PADERBORN UNIVERSITY
The University for the Information Society

Faculty for Computer Science, Electrical Engineering and Mathematics

Synchronized Pushdown Systems for Pointer and Data-Flow Analysis

Johannes Späth

Dissertation

submitted in partial fulfillment
of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors

Prof. Dr. Eric Bodden
Prof. Dr. Karim Ali

Paderborn, March 15, 2019

Abstract

Static data-flow analysis reasons about behaviour of software without executing it. A precise data-flow analysis transforms the program into context-sensitive, flow-sensitive, and field-sensitive approximation of the software. It is challenging to design an analysis of this precision efficiently. Context- and field-sensitive data-flow analysis, if fully precise, is undecidable, and any model of such precision cannot avoid an approximation.

This thesis presents a new data-flow approximation for context-, field- and flow-sensitive data-flow analysis. The solution, called synchronized pushdown systems (SPDS), solves precise distributive data-flow analysis problems by relying on two pushdown systems, one system models field-sensitivity, the other one context-sensitivity. The SPDS then only accepts results that are context- and field-sensitive. SPDS approximates only in corner cases that are rare in practice: at statements where both problems are satisfied but not along the same data-flow path. Experiments comparing SPDS to the standard model for field-sensitivity, k-limited access path, show that SPDS are almost as efficient as k-limiting with $k = 1$ although their precision equals to $k = \infty$.

Static data-flow analysis needs to resolve pointer relations when data escapes to the heap. Flows of pointers are difficult and costly to resolve because pointer relations are non-distributive. Nevertheless, this thesis shows that pointer analysis can be solved by subdividing pointer relations into multiple distributive computations, for each computation a SPDS can be consulted. Based on this design, the thesis presents the demand-driven pointer analysis Boomerang. Apart from relying on efficiently coordinating multiple SPDS, Boomerang minimizes it computational effort by only resolving the minimal part of pointer relations necessary to answer a points-to query.

Another contribution of this thesis is IDE^{al}, a generic and efficient framework for data-flow analyses, e.g., typestate analysis or mining of application programming interfaces (APIs). IDE^{al} resolves pointer relations automatically and efficiently by the help of Boomerang. This reduces the burden of implementing pointer relations into an analysis. Further on, IDE^{al} performs strong updates which makes the analysis sound and precise.

Apart from the fundamental problem of finding the right balance between precision and efficiency of a general static data-flow analysis, this thesis elaborates on a concrete application of Boomerang and IDE^{al} within a data-flow analysis that detects complex security vulnerabilities. Applying this data-flow analysis on large scale shows once more that synchronized pushdown systems enable a promising compromise between efficiency and precision.

Zusammenfassung

Statische Datenflussanalysen analysieren das Verhalten von Software ohne die Software dabei auszuführen. Eine präzise Datenflussanalyse transformiert das Programm in eine kontext-, fluss- und feld-sensitive Approximation der Software. Eine Analyse dieser Präzision effizient zu gestalten und implementieren ist eine Herausforderung. Die Ursache liegt darin, dass kontext- und feld-sensitive Datenflussanalyse, wenn sie vollständig präzise sein soll, ein unentscheidbares Problem darstellt. Daher müssen Approximierungen innerhalb des Modells auftreten und die Analyse verliert an Präzision.

Diese Arbeit präsentiert eine neue Approximation für kontext-, fluss- und feld-sensitive Datenflussanalysen. Die Lösung, die wir mit synchronisierte Kellersysteme (SPDS) bezeichnen, berechnet distributive Datenflussanalyse-Ergebnisse präzise und effizient. SPDS stützt sich dazu auf zwei Kellersysteme. Ein System modelliert Feld-Sensitivität, das andere Kontext-Sensitivität. Das SPDS akzeptiert ein Ergebnis nur, wenn beide Systeme das Ergebnis akzeptieren. SPDS verliert Präzision nur in Spezialfällen, die in der Praxis selten sind: in Fällen, in denen beide Systeme das Ergebnis akzeptieren, aber nicht entlang des gleichen Datenflusspfades. Experimente, die SPDS mit dem Standardmodell für Feld-Sensitivität (k-limited Access Path) vergleichen, zeigen, dass SPDS fast so effizient sind wie k-limiting mit einem Wert von $k = 1$, obwohl SPDS so genau sind wie k-limiting mit $k = \infty$.

Eine statische Datenflussanalyse muss den Heap modellieren und Pointer-Beziehungen auflösen. Pointer-Beziehungen sind nicht überall distributiv und daher ineffizient zu berechnen. Diese Arbeit zeigt, dass Pointer-Beziehungen in mehrere distributive Teilprobleme unterteilt werden können. Jedes einzelne Teilproblem kann effizient mit SPDS gelöst werden. Basierend auf diesem Design stellt diese Arbeit die bedarfsorientierte Pointer-Analyse Boomerang vor. Boomerang minimiert den Rechenaufwand, indem es nur den minimalen Teil der Pointer-Beziehungen berechnet.

Weiterhin präsentiert diese Arbeit IDE^{al}, ein generisches und effizientes Rahmenwerk für Datenflussanalysen, in dem Analysen für Typestate und das Extrahieren der Benutzung von Programmierschnittstellen realisiert werden kann. IDE^{al} löst Pointer-Beziehungen automatisch und effizient mit Boomerang auf. Außerdem führt IDE^{al} starke Updates durch, diese Updates führen zu präzisen und vollständige Analyseergebnissen.

Mit einer Datenflussanalyse zum automatischen Auffinden von Sicherheitsschwachstellen bringt diese Arbeit Boomerang und IDE^{al} in die Anwendung. In einem groß angelegten Experiment wird gezeigt, dass synchronisierte Kellersysteme einen vielversprechenden Ansatz für Datenflussanalysen liefern.

Im Stundenglas der Zeit reift stets der Sand zur Ewigkeit,
verwandelt Trauer, Tränen, Leid zu Sternenstaub der Achtsamkeit.

Winfried Späth

In memory of my dad.

Acknowledgement

This work would not have been possible without the help and support of many people.

Firstly, I would like to thank Eric Bodden for the incredible support on performing my research at Fraunhofer SIT in Darmstadt as well as later on at Fraunhofer IEM in Paderborn. Despite his heavy workload, when requested, Eric would always help me without hesitation and instantly managed to join technical discussion on a deep level. By asking the right questions and providing constructive feedback, he would give me suggestions that would help me to proceed. Also special thanks to him for letting me meander to Oracle Labs in Australia before joining his team in Paderborn again.

Secondly, my deep thanks to Karim Ali, whom I met while he was a PostDoc at Technical University of Darmstadt. Karim guided me continuously in my research. He kept the discussions constantly alive in regular meetings. A special thanks to Karim for his patience in rephrasing, shaping and rewriting many of my first very rough drafts and working through early concepts.

I also would like to thank Anders Møller for carefully reading and examining this thesis, and for making a trip to Paderborn for my PhD defense. Thanks also to Ben Hermann and Heike Wehrheim for joining the examination committee.

Further more, I want to thank all my collaborators. Working with Johannes Lerch led to inspiring technical discussions and insights on the topic of field-sensitivity. Many of these thoughts are continued and completed within this thesis.

Next, I would like to thank Stefan Krüger. Despite being colleagues for years, it was not until recently, that our fields of research merged and both of us were able to benefit from each others work and knowledge.

I would like to express my appreciation to Lisa Nguyen Quang Do for supporting me in benchmarking the work, and for her creativity in illustrations that brightened up the days.

Thanks also to Oracle Labs in Brisbane, and a special thanks to Cristina Cifuentes for the opportunity of interning. In my internship I was able to take on a whole new perspective on the topic of points-to analysis. Interesting discussion with experts in a wide range of areas triggered new ideas and provided me with new insights. In particular, I want to thank, Francois Gauthier and Paddy Krishnan who guided my work.

Further on, I would like to thank all my colleagues at Fraunhofer IEM, University of Paderborn, and Fraunhofer SIT in Darmstadt. Paderborn in particular offered me a helpful, constructive and friendly working environment. Thanks go to my department leader, Matthias Meyer and group leader Matthias Becker, who granted me, when requested, freedom to perform my research. This work would also not have been possible without funding from the Fraunhofer Attract program and an Oracle Collaborative Research Grant.

Last, but not least, I want to thank all my friends for motivating me, keep exploring and picking me up when motivation was lacking. I express a huge appreciation my mum and my dad, and my two brothers, Martin and Christian, who were the first to expose me to computer science.

Contents

1 Introduction

Static program analysis reasons with the semantics of computer programs without actually executing them and has a broad range of applications. Compilers rely on static program analysis to find code transformations optimizing the program's performance. Bug finding frequently uses static analysis to discover unintended behaviour, for instance, inconsistent program states originating from data races [41, 104], program crashes due to unchecked null pointers dereferences [69] or unhandled exceptions raised by misused application programming interfaces (APIs) [27, 39]. Furthermore, it is also proficient in the detection of security vulnerabilities where static analysis discloses privacy leaks [5, 31], SQL injections [59, 62], and executions of untrusted code [52].

Static analysis takes the program's code, either in source or in its compiled form, abstracts the code into a model and checks the model against the property of interest. For instance, a static data-flow analysis detecting privacy leaks traces the flow of sensitive data, e.g., passwords or credit card data, through the model and reports data-flows that are unintended, e.g., when the sensitive data is logged to the console. As the static analysis only approximates the actual code, a finding reported by the static analysis may not constitute a leak at runtime, i.e., the analysis imprecisely reports a *false positive.* The closer the model of the static data-flow analysis resembles the actual data-flow during program execution, the less false positives an analysis reports.

There are various design dimensions of a static analysis fine-tuning its precision. A data-flow analysis can be *intra-* or *interprocedural.* In the former, effects of a call site on a data-flow are over-approximated, while in the latter, effects are precisely modelled by analyzing the called method(s). Additionally, an interprocedural data-flow analysis is precise if it is *context-sensitive*, which means the data-flow analysis correctly models the call stack and the data-flow returns to the same call site it enters the method. A design dimension for object-oriented languages is *field-sensitivity.* A field-sensitive analysis reasons precisely with a data-flow that escapes to the heap when it is stored within a field of an object.

Apart from being precise, a static analysis is also expected to guarantee *soundness.* For example, a compiler only applies a code optimization if the optimization does not change the program's behaviour under *any* given user input. An analysis detecting unchecked null pointer dereferences better finds *all* critical dereferences within the program, a single *false negative*, i.e., if the analysis misses reporting an unchecked flow, it may lead to a program crash.

In practice, no static analysis can find all optimizations, all bugs, or all vulnerabilities within a program (no false negatives) and detect those with perfect precision (no false positives). False positives and false negatives are the fundamental consequence of Rice's theorem [79], which states that checking any semantic properties of a program is an undecidable problem. Consequently, any model for static analysis is forced to over- or under-approximate the actual runtime semantics of the program. Over-approximations add false positives to the result and reduce the *precision* of the analysis, under-approximations introduce false negatives and lower the analysis' *recall*.

Apart from the effect on precision and recall, the approximation is also the influencing factor on the performance of a data-flow analysis. An interprocedural data-flow is less efficient to compute in comparison to an intraprocedural analysis. Adding context- or field-sensitivity to an interprocedural analysis introduces additional complexity within the model and negatively affects the computational effort. Therefore, balancing precision, recall, and performance of a static analysis is a tedious task.

As a first contribution, this thesis proposes a new data-flow model that balances precision and performance while retaining the analysis' recall. The solution, called *synchronized pushdown systems* (SPDS), models a context-, field-, and flow-sensitive data-flow analysis taking the form of two pushdown systems [21]. One system models context-sensitivity, and the other one models field-sensitivity. Synchronizing the data-flow results from both systems provides the final results of the data-flow analysis. A context- and field-sensitive analysis is undecidable [73] and forces SPDS to over-approximate. SPDS, though, are specifically designed to expose false positives *only* in corner cases for which we hypothesize (and confirm in our practical evaluation) that they are virtually non-existent in practice: situations in which an improperly matched caller accesses relevant fields in the same ways as the proper caller would.

Pushdown systems solve context-free language reachability and have been studied intensively [11, 21, 48, 50, 76]. Therefore, SPDS are efficiently solvable by relying on existing efficient algorithms. SPDS are a replacement for the k-limited access-path model [18]. A k-limited access path abstracts how an object is dereferenced from the heap and consists of a local variable entailed by a sequence of field accesses of which the length is at most k. The length of the sequence is limited to k to prevent infinite chains, e.g., when analyzing recursive data-structures. The access path model over-approximates when the field sequence exceeds a length of k. k-limiting is the standard model for field-sensitive data-flow analysis [42] and is widely used [5,6,15,17,18,24,36,100,101]. Analyses with low values of k, e.g., $k = 1, 2, 3$, are efficient to compute but quickly introduce imprecision into the results, higher values of k make the analysis precise but also affect the analysis time exponentially. In our practical evaluation, we compare k-limiting to SPDS and demonstrate that SPDS are almost as efficient as $k = 1$ while being as precise as $k = \infty$.

The second contribution of this thesis tackles points-to analysis. Two distinct variables may access or point to the same object (or memory location), in which case, the two variables are *aliased*. A field-store statement that updates the content of a field of the object via one variable reflects on to the second aliased variable as well. Updating the field of the aliased variable renders points-to analysis a *non-distributive problem* [74, 90]. SPDS only solve the more restrictive distributive problems and cannot compute pointer relations. Despite this fact, with Boomerang, we present a demand-driven pointer analysis that coordinates multiple SPDS and efficiently computes pointer relations by subdividing pointer relations into their distributive and non-distributive parts. Whole program points-to analysis computes points-to sets, i.e., the set of potential runtime memory locations of a variable, for all pointer variables in the program. Unfortunately, whole program points-to analysis, if precise, is difficult to scale. Therefore, Boomerang computes information on-demand by queries (a pointer variable at a program statement). Based on the query, Boomerang computes the minimal data-flows necessary to construct the points-to set for the query variable. We compare Boomerang to two existing demand-driven pointer analyses [93, 104] and are able to show that Boomerang is more precise and efficient.

As a third contribution of this thesism we present IDE^{al}, a generic and efficient pointer-tracking framework for data-flow analyses. Technically, IDE^{al} relies on Boomerang to compute pointer relations efficiently and extends the pushdown systems of SPDS using *weights* [75]. With different weights, IDE^{al} can be instantiated to solve different data-flow problems. We present weights for the detection of misuses of an API, also called a *typestate analysis*, and weights for an analysis to mine API usage patterns [109]. For efficiency, where possible, IDE^{al} propagates aliases in a distributive manner. IDE^{al} performs sound *strong updates*. A weight that is updated on a variable, e.g., an API call changes the typestate of the underlying object, the weight update is also reflected to all other aliased variables. We evaluate an IDE^{al}-based typestate analysis in comparison to a state-of-the-art one [27], where we could measure analysis speed-ups between 3.9× to 99× at the same precision.

Finally, we discuss an IDE^{al} and Boomerang-based analysis for the detection of security vulnerabilities that result from incorrect API usages of the Java Cryptographic Architecture (JCA), a common library for cryptography. We apply the analysis on large scale and run it on 152,996 artifacts of the Maven Central[1] software repository and conclude that the analysis computes results efficiently.

To summarize, the main contributions of this thesis are:

- the concept of synchronized pushdown systems, a novel approximation to context-, field-, and flow-sensitive data-flow analyses,
- the precise on-demand pointer analysis Boomerang that computes points-to set and all aliases,
- the efficient and customizable data-flow framework IDE^{al}, and
- a thorough practical evaluation of the concepts, analysis and frameworks.

This thesis is structured as follows. Chapter 2 introduces to two common data-flow analysis problems that motivate this work. The chapter describes taint and typestate analysis for readers unfamiliar to common terms of static analysis. Next, the chapter Background (Chapter 3) introduces common terms

[1]https://mvnrepository.com/repos/central

of static data-flow analysis and details on Interprocedural Distributive Environment (IDE) and Weighted Pushdown Systems (WPDS), two different but closely related algorithms solving interprocedural data-flows. Chapter 4 presents synchronized pushdown systems, discusses the challenge of undecidability, SPDS' solution to it and elaborates on a worst-case complexity analysis. In Chapter 5, this thesis presents BOOMERANG and provides details on how the combination of multiple forward and backward directed SPDS compute precise pointer information on-demand. Further on, the chapter contains a thorough practical evaluation of BOOMERANG. The chapter concludes by discussing related work on pointer analysis. Chapter 6 presents the data-flow framework IDE^{al} as an extension to BOOMERANG using weights, and explains how IDE^{al} performs strong update. We present an instantiation of IDE^{al} for a typestate analysis and an instantiation of IDE^{al} to mine API usage pattern. Further on, this work evaluates and compares an IDE^{al}-based typestate analysis. In Chapter 7, this thesis discusses *CryptoAnalysis* that builds on BOOMERANG and IDE^{al} and present the experiment and evaluation on Maven Central. In Chapter 8, we compare and correlate different metrics of data-flow analysis regarding their impact on the analysis time. The focus of this chapter is a detailed practical comparison of SPDS, access graphs, and access paths based on the analysis client discussed within Chapter 5 and Chapter 6. The practical comparison completes the pure theoretic view of SPDS in Chapter 4. The thesis concludes in Chapter 9.

2 Motivating Examples

In this chapter, we discuss a general motivation for static data-flow analyses. We highlight two types of static data-flow: *taint analysis* and *typestate analysis*. While taint analysis is primarily used to detect injection flaws and privacy leaks, typestate analysis detects resource leaks and misuses of APIs. The research that we present in this thesis is fundamental, but applies to both types of data-flow analyses.

2.1 Taint Analysis

Injection flaws are the most predominant security vulnerabilities in modern software. Injection flaws occur when untrusted data flows to a command or a query that is interpreted and executed. In 2017, OWASP[1] lists *Injections* as the top category of vulnerabilities with the highest risk of being exploited. A typical example of an injection attack for a database-backed software system is a *SQL injection*. If a software system contains a SQL-injection vulnerability, the database can be compromised and manipulated, and the system is no longer trustworthy. An attacker can read, add, and even remove data from the database.

A system is vulnerable to a SQL injection attack, if the system does not properly *sanitize* user input and uses the input to execute a dynamically constructed SQL command. Figure 2.1 demonstrates a minimal back-end of a web-application vulnerable to a SQL injection. The back-end maps each incoming request to a call to `doGet()`[2] within the application and hands over a `HttpServletRequest` object that represents the request with its parameter. Method `doGet()` loads the user-controllable parameter `"data"` from the request object in line 11 and stores the `String` as value into a `TreeMap`. The `TreeMap` is maintained as field `requestData` of the `Application` object.

Assume the application to persist the map to the database at a later time of execution by calling `writeToDatabase`. The method `writeToDatabase` dereferences the field `this.requestData` to variable `map` in line 20 and iterates over all entries of `map`. For each entry, it constructs and executes two SQL queries (calls in line 26 and 29). The first query string only includes a `key` of the map, whereas the second query contains both, the `key` and the `value` of each map's entry. As the `value` of the map contains untrusted data, the application is vulnerable to a SQL injection attack in line 29 which executes the query string contained in variable `keyValueQuery`. With a correct sequence of characters, the attacker can

[1]https://www.owasp.org/

[2]Throughout this thesis, a term ending in `()` indicates a method. If unambiguous, we omit the parameters of the method.

```java
class Application{
  Map<String,String> requestData = new TreeMap<>();
  Connection conn = ...;


  /** Entry point to the web application.
  * The HttpServletRequest object contains the payload
  * of an incoming request.
  */
  void doGet(HttpServletRequest req, ...){
    String val = req.getParameter("data"); //Untrusted data
    Map<String,String> map = this.requestData;
    map.put("data", val);
  }

  /** Executes two SQL commands to store the content of
  * the Map this.requestData to the database.
  */
  void writeToDatabase(){
    Map<String,String> map = this.requestData;
    Statement stmt = this.conn.createStatement();
    for(Entry<String,String> entry : map.getEntries()){
      String key = entry.getKey();
      String value = entry.getValue();
      String keyQuery = "INSERT INTO keys VALUES (" + key+ ");";
      stmt.executeQuery(keyQuery);//No SQL injection
      String keyValueQuery = "INSERT INTO " + key +
           " VALUES (" + value + ");";
      stmt.executeQuery(keyValueQuery); //SQL injection
    }
  }
}
```

Figure 2.1: A web application vulnerable to a SQL injection attack.

end the SQL insert command and execute any other arbitrary SQL command. For example a command to delete the whole database.

Static data-flow analysis is an effective technique in preventing such injection flaws. However, detecting the SQL injection flaw in the example by means of a data-flow analysis is challenging to implement efficiently if the analysis is required to be precise and sound at the same time (i.e., no false positive and no false negatives). A precise and sound abstraction for the heap is required to model the data-flow through the map.

Injection flaws are detected by a static *taint analysis*, a special form of data-flow analysis. In the case of a taint analysis for SQL injections, a *taint* is any user-controllable (and hence also attacker-controllable and thus untrusted) input to the program. Starting from these inputs, a taint analysis models program execution and computes other aliased variables that are also *tainted*, i.e., transitively contain the untrusted input. When a tainted variable reaches a SQL query, the analysis reports a *tainted flow*. For the code example in Figure 2.1, variable `val` in method `doGet()` is tainted initially. To correctly flag the code as vulnerable, the static taint analysis must model variable `value` in line 24 to be aliased to `val`.

A data-flow analysis trivially detects the alias relationship when the analysis uses an imprecise model. For instance, the *field-insensitive* model taints the whole `TreeMap` object when the tainted variable `val` is added to the `map` in line 13. While field-insensitivity is trivial to model, the analysis results are highly imprecise. Not only are the values of the map tainted, but also any key and the field-insensitive analysis imprecisely marks the constructed SQL query in line 25 as tainted. Therefore, a field-insensitive analysis reports a false positive, as it marks line 26 to execute an unsanitized SQL query.

Field-sensitive data-flow analyses track data-flows through fields of objects and are more precise than field-insensitive analyses. A field-sensitive analysis only reports a single SQL injection for the example. However, the detection of the alias relationship between the variables `value` and `val` is more than non-trivial for a field-sensitive static analysis. The analysis must model the complete data-flow through the map, which spans from the call to `put()` in line 13 to the call in line 24 and involves several accesses to the heap. For instance, at the call to `put()` in line 13, the value `val` escapes as second argument to the callee's implementation of the method `put()` of the class `TreeMap`.

Listing 2.1 shows an excerpt of the callee's code taken from the Java 8 implementation[3] of `TreeMap`. The class contains an inner class `TreeMap.Entry` that lists three fields (`parent`, `right`, and `left`), each of type `TreeMap.Entry`. Method `put()` creates a `TreeMap.Entry` that wraps the inserted element (`value`). The `TreeMap.Entry` is then used to balance the tree (call to `fixAfterInsertion()` in line 36). The method `fixAfterInsertion()` iterates over all `parent` entries and calls `rotateLeft()` to shift around elements within the tree (line 42). The latter method stores to and loads from the fields `parent`, `right`, and `left` of the class `TreeMap.Entry`.

[3]`http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/eab3c09745b6/src/share/classes/java/util/TreeMap.java`

```
33 public V put(K key, V value) {
34   TreeMap.Entry<K,V> parent = //complex computation done earlier
35   TreeMap.Entry<K,V> e = new TreeMap.Entry<>(key, value, parent);
36   fixAfterInsertion(e);
37 }
38 private void fixAfterInsertion(Entry<K,V> x) {
39   while (x != null && x != root && x.parent.color == RED) {
40     //removed many branches here...
41     x = parentOf(x);
42     rotateLeft(parentOf(parentOf(x)));
43   }
44 }
45 private void rotateLeft(TreeMap.Entry<K,V> p) {
46   if (p != null) {
47     TreeMap.Entry<K,V> r = p.right;
48     p.right = r.left;
49     if (l.right != null) l.right.parent = p;
50     //removed 8 lines with similar field accesses
51     r.left = p;
52     p.parent = r;
53   }
54 }
```

Listing 2.1: Excerpt code example of `TreeMap` which is difficult to analyze statically.

The field-sensitive static taint analysis tracks variable `value`, which is the second parameter of method `put()`. To cope with heap-reachable data-flows, field-sensitive analyses commonly propagate data-flow facts in the form of access paths [5, 6, 15, 17, 18, 24, 36, 100, 101]. An access path comprises a local variable followed by a sequence of field accesses, and every field-store statement adds an element to the sequence. The `while`-loop of `fixAfterInsertion` (line 39) in combination with the three field stores (lines 48, 51, and 52) within the method `rotateLeft()` represent a common code pattern[4] that leads to the generation of access paths of all combinations contained in the set $T = \{\texttt{this}.f_1.f_2.\cdots.f_n.\texttt{value} \mid f_i \in \{\texttt{right}, \texttt{left}, \texttt{parent}\}, n \in \mathbb{N}\}$. The data-flow analysis reports the variable `value` of method `writeToDatabase()` to alias to variable `val` of method `doGet()` only if the correct access path exists in the respective set T of the statements retrieving the value from the map (`getEntries()` in line 22 and `getValue()` in line 24).

The set of data-flow facts T is unbounded. Because most static data-flow algorithms require a finite data-flow domain, they typically use k-limiting to limit

[4]Recursive data structures, for instance `LinkedList` and `HashMap`, generate such patterns. Additionally, using inner classes provokes these patterns as the compiler automatically stores the outer class instance within a field of the inner class.

the field-sequence of the access paths to length k [18]. When an access path of length larger than k is generated, the analysis conservatively over-approximates the $(k+1)^{th}$ field. Therefore, not only will the field `value` of a `TreeMap.Entry` of the map be tainted, but any other field will be tainted as well. For example, any `key` inserted into the map imprecisely is tainted as `TreeMap.Entry` has a field `key`. For this particular example, infinitely long field sequences are generated and for any value of k, k-limiting imprecisely reports `key` to alias to `value`.

