, 46, 67).

[Val+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, et al. “Soot - a Java bytecode optimization framework”. In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*. Ed. by Stephen A. MacKay and J. Howard Johnson. IBM, 1999, p. 13 (cit. on pp. 21, 30, 34, 35, 50, 67, 88).

[Vas+20] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. “Every build you break: developer-oriented assistance for build failure resolution”. In: *Empir. Softw. Eng.* 25.3 (2020), pp. 2218–2257. DOI: 10.1007/S10664-019-09765-Y (cit. on p. 1).

[Wan+23] Xizao Wang, Zhiqiang Zuo, Lei Bu, and Jianhua Zhao. “DStream: A Streaming-Based Highly Parallel IFDS Framework”. In: *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2488–2500. DOI: 10.1109/ICSE48619.2023.00208 (cit. on pp. 45, 46, 68).

[Wei+18] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. “Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps”. In: *ACM Trans. Priv. Secur.* 21.3 (Apr. 2018). DOI: 10.1145/3183575 (cit. on pp. 53, 66).

[Wil92] Frank Wilcoxon. *Individual comparisons by ranking methods*. Springer, 1992 (cit. on pp. 37, 90).

[YSX14] Sen Ye, Yulei Sui, and Jingling Xue. “Region-Based Selective Flow-Sensitive Pointer Analysis”. In: *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*. Ed. by Markus Müller-Olm and Helmut Seidl. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 319–336. DOI: 10.1007/978-3-319-10936-7_20 (cit. on p. 21).

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. “Pointer analysis for programs with structures and casting”. In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. PLDI ’99. Atlanta, Georgia, USA: Association for Computing Machinery, 1999, pp. 91–103. DOI: 10.1145/301618.301647 (cit. on p. 2).

[Yu+20] Xiaodong Yu, Fengguo Wei, Xinming Ou, et al. “GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 274–284. DOI: 10.1109/IPDPS47924.2020.00037 (cit. on p. 19).

# List of Figures

# List of Tables