Access graphs represent one approach that avoids k-limiting [29, 45]. They model the "language" of field accesses using an automaton. Access graphs represent the set T finitely and precisely. However, just as access paths, also access graphs suffer from the state-explosion we show in Listing 2.1. In the illustrated situation, the flow-sensitive analysis must store a set similar to T (not necessarily the same) of data-flow facts, i.e., access graphs, at *every* statement, and potentially *every* context where a variable pointing to the map exists. Given the large size of T, computing the data-flow fixed-point for all these statements is highly inefficient, and the use of access graphs does not improve it.

The solution of the synchronized pushdown systems that we present in this theses does not suffer from the state explosion, because a pushdown system efficiently represents millions and even infinitely many access paths in *one* concise pushdown automaton holding data-flow results for *all* statements. We discuss this in more detail in Chapter 4.

2.2 Typestate Analysis

A *typestate analysis* is a static data-flow analysis used, for instance, to detect misuses of APIs and is capable of detecting erroneous API uses at compile time, i.e., before execution. Typestate analyses use an API specification, mostly given in the form of a *finite state machine* (FSM) encoding the intended usage protocol of the API. Based on the specification, the analysis verifies the usage of the API within the code. For example, before an object is destructed, it must be in a state marked as accepting state within the FSM.

The API of the type `java.io.FileWriter` shipped with the standard Java Runtime is a textbook example[5] of an API for which a typestate analysis is helpful in preventing resource leaks. The API can be used to write data from the program to a file on the disk.

To use the API, the developer must first construct a `FileWriter` by supplying a `File` object that the `FileWriter` shall write to. Calling the method `write` on the `FileWriter` object with the respective data as argument tells the `FileWriter` which data shall be written into the `File`. Writing the content of a file to disk is an expensive operation delegated to the operation system and the API delays the respective system calls to the `close()` method of the `FileWriter` object. The API assumes the `close()` method to be called exactly once prior to the destruction of the object. If the user of the API does not call `close()`, the file remains open.

[5]In Java 7, `try-with-resources` blocks were introduced to automatically `close` and release file handles. We assume the developer does not use these syntax elements.

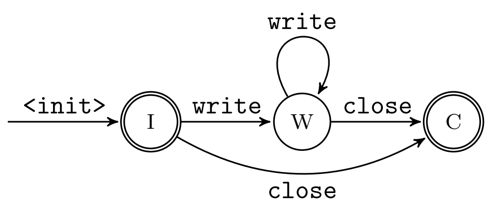


Figure 2.2: The API usage pattern encoded as finite state machine for the class `java.io.FileWriter`.

```
55 class Example{
56   FileWriter writer;
57   public void foo() throws IOException {
58     File file = new File("Data.txt");
59     this.writer = new FileWriter(file);
60     bar();
61     this.writer.close();
62   }
63 }
```

Figure 2.3: Simple, but challenging program to analysis for a typestate analysis.

The file resource is blocked by the process, and other processes may not read and write the same file and the program has a *resource leak*. Additionally, data is never written to the file as the output is only flushed to the file upon calling `close()`.

Figure 2.2 shows the finite state machine that represents a correct usage pattern for the API. The state labeled by I is the initial state. The transition into this state is labeled by `<init>` and refers to the constructor of a `FileWriter` object. The accepting states are the states I and C, the latter is the state in which the `FileWriter` object is correctly closed. All transitions into the C state are labeled by `close`. The state machine lists a third state (W) that the object switches into after a `write` call. In this state, data has been written to the `FileWriter` object but not yet persisted to the actual file on disk. Therefore, it is not an accepting state.

The program in Figure 2.3 shows a code snippet that uses the API. The code constructs a `FileWriter` object and stores it into field `writer` of the `Example` object. After method `bar()` is called, the field `writer` is loaded and the contained `FileWriter` object is `close`d in line 61.

One challenge of a typestate analysis is to perform *strong updates* when the state of an object changes. At the `close()` call in line 61, it is not clear which actual object is closed. If method `bar()` allocates a new `FileWriter` and overwrites the field `writer`, the `FileWriter` allocated in line 59 remains open and the typestate analysis cannot strongly update the state of the latter object. If the analysis detects only a single object to ever be pointed to by field `writer` at statement 61, a strong update can be made. However, the typestate analysis suddenly requires precise points-to information, which is notoriously challenging to obtain efficiently.

Points-to analysis computes points-to information. Despite much prior effort, it is known that a precise points-to analysis does not scale for the whole program [58]. Instead, the typestate analysis only requires points-to information for a rather small subset of all pointer variables, namely the variables pointing to objects that the `FileWriter` is stored within.

In Chapter 5, we present BOOMERANG, a demand-driven, and hence efficient, points-to analysis that computes results for a query given in the form of a pointer variable at a statement. BOOMERANG is precise (context-, flow-, and field-sensitive). We also present the BOOMERANG-based data-flow framework IDE^{al} in Chapter 6, a framework that is powerful enough to encode a typestate analysis that performs strong updates.

3 Background

In this chapter, we discuss the necessary background and terminology to understand this thesis. We define the semantics of the programs that we analyze and provide a brief overview on *data-flow frameworks* (Section 3.2) that describe standard techniques and concepts in static analysis.

Our work is based on weighted pushdown systems (WPDS) [75]. Originally, WPDS stem from the domain of model checking and not from the domain of data-flow analysis. WPDS solves data-flow problems equivalent to Inter-procedural Finite Distributive Subset (IFDS) problems [74] and Inter-procedural Distributive Environment (IDE) problems [80]. WPDS encodes the data-flow results in the form of an automaton, whereas algorithms solving IFDS and IDE problems construct a directed labelled graph, called the *exploded supergraph.* Within this thesis, we switch between the automaton and graph representations where it eases the presentation. Therefore, in this chapter, we also thoroughly explain the correspondence between WPDS and IFDS and IDE.

3.1 Program Semantics

We start this section by providing a definition of the semantics for the program that the static analysis operates on. We assume the code to be in three-address format, which means every statement has at most three operands. Table 3.1 lists all statements whose semantics our static analysis models.

There are *allocation sites*, $x \leftarrow \textbf{new}$. An allocation statement constructs a new object and assigns it to a local variable. There are *local assignment statements*, $x \leftarrow y$. Local assignment statements copy the reference to the object stored in local variable y to x.

A *call site* is a statement of the form $y \leftarrow m(p)$, where m is a called method[1] and p is the argument to the method. The call site may store a return value of the called method in the variable y. For a simpler formal representation, we restrict any method call to be static and to have a single parameter.[2] A virtual call that invokes a method on an object can be formally transformed in a static method call where the object instance flows as parameter to the static method.

A method returns a value via a *return site*, a statement **return** x. It returns the reference stored in variable x to the variable y of a call site.

The remaining statements involve fields. In the three-address format, every statement contains at most one field reference. There are *field store* and *field*

[1]Throughout this thesis, we assume to have access to a pre-computed call graph that is consulted in the case of dynamic dispatch.

[2]Our implementation handles call sites with multiple arguments as well as non-static calls.

Table 3.1: Three-address code that the analysis handles.

Statement	**Notation**
Allocation site	$x \leftarrow \textbf{new}$
Local assignment	$x \leftarrow y$
Call site	$y \leftarrow m(p)$
Return statement	$\textbf{return}\ x$
Static field store	$A.f \leftarrow y$
Static field load	$x \leftarrow A.f$
Non-static field store	$x.f \leftarrow y$
Non-static field load	$x \leftarrow y.f$
Array store of index i	$x[i] \leftarrow y$
Array load of index i	$x \leftarrow y[i]$

load statements, each of which exists in static and non-static form. Let $\mathbb{F}$ be the set of all fields of the classes of a program. A *static field store* has the form $A.f \leftarrow y$ and assigns the reference stored in variable y to the static field f of class A. A *static field load* has the form $x \leftarrow A.f$ and loads from the static field f of class A and stores the reference in local variable x.

Non-static field-store and load statements are fundamental for this thesis and we explicitly highlight their definition.

Definition 1. *A (non-static)* field-store statement *is a statement $x.f = y \in \mathbb{S}$. The variable $x \in \mathbb{V}$ is called the* base variable *of the store, $y \in \mathbb{V}$ the* stored variable *and $f \in \mathbb{F}$ the* stored field*.*

Correspondingly, a field-load statement is defined as follows.

Definition 2. *A (non-static)* field-load statement *is a statement of the form $y = x.f \in \mathbb{S}$. The variable x is called the* base (of the load)*, variable y is the* loaded variable *and field f the* loaded field*.*

The work we present in this thesis is *array-insensitive*, i.e., we model array store ($x[i] \leftarrow y$) and load statements ($x \leftarrow y[i]$) of some index i as non-static field store and load statements to a synthetic field $\mathit{ARRAY} \in \mathbb{F}$. This model disregards the index i of the access and all elements of the array are considered the same.

3.2 Data-Flow Frameworks

Static data-flow analysis originates from the need for program optimization, optimizations that are nowadays performed by most compilers such as dead-branch elimination or removal of unused variables. These program optimizations may clearly not break or change the functionality of the optimized code. Due to this assumption, an optimization may only be made, if it is a valid optimization for *all* paths of the program.

This motivated the design of a general data-flow framework that Kildall proposed first in 1973 [46]. The framework requires the following inputs: (1) a *control-flow graph* C representing the order of execution of the statements ($\mathbb{S}$) of a procedure, (2) a *data-flow domain* $\mathbb{D}$ consisting of *data-flow facts* ($d \in \mathbb{D}$) that abstracts concrete runtime elements (e.g., integer values), (3) a data-flow fact $d_0 \in \mathbb{D}$ that represents the abstract state at the beginning of the procedure, (4) flow functions[3], $f_s\colon \mathbb{D} \to \mathbb{D}$ for each statement $s \in \mathbb{S}$ of the procedure, and (5) a meet operator $\sqcap\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ that combines two data-flow facts to one fact when both meet at control-flow meet points. The goal of Kildall's framework is to compute the *meet-over-all-path* solution (MOP), which for a statement n of the procedure is defined as:

$$MOP(n) = \bigsqcap_{p(e,n) \in C} f^{p(e,n)}(d_0) \in D.$$

Here, $p(e,n)$ is a control-flow path between the entry statement to the procedure e and statement n. Say the path $p(e,n)$ has the form $(e, s_0, \ldots, s_m, n)$, then the function $f^{p(e,n)}$ is the composition of flow functions along the path, i.e., $f^{p(e,n)} = f_e \circ f_{s_0} \circ \cdots \circ f_{s_m} \circ f_n$. Due to loops, a procedure may have infinitely many control-flow paths and the MOP is in general uncomputable. Instead, Kildall suggests an iterative and decidable algorithm to compute the *maximal fixed-point* solution (MFP):

$$\begin{aligned} MFP(e) &= d_0 \\ MFP(n) &= \bigsqcap_{(m,n) \in C} f_n(MFP(m)) \in D. \end{aligned}$$

The initial maximal fixed-point for the entry point e is d_0 and for any control-flow edge $(m,n) \in C$, i.e., m is a predecessor statement of n, the iterative algorithm applies the flow function f_n to the fixed point of any predecessor $MFP(m)$ and merges the results. We say, the data-flow fact *propagates* from m to n. For monotone flow functions, the MFP solution over-approximates the MOP solution.

Kildall's framework establishes a generic solution for *intra-procedural* data-flow analysis. Intra-procedural analysis ignores calls to other procedures completely, unless explicit models for the procedures are specified in the flow functions.

Inter-procedural analysis overcomes the effort of manually modeling the effects of a call for a data-flow analysis. An inter-procedural analysis can be *context-sensitive* or *context-insensitive.* When a data-flow fact enters a method via a call site c (the context), the resulting data-flow fact $MFP(exit)$ at the *exit* statement of the callee method must return to the same context c. If the $MFP(exit)$ returns to any other call site, the analysis is context-insensitive.

A context-insensitive analysis contains data-flow paths that cannot be executed at runtime. These paths are knows as *inter-procedurally unrealizable paths* and make an analysis imprecise as data-flows computed along such paths are false positives. For programs written in object-oriented languages, data-flow analyses

[3]Originally called *gen* and *kill* functions.

that are context-insensitive are too imprecise to report meaningful results [56,64]. Two famous approaches to inter-procedural context-sensitive data-flow analysis are the *call-strings approach* and the *functional approach* [84].

For an analysis that implements the *call-strings approach*, each data-flow fact carries a (finite) sequence of contexts, the call string. The call string of the data-flow fact resembles the execution stack frame. A data-flow fact that enters a method via a call site c pushes the call site (the context) to the string of the respective fact. Hereby, the data-flow fact remembers the call site it returns to, once the fact reaches the exit statement of a method. When the data-flow fact flows back to the call site, it pops the call site from the call string. The call string approach has two main drawbacks. First, methods are potentially analysed multiple times, once for each individual context call string, no matter if the same data-flow information has already been propagated (under a different call string). Second, the call string must be finitely limited to handle recursion, once the limit is hit, the analysis is context-insensitive. In practice, limits of length 1 to 3 are standard to achieve scalable solutions [56, 61].

The *functional approach* constructs functional method *summaries*. A summary models the net transformation of a data-flow fact from the entry to the exit point of a method. The summaries are call-site independent, i.e., the data-flow fact may not contain caller specific information. This allows re-applying summaries at any call site. Yet, the construction of functional summaries is difficult. A summary outlines the transformation of a data-flow fact of some method m. Method m may call another method n, and the summary must include the effect of calling n within m. In general, the summary of any method called by m must be computed prior to the computation of the summary for m. Therefore, most analyses that use the functional approach are *bottom-up* [19, 24, 33]. A bottom-up analysis starts at the leaf methods of the call graph and constructs generic summaries based on abstract input parameters. This approach is opposed to *top-down* that starts at entry points of the call graph (e.g., the `main` method). It is a challenge for bottom-up analyses to keep the summaries as generic as possible while reducing the amount of case splitting necessary when the summary is applied under a concrete context.

The latter motivates *hybrid approaches* [70, 107] that perform top-down and bottom-up analyses at the same time, with the goal to restrict the amount and the complexity of the generated summaries. With the extensions made by Naeem [68], IFDS [74] constructs summaries on-the-fly and can be considered a hybrid functional approach.

3.3 The IFDS Algorithm

The algorithm for solving Inter-procedural Finite Distributive Subset (IFDS) problems [74] is an efficient fixed-point algorithm that can be used to define a flow- and context-sensitive data-flow analysis. Internally, IFDS transforms the data-flow analysis into a reachability problem over a graph. IFDS requires a *supergraph* and flow functions as input. A supergraph is the intra-procedural control-flow graph of the analyzed program (graph C of Kildall's framework) enriched by inter-procedural control-flow edges between caller and callees. The nodes of the supergraph are program statements ($\mathbb{S}$), and edges between them model the control-flow. The flow functions are data-flow problem dependent and describe the effect of each statement on a data-flow fact. Formally, for a statement $s \in \mathbb{S}$, a flow function has the form $f_s : \mathbb{D} \rightarrow \wp(\mathbb{D})$, i.e., a flow function receives a data-flow fact that holds before s as input and outputs a set of data-flow facts that hold *after* the statement. The format of the flow functions differ from Kildall's flow functions, because IFDS restricts the meet operator to be *set union.*

From the supergraph and the flow functions, IFDS generates the *exploded supergraph*, referred to as ESG. Each node $\langle d, s\rangle$ of the ESG is a pair of a statement of the program, $s \in \mathbb{S}$, and a data-flow fact $d \in \mathbb{D}$ of the analysis problem-specific data-flow domain $\mathbb{D}$ which has to be finite for IFDS. The ESG contains a directed edge from node $\langle d_1, s\rangle$ to $\langle d_2, t\rangle$, if t is a control-flow successor statement of s, and if the result of the *flow function* application for statement s to d_1 contains d_2, i.e., $d_2 \in f_s(d_1)$.

IFDS is a worklist algorithm and constructs only the relevant part of the ESG. Whenever a new node $\langle d_1, s\rangle$ in the ESG is generated, the data-flow fact is propagated to all control-flow successor statements t of the statement s. Hereby, the algorithm successively generates new ESG nodes for which the flow functions are re-applied. This process is repeated until no more new ESG nodes are generated, i.e., until a fixed-point of the nodes of the ESG is reached.

As an inter-procedural data-flow analysis, IFDS composes data flows across method boundaries and distinguishes between four different types of flow functions. There are two types of intra-procedural flow functions and two types of inter-procedural flow functions.

The intra-procedural flow functions are the *normal-flow functions* and the *call-to-return-flow functions*. The normal-flow functions specify the transformation of data-flow facts at non-call statements. At call sites, the call-to-return-flow functions propagate data-flow facts at the side of the caller.

IFDS uses two types of inter-procedural flow functions to propagate data-flow facts along control-flow edges connecting caller and callee methods at call sites. The *call-flow functions* map data-flow facts from the caller's scope to those of the potential callees. The *return-flow functions* map data-flow facts at exit points of a callee to the successor statements of the original call site.

IFDS assumes the flow function to be *distributive* functions. Their distributivity in combination with set union as meet operator is key to the efficiency of IFDS. A flow function f is distributive, if for any two data-flow sets $A, B \subseteq \mathbb{D}$ the equation $f(A \cup B) = f(A) \cup f(B)$ holds. Therefore, the result of the application of a distributive function on a set is equal to the application of the function on each individual element of the set and union the results. This property makes it sound and precise to propagate facts $d \in D$ individually. Non-distributive frameworks must instead always propagate entire flow sets $A \subseteq D$.

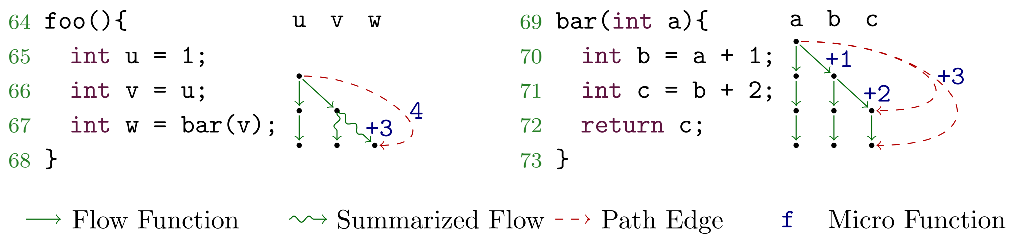
IFDS is a functional approach to data-flow analysis and uses function summaries. Distributivity allows IFDS to store and re-use *point-wise*, procedure *sum-*

maries, i.e., one per data-flow fact. Internally, IFDS constructs intra-procedural path edges. Path edges are directed edges between two *ESG* nodes that summarize the effect of the composition of multiple flow functions. The edges summarize intra-procedural data-flow edges within the *ESG* and are shortcuts in the graph. Every path edges is anchored in an *ESG* nodes whose statement is the first statement of a method. The flow functions iteratively extend the path edges. A path edge that reaches a return statement of a method is promoted as a summary. Hereby, IFDS constructs the summaries on-the-fly. The summary encodes how a data-flow fact entering a method is transformed transitively within the method. This encoding allows IFDS to reuse the summary at any call site as soon as the matching individual fact is seen again, which yields context-sensitivity. Technically, IFDS stores the contexts in the *incoming set* [68]. The incoming set stores a path edge reaching a call site in combination with the data-flow fact entering the callee method. When a path edge reaches the exit statement of the callee the propagation of the path edge stored within the incoming set is continued at the call site.

3.4 The IDE Algorithm

Sagiv et al. [80] extended IFDS to Interprocedural Distributive Environments (IDE) by additionally associating *lattice values* to each node of the *ESG*. A lattice value is an element of a bounded-height semi-meet-lattice $\mathbb{L}$. A *meet-lattice* is a partially ordered set such that any two elements of the set have a greatest lower bound with respect to the order. It is *bounded* in height, if the lattice has one greatest element.

In addition to the output of IFDS, IDE generates an environment for each statement $s \in \mathbb{S}$ of the program. An *environment* is a function $env_s{:}\,\mathbb{D} \to \mathbb{L}$ and maps a data-flow fact at the statement s to its corresponding lattice value. The environment is computed by *environment transformers*. Environment transformers are functions $t{:}\,Env(\mathbb{D},\mathbb{L}) \to Env(\mathbb{D},\mathbb{L})$, where $Env(\mathbb{D},\mathbb{L})$ is the set of all environments, i.e., an environment transformer maps one environment to another one. The environment transformers describe the effect of a statement on the lattice value for a particular data-flow fact. IDE requires those environment transformers to be *distributive*: $(t(\sqcap_i env_s^i))(d) = \sqcap_i(t(env_s^i))(d)$ for any $d \in \mathbb{D}, s \in \mathbb{S}$ and any infinite set of environments $env_s^1, env_s^2, ... \in Env(\mathbb{D},\mathbb{L})$. This property allows representing one environment transformer by multiple *micro functions* or *edge functions* [9]. The edge functions are additional input to IDE and are similar to the flow functions. Each edge of the *ESG* is labeled by one edge function. Each edge function has the form $f{:}\,\mathbb{L} \to \mathbb{L}$. IDE successively composes the edge functions along the data-flow when a flow function extends a path edge. Therefore, the composition of two functions $f_1 \circ f_2$ must be defined. Additionally, the edge functions must have a *meet* ($\sqcap$) operation. The meet of two edge functions define which edge function to propagate when the same path edge is generated along different control-flow branches with different associated edge functions.



```
64 foo(){
65   int u = 1;
66   int v = u;
67   int w = bar(v);
68 }
```

```
69 bar(int a){
70   int b = a + 1;
71   int c = b + 2;
72   return c;
73 }
```

Figure 3.1: Linear-constant propagation modeled in IDE.

Similar to IFDS, IDE is a fixed-point algorithm. During the *ESG* construction, the corresponding edge functions are composed, met, and propagated, once the construction of the *ESG* is done, the resulting edge functions are evaluated to yield the final lattice values associated with each node of the *ESG*, i.e., the environment. The latter process is called *Phase 2* [80] of IDE.

Example 1. Figure 3.1 shows an example that uses IDE to perform linear-constant propagation [80]. Linear-constant propagation propagates integer constants through the program and computes which variables contain a constant linear integer value. Next to the code, the figure depicts the *ESG* and some path edges that IDE generates during its fixed-point iteration. For this example, the data-flow domain $\mathbb{D}$ is the set of local variables $\mathbb{V}$. Each data-flow fact d (the local variable) is shown at the top of the column where the node is drawn. Nodes are placed between two statements, because each node represents a fact that holds after and before a statement. We use the notation `u`@65 to refer to the *ESG* node *after* line 65 with data-flow fact `u`.

Linear-constant propagation starts at assignments of constant integers to variables, here at line 65. Therefore, IDE computes graph reachability starting from the *seed ESG* node `u`@65. Line 65 assigns the constant `1` to variable `u`. The succeeding assignment `v = u` (line 66) transfers the value of `u` to `v`. The flow function of the assignment captures the data-flow; the constant flows to `v` and variable `u` stays constant. Therefore, Figure 3.1 draws the two straight edges (`u`@65 to `v`@66 and `u`@65 to `u`@66) labeled flow functions. The data-flow fact `v`@66 holds before the call to `bar` (line 67). The variable `v` is used by the call site and the call-flow functions are applied. The call-flow function maps the argument `v` to the formal parameter variable `a`. The analysis continues to constructs the *ESG* within method `bar`. When we ignore the increase of the integer values, parameter `a` flows to variable `b` which then flows to `c`. Hence, `a` flows transitively to the return variable `c`. The transitive flow is captured within the path edges of IDE (and IFDS). Figure 3.1 highlights the path edges in `bar` as dashed edges from `a`@69 to `b`@71 and `a`@69 to `c`@72. For the ease of presentation, we do not draw all path edges.

Upon generation of the path edge `a`@69 to `c`@72, IDE stores the edge as summary. It summarizes the data-flow information that `a` flows to `c`. This summary is then applied at the call site to `bar`. In Figure 3.1, the application of the path edge at the call site context is highlighted by the meandered edges labeled as summarized flows. The summarized flows extend the path edges within `foo`, where IDE generates the path edge `u`@65 to `w`@67. The edges carry the semantics that there exists a data-flow relation from `u` to `w`, and as `u` is constant, `w` is constant too.

IFDS and IDE construct the *ESG* in the same way. Unlike IFDS, IDE additionally allows tracking the concrete integer values associated to each variable at each statement. The integer values are propagated as edge functions of IDE.

For linear-constant propagation, a lattice value is a set of integers, i.e., the environment associates to each *ESG* node a set of integers. In Figure 3.1, the edge functions for IDE problem are depict as labels to the edges of the *ESG*. When there is no label to an edge, the edge function is the identity function that does not change the integer value. To the flow at statement `u = 1`, IDE assigns the constant edge function $\lambda v.1$, here denoted just by `1`. Within `bar`, the flow from `a` to `b` at the statement `b = a + 1` (line 70) receives the edge function $\lambda v.v + 1$, denoted by `+1`. This edge function simply increases every incoming lattice value by one. In the same way, the next statement in line 71 increases the value by `+2`. Those two edge functions are composed when the path edge `a`@69 to `c`@72 is generated. The composition of `+1` ∘ `+2` yields the edge function `+3` that is associated to the path edge. This path edge is promoted to a summary. The summary states that the value of variable `a` flows to `c` and additionally increases the lattice value by three.

In Phase 2 of IDE, the final lattice values are computed. Phase 2 propagates calling context-dependent information of the environments down to the callees. Before Phase 2, IDE computes that `b` within `bar` is `a` increased by one, but IDE does not propagate the actual value of `a` (nor `b`). The actual value of `a` is `1` but only under the call site context in line 67. For other call sites, the value may differ and the summary should not be restricted to this value to be as reusable as possible. Therefore, only Phase 2 propagates this value in a top-down manner along the call-flow functions to the callees and computes the final lattice values. In the example, Phase 2 computes that `b` is constant and equals to `2` and `c` is equal to `4`.

3.5 Pushdown Systems

Pushdown systems (PDS) were originally developed for model-checking [21, 28, 48]. A pushdown system consists of rules that correspond to the flow functions of IFDS. The application of an algorithm called post* [11, 21, 75, 82] solves the pushdown system and computes the same graph reachability problem IFDS computes. A pushdown system is leaner than IFDS in terms of data structures. IFDS stores the *ESG* in terms of path edges and maintains the contexts for each path edge within the incoming set [68]. Opposed to this, the algorithm post* produces an automaton (or equivalently a finite state machine) that holds both pieces of information, the path edges and their contexts.

Definition 3. *A* pushdown system *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$*, where* P *and* Γ *are finite sets called the* control locations *and the* stack alphabet*, respectively. A* configuration *is a pair* $\langle\langle p, w \rangle\rangle$*, where* $p \in P$ *and* $w \in \Gamma^*$*, i.e., a control location* *with a sequence of stack elements. The finite set* Δ *is composed of* rules*. A rule has the form* $\langle\!\langle p,\gamma\rangle\!\rangle \to \langle\!\langle p',w\rangle\!\rangle$*, where* $p,p' \in P$*,* $\gamma \in \Gamma$ *and* $w \in \Gamma^*$*.*

The rules of a pushdown system define a transition relation $\Rightarrow$ between configurations of $\mathcal{P}$: If $\langle\!\langle p,\gamma\rangle\!\rangle \to \langle\!\langle p',w\rangle\!\rangle$, then $\langle\!\langle p,\gamma w'\rangle\!\rangle \Rightarrow \langle\!\langle p',ww'\rangle\!\rangle$ for all $w' \in \Gamma^*$.

The length of w determines the type of the rule. A rule with $|w| = 1$ is called a *normal rule*, one with length 2 a *push rule*, and a rule of length 0 a *pop rule*. If the length of w is larger than 2, the rule can be decomposed into multiple push rules of length 2. A push rule pushes a stack element on the stack, a pop rule pops the last element off the stack. The normal rules do not change the length of the stack of the configuration.

When the PDS encodes an IFDS instance, the control locations are the data-flow facts of $\mathbb{D}$, the stack alphabet is $\mathbb{S}$, the set of statements of the program. The set of rules correspond to the edges of the *ESG*. A normal rule matches the intra-procedural data-flows, i.e., the normal-flow and the call-to-return-flow functions. The push rules resemble the call-flow functions, the pop rules are the equivalent to the return-flow functions.

When the individual stack elements of a sequence of $w \in \Gamma^*$ are relevant in this thesis, we write $w = w_0 \cdot w_1 \cdot \ldots \cdot w_n$. If unambiguous, we omit the $\cdot$ symbols and write $w = w_0w_1\ldots w_n$ instead. A configuration $\langle\!\langle p, w_0 \cdot w_1 \cdot \ldots \cdot w_n\rangle\!\rangle$ encodes an *ESG* node $\langle p, w_0\rangle$ with additional stack information. For a data-flow analysis, the remaining stack sequence $w_1 \cdot \ldots \cdot w_n$ tracks the calling context, in terms of call sites, over which the data-flow occurred.

Example 2. Table 3.2 lists all rules for the pushdown system that corresponds to the linear-constant propagation example discussed in Example 1 for IDE. For this example, we ignore the column *Weight*, as a PDS (analog to IFDS) cannot model the actual value of an integer. For any edge in the *ESG*, the PDS has a corresponding normal rule. Table 3.2 also lists the push and pop rules for the example. We discuss the push and pop rules in more detail. The push rule $\langle\!\langle \mathtt{v}, 66\rangle\!\rangle \to \langle\!\langle \mathtt{a}, 69 \cdot 67\rangle\!\rangle$ maps the argument `v` to the parameter `a`. This rule corresponds to the respective call-flow function of IFDS/IDE. Additionally, the rule replaces the top most element (the predecessor of the call site in line 66) of the stack by the first statement (line 69) of the called method and pushes the the call site (line 67) to the stack. The pop rule $\langle\!\langle \mathtt{c}, 72\rangle\!\rangle \to \langle\!\langle \mathtt{w}, \epsilon\rangle\!\rangle$ maps back the returned value `c` to variable `w`. The variable `w` is the assigned variable at the call site in line 67. In addition to that, the pop rule has an ϵ as a stack element of the target configuration. This setup means that when the rule is applied, the stack element 72 is replaced by ϵ. In other words, it removes 72 from the stack.

Taking the transitive closure of the transition relation $\Rightarrow$ (denoted by $\Rightarrow^*$) from a starting configuration c rises a set of reachable configurations called $post^*(c) = \{c' \mid c \Rightarrow^* c'\}$. Speaking in terms of IFDS, $post^*(c)$ is the set of all transitively reachable *ESG* nodes *and* their contexts starting from a given node c. The set can potentially be infinite, however, the set of configurations is regular[4] and it

[4]Despite the fact that pushdown systems solve context-free reachability problems, similar to IFDS, a pushdown system only constructs inter-procedural realizable path.

Table 3.2: Rules of the PDS for the example in Figure 3.1.

Normal Rule	Weight
$\langle\!\langle \texttt{u}, 65\rangle\!\rangle \rightarrow \langle\!\langle \texttt{u}, 66\rangle\!\rangle$	1
$\langle\!\langle \texttt{u}, 66\rangle\!\rangle \rightarrow \langle\!\langle \texttt{u}, 67\rangle\!\rangle$	
$\langle\!\langle \texttt{u}, 65\rangle\!\rangle \rightarrow \langle\!\langle \texttt{v}, 66\rangle\!\rangle$	
$\langle\!\langle \texttt{v}, 66\rangle\!\rangle \rightarrow \langle\!\langle \texttt{v}, 67\rangle\!\rangle$	
$\langle\!\langle \texttt{a}, 69\rangle\!\rangle \rightarrow \langle\!\langle \texttt{a}, 70\rangle\!\rangle$	
$\langle\!\langle \texttt{a}, 70\rangle\!\rangle \rightarrow \langle\!\langle \texttt{a}, 71\rangle\!\rangle$	
$\langle\!\langle \texttt{a}, 71\rangle\!\rangle \rightarrow \langle\!\langle \texttt{a}, 72\rangle\!\rangle$	
$\langle\!\langle \texttt{a}, 69\rangle\!\rangle \rightarrow \langle\!\langle \texttt{b}, 70\rangle\!\rangle$	+1
$\langle\!\langle \texttt{b}, 70\rangle\!\rangle \rightarrow \langle\!\langle \texttt{b}, 71\rangle\!\rangle$	
$\langle\!\langle \texttt{b}, 71\rangle\!\rangle \rightarrow \langle\!\langle \texttt{b}, 72\rangle\!\rangle$	
$\langle\!\langle \texttt{b}, 70\rangle\!\rangle \rightarrow \langle\!\langle \texttt{c}, 71\rangle\!\rangle$	+2
$\langle\!\langle \texttt{c}, 71\rangle\!\rangle \rightarrow \langle\!\langle \texttt{c}, 72\rangle\!\rangle$	

Push Rule	Weight
$\langle\!\langle \texttt{v}, 66\rangle\!\rangle \rightarrow \langle\!\langle \texttt{a}, 69 \cdot 67\rangle\!\rangle$	

Pop Rule	Weight
$\langle\!\langle \texttt{c}, 72\rangle\!\rangle \rightarrow \langle\!\langle \texttt{w}, \epsilon\rangle\!\rangle$	

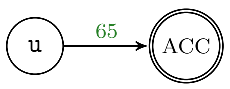
can be finitely represented by a finite automaton.

Definition 4. *Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, a* $\mathcal{P}$-automaton *is a finite non-deterministic automaton $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of* states*, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the set of transitions and $F \subseteq Q$ are the* final states*. The* initial states *are all control locations P of the pushdown system $\mathcal{P}$. A configuration $\langle\!\langle p, w\rangle\!\rangle$ is* accepted *by $\mathcal{A}$, if the automaton contains a path from state p to some final state $q \in Q$ such that the word along the path is equal to w. We write $\langle\!\langle p, w\rangle\!\rangle \in \mathcal{A}$ for an accepted configuration.*

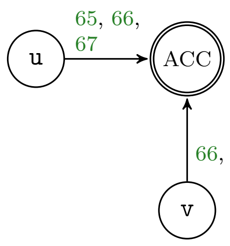
The $\mathcal{P}$-automaton encodes the set $post^*(c)$. Algorithm post* computes the set and requires as input a $\mathcal{P}$-automaton which accepts the initial configuration c. According to the rules of the pushdown system, the algorithm saturates the automaton with transitions, i.e., new transitions are added to the automaton until a fixed-point is reached. The saturation process is similar to IFDS' and IDE's construction of path edges for the realizable paths. We demonstrate the computation process for the $\mathcal{P}$-automaton and show correspondence to IFDS and IDE based on the linear-constant propagation example in Example 1.

Example 3. Table 3.2 lists the pushdown system for a linear-constant propagation performed on the code in Figure 3.1. Figure 3.2 presents the automaton that post* computes based on this pushdown system. The figure depicts the saturation process stepwise. The automaton drawn in Figure 3.2a shows the initial automaton that is input to post*.

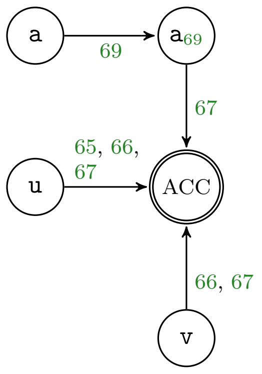
Linear-constant propagation starts at any assign statement assigning a constant value to a variable, for instance line 65 which assigns `u`, and the initial automaton accepts the configuration $\langle\!\langle \texttt{u}, 65\rangle\!\rangle$. This configuration is the start configuration of the two normal rules $\langle\!\langle \texttt{u}, 65\rangle\!\rangle \rightarrow \langle\!\langle \texttt{u}, 66\rangle\!\rangle$ and $\langle\!\langle \texttt{u}, 65\rangle\!\rangle \rightarrow \langle\!\langle \texttt{v}, 66\rangle\!\rangle$. Therefore, post* adds two transitions to the automaton. One transition from `u`



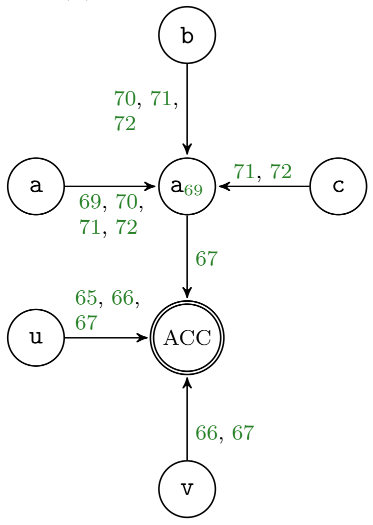
(a) The initial automaton.



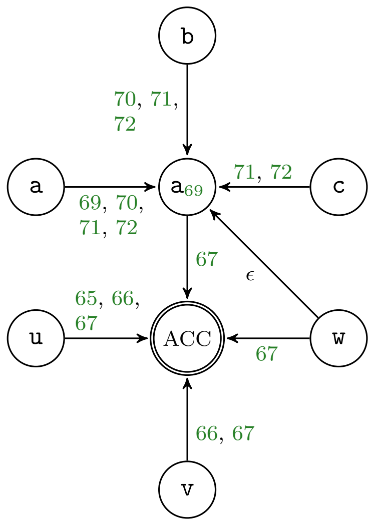
(b) Saturation of method `foo`.



(c) Applying the push rule.



(d) Automaton before application of the pop rule.



(e) The final automaton after saturation.

Figure 3.2: The successive construction of the $\mathcal{P}$-automaton for the pushdown system for Figure 3.1.

to the final state with label 66 and a second one with the same label and target but from `v`. Adding those rules means that $\langle\langle \texttt{u}, 66\rangle\rangle$ and $\langle\langle \texttt{v}, 66\rangle\rangle$ are accepted configurations. When all normal rules of method `foo` are applied, the automaton is the automaton drawn in Figure 3.2b.

At this state, the automaton accepts the start configuration of the push rule $\langle\langle \texttt{v}, 66\rangle\rangle \rightarrow \langle\langle \texttt{a}, 69 \cdot 67\rangle\rangle$. Therefore, the configuration $\langle\langle \texttt{a}, 69 \cdot 67\rangle\rangle$ must be accepted, which yields the automaton drawn in Figure 3.2c. The application of the rule adds the intermediate state $\texttt{a}_{69}$ to the automaton. From this intermediate state, the transition to the final state with stack symbol 67 as label is added. Additionally, a transition into $\texttt{a}_{69}$ from state `a` with label 69 is added. This renders configuration $\langle\langle \texttt{a}, 69 \cdot 67\rangle\rangle$ accepted.

Figure 3.2d shows the automaton when saturation is finished within method `bar`. All normal rules are applied and the appropriate transitions are added. The automaton encodes that variables `a`, `b`, and `c` are data-flow reachable under stack 67.

At this point, post* applies the pop rule $\langle\langle \texttt{c}, 72\rangle\rangle \rightarrow \langle\langle \texttt{w}, \epsilon\rangle\rangle$. Figure 3.2e shows the final automaton when this rule is applied. There is a transition out of `c` with label 72 and target state $\texttt{a}_{69}$. Hence, the rule dictates adding the ϵ transition from state `w` to target $\texttt{a}_{69}$. Due to the ϵ-transition, the configuration $\langle\langle \texttt{w}, 67\rangle\rangle$ is accepted. In terms of the data-flow, the acceptance of the configuration proves the data-flow connection between `u`@64 and `w`@67. The concrete constant value that is propagated along with the data-flow is computed by adding weights to the pushdown system, resulting in a *weighted pushdown system*. Therefore, the unweighted pushdown system corresponds to IFDS, while adding weights to the pushdown system corresponds to solving an IDE problem.

We highlight the correspondence between IFDS/IDE and post* on basis of the automaton drawn in Figure 3.2d. For Example 1, we visualized the concept of path edges in IFDS/IDE in Figure 3.1. Figure 3.1 lists two path edges in the form of dashed edges belonging to `bar`. In the $\mathcal{P}$-automaton in Figure 3.2d, those path edges correspond to the two transitions out of state `c` into state $\texttt{a}_{69}$. In IFDS the path edges are used as summaries. The algorithm post* can be summarized similarly [49]. The sub-automaton rooted in $\texttt{a}_{69}$ can be re-used as a summary for `bar`.

3.6 Weighted Pushdown Systems

Pushdown system subsequently have been extended by Schwoon et al. to *weighted pushdown systems* (WPDS) where each rule receives an additional *weight* [83]. IDE problems can be encoded as WPDS where the weights correspond to the edge functions of the IDE problem.

The weights for a pushdown system are elements of a *weight domain*. The weight domain has to satisfy the following assumptions to guarantee termination of algorithm post*:

Definition 5. *A* bounded idempotent semiring *(or* weight domain*) is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set whose elements are called* weights*,* $\overline{0}, \overline{1} \in D$*, and*

$\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that

1. *$(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$.*
2. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$$

3. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D, a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
4. *In the partial order $\sqsubseteq$ defined by $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

The terminology of weighted pushdown systems and IDE slightly differ, but they have the same principal concepts. There is a one-to-one correspondence between the two. IDE associates lattice values to each node of the *ESG*. The lattice values originate from a bounded distributive meet-lattice. Any bounded distributive lattice is also a weight domain. The meet operation of the lattice, $\sqcap$, is the same operator as $\oplus$. IDE takes the composition, in notation $\circ$, of the edge function along valid data-flow paths, hereby it extends the functions. In the terminology of weighted pushdown systems, the binary operator $\otimes$ extends two weights to a new one.

Linear-constant propagation is one application of IDE and can be encoded equivalently in WPDS. Reps et al. [76] discuss how linear-constant propagation encodes as weight domain and provides a proof of the required properties to comply as weight domain.

Example 4. We lift Example 3 to a weighted pushdown system. A weighted PDS expects a mapping of each pushdown rule to a weight in the weight domain. Table 3.2 lists the pushdown system rules with their weights for a linear-constant propagation performed on the code in Figure 3.1. When the cell for the weight in Table 3.2 is empty, the weight corresponds to the identity element, i.e., $\overline{1}$. The statement does not have any impact on the weight.

For example, the rule $\langle\!\langle \texttt{u}, 65 \rangle\!\rangle \rightarrow \langle\!\langle \texttt{u}, 66 \rangle\!\rangle$ carries weight `1` that encode that `u` in line 65 is assigned the constant value `1`. The rule $\langle\!\langle \texttt{a}, 69 \rangle\!\rangle \rightarrow \langle\!\langle \texttt{b}, 70 \rangle\!\rangle$ has weight `+1` associated. The rule indicates the integer value of `a` flows to variable `b` at statement 69, at the same time, its value is increased by one. A similar effect holds for rule $\langle\!\langle \texttt{b}, 70 \rangle\!\rangle \rightarrow \langle\!\langle \texttt{c}, 71 \rangle\!\rangle$ that has the weight `+2` associated.

For a pushdown system, the $\mathcal{P}$-automaton encodes all reachable configurations, i.e., all reachable *ESG* nodes of a data-flow problem given one start node. For a weighted pushdown system, a weighted $\mathcal{P}$-automaton is used. Each automaton's transition also carries a weight. The transitions of the automaton correspond to the path edges of an IFDS solution. As IDE associates an edge function to each path edge, in the terminology of weighted pushdown system, each $\mathcal{P}$-automaton's transition receives a weight. During construction of the weighted automaton, the weights are extended and combined.

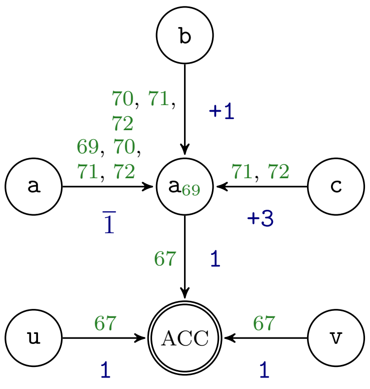


Figure 3.3: Partial post*-saturated weighted automaton for the WPDS of a linear-constant propagation performed on the code from Figure 3.1.

Example 5. Figure 3.3 shows a weighted pushdown automaton for the linear-constant propagation. It is the same automaton as computed for Example 3, just enriched by weights. We only depict the part of the automaton relevant for method `bar()` and relevant transitions with `foo()`. In the example, all transitions between two states are labeled by the same weight and instead of drawing the same weights multiple times, the weight's labels are drawn only once.

We discuss some of the transitions of the automaton. For instance, the weighted automaton contains transitions from `u` and `v` to the accepting state labeled by statement 67 with weight `1`. This encodes that the variables are constant with value `1` at the statement. Furthermore, the weighted automaton has a transition from state `a` to $\texttt{a}_{69}$ with label 72 and weight $\bar{1}$. The semantics of this transition is that there is a data-flow between `a`@69 and `a`@72 such that the weight remains unchanged: Whatever integer values flow into `bar()`, at the end of the method, variable `a` still holds the same integer value.

The automaton also has a transition `b` to $\texttt{a}_{69}$ with label 72 and weight `+1`. The integer value from the parameter `a` flows to `b`, but the integer value is increased by one.

The weighted automaton also contains a transition `c` to $\texttt{a}_{69}$ with label 71 and weight `+3`. This transition is the result of composing the two rules $\langle\!\langle \texttt{a}, 69\rangle\!\rangle \rightarrow \langle\!\langle \texttt{b}, 70\rangle\!\rangle$ with weight `+1` and $\langle\!\langle \texttt{b}, 70\rangle\!\rangle \rightarrow \langle\!\langle \texttt{c}, 71\rangle\!\rangle$ with weight `+2`. The two weights of the rules are extended and yields $\texttt{+1} \otimes \texttt{+2} = \texttt{+3}$.

The weighted automaton does not explicitly maintain the concrete integer values associated to the variables within `bar()`. For instance, the weights of the automaton do not encode variable `c` to hold the value `4` when called from call site in line 67. Still, the actual values within the callee can be computed by extending the weights along the edges of the automaton created by push rules. For instance, extending the weight `1` of the transition out of state $\texttt{a}_{69}$ into the accepting state by the weight for the transition out of `c` yields the weight under the respective calling context: $\texttt{1} \otimes \texttt{+3} = \texttt{4}$. This is equivalent to the computation of Phase 2 of IDE.

4 Synchronized Pushdown Systems

In Chapter 3, we familiarized the reader with pushdown systems and their application to inter-procedural context- and flow-sensitive data-flow analysis. This chapter presents the first main contribution of the thesis: *Synchronized Pushdown Systems*, a technique to incorporate field-sensitivity into a context- and flow-sensitive data-flow analysis that hereby achieves more precise data-flow results efficiently.

First, this chapter motivates and discusses a pushdown system for a field- and flow-sensitive data-flow analysis. We call this pushdown system the *field-PDS*. The field-PDS is a replacement for the concept of access paths, a widely used abstraction for field- and flow-sensitive analyses. Access paths require coarse over-approximations that hinder precision and scalability of data-flow analyses.

Second, the chapter recaps the pushdown system for context- and flow-sensitive data-flow problems. We call this system the *call-PDS*. Subsequent, the call-PDS and the field-PDS are synchronized to yield *synchronized pushdown systems* (SPDS). We show how appropriate synchronization of the two systems solves a precise data-flow analysis whose results are context-, field-, and flow-sensitive.

In general, field-sensitive and context-sensitive analysis is undecidable [73], which forces SPDS to over-approximate. Though, the over-approximation SPDS introduce, are specifically crafted to expose false positives *only* in corner cases, in situations, in which an improperly matched caller accesses relevant fields in the same ways as the proper caller would. In this chapter, we hypothesize that such cases are virtually non-existent in practice and confirm the hypothesis in our detailed practical comparison of access paths and SPDS in Section 8.1.

We published the work on SPDS at the 2019 Symposium on Principles of Programming Languages (POPL) [89]. Verbatim parts of our POPL publication are included in this chapter.

4.1 Imprecise and Inefficient Field Abstractions

In Section 2, we discussed a taint analysis based on a program which uses a `TreeMap` to store and load tainted data. A static data-flow analysis can only be sound and precise if data-flows through the map and its fields are correctly abstracted at field store and load statements. There exists a variety of different *field abstractions* [18,42] addressing this problem, however, we found that none of the existing abstractions is precise and efficient at the same time. The field-PDS is a replacement abstraction that is fully precise and at the same time efficient.

Field-based Domain The data-flow domain of a field-based analysis is the set $\mathbb{V} \cup \mathbb{F}$, i.e., a data-flow fact is either a local variable or a field name. At a field-

```
74 foo(u, v, w){
75   v.f = u;
76   x = w.f;
77   w.g = v;
78 }
```

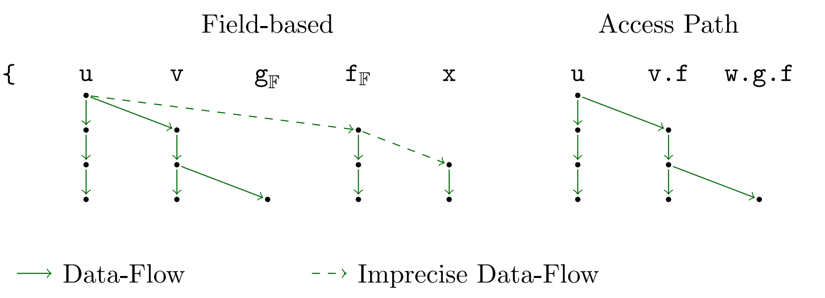


Figure 4.1: Field-based abstraction compared to the access path data-flow model.

store statement $x.f = y$, the field-based abstraction generates two new data-flow facts. It generates one data-flow fact for the base variable x, and one for the stored field f.

In Figure 4.1, we provide an example of an exploded supergraph constructed by a field-based analysis. Assume a (taint or typestate) analysis to track the data-flow fact `u` at the beginning of `foo`. The first statement of `foo` in line 75 is the field store `v.f = u`. The analysis generates two data-flow facts: the variable `v` and the field $\texttt{f}_{\mathbb{F}}$. We use the subscript $\mathbb{F}$ to indicate that the data-flow fact is a field. The subsequent line 76 is a field-load statement (`x = w.f`) that loads field $\texttt{f}_{\mathbb{F}}$ again. Due to the previously generated data-flow fact $\texttt{f}_{\mathbb{F}}$, the analysis assumes the tracked data to flow to the loaded variable `x`. This data-flow is imprecise: If variables `v` and `w` do not alias, variables `x` and `u` do not contain the same data. In the figure, the imprecise data-flows are highlighted as dashed edges.

Access-Path Domain The access path-domain is a more precise abstraction than the abstraction used in the field-based domain. An access path is an element of $\mathbb{V} \times \mathbb{F}^*$. We write an access path as $y.f_0 \cdot f_1 \cdot \ldots \cdot f_n$ where $y \in \mathbb{V}$ is a local variable, the *base*, and $f_0 \cdot f_1 \cdot \ldots \cdot f_n$ is a finite sequence of fields, i.e., $f_i \in \mathbb{F}$. In Figure 4.1, aside from the propagations for the field-based abstraction, the same data-flow problem is solved with an analysis based on an access path-domain. The solution of this data-flow problem is more precise than an analysis using the field-based abstraction. At the field store `v.f = u` in line 75, the access path `v.f` is generated. The subsequent statement `x = w.f` does not load this field, because the base of the access path, `v`, differs from the base of the field-load statement `w` and we assume the variables are not aliased. The length of the sequence of fields of the access path grows with every encountered field-store statement. The data-flow fact `v.f` reaches the field store statement (`w.g = v`) in line 77, and after that statement, the tracked data is also accessible by de-referencing `w.g.f`.

The access-path domain has a problem that leads to undecidability: an access path may grow infinitely long for programs that contain control-flow backward edges, such as loop constructs and recursive methods. Figure 4.2 shows a minimal example program which generates infinitely long sequences of access paths as of a simple `while` loop. Suppose the data-flow analysis propagates parameter `a`. At the end of the loop in line 84, the data is stored in the access paths `b.f` and `a.f`. As the loop may be executed a second time, the control flow graph has a backward edge from the last statement of the loop to the first one. Therefore, the data-flow

```
foo(a){
  while(...){
   b = new B();
   b.f = a;
   a = b;
  }
}
```

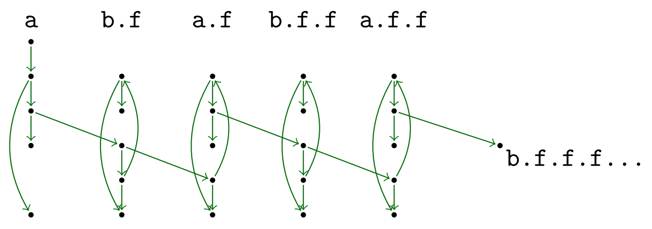
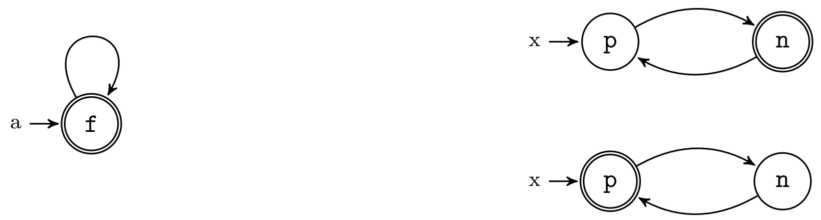


Figure 4.2: Infinite number of propagations generated by an analysis using the access-path model.



(a) Access graph for access path `a(.f)*` generated for example of Figure 4.2.

(b) Access graph for all access paths `x(.p.n)*` and `x(.p.n)*.p`.

Figure 4.3: Cyclic access graphs that an analysis generates when analyzing cyclic data structures.

algorithm re-injects the data-flow facts and propagates all facts a second time through the loop. The additional iteration yields the access paths `a.f.f` and `b.f.f`, which another time differ from the results of the previous iteration. The static analysis cannot decide how often the loop is executed and assumes infinitely many executions, which cause the access paths' field sequences to grow infinitely long. Consequently, the data-flow algorithm cannot reach a fixed point and does not terminate. The approach k-limiting [18] cuts the sequence at a length of k and over-approximates the $(k+1)^{th}$ field by a $*$ symbol that imprecisely allows any field to be loaded from. A k-limited analysis with a larger value of k is more precise, but also less efficient than an analysis with a smaller value. The larger k, the more access paths the analysis potentially generates.

Access-Graph Domain Access graphs [29, 45] are a more precise abstraction than access paths. Instead of modeling the field accesses in the form of a sequence, paths in a graph model the field sequences. The graph's nodes are labeled by fields, and each paths through the graph forms the sequence of fields of an access path. The graph representation does not need to k-limit, because infinitely long sequences simply correspond to cyclic paths in the graph.

Figure 4.3a illustrates an access graph that an analysis generates for the code in Figure 4.2. The figure represents all access paths `a(.f)*` as a single access graph. The Kleene-star ($*$) denotes a regular-expression-like syntax, the access path can repeat field `f` infinitely often. Hence, a single access graph suffices to represent the infinite number of access paths and k-limiting is not required. Still, the access graph model is inefficient. Access graphs may slightly differ at every statement in the program (and potentially under every context), which is why the analysis must maintain the access graphs separately per statement and context. In Figure 4.3b, we show two access graphs, the upper one represents all access paths of form `x(.p.n)*`, the lower one all of form `x(.p.n)*.p`. Both graphs differ in their accepting states (denoted as double circles). Efficient and distributive analysis frameworks, e.g., IFDS, cannot merge both access graphs to a simpler and more concise representation, because they only support the merge operator set union which maintains both graphs individually. Such individual propagation hinders scalability.

4.2 Field-Pushdown System

Field store and load statements can be modeled precisely as a pushdown system. The pushdown system overcomes the imprecision of k-limiting and renders the analysis more efficient. Instead of maintaining access graphs per statement, the pushdown system generates a single graph that encodes all field accesses at all statements. We first provide a formal definition of the system in the form of the pushdown system's rules before we demonstrate the pushdown system on examples.

Definition 6. *The* field-PDS *is the pushdown system* $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$*. A control location of this system is a pair of a variable and a statement. We use* $x@s$ *for an element* $(x, s) \in \mathbb{V} \times \mathbb{S}$*. The notation emphasizes that fact* x *holds* at *statement* s*. The pushdown system pushes and pops elements of* $\mathbb{F}$ *to and from the stack. An empty stack is represented by the* ϵ *field.*

A configuration of the field-PDS is an element of $\mathbb{V} \times \mathbb{S} \times \mathbb{F}^*$ and we write it as $\langle\!\langle x@s, f_0 \cdot f_1 \cdot \ldots f_n \rangle\!\rangle$. The configuration can be read as follows, at statement s the data-flow resides in the access path $x.f_0 \cdot f_1 \cdot \ldots f_n$.

In the following, we construct the set of rules $\Delta_{\mathbb{F}}$ as the disjoint union of the sets of normal ($\Delta_{\mathbb{F}}^{normal}$), push ($\Delta_{\mathbb{F}}^{push}$), and pop ($\Delta_{\mathbb{F}}^{pop}$) rule sets.

4.2.1 Normal Rules

The field-PDS $\mathcal{P}_{\mathbb{F}}$ pushes and pops fields to and from its stack. The statements that push and pop the fields are the store and load statements. All other statements maintain the field stack unchanged and constitute as normal rules to the field-PDS.

We construct the normal rules by the help of the function *normalFieldFlow*. The function maps from $\mathbb{V} \times \mathbb{S}$ to $\wp(\mathbb{V})$. Table 4.1 lists the *normalFieldFlow* function. The first two columns describe the inputs, while the third column contains the respective output set $O \subseteq \mathbb{V}$ of the function.

Table 4.1: The function *normalFieldFlow* for $\mathcal{P}_{\mathbb{F}}$. Within the comment column t refers to the input statement.

Variable ($\mathbb{V}$)	**Statement** ($\mathbb{S}$)	**Out** ($\wp(\mathbb{V})$)	**Type**	**Comment**
x	$x \leftarrow *$	$\varnothing$	intra	kill x at any assignment to x
y	$x \leftarrow y$	$\{x, y\}$		y copied to x at t
y	$x.f \leftarrow y$	$\{y\}$		info on y is retained at t
y	$x \leftarrow y.f$	$\{y\}$		info on y is retained at t
y	$A.f \leftarrow y$	$\{y, t\}$		generate static field $A.f$
$A.f$	$A.f \leftarrow y$	$\varnothing$		kill static field $A.f$
p_i	$m(p_1, p_2, \ldots, p_n)$	$\{q_i\}$	inter	p_i copied to formal q_i
q_i	**return**	$\{p_i\}$		q_i copied to actual p_i
x	**return** x	$\{y\}$		x copied to assigned value y

Assume $\mathcal{P}_{\mathbb{F}}$ accepts a configuration $\langle\langle x@s, g_0 \cdot \ldots \cdot g_n \rangle\rangle$ and let t be an intraprocedural control-flow successor of s. Assume further that $y \in$ *normalFieldFlow*(x, t), then $\mathcal{P}_{\mathbb{F}}$ has a rule:

$$\langle\langle x@s, g_0 \rangle\rangle \to \langle\langle y@\tilde{t}, g_0 \rangle\rangle \in \Delta_{\mathbb{F}}^{normal}.$$

It is $\tilde{t} = t$, unless t is a call site or a return statement (cases for which Table 4.1 lists inter as type). If t is a call site, $\tilde{t}$ is the first statement of the callee. For a return statement t of a method m, $\tilde{t}$ is defined as any intra-procedural control-flow successor of any call site calling m.

The start configuration and the target configuration of the rules have the same field as the stack location. Therefore, none of these rules add or remove an element from the stack.

For an assignment statement $t{:}\, x \leftarrow y$, the field-PDS contains the normal rules $\langle\langle y@s, g_0 \rangle\rangle \to \langle\langle y@t, g_0 \rangle\rangle$ and $\langle\langle y@s, g_0 \rangle\rangle \to \langle\langle x@t, g_0 \rangle\rangle$. At the successor statement t of s, the data is reachable via variables x and y. A field-store statement $t{:}\, x.f \leftarrow y$ gives rise to only the normal rule $\langle\langle y@s, g_0 \rangle\rangle \to \langle\langle y@t, g_0 \rangle\rangle$. The normal rules of $\mathcal{P}_{\mathbb{F}}$ do *not* handle the store to field f of variable x, which is instead taken care of by a push rule.

The field-PDS models control-flow explicitly in the control locations of the pushdown system. $\mathcal{P}_{\mathbb{F}}$ is defined context-insensitively. At a return statement of a method m, data-flow propagates to all call sites of m.

4.2.2 Push Rules

When a configuration $\langle\langle y@s, g_0 \cdot \ldots \cdot g_n \rangle\rangle$ is accepted by $\mathcal{A}_{\mathbb{F}}$ and some successor t of s is a field-store statement, i.e., $t{:}\, x.f \leftarrow y$, $\mathcal{P}_{\mathbb{F}}$ lists the push rule

$$\langle\langle y@s, g_0 \rangle\rangle \to \langle\langle x@t, f \cdot g_0 \rangle\rangle \in \Delta_{\mathbb{F}}^{push}.$$

```
86 foo(u, v, w){
87   v.f = u;
88   x = w.f;
89   if(...){
90     w.g = v;
91   } else {
92     w.h = v;
93   }
94 }
```

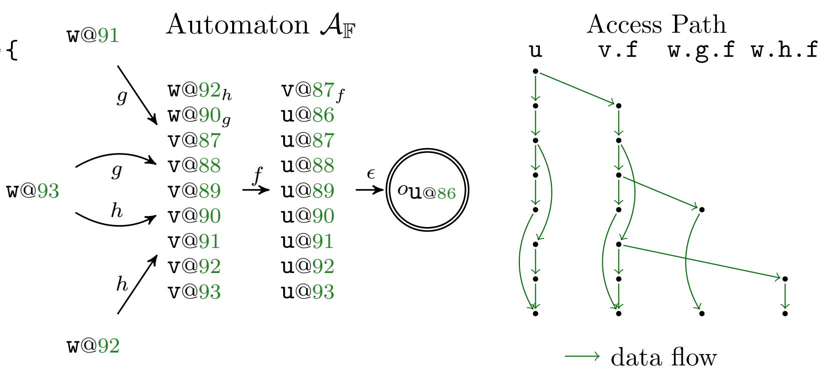


Figure 4.4: A post*-saturated $\mathcal{A}_{\mathbb{F}}$ when initialized with the configuration $\langle\langle$u@87$, \epsilon\rangle\rangle$ and saturated with $\mathcal{P}_{\mathbb{F}}$ listed in Table 4.2. Next to it, the same information represented as standard data-flow graph with an access-path domain.

Due to the push rule, post* marks the configuration $\langle\langle x@t, f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n\rangle\rangle$ as accepting as well. Therefore, an access path $y.g_0 \cdot g_1 \cdot \ldots \cdot g_n$ reaching statement t, generates the access path $x.f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n$ to hold after statement t, i.e., the field f is prepended to the access path.

A field-store statement indirectly updates the field of any variable that is must-aliased. $\mathcal{P}_{\mathbb{F}}$ does not model the update of the must-alias. This makes the resulting SPDS deliberately unsound. Such indirect updates of aliased variables are content of Chapter 5.

4.2.3 Pop Rules

The pop rules correspond to the runtime semantics of a field-load statement. For an accepting configuration $\langle\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n\rangle\rangle$ where the successor statement t of s is a load statement t: $x \leftarrow y.f$, the field-PDS $\mathcal{P}_{\mathbb{F}}$ lists a pop rule of form:

$$\langle\langle y@s, f\rangle\rangle \to \langle\langle x@t, \epsilon\rangle\rangle \in \Delta_{\mathbb{F}}^{pop}$$

The accepting configuration $\langle\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n\rangle\rangle$ induces the configuration $\langle\langle x@t, g_0 \cdot \ldots \cdot g_n\rangle\rangle$. In other words, when the access path $y.f \cdot g_0 \cdot \ldots \cdot g_n$ holds after statement s, the analysis continues to propagate the access path $x.g_0 \cdot \ldots \cdot g_n$ to hold after statement t.

Example 6. Figure 4.4 shows an example program code with three field-store and one field-load statements. $\mathcal{P}_{\mathbb{F}}$ modeling the code's data-flow is shown in the form of the rule set $\Delta_{\mathbb{F}}$ in Table 4.2. Table 4.2 lists normal, push, and pop rules for the data-flows in method `foo()`. For example, the normal rule $\langle\langle$u@86$, *\rangle\rangle \to \langle\langle$u@87$, *\rangle\rangle$ encodes that data flows from u@86 to u@87. The Kleene-star ($*$) at the stack location of the rule is a wildcard that can be replaced by any field $g \in \mathbb{F}$, i.e., the representation actually bundles multiple rules. The semantics of the rule is that any data stored in any field dereferenced from u at statement 86 is propagated to the successor statement 87, because statement 86 does not modify u.

Table 4.2: The rule set $\Delta_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$ for the code shown in Figure 4.4.

Normal Rules	
$\langle\!\langle \mathtt{u@86}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@87}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@88}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@89}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@87}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@88}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@89}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@90}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@88}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@89}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@90}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@93}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@89}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@90}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@89}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@91}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@90}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@93}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@91}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@92}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@89}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@91}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{v@92}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@93}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@91}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@92}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{w@90}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@93}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{u@92}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{u@93}, * \rangle\!\rangle$	$\langle\!\langle \mathtt{w@92}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@93}, * \rangle\!\rangle$
$\langle\!\langle \mathtt{v@87}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@88}, * \rangle\!\rangle$	

Push Rules
$\langle\!\langle \mathtt{u@86}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@87}, \mathtt{f} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathtt{v@89}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@90}, \mathtt{g} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathtt{v@91}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@92}, \mathtt{h} \cdot * \rangle\!\rangle$

Pop Rules
$\langle\!\langle \mathtt{w@87}, \mathtt{f} \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{x@88}, \epsilon \rangle\!\rangle$

The rule set $\Delta_{\mathbb{F}}$ contains three push rules, each of which matches a field-store statement. For example the push rule $\langle\!\langle \mathtt{u@86}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@87}, \mathtt{f} \cdot * \rangle\!\rangle$ encodes that any data stored in `u@86` flows to `v@87` at the same time pushing `f` to the top of the stack. We also use the Kleene-star notation, because the field `f` is pushed, no matter which field is on the stack.

Each field-load statement matches a pop rule. The presented $\mathcal{P}_{\mathbb{F}}$ lists the pop rule $\langle\!\langle \mathtt{w@87}, \mathtt{f} \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{x@88}, \epsilon \rangle\!\rangle$. When variable `w` reaches statement 87, i.e., `w@87` is propagated, and the stack topmost element is the field `f` (the tracked data is stored at least below field `f`), the data-flow continues to `x@88`, and field `f` is popped from the stack.

Based on $\mathcal{P}_{\mathbb{F}}$, algorithm post* can answer reachability queries over the system described in Table 4.2. The resulting post*-saturated $\mathcal{P}$-automaton, which we refer to as the *field automaton* $\mathcal{A}_{\mathbb{F}}$, contains field-sensitive and flow-sensitive data-flow results[1]. We assume $\mathcal{A}_{\mathbb{F}}$ to initially contain the transition $\mathtt{u@86} \xrightarrow{\epsilon} o_{\mathtt{u@86}}$. We label the accepting state of the automaton by $o_{\mathtt{u@86}}$, because it refers to the abstract object stored in variable `u` at the beginning of method `foo()`.

The transition labels of $\mathcal{A}_{\mathbb{F}}$ are elements of $\mathbb{F}$, i.e., fields of the program. The abstract object $o_{\mathtt{u@86}}$ is stored inside field `f` of variable `v` at the statement in line 87. The code then branches and in line 90, variable `v` is stored inside field `g` of some object pointed-to by `w`, line 92 stores variable `v` to field `h` of `w`. Therefore, in line 93, the abstract object $o_{\mathtt{u@86}}$ is transitively accessible either via access path `w.g.f` or via `w.h.f`. $\mathcal{A}_{\mathbb{F}}$ encodes this information as it accepts the two words[2] $\mathtt{g} \cdot \mathtt{f} \cdot \epsilon$ and $\mathtt{h} \cdot \mathtt{f} \cdot \epsilon$ starting from node `w@93`. It is also important to note that the field automaton does not contain a state with variable `x`. Data from `u` does only flow to `v.f` but cannot be loaded from `w.f`[3] , and `x` never becomes reachable.

Next to the $\mathcal{P}$-automaton, Figure 4.4 also shows the same data-flow analysis

[1]For a simpler representation of the automaton, we merged states of transitions with the same field label of the automaton.

[2]An *accepted word* $w = w_1 \cdot w_2 \cdot \dots \cdot w_n \in \mathbb{F}^*$ of $\mathcal{A}_{\mathbb{F}}$ is a path from a some node to the accepting state such that the concatenated transition labels form w.

[3]Chapter 5 discusses the case that `v` and `w` are aliases.

```
95  foo(a){
96    while(...){
97      b = new B();
98      b.f = a;
99      a = b;
100   }
101 }
```

$\langle\!\langle \texttt{a@97}, * \rangle\!\rangle \to \langle\!\langle \texttt{b@98}, \texttt{f} \cdot * \rangle\!\rangle$

$\langle\!\langle \texttt{a@99}, * \rangle\!\rangle \to \langle\!\langle \texttt{a@96}, * \rangle\!\rangle$

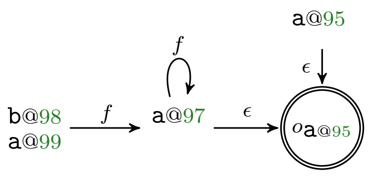


Figure 4.5: An example illustrating the finite representation that $\mathcal{P}_{\mathbb{F}}$ encodes for an infinite set of access paths.

but encoded in a data-flow graph (exploded supergraph) for an access-path based analysis. The example code does not contain a loop and only finitely many access paths are generated. Therefore, both representations encode the same information, and there is a unique transformation between the two. For instance, $\mathcal{A}_{\mathbb{F}}$ accepts configuration $\langle\!\langle \texttt{w@93}, g \cdot f \cdot \epsilon \rangle\!\rangle$. This configuration corresponds to the node with label `w.g.f` for statement 93 in the access-path based representation on the right.

However, $\mathcal{A}_{\mathbb{F}}$ encodes the same information more concisely. The access-path representation requires an explicit enumeration of the fields, `w.g.f` and `w.h.f` are encoded individually. Opposed to that, $\mathcal{A}_{\mathbb{F}}$ shares the information that prior to the branch the data-flow is stored in field `f`. $\mathcal{A}_{\mathbb{F}}$ only needs to store the two transitions labeled g and h out of `w@94`. The outgoing transition of the target node labeled by f encodes the remaining field of *both* access paths `w.g.f` and `w.h.f`. The automaton $\mathcal{A}_{\mathbb{F}}$ concisely merges the information sharable between multiple data-flow paths. It follows that the more branched field-store statements the analyzed code contains, the more efficient the automaton representation becomes.

Example 6 demonstrates that $\mathcal{A}_{\mathbb{F}}$ encodes the same information as a data-flow analysis based on access path. For this example code, a transformation from the automaton representation into the exploded supergraph is possible. However, $\mathcal{A}_{\mathbb{F}}$ can also encode access paths of infinite length and a transformation into a exploded supergraph (with finitely many nodes) is impossible.

Example 7. Figure 4.5 shows a minimal code to generate an infinite amount of access paths that requires k-limiting.

Figure 4.5 lists a subset of the rules of $\mathcal{P}_{\mathbb{F}}$, and next to it, the relevant transitions of $\mathcal{A}_{\mathbb{F}}$ that post* generates when tracing the abstract object $o_{\texttt{a@95}}$. Initially, $\mathcal{A}_{\mathbb{F}}$ accepts the configuration $\langle\!\langle \texttt{a@95}, \epsilon \rangle\!\rangle$. Between the statements from line 95 to line 97, variable `a` is not overwritten and configuration $\langle\!\langle \texttt{a@97}, \epsilon \rangle\!\rangle$ becomes accepting. Next, the push rule $\langle\!\langle \texttt{a@97}, * \rangle\!\rangle \to \langle\!\langle \texttt{b@98}, \texttt{f} \cdot * \rangle\!\rangle$ is applied and yields the accepting configuration $\langle\!\langle \texttt{b@98}, f \cdot \epsilon \rangle\!\rangle$. Because statement 99 transfers data-flow from `b` to `a`, the configuration $\langle\!\langle \texttt{a@99}, f \cdot \epsilon \rangle\!\rangle$ turns accepting. This configuration flows back to `a@96`, because the rule $\langle\!\langle \texttt{a@99}, * \rangle\!\rangle \to \langle\!\langle \texttt{a@96}, * \rangle\!\rangle$ encodes a control-flow backward edge from the end of the loop to the entry. Line 96 and line 97 do not overwrite variable `a` and due to the push rule $\langle\!\langle \texttt{a@97}, * \rangle\!\rangle \to \langle\!\langle \texttt{b@98}, \texttt{f} \cdot * \rangle\!\rangle$,

```
102 foo(u, v, w){
103   v.f = u;
104   w.g = v;
105   x = w.g;
106   y = x.g;
107 }
```

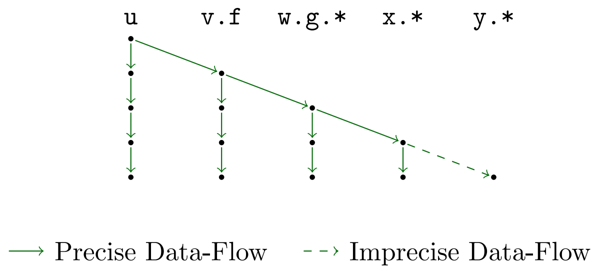


Figure 4.6: Imprecision of an k-limited access-path based analysis and $k = 1$.

post* inserts the self-loop edge with label f for state a@98.

Once post* saturates $\mathcal{A}_{\mathbb{F}}$, it encodes all sequences of possible access paths. When required, these can be extracted from $\mathcal{A}_{\mathbb{F}}$ in the form of a regular expression. Assume we would like to know how a@95 is accessible in line 99 from variable b, i.e., from the node b@99. All path(s) from this node to the accepting state are covered by the regular expression (f)+. The data may be stored in any access path with base variable b and arbitrarily many field accesses f.

Example 7 discusses the clear benefit of $\mathcal{P}_{\mathbb{F}}$ over the access-path model as $\mathcal{A}_{\mathbb{F}}$ easily represents an infinite number of access paths concisely. Apart from the concise representation, $\mathcal{P}_{\mathbb{F}}$ is also more precise than k-limiting which computes imprecise results through over-approximation.

Example 8. Figure 4.6 depicts a method foo() with two field-store and two field-load statements. Assume we use a data-flow analysis with an access-path based data-flow domain. The analysis uses k-limiting with a limit $k = 1$, i.e., an access path with only a single field is allowed. As soon as an access path with more than one field is constructed, it is over-approximated. In the example, we assume to track the data-flow stored in u at the beginning of method foo(), i.e., the abstract object $o_{\texttt{u@102}}$. Object $o_{\texttt{u@102}}$ is stored within field f of v, and the analysis generates the access path v.f at line 103. The next statement stores v in w.g. After execution of the statement, the data is accessible via w.g.f. However, this access path is of length 2 and a k-limit of 1 requires an over-approximation of this data-flow fact. Instead of maintaining the precise fact w.g.f, the access path w.g.* is propagated. The latter access path is less precise as * symbolically represents any field and not only f.

In line 105, method foo() loads w.g and stores it in x. The analysis precisely generates the access path x.*. The next statement in line 105 loads field g of x. The propagated access path x.* matches x.g, and the data-flow continues with the access path y.*. In Figure 4.6, this data-flow is highlighted as imprecise, because, at runtime, the traced abstract object stored in u is *not* transferred to y, because the field load of g does not match the last store of field f.

As a consequence, an access-path based analysis with a k-limit of $k = 1$ generates an imprecise data-flow on this example. While the wildcard is required to transform the infinite sized domain of access paths into a finite domain, the automaton representation $\mathcal{A}_{\mathbb{F}}$ does not require such an approximation, because

$\mathcal{A}_{\mathbb{F}}$ encodes an infinite number of paths, each of which correspond to one access path.

For the code of Figure 4.6, a limit of $k = 2$ suffices for a k-limited analysis to be precise. However it is straightforward to change the code snippet such that the limit of $k = 2$ does not suffice either (e.g., by adding an additional field-store statement `u.h = w`). Despite that such code looks synthetic, successive field load and field-store statements are realistic when objects are stored and loaded via *getter* and *setter* methods.

4.3 Call-Pushdown System

The rules of $\mathcal{P}_{\mathbb{F}}$ are defined context-insensitively. At return statements, the rules map data-flows to all call sites of the returning method and not just the call site that the data-flow actually entered the method through. Opposed to $\mathcal{P}_{\mathbb{F}}$, the pushdown system we discuss in this section, the *call-PDS*, models context-sensitivity. We have already sketched the pushdown system within Example 3. For completeness, we provide a full definition within this section.

Definition 7. *The* call-PDS *is the pushdown system* $\mathcal{P}_{\mathbb{S}} = (\mathbb{V}, \mathbb{S}, \Delta_{\mathbb{S}})$. *The pushdown systems control location are local variables of the program, and the stack elements are the statements of the program. We call the* $\mathcal{P}$*-automaton that* post* *saturates based on* $\mathcal{P}_{\mathbb{S}}$ *the* call automaton *and denote the automaton as* $\mathcal{A}_{\mathbb{S}}$.

We subdivide the rules set $\Delta_{\mathbb{S}}$ into the normal ($\Delta_{\mathbb{S}}^{normal}$), push ($\Delta_{\mathbb{S}}^{push}$), and pop ($\Delta_{\mathbb{S}}^{pop}$) rule sets and describe their construction based on the program semantics.

4.3.1 Normal Rules

The normal rules encode the intra-procedural data-flow of the analysis. We describe the normal rules of $\mathcal{P}_{\mathbb{S}}$ in terms of their flow functions. A flow function f takes as input variable $x \in \mathbb{V}$ and a statement $s \in \mathbb{S}$, and returns a set $D \subseteq \mathbb{V}$ of variables.

When the automaton $\mathcal{A}_{\mathbb{S}}$ accepts a configuration $\langle\!\langle x, s \rangle\!\rangle$, and t is an intraprocedural control-flow successor s, the flow function f is applied with arguments x and t. For any element y of the result set, i.e., $y \in f(x, t)$, the $\mathcal{P}_{\mathbb{S}}$ lists a rule of the form $\langle\!\langle x, s \rangle\!\rangle \to \langle\!\langle y, t \rangle\!\rangle$.

Table 4.3 describes the normal-flow functions for the relevant statements. The first row describes statements that kill the respective data-flows. Any assignment statement such that the destination of the assignment is a local variable (allocation sites, call sites, local-assign statements, or field-load statements) *kills* the local variable. The statement assigns a new value to the local variable and the analysis flow functions *strongly updates* the variable. Such strong update is possible only as the analysis models a flow-sensitive analysis.

The remaining rows of Table 4.3 assume the source of the assign statement matches the incoming data-flow abstraction. For example, the second row models

Table 4.3: Normal-Flow Function for the call-PDS.

Statement ($\mathbb{S}$)	**In** ($\mathbb{V}$)	**Out** ($\wp(\mathbb{V})$)
$x \leftarrow *$	x	$\varnothing$
$x \leftarrow y$	y	$\{x, y\}$
$x.f \leftarrow y$	y	$\{x, y\}$
$x \leftarrow y.f$	y	$\{x, y\}$

the flow of the data-flow abstraction representing y at an assign statement of form $x \leftarrow y$. The analysis continues the data-flow with x and y. Therefore, the normal flow function for a statement $x \leftarrow y$, and input variable y produces the out set $\{x, y\}$.

We want to highlight the semantics of $\mathcal{P}_{\mathbb{S}}$ for the field-store and field-load statements. Opposed to $\mathcal{P}_{\mathbb{F}}$, $\mathcal{P}_{\mathbb{S}}$ ignores the fields of the field stores and field loads, and data-flow continues to and from the base variable of the field store and load. Assume a field store, $x.f \leftarrow y$ and a data-flow to y that holds before the statement. The out set is $\{x, y\}$, the same set as for a normal assign statement $x \leftarrow y$, which means that the field f is not represented within the rules of $\mathcal{P}_{\mathbb{S}}$. The same holds for a field-load statement, $x \leftarrow y.f$, if the analysis encounters a data-flow fact y prior to the statement, it continues with x, i.e., the out set is also $\{x, y\}$. A check if the field load is actually feasible is ignored. In other words, $\mathcal{P}_{\mathbb{S}}$ is field-insensitive.

We want to pinpoint the reader to a difference between $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$. Both systems are flow-sensitive but each system encodes flow-sensitivity in its own way. $\mathcal{P}_{\mathbb{S}}$ uses the stack symbols explicitly to encode control-flow. Opposed to that, $\mathcal{P}_{\mathbb{F}}$ makes control-flow explicit in the control locations and not in the stack.

4.3.2 Push Rules

The push rules of $\mathcal{P}_{\mathbb{S}}$ model inter-procedural data-flows from a call site to the callee's start points. Assume the automaton $\mathcal{A}_{\mathbb{S}}$ to accept a configuration $\langle\!\langle p, s \rangle\!\rangle$, where a successor t of s is a call site $m(p)$, i.e., with parameter p. Further let e be the first statement of the callee m. Then, $\mathcal{P}_{\mathbb{S}}$ has a push rule of the form

$$\langle\!\langle p, s \rangle\!\rangle \to \langle\!\langle q, e \cdot c \rangle\!\rangle \in \Delta_{\mathbb{S}}^{push}$$

where q is the formal parameter of the callee. The push rule replaces the stack element s by e and s (in this order) within the stack. After the push rule is applied, the analysis first continues along any control-flow successor of e, because the stack element e is the top element of the stack. When the return statement of the callee m is reached, the top element of the stack is popped and the data-flow continues at after the call site c that has been pushed on the stack when the data-flow enters the callee.

4.3.3 Pop Rules

The pop rules of $\mathcal{P}_{\mathbb{S}}$ are the inverse of the push rules and map data-flow information from return statements of callees back to their corresponding call sites. Assume we have a return statement r : **return** x of some callee method m with formal parameter q and a call site $y \leftarrow m(p)$ that invokes m. Then the following two rules:

$$\langle\!\langle q, r\rangle\!\rangle \to \langle\!\langle p, \epsilon\rangle\!\rangle \in \Delta_{\mathbb{S}}^{pop}$$
$$\langle\!\langle x, r\rangle\!\rangle \to \langle\!\langle y, \epsilon\rangle\!\rangle \in \Delta_{\mathbb{S}}^{pop}$$

are contained in the rule set of $\mathcal{P}_{\mathbb{S}}$. The first rule maps the formal parameter back to the argument variable p at the call site, while the second rule maps the returned value x to the variable y, the returned variable is assigned to at the call site. The pop rules replace the current stack symbol r by an ϵ and, when applied by post*, removes the element r from the stack.

4.4 Synchronizing Call and Field-PDS

Section 4.2 and Section 4.3 discussed the two pushdown systems $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ individually. However, while $\mathcal{P}_{\mathbb{S}}$ is defined field-*in*sensitively, $\mathcal{P}_{\mathbb{F}}$ is context-*in*sensitive and each system retains a precision advantage over the other one.

In this section, we address the question of how to construct *one* analysis that combines the precision benefits of both pushdown systems. Both analyses encode their results in their respective field and call automata ($\mathcal{A}_{\mathbb{F}}$ and $\mathcal{A}_{\mathbb{S}}$). The key idea for a context-, flow-, and field-sensitive analysis is to *synchronize* these automata. Intuitively, a configuration of the more precise analysis is accepted, only if the field automaton *and* the call automaton accept the configuration.

Definition 8. *For the call-PDS $\mathcal{P}_{\mathbb{S}} = (\mathbb{V}, \mathbb{S}, \Delta_{\mathbb{S}})$ and the field-PDS $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$, the* synchronized pushdown systems *are the quintuple* SPDS = $(\mathbb{V}, \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}}, \Delta_{\mathbb{S}})$. *A configuration of SPDS extends from the configuration of each system: A* synchronized configuration *is a triple $(v, s, f) \in \mathbb{V} \times \mathbb{S}^{+} \times \mathbb{F}^{*}$, which we denote as $\langle\!\langle v.f_1\cdot\ldots\cdot f_m @ s_0^{s_1\cdot\ldots\cdot s_n}\rangle\!\rangle$ where $s = s_0\cdot s_1\cdot\ldots\cdot s_n$ and $f = f_1\cdot\ldots f_m$. For synchronized pushdown systems we define the set of all reachable synchronized configurations from a start configuration $c = \langle\!\langle v.f_1\cdot\ldots\cdot f_m @ s_0^{s_1\cdot\ldots\cdot s_n}\rangle\!\rangle$ to be*

$$post_{\mathbb{SF}}(c) = \{\langle\!\langle w.g@t_0^{t_1\cdot\ldots\cdot t_n}\rangle\!\rangle \mid \langle\!\langle w@t_0, g\rangle\!\rangle \in post_{\mathbb{F}}^{*}(\langle\!\langle v@s_0, f\rangle\!\rangle) \wedge \langle\!\langle w, t\rangle\!\rangle \in post_{\mathbb{S}}^{*}(\langle\!\langle v, s\rangle\!\rangle)\}. \tag{4.1}$$

Hence, a synchronized configuration c is accepted if $\langle\!\langle v, s_0\cdot\ldots\cdot s_n\rangle\!\rangle \in \mathcal{A}_{\mathbb{S}}$ and $\langle\!\langle v@s_0, f_1\cdot\ldots\cdot f_m\rangle\!\rangle \in \mathcal{A}_{\mathbb{F}}$ and $post_{\mathbb{SF}}(c)$ can be represented by the automaton pair $(\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$, which we refer to as $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}}$.

```
108 bar(u, v){
109   v.h = u;          114 foo(p){
110   w = foo(v);       115   q.g = p;
111   x = w.g;          116   return q;
112   y = x.f;          117 }
113 }
```

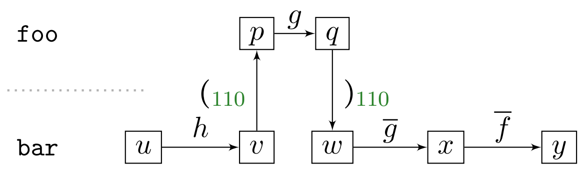


Figure 4.7: A code snippet and a labeled graph representation of the code.

Example 9. Figure 4.7 shows a code snippet where data flows inter-procedurally and is stored and loaded into a field of an object. Below the snippet, we depict a graph representation of the code to help illustrate the data-flow throughout the code. The nodes of the graph represent program variables; horizontal edges between them correspond to field push and pop rules. The edges are labeled with the names of the fields. A field label with a line on top, e.g., $\overline{f}$, means the field f is loaded (a pop rule). For field-stores, the field is not overlined (push rule). The vertical edges resemble push and pop rules in $\mathcal{P}_{\mathbb{S}}$. We label these edges with opening and closing parentheses. An opening parenthesis "(" matches a push rule, the closing parenthesis ")" corresponds to a pop rule. The line number in the subscript refers to the call site that is pushed to the stack.

Assume a context-, flow-, and field-sensitive data-flow analysis to track the object pointed to by u@108. We refer to this abstract object by $o_{\texttt{u}@108}$. Additionally, assume we want to infer whether $o_{\texttt{u}@108}$ is accessible by y@112. The actual data-flow is best understood within the graph representation which contains a path from u to y. The labels along this path concatenate to form the sequence (or word) $h \cdot (_{110} \cdot g \cdot)_{110} \cdot \overline{g} \cdot \overline{f}$. The parentheses $(_{110}$ and $)_{110}$ are properly matched. This means the path is realizable in terms of context-sensitivity, i.e., a valid execution path. However, the path is not feasible in terms of field accesses. The field store g is properly matched against the load $\overline{g}$, but the field store h does not match the load of $\overline{f}$. In other words, there is no data-flow connection between u@108 and y@112, which means the latter does not point to $o_{\texttt{u}@109}$.

In the following, we discuss that synchronized pushdown systems prove the missing data-flow connection as they computes $\langle\!\langle y.\epsilon@112^{\epsilon}\rangle\!\rangle \notin post_{\mathbb{SF}}(\langle\!\langle u.\epsilon@108^{\epsilon}\rangle\!\rangle)$. For the data-flow analysis, we first construct $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}} = (\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$ such that the automaton accepts the configuration $\langle\!\langle \texttt{u}.\epsilon@108^{\epsilon}\rangle\!\rangle$. Therefore, $\langle\!\langle \texttt{u}, 108\rangle\!\rangle \in \mathcal{A}_{\mathbb{S}}$ and $\langle\!\langle \texttt{u}@108, \epsilon\rangle\!\rangle \in \mathcal{A}_{\mathbb{F}}$.

We then apply post* to both automata and compute the set $post_{\mathbb{SF}}(\langle\!\langle u.\epsilon@108^{\epsilon}\rangle\!\rangle)$. The set is represented by the two automata depicted in Figure 4.8. $\mathcal{A}_{\mathbb{S}}$ accepts the configuration $\langle\!\langle \texttt{x}, 111\rangle\!\rangle$ and $\mathcal{A}_{\mathbb{F}}$ accepts $\langle\!\langle \texttt{x}@111, h \cdot \epsilon\rangle\!\rangle$. Therefore, the synchronized configuration $\langle\!\langle \texttt{x}.h \cdot \epsilon@111^{\epsilon}\rangle\!\rangle$ is accepted. Since $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}}$ is constructed based on

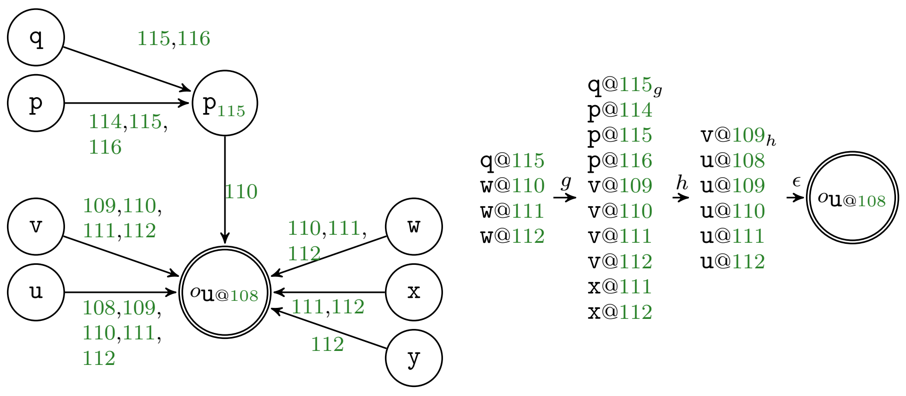


Figure 4.8: The post* saturated $\mathcal{A}_{\mathbb{S}}$ (left) and $\mathcal{A}_{\mathbb{F}}$ (right) for the example in Figure 4.7.

the initial synchronized configuration $\langle\langle \mathtt{u}.\epsilon@108^{\epsilon}\rangle\rangle$, accessing field `h` of `x` (line 111) retrieves the same object as stored in variable `u` at statement 108. In other words, object $o_{\mathtt{u@108}}$ is accessible via access path `x.h` after statement 111. The next line, statement 112, loads the field `f` of variable `x`. Due to the field-load statement, $\mathcal{P}_{\mathbb{F}}$ lists the pop rule $\langle\langle \mathtt{x@111}, \mathtt{f}\rangle\rangle \to \langle\langle \mathtt{y@112}, \epsilon\rangle\rangle$. $\mathcal{A}_{\mathbb{F}}$ does not contain a transition out of state `x@111` with label `f`. Therefore, the pop rule cannot be applied, consequently `y@112` does not become a state of $\mathcal{A}_{\mathbb{F}}$.

Despite $\mathcal{A}_{\mathbb{S}}$ accepting the configuration $\langle\langle \mathtt{y}, 112\rangle\rangle$, $\langle\langle \mathtt{y@112}, \epsilon\rangle\rangle$ is not an accepted configuration for $\mathcal{A}_{\mathbb{F}}$. In turn, $\langle\langle y.\epsilon@112^{\epsilon}\rangle\rangle \notin \mathit{post}_{\mathbb{SF}}(\langle\langle u.\epsilon@108^{\epsilon}\rangle\rangle)$.

4.4.1 Undecidability and Required Approximations

The “synchronized” combination of the two automata, as we present it above, raises the question whether a tighter integration of both automata would not be possible and beneficial. Unfortunately, as Reps [73] shows, context-sensitive data-dependence analysis is generally undecidable: it can be mapped to a reachability problem on a graph with two interleaved context-free languages (CFL), which means a word formed along *one* path in the graph must form a correct word in *both* CFLs. In Example 9, we see that a context- and field-sensitive data-flow analysis is equivalent to a reachability problem of two CFLs: one language ($L_{\mathbb{F}}$) for field stores and loads (e.g., f and $\overline{f}$), and a second one ($L_{\mathbb{S}}$) matching call and return flows (e.g., $(_{110}$ and $)_{110}$).

Computing the set $\mathit{post}_{\mathbb{SF}}$ for a synchronized pushdown system is decidable, because the set is merely the conjunction of the sets $\mathit{post}^{*}_{\mathbb{S}}$ and $\mathit{post}^{*}_{\mathbb{F}}$. The essential difference is that a SPDS computes both sets along potentially *different* control-flow paths. Interestingly, we find that this approximation leads to imprecision only under unusual circumstances, which makes SPDS precise in practice. The following example demonstrates that potential precision loss.

Example 10. Figure 4.9 extends the code snippet provided in Figure 4.7 with two new methods. The method `baz()`, similar to `bar()`, calls `foo()` after storing a field, and the method `qux()` that calls both methods `baz()` and `bar()` (lines 119

```
118 qux(a, b, c){      122 baz(r, s){
119   bar(a, b);       123   s.f = r;
120   baz(a, c);       124   t = foo(s);
121 }                  125 }
```

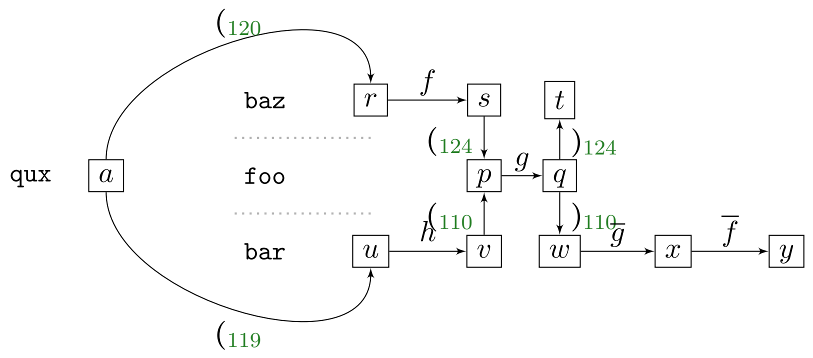


Figure 4.9: Code snippet that extends Example 9 from Figure 4.9 and the updated graph representation.

and 120). The first parameter of both calls from `qux()` to `baz()` and `bar()` is the same variable `a`. Below the code, we also show the complete and updated graphical representation for the code of Figure 4.9. The earlier representation is extended with the variable nodes for `baz()` and `qux()` and the respective edges.

Assume we want to know if there is a data-flow path from a to y. The graph contains two paths between the nodes. One path that contains node u generates the word $w_1 = (_{119}{\cdot}h \cdot (_{110}{\cdot}g{\cdot})_{110} \cdot \overline{g} \cdot \overline{f}$. The sequence of labels along the other path forms $w_2 = (_{120}{\cdot}f \cdot (_{124}{\cdot}g{\cdot})_{110} \cdot \overline{g} \cdot \overline{f}$. In combination, both paths introduce imprecision into the analysis, because they make the analysis report a flowing to y, despite it being impossible at runtime.

Along the path of the word w_1, and since more opening parentheses are acceptable, the call parenthesis are properly matched, $(_{119}(_{110})_{110}$. However, the field stores and loads are not properly matched, $h \cdot g \cdot \overline{g} \cdot \overline{f}$. For the second path with the word w_2, the situation is the other way around. The field stores and loads are properly matched, $h \cdot g \cdot \overline{g} \cdot \overline{h}$, while the call parentheses of the word do not match, $(_{120}{\cdot}(_{124}{\cdot})_{110}$.

To conclude, the set $post^*_{\mathbb{S}}$ contains all configurations m reachable from configuration n such that the word on a path p_1 between n and m forms a word in $L_{\mathbb{S}}$. Opposed to this, $post^*_{\mathbb{F}}$ contains all configurations m such that a path p_2 from n to m forms a word in $L_{\mathbb{F}}$. However, the path p_1 and p_2 may differ. As we showed, it is possible to construct examples where synchronized pushdown systems do not precisely solve the data-flow problem. Yet, our empirical evaluation reveals *no* practical occurrence of this over-approximation (see Section 8.1.2). Therefore, we hypothesis that: An improperly matched call site does not induce a properly matched field access (and vice versa).

4.4.2 Worst-Case Complexity Analysis

We next discuss the worst-case complexity for the computation of $post_{\mathbb{SF}}(c)$ for SPDS, and compare it to a context-sensitive and flow-sensitive analysis that uses a k-limited access-path representation. For the comparison, we also encode the latter analysis as an analysis based on a single pushdown system. For a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$, algorithm post* constructs the $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ with a complexity of $\mathcal{O}(|P|\ |\Delta|\ (|Q| + |\Delta|)\ +\ |P|\ |\delta|)$ for both time and space [21].

Synchronized Pushdown Systems A SPDS computes two independent *post** sets for $\mathcal{P}_{\mathbb{F}}$ and $\mathcal{P}_{\mathbb{S}}$, hence the worst-case complexity is the maximum of any of the two post* computations. The control locations of $\mathcal{P}_{\mathbb{S}}$ are the program variables involved in queried data flows, an upper bound of which is $|P| = |\mathbb{V}|$. The out-set of a data-flow at an assignment statement has at most two[4] variables. For every other statement, the out-set contains one or zero elements. There is a data-flow for every edge (at most $|\mathbb{S}|^2$) in the inter-procedural control-flow graph, and the number of rules can be approximated by $|\Delta| = 2|\mathbb{V}||\mathbb{S}|^2$. $\mathcal{A}_{\mathbb{S}}$ has one state per variable and an intermediate state for each variable that flows at a call site to a callee, hence the number of $\mathcal{A}_{\mathbb{S}}$ states is $|Q| = |\mathbb{V}| + |\mathbb{V}||\mathbb{S}| \le 2|\mathbb{V}||\mathbb{S}|$. Each transition of $\mathcal{A}_{\mathbb{S}}$ is labeled by a statement, and $\mathcal{A}_{\mathbb{S}}$ has at most $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^3$ edges and computing post* for $\mathcal{P}_{\mathbb{S}}$ has a worst-case complexity of

$$\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^2(|\mathbb{V}||\mathbb{S}| + |\mathbb{V}||\mathbb{S}|^2) + |\mathbb{V}|^3|\mathbb{S}|^3) = \mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^4).$$

The control locations of $\mathcal{P}_{\mathbb{F}}$ are pairs of variables and statements, and we approximate the control locations by $|P| = |\mathbb{V}||\mathbb{S}|$. In practice, the variable of a control location of $\mathcal{P}_{\mathbb{F}}$ must be local to the method of the statement of the control location, which greatly reduces the size of the set P. The number of rules of $\mathcal{P}_{\mathbb{F}}$ is bounded by $|\Delta| = 2|\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|$, because at an assignment statement the analysis applies at most two rules for every field. In the worst case, for each variable at each statement, a push rule creates an intermediate state which bounds the states of $\mathcal{A}_{\mathbb{F}}$ by $|Q| = |\mathbb{V}||\mathbb{S}||\mathbb{F}|$. The size of the transitions set of $\mathcal{A}_{\mathbb{F}}$ can be approximated by $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^2|\mathbb{F}|^3$, because, between each of the states, there can be a transition labeled by a field. From these approximations, the complexity of the computation of post* for $\mathcal{P}_{\mathbb{F}}$ evaluates to $\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^3|\mathbb{F}|(|\mathbb{V}||\mathbb{S}||\mathbb{F}| + |\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|) + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^3)$ which reduces to

$$\mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^5|\mathbb{F}|^2 + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^3). \tag{4.2}$$

This complexity dominates the complexity for $\mathcal{P}_{\mathbb{S}}$, therefore, the worst-case complexity for SPDS is the same as of $\mathcal{P}_{\mathbb{F}}$.

[4]At a field-store statement $x.f \leftarrow y$, we assume y to flow to x only but not to any alias of x. We discuss aliasing in Chapter 5.

Access Paths with k-limiting For comparison, we assume the k-limited access-path based analysis (AP^k) to be encoded as a pushdown system similarly to $\mathcal{P}_{\mathbb{S}}$, i.e., call sites correspond to push rules and return statements to pop rules of the system. Instead of using variables ($\mathbb{V}$) as control locations, the control locations for the k-limited analysis are access paths, i.e., a local variable followed by a k-limited sequence of fields. Hence, $|P| = |\mathbb{V}||\mathbb{F}|^k$. The size of the rule set is at most $|\Delta| = 2|\mathbb{V}||\mathbb{F}|^k|\mathbb{S}|^2$ because, for every edge of the control flow graph, one access path is mapped to at most two access paths.[5] The pushdown system's stack alphabet is $\mathbb{S}$, which limits the size of the state set of the $\mathcal{P}$-automaton to $|Q| = 2|\mathbb{V}||\mathbb{F}|^k|\mathbb{S}|$, and the size of the transitions set to $|\delta| = 4|\mathbb{V}|^2|\mathbb{S}|^3|\mathbb{F}|^{2k}$. For AP^k it results a worst-case complexity of $\mathcal{O}(|\mathbb{V}|^2|\mathbb{S}|^2|\mathbb{F}|^{2k}(|\mathbb{V}||\mathbb{S}||\mathbb{F}|^k + |\mathbb{V}||\mathbb{S}|^2|\mathbb{F}|^k) + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^{3k})$ which simplifies to

$$\mathcal{O}(|\mathbb{V}|^3|\mathbb{S}|^4|\mathbb{F}|^{3k} + |\mathbb{V}|^3|\mathbb{S}|^3|\mathbb{F}|^{3k}). \tag{4.3}$$

We now compare the analysis complexity of AP^k (4.3) to the complexity of SPDS (4.2). The complexities differ in two parts. First, AP^k multiplies the exponent of all $|\mathbb{F}|$ factors by the value k. Second, SPDS increases $|\mathbb{S}|^4$ to $|\mathbb{S}|^5$. The additional factor $|\mathbb{S}|$ is introduced by automaton $\mathcal{A}_{\mathbb{F}}$, as its states refer to statements in addition to variables.

It is expected that for some $k > 0$, SPDS is more performant than AP^k for data-flows that are assigned to many fields and at the same time reach few statements. Additionally, the complexity estimates show that the larger k, the more time and space AP^k requires.

We perform practical experiments comparing SPDS to AP^k in Chapter 8. The experiments showcase that, in practice, SPDS are almost as efficient as AP^k when $k = 1$, although SPDS delivers results as precise as AP^k with $k = \infty$.

4.5 Related Work

In collaboration with Lerch et al., we designed IFDS-APA [51, 53] to solve the same problem of having a context-, flow-, and field-sensitive analysis. This earlier formulation does not rely on pushdown systems. Instead, it takes a CFL-reachability approach to solve the problem. To become decidable, this formulation requires either the language of field stores and loads or the language of matching call and returns to be over-approximated by a regular language. This additional (and imprecision-introducing) computation step is not necessary in SPDS. Our implementation attempts to lift BOOMERANG [90] to IFDS-APA failed due to this complex over-approximation step which makes the implementation hard to realize. These difficulties motivate the design of SPDS, which thoroughly relies on existing well-established research.

Other than IFDS-APA, prior research on data-flow analyses that are context-, flow-, and field-sensitive is rare. Andromeda [100] and FlowDroid [5] are two precise taint analyses of these dimensions, and both use k-limiting. These analysis can benefit from replacing their access path representation by SPDS.

[5]Similar to SPDS, we also ignore aliasing here.

In recent work, Zhang et al. [106] introduce linear conjunctive language (LCL) reachability and show how the interleaved matching-parentheses problem of field-sensitive and context-sensitive data-flow analysis can be over-approximated by a LCL. Their work presents a new algorithm to solve LCL-reachability and base their work on trellis automata, instead, we show that we can formulate the problem in two pushdown systems and rely on existing algorithms and improvements [21, 50, 77]. Zhang et al. also base their approach on the hypothesis that we discuss in Section 4.4.1, and the two ideas are closely related, the main difference is that both approach the problem from two different perspectives (language-formulation and pushdown systems).

There are various approaches for encoding context-sensitive and field-sensitive (but mostly flow-insensitive) alias or points-to analyses as two CFL-reachability (or Dyck-reachabilty) problems [14, 91, 93, 102, 105]. In Chapter 5, we provide a detailed discussion of these approaches with respect to points-to analysis. However, all CFL-approaches share the same over-approximation. To guarantee decidability, the approaches approximate either the CFL for field stores/loads or the CFL for call/returns by a regular language.

One points-to analysis that we want to highlight here is the analysis by Li and Ogawa [57]. Their analysis builds on weighted pushdown systems but models both context- and field-sensitive in a *single* weighted pushdown system. As Reps proved [73], this formulation is equal to an undecidable data-dependence analysis and the authors require to explicitly address decidability.

In Section 4.1, we discuss related work on access graphs [29, 45, 90], which is a similar representation to $\mathcal{A}_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$. Access graphs allow a finite representation of the potentially infinite number of access paths. However, to be flow-sensitive, such approaches maintain access graphs per statement. Similarly, also *alias graphs* [43], a field abstraction proposed as an efficient data-flow model for must-aliasing access paths, must store information statement-wise. Using SPDS, the single automaton $\mathcal{A}_{\mathbb{F}}$ is sufficient to encode all field accesses at all statements.

5 Boomerang

Chapter 4 presented synchronized pushdown systems as a solution to efficient and precise context-, field- and flow sensitive data-flow analysis. In this chapter we show how multiple SPDS can be combined to solve pointer relations.

Points-to analysis computes memory locations for variables of reference type or *pointer variables*. A points-to analysis is a static analysis that computes potential objects a pointer variable may point to at runtime. During the program execution, a pointer variable points to exactly one object, however the object that is referenced may differ dependent on the program execution path. A static points-to analysis reason about all program execution and over-approximates the actual runtime object by a set of potential objects. The set is called the *points-to set* and abstracts the objects in the form of the allocation site statements of the object.

Information about potential allocations sites of a pointer variable is necessary for many static analyses. Points-to information is helpful for program refactoring [23] and program optimizations [16], to generate program call graphs [32, 55, 97] or during the propagation of taint [5, 100] or typestate information [26, 88].

Despite the fact that points-to analysis is a decades-long researched topic, every client (call graph construction, data race, taint or typestate analysis, etc.) has its own requirements for the supporting points-to analysis and, unfortunately, there is no one-fits-all-solution for pointer analysis [38]. While call-graph construction algorithms require the types of the allocations sites, data-race clients intersect two points-to sets to obtain alias information, and taint or typestate analyses that uses a storeless heap model (access path, access graphs, or SPDS) require all aliases to a given access path.

In this chapter, we instantiate multiple synchronized pushdown systems and present the design of a demand-driven pointer analysis crafted for taint and typestate analysis clients. We call this analysis Boomerang. To ease presentation, we first formulate a whole-program context-, field-, and flow-sensitive points-to analysis as a purely control-flow forward-directed analysis. Next, we show how Boomerang intertwines a forward-directed analysis with an additional backward-directed analysis to compute points-to sets on-demand.

We published Boomerang at the 2016 European Conference on Object-Oriented Programming (ECOOP) [90]. At the time, Boomerang used access graphs as heap model which, in this thesis, we replace by SPDS.

5.1 Non-Distributivity of Pointer Information

The computation of points-to relations is non-trivial as the flow functions for points-to analysis are non-distributive. For SPDS, the flow functions (i.e., push-

```
126 foo(){
127   u = new;
128   v = u;
129   x = new;
130   u.f = x;
131   y = v.f;
132 }
```

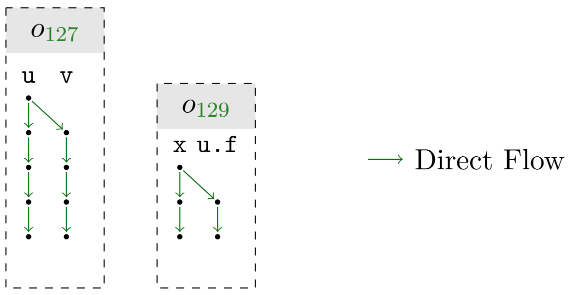


Figure 5.1: Points-to analysis is non-distributive.

down rules) are distributive over the merge operator, which is also why the SPDS as described in Chapter 4 models only direct data flows that do not take aliasing into account.

Example 11. Figure 5.1 shows example code for which a SPDS is not able to compute all points-to relations. SPDS is not able to reason that, at runtime, variables `x` and `y` point-to the same object.

The program code contains two allocation sites in line 127 and 129, and the figure depicts two SPDS, one for o_{127} and o_{129}. The data-flows for each objects are drawn as exploded supergraphs depicted in a box labeled by the corresponding object. Each data-flow is rooted at the allocation site of the object from which the exploded supergraph is constructed. From the exploded supergraph, *some* points-to relation can be extracted but not *all*. For example, the graph for object o_{127} contains nodes `u` and `v` at the statement in line 130. This means that the variables at the respective statement point to the object o_{127}. In other words, each variable of the node of the exploded super graph for one object is aliased to each other variable.

Yet, the extracted points-to and alias information is not complete when field stores are involved in the data-flow. Consider the second data-flow of o_{129}. The data-flows of this SPDS lists the access paths `x` and `u.f`. But at runtime, object o_{129} is also pointed to by `v.f` and `y` as `u` and `v` are aliased. SPDS do not compute a node for variable `y`, and `y` has an empty points-to set.

The root cause that the flow functions for a points-to analysis are non-distributive is field store statements. Consider the store statement `u.f = x` in line 130. The pushdown rules for $\mathcal{P}_\mathbb{F}$ at this statement list the push rule $\langle\!\langle \texttt{x}@129, *\rangle\!\rangle \to \langle\!\langle \texttt{u}@130, \texttt{f}\cdot *\rangle\!\rangle$ and the normal rule $\langle\!\langle \texttt{x}, 129\rangle\!\rangle \to \langle\!\langle \texttt{u}, 130\rangle\!\rangle$ for $\mathcal{P}_\mathbb{S}$. None of the rules capture the flow-to relation between `x` and `v.f`.

In practice, distributivity means that the result of applying the flow function only depends on the statement (`u.f = x`) and the incoming data-flow fact (`x` points-to the object o_{129}), but not on any other information. The information that `v` and `u` are aliases it not derivable from within the flow function.

We now provide a formal view on the consequence of the distributive propagation of SPDS. Let a be an allocation site statement in the program and let the pair $o_a := (\mathcal{A}^a_\mathbb{S}, \mathcal{A}^a_\mathbb{F})$ represent the set $post_{\mathbb{SF}}(\langle\!\langle v.\epsilon@a^\epsilon\rangle\!\rangle)$, where v is the variable allocated in a. Then it is $\forall x \in \mathbb{V}, f \in \mathbb{F}^*, t \in \mathbb{S}$ and $c \in \mathbb{S}^*$:

$$\langle\!\langle x.f@t^c \rangle\!\rangle \in o_a \Rightarrow x.f@t \text{ points-to } o_a \text{ under calling context } c. \tag{5.1}$$

The rules for $\mathcal{P}_\mathbb{S}$ and $\mathcal{P}_\mathbb{F}$ model the direct data-flow for each statement and capture the direct points-to relations. However, the reverse direction is *not* true:

$$x.f@t \text{ points-to } o_a \text{ under calling context } c \not\Rightarrow \langle\!\langle x.f@t^c \rangle\!\rangle \in o_a. \tag{5.2}$$

In Example 11, for instance, the SPDS for o_{129} contains the synchronized configuration $\langle\!\langle$`u.f`$@130^\epsilon\rangle\!\rangle$ which means, access path `u.f` after statement 131 points to the object o_{129}. However, $\langle\!\langle$`v.f`$@131^\epsilon\rangle\!\rangle$ points-to the same object but the SPDS for o_{129} misses this configuration.

The missing alias relation is due to flow-sensitivity. The data-flow is not detected if the alias relationship of the variables `v` and `u` is established prior to the field store statement. If one rewrites the code such that statement 128 follows statement 130, the data-flow for o_{129} correctly lists the access paths `v.f` and `y`.

Most of the existing points-to analyses [1,12,37,55,63,91,94,98,104] are flow-insensitive and do not consider the order of the control-flow, which means that the presented missing data-flow cannot occur. We say that a flow-sensitive data-flow analysis is *alias-sensitive* when the following holds:

Definition 9. *Assume a field store statement $x.f = a$ and a control-flow-succeeding field load statement $b = y.f$ such that x and y at both statements point to the same object, i.e., they* alias *with each other. Further on, this alias relationship is established at a control-flow-preceding statement of the field store. Then, a flow-sensitive and field-sensitive data-flow analysis is* alias-sensitive*, if the analysis establishes a data-flow connection between the stored variable a of the field store and the loaded variable b of the field load.*

Every context-, field-, and flow-sensitive analysis that is also alias-sensitive is a points-to analysis. As motivated in the example, a single SPDS does not produce alias-sensitive results.

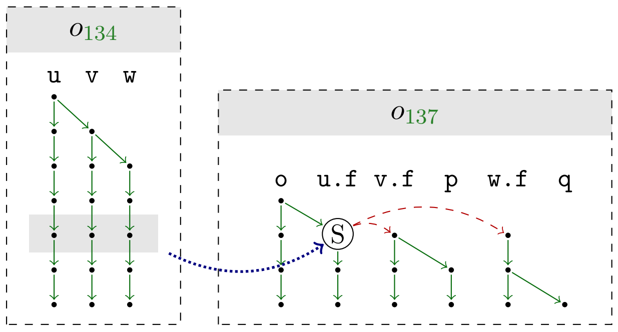
5.2 Forward-Directed Points of Aliasing

A *single* SPDS is not expressive enough to conduct points-to analysis, but points-to analysis is expressible by combining *multiple* SPDS. For the ease of explanation, we first discuss the idea based on a whole-program points-to analysis, which means the analysis is forward-directed only.

A whole-program points-to analysis computes complete points-to sets for all variables in the program. This requires forward propagation of data-flow from *any* allocation statement. The points-to analysis based on SPDS maintains a SPDS per allocation statement, i.e., per abstract object.

At particular statements in the program, the synchronized pushdown system must interact and exchange information. The interaction is required at what we call "*points of aliasing*". Data-flow propagated along field-store statements

```
133 foo(){
134   u = new;
135   v = u;
136   w = v;
137   o = new;
138   u.f = o;
139   p = v.f;
140   q = w.f;
141 }
```



⟶ Direct Flow - - › Indirect Flow Ⓢ Store Point of Aliasing

Figure 5.2: Indirect flows at a store point of aliasing.

and call sites generates a points of aliasing. Apart from the statement the point of aliasing is registered at, a point of aliasing lists two synchronized pushdown automata $o_a := (\mathcal{A}^a_{\mathbb{S}}, \mathcal{A}^a_{\mathbb{F}})$ and $o_b := (\mathcal{A}^b_{\mathbb{S}}, \mathcal{A}^b_{\mathbb{F}})$. At a point of aliasing transitions are copied over from $\mathcal{A}^a_{\mathbb{S}}$ to $\mathcal{A}^b_{\mathbb{S}}$ and from $\mathcal{A}^a_{\mathbb{F}}$ to $\mathcal{A}^b_{\mathbb{F}}$.

Once all information is copied over from one automaton to the other, post* is applied again to re-saturate the automaton. In terms of the data-flow analysis, this means new data-flow paths become reachable. These new paths may involve new points of aliasing. The described iterative process is repeated until no new point of aliasing is unveiled and all automata are fully post*-saturated.

5.2.1 Field-Store Point of Aliasing

A field-store point of aliasing is registered at a field store statement of the form $s : x.f = y$. The whole program points-to analysis starts propagating a SPDS for every allocation site a. Therefore, assume the SPDS for the abstract object o_a such that configuration $\langle\!\langle y.g@s^c \rangle\!\rangle \in o_a$. This means, the abstract object allocated at statement a, is accessible via the access path $y.g$ at statement s under some context $c \in \mathbb{S}^*$. Then the points-to analysis registers a point of aliasing for the field store statement s. At the field-store statement, the flow functions (see Table 4.1 and Table 4.3) propagate $\langle\!\langle y.g@s^c \rangle\!\rangle$ to $\langle\!\langle x.f.g@t^c \rangle\!\rangle$ for any successor t of s. However, after statement s, object o_a is also accessible *indirectly* via any alias of x (see Example 11).

Because the whole-program points-to analysis starts at any allocation sites, the points-to analysis also constructs a SPDS for the object o_b such that x points-to o_b, i.e., $\langle\!\langle x.\epsilon@s^d \rangle\!\rangle \in o_b$. The field and call automaton for o_b encode aliasing access paths other than $x.\epsilon$ that also reference the object o_b at statement s. From the automata of o_b, the analysis copies a subset of transitions (*indirect flow edges*) to the automata for o_a such that o_a is also accessible via the indirect aliases.

Example 12. Figure 12 depicts points of aliasing and the generation of indirect flow edges. The code snippet, allocates two objects in line 134 and line 137, respectively. Aside of the program code, two exploded supergraphs for the respective allocation sites are drawn.

The object o_{137} generates a field-store point of aliasing in line 138 as highlighted by the exploded super graph node marked as Ⓢ. The object is stored within field `f` of a second object that is accessible via `u`.

The data-flow propagations for object o_{134} generate a data-flow fact `u`@138, i.e., the analysis computes that `u` at the field-store statement points to o_{134}. Additionally, the exploded super graph also contains the two nodes `v`@138 and `w`@138. All of the variables originate from o_{134} and the variables alias as they point to the same object. The aliasing variables are passed over to the SPDS of o_{137} as highlighted by the dotted blue arrow from the box of o_{134} to o_{137}. Based on information that `v` and `w` are aliases of `u`, the point of aliasing derives that object o_{137} is indirectly accessible via `v.f` and `w.f` after the field-store statement. The indirect data-flow edges are added to the exploded supergraph, and the analysis continues the re-saturation (or fixed point computation). Due to the indirect flow edge, the data-flow for object o_{137} generates the two nodes `p`@139 and `q`@140, and the analysis correctly identifies variables `p` and `q` at the respective statements as pointers to o_{137}.

In the example, the analysis adds indirect edges based on aliases that are plain local variables (`v` and `w`), i.e., access paths with an empty sequence of fields. It is also possible that the alias relationship is not based on local variables, but that an alias is an access path with a non-empty sequence of fields.

Example 13. Figure 5.3 shows a code snippet where an object is nested within fields of an other object with a depth of 2. At runtime, the variable `q`@149 points to the object allocated in line 146. For a points-to analysis, the nesting in fields is more difficult to analyze soundly.

After statement 145, access path `u.f` and variable `w` are pointers to the same object (o_{144}). At statement 147, object o_{146}, referenced by variable `o`, is stored in `w.g`. Because `w` and `u.f` are aliased, also field `g` of `u.f` is updated. After the field-store statement, `u.f.g` points to o_{146}.

The content of the access path `u.f.g` is loaded within the following two subsequent field load statements. The first statement in line 148 dereferences field `f` of `u` and stores the content to `p`. The subsequent field load then loads field `g` of `p` and stores the field's content into variable `q`. Therefore, `q` points-to o_{146} at runtime.

For the static points-to analysis to model the correct runtime behaviour of the program, the field store point of aliasing in line 147 does not only add indirect flows for variables, but appends the stored field, `g`, to all aliasing access paths of the base variable, e.g., `g` is appended to the alias `u.f`.

Technical View

The last two examples motivate that the analysis has to extract all aliasing access paths at a statement from the results. We now want to provide a more technical view on how the analysis extracts this information.

```
142 foo(){
143   u = new;
144   w = new;
145   u.f = w;
146   o = new;
147   w.g = o;
148   p = u.f;
149   q = p.g;
150 }
```

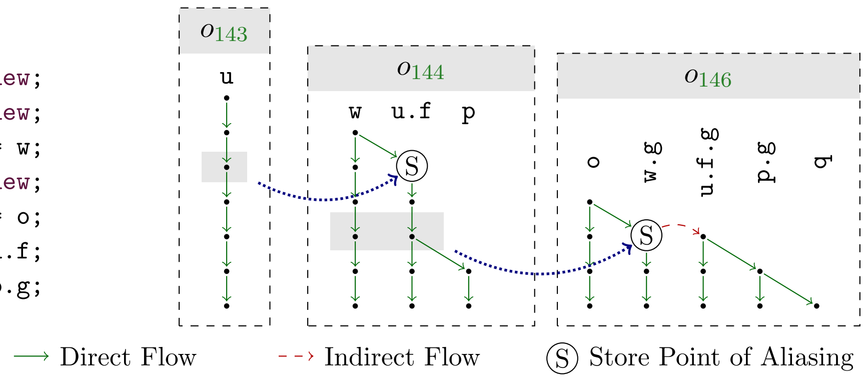


Figure 5.3: Access path can point to the same object (here `w` and `u.f`). The point of aliasing in line 147 must indirectly generate the access path `u.f.g` when data-flow is stored in `w.g`.

The responsibility of the points of aliasing is to add transitions to $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{F}}$ of the respective SPDS of an object such that the implication in equation (5.2) also holds.

Definition 10. *The* set of aliases of an object allocation a at statement s *is the set* $aliases_a(s) := \{\langle\!\langle v.f@s^c \rangle\!\rangle \in o_a \mid v \in \mathbb{V}, f \in \mathbb{F}^*, c \in \mathbb{S}^*\}$*. By definition, all access paths* $v.f$ *in the set point to allocation site* a*; any two access paths of the set are* aliased*.*

The latter set is a subset of the set $post_{\mathbb{SF}}$ as defined in (4.1) and representable as sub-automata of the $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{F}}$.

Definition 11. *A* field-store point of aliasing *is a triple* (s, o_a, o_b) *of a field store statement* $s : x.f = y$ *and two objects* o_a *and* o_b*. Additionally it holds, that* $\langle\!\langle y.g@s^c \rangle\!\rangle \in o_a$ *for some* $g \in \mathbb{F}^*$ *and* $\langle\!\langle x.\epsilon@s^d \rangle\!\rangle \in o_b$ *(for some arbitrary call stacks* $c, d \in \mathbb{S}^*$*).*

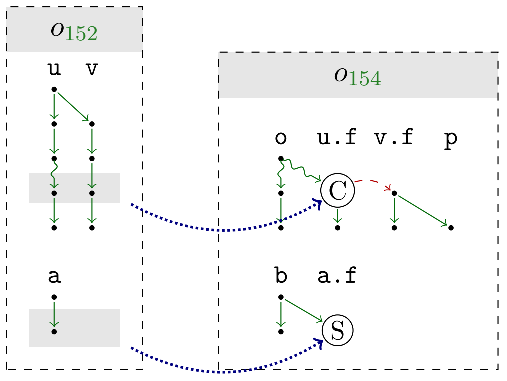
The term $\langle\!\langle x.\epsilon@s^d \rangle\!\rangle \in o_b$ encodes the points-to relation of the base variable of the field store. At the field store statement s, access path $x.\epsilon$ points to allocation site b (under call stack d).

At the point of aliasing the object o_a flows to field f of o_b. At any control-flow successor t of s, object a can be loaded via f not only from variable x, the direct flow, but also from any access path that aliases to $x.\epsilon$. The latter access paths are collected in the set $aliases_b(s)$ and the analysis adds the *indirect* transitions into the automaton of o_a such that:

$$\langle\!\langle v.f \cdot g@t^c \rangle\!\rangle \in o_a, \text{ where } \langle\!\langle v.g@s^c \rangle\!\rangle \in aliases_b(s)$$

holds. The field f is concatenated to the sequence of fields g and yields $f \cdot g$. For accessing o_a, f must be loaded first (and the fields remaining in the field sequence g).

```
151 foo(){
152   u = new;
153   v = u;
154   o = new;
155   setF(u,o);
156   p = v.f;
157 }
158 setF(a,b){
159   a.f = b;
160 }
```



⟶ Direct Flow ⇝ Summarized Flow ⇢ Indirect Flow Ⓒ Call Ⓢ Store

Figure 5.4: Indirect flows at a store point of aliasing within the caller context.

5.2.2 Call-Site Point of Aliasing

At a field-store statement, a point of aliasing injects all aliasing access paths as indirect flows into the analysis. By definition, the set $aliases_a(s)$ contains only access paths for which the base variable is a local variable of the method of the field-store statement s. As a consequence only all aliasing access paths *within the scope of the method* are added at a field-store point of aliasing. When the flow continues in a different analysis scope, i.e., it returns to a call site from a callee method, aliases may be missing if the flow at the call site is not explicitly handled. *Call-site points of aliasing* are similar to field-store point of aliasing and handle necessary re-propagations of the missing aliases.

Example 14. Alias-sensitivity also requires alias handling at data-flows changing their method scope as demonstrate in the code in Figure 5.4. At runtime, `p` at statement 156 holds a reference to object o_{154}, i.e., variable `o` and `p` are aliased.
In line 159, method `setF` stores the second parameter, variable `b`, within field `f` of the first parameter, named `a`. This generates a field-store point of aliasing. In the example, in the exploded supergraph for o_{154}, the point of aliasing is the node `a.f`@159 marked as Ⓢ. This point of aliasing does *not* generate any indirect flows; at the statement in line 159, variable `a` is the only visible reference to object o_{152}.
The data-flow fact `a.f` has a local variable that is a parameter to `setF()` and returns as `u.f` to the call site in method `foo()` (line 155). The control-flow succeeding statement of the call site loads field `f` of variable `v` which is aliased to `u`. The analysis misses this data-flow, the access path `v.f` is never generated, because the alias relationship between `u` and `v` is established before the call site in line 155, and at the field-store point of aliasing, variable `u` and `v` are out of scope.
When a data-flow returns from a callee to a call site, and a field was written within the callee, a call-site point of aliasing is registered at the call site to

```
161 context1(taint){
162   x = new X;
163   y = new X;
    //taint, x and y do not alias
164   foo(x,y,taint);
165 }
```

```
166 context2(notaint){
167   u = new X;
168   v = u;
169   //no taint, u and v are aliased
170   foo(u,v,notaint);
171 }
```

```
172 foo(a, b, data){
173   a.f = data;
174   aliasData = b.f;
175   sink(aliasData);
176 }
```

Figure 5.5: A taint analysis example to demonstrate precise and demand-driven pointer analysis.

model indirect flows. In Figure 5.4, the call site point of aliasing is the exploded supergraph node `u.f`@155 that is highlighted as ⓒ.

At the field-store point of aliasing ⓢ, the analysis registered that object o_{154} flows to some field of o_{152}. At the call site, for any access path that originates from object o_{152} and that bypasses the call site, an indirect flow edge is injected into the analysis. Concretely, in the example, it means that from `u.f` also `v.f` is generated: The variables `u` and `v` are aliases at the call site in line 155.

5.3 Demand-Driven Points-To Analysis

Until now, we assumed a whole-program points-to analysis which starts propagating at *any* allocation site within the program. Instead of trying to improve scalability of a whole-program points-to analysis, we argue that whole-program points-to information is frequently not necessary, in particular, for taint or typestate analyses client. These *client analyses* of a points-to analysis mostly require focused and local points-to information as taint or typestate flows span a restricted, rather small code region.

In addition to focused points-to information, these clients require precise information. Imprecision within the points-to results propagate into the client's results, introduce false positives and hereby render the results less understandable [38]. Even worse, a less precise points-to analysis returns more (spurious) variables to alias. In turn, the client analysis considers spurious variables activating more data-flow propagations, which results both in imprecise results and longer runtime of the client analysis.

Example 15. Figure 5.5 provides an example of a taint analysis that requires points-to information. Assume the data-flow analysis to start in method `context1` where variable `taint` is tainted. The methods then constructs two objects (o_{162} and o_{163}) that are accessible by variables `x` and `y`. At the call site in line 164, these variables as well as variable `taint` flow as parameters to method `foo()`. This method stores the taint that resides in the variable `data` to field `f` of parameter `a` (line 173). The variable `a` references the same object as variable `x`, i.e., object o_{162}. At the subsequent statement in line 174, the field `f` is loaded from the object that is referenced by `b`. At this field load statement, variable `b` references the object o_{163}. As the taint never flows to object o_{163}, field `f` does *not* load tainted information in line 174.

The taint analysis only needs points-to information for the object o_{162}, because this object is the only one interacting with the tainted data. A whole-program points-to analysis computes unnecessary points-to information for all of the three allocated objects (o_{162}, o_{163}, and o_{167}).

Additionally, a whole-program points-to analysis may also introduce false positives to the taint analysis client. At runtime, variable `aliasData` never holds tainted information, and a precise taint analysis does not need to propagate variable `aliasData` as tainted. In the case the taint analysis demands a (context-insensitive) whole-program points-to analysis for aliases of variable `a` within `foo()`, variable `b` is reported as alias. The context-insensitive analysis does not distinguish between the two calling context of `foo()` in `context1` and `context2`. For the latter, variables `a` and `b` are aliased. With this imprecise points-to information, variable `b.f` and transitively `aliasData` are propagated as tainted and a false-positive tainted data-flow will be reported.

Apart from the negative effect of the over-approximation of the points-to analysis on precision, the approximation also affects the taint analysis efficiency. A precise points-to analysis avoids the unnecessary propagation of variable `aliasData`.

The requirements of the taint and typestate analysis clients motivate the design of demand-driven yet precise analysis. Instead of computing the points-to flows for every variable at every statement in the program, it suffices to compute the points-to set for a *pointer query*, i.e., a variable at a statement (optionally enriched by a calling context encoded in $\mathcal{A}_{\mathbb{S}}$).

We present Boomerang, a highly precise (context-, field-, and flow-sensitive) demand-driven pointer analysis that satisfies these requirements. For the computation of a points-to query, Boomerang introduces a control-flow backward directed analysis. The name Boomerang is chosen as the analysis alternates between forward and backward analysis passes.

Standard demand-driven points-to analyses [91, 104] deliver points-to sets for pointer variables. Boomerang's instead computes *rich* results that ease the integration with taint and typestate analyses. In addition to the relevant allocation sites, the results contain all aliasing access paths at the query statement that may also be used to access the same object.

5.3.1 Backward Analysis

The responsibility of the backward analysis is to compute the allocation sites, after which the forward analysis computes all local variables (or access paths) that point to these allocation sites.

```
177 context1(t){
178   x = new X;
179   y = new X;
180   foo(x,y,t);
181 }
182 foo(a,b,d){
183   a.f = d;
184   e = b.f;
185   sink(e);
186 }
```

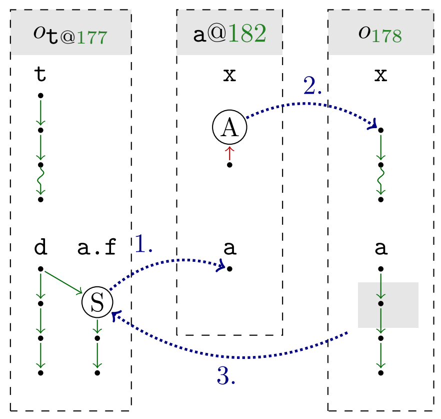


Figure 5.6: Indirect flows at a store point of aliasing.

Example 16. The example in Figure 5.6 presents the interaction of the forward and backward analyses when a demand-driven pointer analysis is integrated into a taint analysis. The example re-uses the code snippet from Example 15 and omits method `context2`, the second calling context of method `foo()`. Assume the taint analysis to propagate the variable `t`, the first parameter to method `context1`. In Figure 5.6, the taint stored in variable `t` points-to some abstract (tainted) object. We denote this abstract taint object as $o_{\mathtt{t}@177}$. The taint analysis forward-propagates the variable as data-flow fact according to the assignment chain of the variables. The tainted data escapes to `foo()` as variable `d`, where the data is stored within a field of some other (yet) unknown object at the store statement in line 183. This statement is a field-store point of aliasing at which the backward analysis is triggered.

The dotted blue arrow labeled 1. visualizes the following step the analysis takes: At the field-store point of aliasing, a backward query for the base of the field store, `a`@182, is triggered. Starting from this variable at that particular statement, backward flow is computed. The backward flow reaches the entry statement of method `foo()` where the flow continues to the call site of `foo()` in the method scope `context1` (line 180)[1]. In this scope, the variable of interest is `x` and the analysis discovers the variable's allocation site in line 178. The backward analysis marks this allocation site as an allocation site for the forward analysis. The arrow marked with 2. shows this analysis step.

From the allocation site, the forward analysis computes the data-flow as a regular whole-program points-to analysis based on SPDS. Any aliasing information that reaches the field-store point of aliasing that originally triggered the backward query will be used as indirect flow edge. In Figure 5.6, this is highlighted as the dotted blue arrow labeled 3. To keep the explanation simple, the example is designed such that there are no indirect edges added at the point of aliasing

[1]The backward analysis may only propagate within methods the forward analysis visits which means the backward analysis does not propagate within `context2` (see Figure 5.5).

```
187 foo(v,w){
188   w.g = v;
189   x = w.g;
190 }
```

Forward Rules	Backward Rules
$\langle\!\langle \texttt{v@187}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{w@188}, \texttt{g} \cdot * \rangle\!\rangle$	$\langle\!\langle \texttt{w@188}, \texttt{g} \rangle\!\rangle \rightarrow \langle\!\langle \texttt{v@187}, \epsilon \rangle\!\rangle$
$\langle\!\langle \texttt{w@188}, \texttt{g} \rangle\!\rangle \rightarrow \langle\!\langle \texttt{x@189}, \epsilon \rangle\!\rangle$	$\langle\!\langle \texttt{x@189}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{w@188}, \texttt{g} \cdot * \rangle\!\rangle$

Figure 5.7: Field push and pop rules of the backward pushdown system of fields are swapped from the forward pushdown system of fields.

and the taint analysis precisely suppresses a taint warning at the sink statement.

There are various ways to perform a backward analysis. We chose to reverse the control-flow graph [9]. The analysis interprets the control-flow successors as control-flow predecessors, return statements, i.e., method exit points become the method's entry points. Accordingly, method entry statements are turned into exit statements. All flow functions are reversed, which means the start and target configurations of the (normal) rules are swapped and a push rule turns into pop rule and pop a rule becomes a push rule. For instance, a field-load statement generates a field-push rule for $\mathcal{P}_{\mathbb{F}}$ of the backward analysis. A field-store statement pops a field off the stack and corresponds to a pop rule in $\mathcal{P}_{\mathbb{F}}$.

Alternatively, a backward analysis could also be computed by algorithm pre* [22] which is the reverse algorithm to post*. pre* takes a pushdown system and computes a backward (opposed to a forward) reachability query. Implementation-wise one would need a synchronized version of pre*, and we chose to reverse the control-flow instead.

Example 17. In Figure 5.7 we show an excerpt of the push and pop rules for the backward and forward directed $\mathcal{P}_{\mathbb{F}}$. The statement in line 189 is a field-load statement accessing field `g`. For the backward analysis, the control-flow successor of statement 189 is statement 188. A backward analysis that searches the allocation of `x` at the end of `foo()`, i.e., statement 189, propagates to `w.g` at statement 188 and the backward analysis' rule set contains the push rule of form $\langle\!\langle \texttt{x@189}, * \rangle\!\rangle \rightarrow \langle\!\langle \texttt{w@188}, \texttt{g} \cdot * \rangle\!\rangle$. The push rule is the equivalent of the forward analysis' pop rule $\langle\!\langle \texttt{w@188}, \texttt{g} \rangle\!\rangle \rightarrow \langle\!\langle \texttt{x@189}, \epsilon \rangle\!\rangle$.

The (backward) succeeding statement of the field-load statement is a field-store statement (line 188). For the backward analysis this means, field `g` is removed from the stack and a field pop rule, $\langle\!\langle \texttt{w@188}, \texttt{g} \rangle\!\rangle \rightarrow \langle\!\langle \texttt{v@187}, \epsilon \rangle\!\rangle$, is generated for the backward analysis. The forward analysis generates the opposite rule: The field-store statement corresponds to a field push rule.

5.3.2 Field-Load Point of Aliasing

Alias-sensitivity is not only a challenge for the forward analysis; the backward analysis also misses allocation sites if alias-sensitivity is not encoded within SPDS. Symmetric to the field-store point of aliasing, the backward analysis misses aliases due to field-load statements.

```
191 foo(){
192   u = new;
193   v = u;
194   o = new;
195   u.f = o;
196   p = v.f;
197 }
```

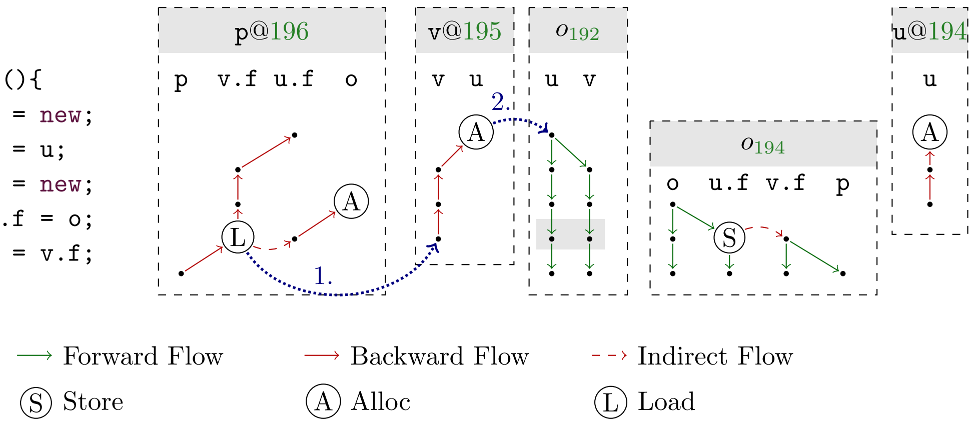


Figure 5.8: Symmetric to the forward analysis, the backward analysis requires a point of aliasing for field-load statements.

Example 18. Figure 5.8 depicts a code snippet that requires alias-sensitivity for the backward analysis and visualizes its data-flow analysis solution. Assume an analysis to query for points-to information of the variable `p` at the last statement of method `foo()`. The demand-driven analysis backward-searches for allocations of `p`. The last statement (line 196) of method `foo()` is a field-load statement and the backward analysis generates a push rule for the assignment of the load. Instead of searching for `p`, the analysis propagates `v.f`. However, the code never assigns `v.f`.[2] The backward analysis cannot find any allocation and no forward analysis is triggered. The static analysis would compute an empty points-to set for the pointer query variable `q`. But at runtime, variable `p` points to the object o_{194}, which means that the static analysis is unsound.

The problem is that variables `v` and `u` are aliased, which is why the backward analysis must also search for allocations of `u.f` after the field-load statement in line 196. The concept is symmetric to the forward analysis: During the forward analysis, during application of field push rules, field-store points of aliasing are generated. For the backward analysis field-load statement generate field-push rules and a point of aliasing is required for the backward analysis.

Figure 5.8 visualizes the process of the analysis. Boxes labeled by objects, i.e., o_{192} match forward SPDS, boxes labeled by variable at statements, e.g., `p`@196, group data-flows belonging to backward directed analyses. In Figure 5.8, we label field-load point of aliasing by Ⓛ. The backward analyses for `p`@196 generates a field-load point of aliasing at statement `p = v.f` in line 196. At the point of aliasing a second backward subquery is triggered. The backward analysis requires all-aliases to the base variable of the load (variable `v`). The backward analysis of this subquery discovers the allocation site of `v` in line 192. From this allocation site on, all aliases are computed in the forward analysis of object o_{192}.

Once the backward subquery is completed, the analysis has detected that `v` and `u` are aliases. An indirect flow edge from `v.f` to `u.f` is added to the backward analysis of `p`@196. The data-flow fact `u.f` continues to the field-store statement

[2]We assume constructors do not initialize any fields.

```
198 foo(){
199   u = new;
200   v = new;
201   u.f = v;
202   setG(u);
203   p = u.f;
204   q = p.g;
205 }
206 setG(a){
207   b = a.f;
208   c = new;
209   b.g = c;
210 }
```

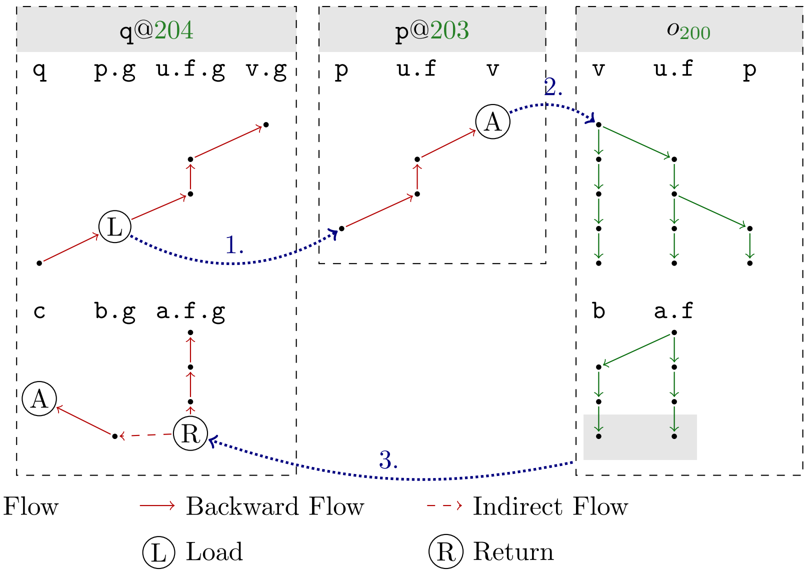


Figure 5.9: An example illustrating return site point of aliasing.

`u.f = o` in line 195. According to the statement, the backward analysis searches for allocation sites to variable `o`. The latter data-flow fact unravels the expected allocation site in line 194 from which the forward analysis starts its propagation.

5.3.3 Return Site Point of Aliasing

As discussed in Section 5.2.2, the forward analysis records call-site points of aliasing. These add additional indirect data-flow edge when a data-flow fact is mapped from a callee to a call site. The call-site point of aliasing collects missing aliases that are established prior to a call site and generates indirect flow edges. The backward analysis also misses aliases when a fact is mapped from call site to the return statements of a respective callee. Within the new callee scope, indirect aliases to the data-flow fact may exist and the allocation site the backward analysis is searching for may be stored within a field of an indirect alias. *Return site point of aliasing* of the backward analysis are the symmetric concept to call site point of aliasing of the forward analysis.

Example 19. Figure 5.9 shows a code snippet for which the backward analysis requires a return point of aliasing to find the right allocation site. Assume a query to `q` in line 205. At runtime, variable `q` contains the object o_{208}. The backward analysis follows the assignment chain of `q` and generates the access paths `p.g` and subsequently `u.f.g` in lines 203 and 204. The data-flow fact `u.f.g` flows to method `setG()` where it is mapped to `a.f.g`. Within this scope, the prefix access path `a.f` aliases to `b`. Method `setG()` updates field `g` of the aliasing variable `b` in line 209 to hold object o_{208}.

For the backward analysis to detect the allocation site 208, upon entering the scope of `setG()`, the analysis must search for `b.g` in addition to `a.f.g`. A return point of aliasing introduces indirect aliasing edges to cover the data-flows of aliases of the entering access path.

Figure 5.9 also visualizes the construction of the data-flow for the pointer query `q`@204. The query starts a backward analysis, which generates the data-flow contained in the box labeled by `q`@204. The backward analysis generates a field-load point of aliasing (Ⓛ) at statement in line 204 and the data-flow further propagates into method `setG()` where a return site point of aliasing (Ⓡ) is generated at statement in line 209. The field-load point of aliasing triggers the backward query `p`@203 which flows to the allocation site (Ⓐ) in line 200. From this allocation site the forward analysis of o_{200} propagates its data-flow through `foo()` and `setG()`. Within `setG()`, the analysis records `b` to be an alias to `a.f` at the statement of Ⓡ (line 209). The return point of aliasing processes the aliases and replaces the prefix `a.f` of `a.f.g` by `b` and adds an indirect flow edge to propagate `b.g`. Due to the generation of `b.g`, the backward analysis of `q`@204 reports the allocation site in line 208.

5.4 Unbalanced Returns of Allocation Sites

A points-to analysis that distinguishing objects only by their allocation site statement is not sufficient when the analysis is context-sensitive. A simple method that creates an object and returns the object, instantiates one object per method call. For instance, the factory pattern typically induces such cases. A factory method creates an object and returns the instance to the call site. It follows that each call site of the factory method allocates its own instance.

The created objects escape the factory method. In terms of a data-flow analysis, this means the flow is an *unbalanced return* [52], i.e., the data-flow is rooted in some callee and returns to its callers *without* actually matching any call site when returning. In cases of unbalanced returns, Boomerang distinguishes objects not only by their allocation sites but also by their call sites that return the object.

Example 20. Figure 5.10 presents a data-flow and Boomerang's model of unbalanced returns. Method `foo()` of Figure 5.10 calls method `create()` in lines 218 and 220 to create two[3] individual objects allocated at the same allocation site (line 212). Within method `foo()`, a context-sensitive data-flow analysis can distinguish both instances.

Boomerang separates the two objects within method `foo()` and maintains its context-sensitivity. Right of the code, Figure 5.10 depicts the automaton $\mathcal{A}_{\mathbb{S}}$ for the data-flow of object o_{212}. Within method `create()`, Boomerang does not distinguish the two objects and the automaton only contains the transitions from `u` to o_{212} with labels 212 and 213. When the data-flow returns unbalanced to method `foo()`, Boomerang creates two new (synthetic) accepting states within $\mathcal{A}_{\mathbb{S}}$. The new states are uniquely identified by the name of the object (o_{212}) and the unbalanced call sites that returns the object (218 and 220).

[3] Boomerang is not path-sensitive and does not argue that only a single object exists at runtime.

```
211 create(){
212   u = new;
213   return u;
214 }
215
216 foo(){
217   if(...){
218     b = create();
219   } else {
220     b = create();
221   }
222   c = b;
223   bar(c);
224 }
225
226 bar(x){
227   y = x;
228 }
```

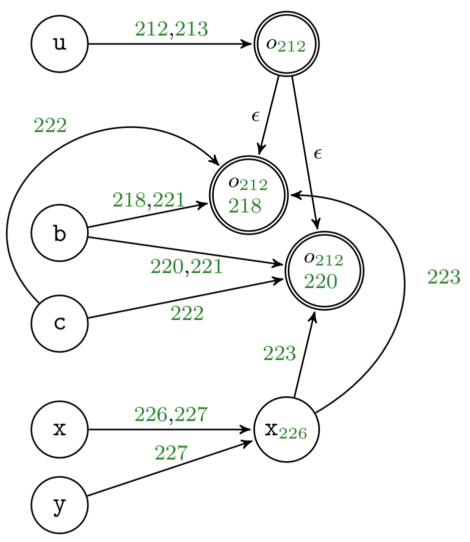


Figure 5.10: An example illustrating unbalanced context modeling in Boomerang.

The automaton contains transitions from variables `b` and `c` of `foo()` to these newly created states. Therefore, Boomerang is able to reason that `b`@221 (respectively `c`) may point to any of the two returned objects.

The automaton explodes the states of allocation sites for each unbalanced call sites within `foo()`. This increases the cost of the computation of the data-flow within `foo()`. For instance, post* computes two transitions from `b` labeled by 221 that differ only in their target states.

Despite distinguishing the data-flows within `foo()` which increases the computational effort, the data-flow of both objects continue to `bar()`. Within `bar()` Boomerang does not require to separate the two data-flows anymore, instead the automaton contains the intermediate state $\texttt{x}_{226}$ and computes that `x` and `y` point to this state. The latter intermediate state is the source state of transitions that reflect the argument variable `x` of `bar()` to point to either of the two objects within `foo()`. Therefore, within `bar()` and any transitive caller, computational effort is saved as the data-flow for the two objects is *not* explicitly separated.

5.5 Evaluation

In this section, we evaluate an implementation of the demand-driven points-to analysis Boomerang and compare it to existing demand-driven pointer analyses.

5.5.1 Implementation

We have implemented Boomerang based on the static analysis framework Soot.[4] Our implementation is publicly available.[5]. We originally presented Boomerang at ECOOP 2016 [90]. At the time, the implementation used access graphs. In 2017, we implemented post* and SPDS and replaced the heap model in Boomerang by SPDS which led to re-implementing Boomerang. For the re-implementation, we use the *Observer Pattern* in most places. This pattern is helpful to implement Boomerang's points of aliasing that depend on the two automata $\mathcal{A}_{\mathbb{F}}$ and $\mathcal{A}_{\mathbb{S}}$ that encode the set $aliases_a(s)$. Points of aliasing constantly add transitions to the two automata. This triggers the re-computation of post* which in turn affects other points of aliasing. The observer pattern eases the implementation and avoids uses of worklists[6] as each observer processes each request synchronously. In the Boomerang implementation from 2016, we noticed that these worklist easily introduce non-determinism if they are not updated correctly. Our implementation is rigorously tested with more than 300 test cases that cover the common features a pointer analysis should handle (field stores and loads, interprocedural flows, static fields, array etc.). Apart from small test cases that test certain features explicitly, Boomerang also ships with test cases computing complex data-flows through methods of the Java Runtime Library. For instance, test cases storing and loading elements in a `java.util.Collection`, e.g., `HashSet`, `LinkedList`, `ArrayList`, `TreeSet` and others.

Based on our implementation, we evaluate the precision and the performance of Boomerang through answering the following research questions:

- **RQ1** How precise is the pointer information delivered by Boomerang compared to other existing pointer analyses?
- **RQ2** How does the use of Boomerang affect the performance and precision of a taint analysis?
- **RQ3** How does Boomerang perform in comparison to other pointer analyses when using data-race analysis as a client?

The long interest in the area of points-to analysis led to a large variety of implementations of different flavours of pointer analysis. For this comparison, we focus on two pointer analysis implementations that are feature-similar to Boomerang. Both analyses are designed for program analysis of Java, are implemented on top of the analysis framework Soot, are demand-driven, and feature field- and context-sensitivity. The major contrast to Boomerang lies in their flow-insensitivity and their weaker output format: neither of the two analyses returns all aliasing access paths at a query statement.

[4]`https://github.com/sable/soot`

[5]`https://github.com/CROSSINGTUD/WPDS`

[6]An observer for $\mathcal{A}_{\mathbb{F}}$ or $\mathcal{A}_{\mathbb{S}}$ may not register itself recursively to guarantee termination. All observers of an automaton are maintained in a set and must define `hashCode()` and `equals()` methods.

Refinement-Based Context-Sensitive Points-To Analysis for Java (2006) by Sridharan and Bodík [91] (hereafter denoted SB) is an analysis that refines pre-computed points-to analysis results. The implementation relies on the context-insensitive points-to analysis shipped within SPARK [55] and refines the results of a pointer query by traversing SPARK's pointer-assignment graph. For difficult points-to queries that require traversal of many nodes, the time for the refinement phase takes several minutes. As a fallback, SB specifies an analysis budget. Once exceeded, SPARK's context-insensitive points-to results are returned. A drawback of SB is that it requires the pre-computation of the points-to based call graph by SPARK. In our experiment on Maven Central (Section 7.3), we observed that the construction of context-insensitive call graphs does not scale and SB cannot be applied.

Demand-Driven Context-Sensitive Alias Analysis for Java (2011) by Yan et al. [104] (hereafter denoted DA) is an analysis crafted for a limited set of client analyses such as a data-race analysis. In comparison to a points-to analysis that returns a points-to set of allocation sites for *one* particular variable, an alias analysis takes *two* variables as input and returns `true` in the case both variables alias, i.e., their points-to sets share at least one allocation site. An alias does not need to compute the complete points-to sets for both variables but can terminate quickly as soon as one shared allocation site is found. Therefore, an alias analysis can be more efficient. DA pre-computes intra-procedural method-wise alias relationships for variables on-the-fly and combines the information for every new query.

5.5.2 Precision and Recall on PointerBench

Evaluating any static analysis with respect to precision and recall requires knowledge of a ground truth which marks the correct, expected output for an analysis. Precision and recall values for the static analysis are computable based on deviations from the ground truth. For a static points-to analysis, the ground truth comprises the points-to sets for a variable. For an alias analysis, the ground truth comprises pairs of aliased variables. Deriving the ground truth for a pointer or alias analysis on real-world software is difficult and requires error-prone manual effort. A points-to analysis computes a solution across all execution paths and correctly labelling and specifying the complete points-to sets for all variables for all paths is not viable.

Instead of labelling real-world software, we propose PointerBench as a micro-benchmark suite for pointer analysis with a labeled ground truth. The benchmark's test programs are designed to test for analysis design dimensions (e.g., context or field-sensitivity) separately. Hence, true positives, false positive and false negative highlight strength and weaknesses of the pointer analysis under test.

The PointerBench Micro-Benchmark. PointerBench contains 36 specially crafted small programs that depict common pointer analysis issues for Java (e.g.,

		Alias pairs			Allocation sites	
	Tests	Boomerang	DA	SB	Boomerang	SB
Basic	SimpleAlias	✓	✓	✓	✓	✓
	Interprocedural	✓✓✓	✓✓✓	✓✓✓	✓✓	✓✓
	ReturnValue	✓✓✓✓✓✓	✓✓✓✓✓⊖	✓✓✓✓✓✓	✓✓✓	✓✓✓
	Parameter	✓✓	✓✓	✓✓	✓✓	✓✓
	Loops	✓✓✓✓⊕	✓✓✓✓⊕	✓✓✓✓⊕	✓✓✓	✓✓✓
	Recursion	✓✓✓	✓✓✓	✓✓✓	✓	✓
	Branching	✓	✓	✓	✓✓	✓✓
General Java	ContextSensitivity	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓✓✓✓✓	✓✓✓✓✓✓
	ObjectSensitivity	✓✓	✓✓	✓✓	✓✓	✓✓
	FieldSensitivity	✓✓✓✓	✓✓✓✓	✓✓✓✓	✓✓	✓✓
	FlowSensitivity	✓	⊕	⊕	✓	✓⊕
	StrongUpdate	✓✓	✓⊕	✓⊕	✓✓	✓✓
Corner Cases	OuterClass	✓✓	✓⊕	✓⊕	✓	✓⊕
	SuperClasses	✓	✓	✓	✓	✓⊕
	StaticVariables	✓	✓	✓	✓	✓
	Null	✓✓	✓✓	✓✓		
	Exception	✓✓	✓✓	✓✓	✓✓	✓✓
	Interface	✓	✓	✓	✓	✓
Collections	List	✓✓✓⊕⊕	✓✓✓⊕⊕	✓✓✓⊕⊕	✓✓⊕⊕	✓✓⊕⊕
	Map	✓⊕	✓⊕	✓⊖	✓⊕	⊖
	Set	✓✓⊕	✓⊕⊕	⊕⊕⊕	✓⊕	⊖
	Array	✓⊕	✓⊕	✓⊕	✓⊕	✓⊕
	Recall	1.0	0.98	0.98	1.0	0.95
	Precision	0.89	0.81	0.81	0.88	0.86

Table 5.1: The precision and recall of Boomerang, SB, and DA with respect to alias pairs and allocation sites on PointerBench. ⊖= false negatives, ⊕= false positives, ✓= true positives. For the test cases `Set` and `Map`, SB times out after 10 minutes.

handling collections, field-sensitivity, context-sensitivity). Each program contains special API-calls that describe points-to and alias queries. The points-to sets of the queries are allocation sites whose type implement a certain interface. A static pre-analysis extracts and parses the required statements and provides may information about all aliases, non-aliases, and allocation sites of a particular variable in the program. Using PointerBench, alias, points-to, or any kind of pointer analysis can be compared against one another based on the same ground truth. PointerBench is open source[7] and we encourage contributions.

Experimental Setup. For each program and each analysis (Boomerang, SB, and DA), the specified points-to and alias queries on PointerBench is extracted and triggered. A query (m, v) to SB comprises a local variable v and the method m the variable belongs to. The query result is a points-to set of the given variable with each allocation site enriched by a calling context. A query for DA consists of two local variables and the methods they belong to: (m_1, v_1, m_2, v_2). The result is a boolean value stating whether the two variables v_1 and v_2 in the given

[7]Available for download at `https://github.com/secure-software-engineering/PointerBench`

methods m_1 and m_2 may alias or not. Although this interface is convenient for some client analyses (e.g., data-race analysis), points-to sets cannot be derived from alias information. For Boomerang, a pointer query is a local variable v and a statement s (Boomerang is flow-sensitive). Boomerang returns results in the form of a map. The map's keys are allocation sites o_a. Each associated value is the set $aliases_a(s)$ that encodes all access paths aliasing to v.

Due to the difference in analysis types, returned information and query format, we compare the precision of SB, DA, and Boomerang on their common basis: alias information. We use DA's query format as this information can be derived by all three analyses. For SB, an alias query is mapped to two points-to queries, one for each variable of the alias query. If the intersection of the two points-to sets is non-empty, the two variables may alias. Boomerang returns all aliases of an object a as set $aliases_a(s)$ directly and a single points-to query for one variable suffices. In addition to the evaluation of alias information, we additionally compare Boomerang's points-to sets to SB's points-to sets.

Results for Alias Pairs. In Table 5.1, we report the true positives, false positives, and false negatives for each pair of aliasing/non-aliasing variables. Across all the programs in PointerBench, Boomerang achieves 100% recall and 89% precision, DA achieves 98% recall and 81% precision, and SB achieves 98% recall and 81% precision with respect to alias pairs.

DA reports a false negative for the test case `ReturnValue`. This test case creates an object and passes it to a static method that returns its parameter (i.e., it is an identity function). In the caller of that function, the argument to the call and the return value should alias. DA incorrectly models static methods, which leads to this false negative.

In the group `Collections`, the test cases contain operations on `TreeSet` and `HashMap` (e.g., inserting and retrieving elements). These collections store elements in arrays, but all three analyses are array-insensitive. Therefore, assignments to array elements are treated as assignments to a synthetic field `ARRAY` modeling all of the array's content, disregarding the index. Additionally, all analyses are path-insensitive: an object added to a set aliases with all other objects contained in that set (unless they cannot alias by their type). This imprecision constitutes to the precision loss for all three analyses in `Collections`.

For the test case `FlowSensitivity`, DA and SB report a false positive. The test case triggers a query for `b` before a statement `b = a` which updates `b` to alias with `a`. Prior the statement, the alias relationship between the two variables does not hold, the two flow-insensitive pointer analyses DA and SB cannot detect it.

Results for Allocation Sites. We now compare the two analyses Boomerang and SB. DA delivers only alias results and hence cannot compute allocation sites. For each program in PointerBench, the last two columns of Table 5.1 show false negatives, false positives and true positives in terms of allocation sites reported by Boomerang and SB.

Across all test cases in PointerBench, Boomerang achieves 100% recall, and SB 95%. The lower recall value for SB is due to the analysis timeouts on the test cases `Map` and `Set`. For these test cases, SB's analysis runs into a timeout after 10 minutes and the analysis is not able to compute points-to sets. We consider these timeouts as empty points-to sets which constitutes to two false negatives for SB. With respect to precision, Boomerang and SB achieve 88% and 86%, respectively. The main reason for SB's drop in precision are the test cases `FlowSensitivity`, `OuterClass` and `SuperClasses`. The analysis SB is not flow-sensitive and the test case `FlowSensitivity` strongly updates a variable. A strong update requires flow-sensitivity. The test cases `OuterClass` and `SuperClasses` store and load an object from fields of respective classes. In both target programs, a field store strongly updates the content of a field. The strong update requires a flow-sensitive analysis. Therefore, Boomerang is more precise for these target programs.

Summary. In summary, on the micro-benchmark suite PointerBench, all three analyses are highly precise when used for alias information. Boomerang achieves slightly higher precision than DA and SB. Also when used to query points-to information, Boomerang is slightly more precise than SB as Boomerang is flow-sensitive and can perform strong updates.

5.5.3 Integration with a Taint Analysis for Android

The usefulness of a demand-driven pointer analysis is best evaluated in combination with a concrete client analysis. FlowDroid is a context-, field-, and flow-sensitive taint analysis for Android [5]. In this experiment, we compare the performance of different pointer analyses (Boomerang, SB, DA and FlowDroid's default alias analysis) when integrated into the taint analysis FlowDroid on real-world applications from the Google Play Store.

Experimental Setup. Like many taint or typestate analyses, FlowDroid requires alias information at points of aliasing, i.e., at field-store statements and at call sites. For a taint analysis, the logic is the same as for a points-to analysis and at these statements, points of aliasing are required such that FlowDroid indirectly taints other data-flow facts (FlowDroid access paths). At those statements, FlowDroid needs to obtain all aliases of the tainted access path. FlowDroid then taints those aliases and continues the taint propagation.

The interface for SB delivers pairwise alias information of variables, SB delivers points-to sets in the form of allocation sites. Neither of the two interfaces returns all aliases directly in the form of access paths. Therefore, integrating DA or SB into FlowDroid needs additional post-processing for the computation of all-alias sets.

The post-processing is as follows. When a given FlowDroid access path `a.f.g` is tainted at a field-store statement (`a.f = taint`) or returns from a call site to a new scope (`taint(a)`), the following operation must be performed. For every local variable `d` of the method containing the field-store or the call site, the analysis checks if the variable `d` aliases with `a`. In such case, the access path `d.f.g` is added to A, the set of all aliasing access paths. Similarly, for every field-load statement `c = d.f`, where the field matches the first field of the access path `a.f.g`, the analysis checks whether `d` and `a` alias. If so, the access path `c.g` is also added to A. For every field-write statement `d.h = t`, the analysis checks for aliasing between `t` and `a`. If they alias, the access path `d.h.f.g` is added to the set A. Each alias within set A is injected into FlowDroid's data-flow propagation.

In contrast, when using Boomerang, FlowDroid requires only one query per field store or call site. Boomerang directly returns a set containing all aliases.

In his thesis [3], Arzt evaluates FlowDroid on a set of 25 realistic real-world applications. For this experiment, we use the same set of applications and ran FlowDroid with the three different pointer analysis integration. As a baseline for comparison, we also run FlowDroid in its default configuration[8] that includes its own intertwined alias strategy [5]. The experiment was conducted on an Intel Xeon E5-2680, 2.40 GHz machine with 16 processors and 128 GB of memory using JDK version 1.8.0_171. We limit the overall analysis time for each application to 60 minutes.

Results. Table 5.2 lists the analysis times and the number of reported taint flows for FlowDroid configured to use the different pointer analyses. FlowDroid supported by Boomerang successfully analyzed 23 of the 25 applications within the allocated time budget of 1 hour. FlowDroid supported by DA terminated its analysis on 11 of all applications. With the integration of SB in FlowDroid, the taint analysis successfully terminates on only 2 out of all 25 applications. FlowDroid in its default configuration terminates on 21 applications. This clearly shows that the rich query format Boomerang offers significantly impacts the analysis time of a client analysis.

Table 5.2 also lists the analysis times of FlowDroid in the case the analysis terminated. These times are the average of 5 independent runs. In 22 of all applications the integration of Boomerang into FlowDroid outperforms the default configuration of FlowDroid. Averaged across the terminated runs, Boomerang improves the performance of FlowDroid by a factor of 2.3×. We argue that the main reason for this performance improvement are the SPDS of Boomerang. FlowDroid models access paths explicitly. We compare these two heap models more carefully in Section 8.1.

Table 5.2 shows that the choice of the alias analysis also affects the number of taint flows reported by FlowDroid. We checked our implementation of the integration of Boomerang into FlowDroid by relying on the large set of test cases available for heap flows in FlowDroid's implementation. We found some test cases for which the integration with Boomerang in FlowDroid delivers more precise results than the default configuration of FlowDroid. In particular, these cases include programs where a data-flow is propagated via an access path

[8]Except that we only use a single thread. The implementations of DA, SB and Boomerang are not multi-threaded.

Table 5.2: Analysis times and reported taint flows of FlowDroid with different alias strategies. Timeouts are marked by ✗.

Application		FlowDroid	Boomerang	DA	SB
Burger King	Time (in s)	0.3	0.3	49.0	✗
	#Flows	7	7	11	-
Adobe Reader	Time (in s)	1298.2	26.9	1260.1	✗
	#Flows	9	8	14	-
Amazon Kindle	Time (in s)	1178.8	510.4	✗	✗
	#Flows	-	51	-	-
AutoScout24	Time (in s)	5.1	1.5	29.7	71.1
	#Flows	7	6	6	6
CNN	Time (in s)	1528.6	2.9	✗	✗
	#Flows	33	33	-	-
eBay	Time (in s)	1289.2	37.0	✗	✗
	#Flows	27	30	-	-
Facebook	Time (in s)	0.6	0.2	✗	✗
	#Flows	3	3	-	-
Facebook Messenger	Time (in s)	✗	✗	✗	✗
	#Flows	-	-	-	-
Google Plus	Time (in s)	✗	✗	✗	✗
	#Flows	-	-	-	-
Instagram	Time (in s)	1235.3	0.8	1654.5	✗
	#Flows	-	6	-	-
LinkedIn	Time (in s)	845.3	846.6	✗	✗
	#Flows	82	83	-	-
Microsoft Outlook	Time (in s)	924.2	4.1	✗	✗
	#Flows	26	18	-	-
Microsoft Word	Time (in s)	5.8	2.0	871.0	✗
	#Flows	22	18	24	-
Netflix	Time (in s)	39.0	2.5	✗	✗
	#Flows	23	17	-	-
Pokemon Go	Time (in s)	3.8	2.8	✗	✗
	#Flows	23	24	-	-
Opera	Time (in s)	4.8	3.0	2.1	3.5
	#Flows	14	20	20	20
PayPal	Time (in s)	5.4	6.8	756.3	✗
	#Flows	18	18	21	-
Pinterest	Time (in s)	✗	1429.0	✗	✗
	#Flows	-	42	-	-
Angry Birds	Time (in s)	773.2	1.3	786.7	✗
	#Flows	-	22	31	-
Skype	Time (in s)	3.8	3.0	✗	✗
	#Flows	23	21	-	-
Tinder	Time (in s)	1370.3	13.8	1160.1	✗
	#Flows	14	12	15	-
Wetter.com	Time (in s)	✗	257.0	✗	✗
	#Flows	-	67	-	-
Offi Journey Planer	Time (in s)	11.9	5.3	225.0	✗
	#Flows	1	1	4	-
VLC for Android	Time (in s)	5.6	0.6	✗	✗
	#Flows	1	1	-	-
Telekom Mail	Time (in s)	1009.1	1140.0	1143.9	✗
	#Flows	26	28	28	-

and the base variable of the access path is known to be `null`. For these case, the program does not leak any data as it crashes earlier in a `NullPointerException`. Another test case shows an over-approximation due to the *activation statement* that FlowDroid's original implementation requires [3]. Each of FlowDroid's data-flow facts is enriched by an activation statements, a taint propagated backward is deactivated and only activated once it bypasses the activation statement during the forward propagation. With the integration of Boomerang the activation statement is not necessary.

We assume these cases to also occur on the real-world applications and affect the number of reported taint flows. Unfortunately, we do not have a ground truth for the taint flows within these applications and we cannot argue which integration misses findings or introduces new spurious ones.

Summary. The analysis time of FlowDroid reduces significantly when switching from DA or SB analyses to Boomerang's alias integration. The taint analysis client FlowDroid makes full use of the rich query results returned by Boomerang. The integration of Boomerang into FlowDroid reduces the analysis time by a factor of 0.44× in comparison to the default FlowDroid taint analysis.

5.5.4 Data-Race Client on DaCapo

In this experiment we compare the three demand-driven points-to analyses on a second data-flow client for pointer analysis, a data-race client.

2006 DaCapo Benchmark Suite. The DaCapo 2006 benchmark suite [7] contains a collection of 11 different realistic, general purpose and freely available Java programs. The programs have between 2,795 and 12,450 (geometric mean: 5,768) declared methods. The benchmark suite is widely accepted and heavily used for testing and benchmarking of dynamic analyses, but also used in benchmarking of static analyses.

Data Race. A data race is a programming error that causes the execution of concurrent programs to be non-deterministic. In Java, a data race occurs when two threads access the same field of the same object and at least one access is writing to the field. When both field accesses are not properly locked in synchronized blocks, the field accesses execute in arbitrary order and can render the program execution non-deterministic. The non-determinism makes the error difficult to reproduce, and hence to debug and fix. A static analysis can suggest which pairs of field accesses may potentially race.

Experimental Setup. A static data-race analysis detects potential data races within a program. For any pair of statements, hereafter called *data-race pair*, that access the same field, say `f`, a data-race analysis checks whether the base variables of both accesses are aliased to each other. Say one of the two statements

Table 5.3: Precision and performance results for the datarace analysis with the demand-driven pointer analysis DA, SB, and Boomerang.

Benchmark	ANTLR	BLOAT	CHART	ECLIPSE	FOP	HSQLDB	JYTHON	LUINDEX	LUSEARCH	PMD	XALAN
Checked Pairs	5513	221799	841	3805	541	9173	32152	1364	4595	113634	54
Pruned Pairs	6 310 4	10 1468 0	0 45 0	115 287 0	0 161 0	63 254 0	5 1121 5	5 632 284	76 257 282	0 558 6	3 52 0
Precision Gain (in %)	0.1 5.6 0.1	0.0 0.7 0.0	0.0 5.4 0.0	3.0 7.5 0.0	0.0 29.8 0.0	0.7 2.8 0.0	0.0 3.5 0.0	0.4 46.3 20.8	1.7 5.6 6.1	0.0 0.5 0.0	5.6 96.3 0.0
Timeouts	160 4907 2573	1800 220259 221799	0 667 440	0 2236 1885	0 55 18	1045 8736 9173	3047 30745 26988	28 318 271	184 3710 1963	7187 112829 24999	0 0 0

▨ SB 🟥 Boomerang ▨ DA

stores `x.f` and the other statement loads `y.f`, then a data race may occur when `x` and `y` are aliased. The program may behave differently depending on which statement executes first. For a data race, at least one of the two statements must write the field and change its value. The data-race client is a worst-case client for a demand-driven alias analysis as it triggers queries for any pair of statements accessing the same field. Hereby, it easily triggers queries throughout the entire program.

We run the data-race analysis on all 11 benchmarks of DaCapo. We pre-compute a SPARK-based context-insensitive call graph and compute all call-graph reachable data-race pairs for which SPARK's points-to analysis reports the base variables to alias. We then hand over the data-race pairs (in total 393,471) to the more precise pointer analyses (SB, DA, and Boomerang). A data race is a client for a pointer analysis with little requirements. Pairwise alias information, similar to what DA delivers, suffices and each data-race pair maps to exactly one alias query. For the two demand-driven points-to analyses (Boomerang and SB), queries to compute the points-to sets are triggered and the points-to sets are intersected to check if the two base variables of the field accesses alias. A non-empty intersection means the variables alias and the pair of statements is a potential data race. This analysis undermines the power of the points-to analysis, because the points-to sets are not further used. To limit the overall execution time for the many data-race pairs to check against, each data-race pair was granted 1 second to return its points-to result. If the pointer-analysis query times out, the data-race pair is conservatively reported as a potential data-race pair.

Precision Results. The precision results for this experiment are listed in Table 5.3. In the row *Checked Pairs*, the table shows the total number of datarace pairs that were checked for. This number excludes the pairs SPARK points-to analysis labels as non-aliasing. The row *Pruned Pairs* shows the number of pairs for which the analysis proves that a datarace does not exist, ergo the higher this number, the more precise the analysis. In addition to the pruned pairs, Table 5.3 lists the *Precision Gain* computed as

$$\text{Precision Gain} = \frac{\text{Pruned Pairs}}{\text{Checked Pairs}}$$

The results show that, across all benchmarks, the precision gain is higher for the pointer analysis Boomerang than for DA and SB. A datarace client based on Boomerang reports fewer candidates as datarace pairs to the user.

Performance Results. We discuss the performance of the datarace client in the form of the number of datarace pairs for which the computation takes more than one second, the time budget we allocated for each query. Apart from the number of pruned pairs, Table 5.3 lists the number of *Timeouts* which is the number of datarace pairs for which the respective pointer analysis analysis exceeds the time budget. Therefore, the higher the number of timeouts, the less efficient an analysis is.

The number of timeouts for the alias analysis DA is significantly smaller across all DaCapo benchmark programs. On geometric average, the analysis DA times out on 2.4% of all queries. Opposed to this, the pointer analysis SB times out for 27.3% and Boomerang even for 42.0% of all queries. Despite the fact that Boomerang performs worst, Boomerang also computes the most precise information. The efficiency results also indicate that the datarace client is a mal-suited client for points-to analysis and performs better with an alias analysis. The overhead to compute full points-to sets is unnecessary for a datarace client and avoided by the alias analysis DA. We further elaborate on the high fraction of timeouts and their origins in Chapter 8.

Summary. Boomerang is mal-suited for a datarace client. The datarace client is the perfect client for an alias analysis such as DA, because it leaves computed points-to sets unutilised. Moreover one must question whether it's not a client for which one should rather conduct a whole-program analysis in the first place.

5.6 Related Work

The intense amount of research has lead to pointer analyses with numerous kinds of approximations, each of which balances precision and scalability in a different manner and targets different clients. We split the large body of work into *whole-program* and *demand-driven* pointer analysis. Whole-program pointer analyses compute points-to sets for every pointer variable in the program and are mostly applied during call graph construction, for the detection of casts that may fail or for data-race clients. Demand-driven approaches are designed for clients such as just-in-time compilers, interactive development environments, taint or typestate analysis. These clients typically require targeted pointer information about limited parts of the code, thus raising pointer queries.

5.6.1 Whole-Program Pointer Analyses

Despite the fact that BOOMERANG is a demand-driven pointer analysis, we found flow-sensitivity to be a second major difference to many existing whole-program pointer analyses. Further on, we thus group the research into *flow-sensitive* and *flow-insensitive* approaches.

Flow-Insensitive Analyses

There are *Andersen-* [1] and *Steensgard-* [94] style whole-program points-to analyses. Both compute flow-insensitive points-to sets. Andersen-style analyses are subset-based; after an assignment statement $y \leftarrow x$ it holds $pts(y) \supseteq pts(x)$. Steensgard-style analyses are equality-based which means forcing the points-to sets to be equal after an assign statement, i.e., $pts(y) = pts(x)$. Steensgard-style analyses are more efficient to compute than Andersen- style, but produce less precise results.

The large amount of existing pointer analyses motivates the design of frameworks for pointer analysis such as SPARK [55] and DOOP [12]. SPARK is a pointer analysis framework that ships as part of the static analysis framework SOOT.[9] SPARK implements an Andersen-style context-insensitive and flow-insensitive pointer analysis and allows to compare implementation details and their impact on the analysis: Points-to sets can be implemented as `HashSets` or `BitVectors`, but also any hybrid version of the two. SPARK's context and flow-insensitive pointer analysis is the standard points-to based call graph construction algorithm in SOOT.

In DOOP [12], points-to analysis algorithms are specified in the declarative programming language Datalog. The relational formulation of Datalog is a perfect fit for Andersen-style points-to analysis and enables Datalog-specific automatic memory and runtime optimizations. For instance, the authors of DOOP report a significant performance boost by porting the Datalog rules to a different Datalog solver [2].

DOOP performs a top-down points-to analysis and as Andersen-style analysis uses the store-based heap abstraction [42]. For context-sensitivity, the store-based heap abstraction models object allocations with additional k-limited contexts string. The definition of a context can be arbitrarily modeled in the form of object allocations (object-sensitivity) or call sites (call-site context-sensitivity). The analysis merges contexts when the context length exceeds k. Various DOOP-based research exists trying to find the right balance between precision and scalability in different configurations of context-sensitivity [44, 86, 87].

Recently, Tan et al. [98] propose the design of a precise and efficient points-to analysis for *type-based* data-flow clients. Call-graph construction algorithms, devirtualization, or may-fail casting clients do not require precise information about actual allocation sites of variables, instead the actual runtime types of the object suffice. From a fast but imprecise points-to pre-analysis a field points-to graph is extracted. The graph is used to pre-compute *type-consistent* objects. Two

[9] `https://github.com/Sable/soot`

objects are type-consistent if for all of their fields the object stored in the fields are of the same type. For two type-consistent objects, their points-to sets can be merged during the more precise points-to analysis without sacrificing precision for type-based data-flow clients. While the idea is similar to a simple variable type analysis (VTA), their analysis requires a complete points-to graph along which the type-consistent checks are performed. We cannot apply a similar idea to BOOMERANG, because the clients we address require more precise information than purely the types of the allocation sites.

Flow-Sensitive Analyses

In contrast to a flow-insensitive analysis that stores points-to set per variable and method of the analyzed program, a flow-sensitive analysis requires one to maintain points-to sets per variable and statement of the program. Storing points-to set statement-wise not only drastically increases the memory consumption of the analysis, but also prolongs the computation time for the fixpoint of the data-flow propagation. Therefore, some of these analyses chose to be context-insensitive instead.

In the pointer analysis for C presented in [35], the challenges of flow-sensitive analysis are addressed by a *sparse analysis*. Instead of propagation points-to sets from statement to statement, the sets are sparsely propagated along the *def-use chain* of variables. The def-use chain contains the definition statement of the variable and statement that uses the variable. A points-to analysis itself computes such def-use chains. Therefore, Hardekopf and Lin propose a staged analysis where an earlier and cheaper pointer analysis is consulted for the def-use chains. The downside of this approach is that spurious edges introduced during the cheaper pointer analysis imprecisely conflate results of the later stage. Shi et al. [85] call this problem the "pointer trap". In their work they also present PINPOINT, a flow-sensitive and demand-driven value analysis which partially overcomes the trap when resolving pointer relations. We discuss PINPOINT in more detail in Section 5.6.2.

Statement-wise points-to sets are more precise than flow-insensitive points-to sets because of strong updates. At field-store statements points-to information can be strongly updated when a points-to set of the base variable must point to a single runtime object. Lhoták and Chung present an analysis for C [54] that computes flow-sensitive points-to results for variables that must point to a single allocation site. Hereby, strong updates are still performed. The analysis switches to flow-insensitive results as soon as a the size of the points-to set is larger than one. They report their analysis to be almost as efficient as a flow-insensitive analysis while still performing 98% of the strong updates a full flow-sensitive analysis would perform. Opposed to BOOMERANG, this analysis is also context-insensitive.

De and D'Souza [17] present a context-sensitive alias analysis for Java that also performs strong updates. The authors claim a scalable analysis that is achieved by the storeless access path-based model with k-limiting. Their experiments are run with a k-limit with $k = 2$ and $k = 3$. For context-sensitivity, call strings of length 1 are used. Additional efficiency and precision may be gained by replacing their model with the synchronized pushdown system suggested in this work. As an alias analysis their analyses fits the needs of a data-race client, but as points-to sets are not computed other points-to clients cannot use their results.

Feng et al. [24] propose a bottom-up context-sensitive points-to analysis. The analysis starts analyzing methods without callees, generates generic summaries, and traverses the call graph in a bottom-up manner to apply the summary at call sites. Bottom-up points-to analyses need to analyse methods only once, but are challenging to implement. The actual type of an object for instance, may depend on a caller. Therefore, at a virtual call site, the actual callee method depends on the caller and a summary has to encode all possible execution but should be refined when applied. For refining the applications at application time, the authors present constraint-based summaries based on a storeless heap model that is similar to access paths. The constraints depend on the actual type and alias relationships within callers. Their experiments show a significant improvement in the running time over top-down object-sensitive points-to analysis. The pushdown systems (as used by BOOMERANG) can also use summaries [49]. Extending the pushdown systems summaries by the constraints Feng et al. present is an interesting approach and could yield additional precision and efficiency for BOOMERANG.

5.6.2 Demand-Driven Pointer Analyses

Demand-driven pointer analyses compute answers to pointer queries for data-flow analyses clients. The queries can be triggered for pointer variables (at arbitrary statements) within the program. As for the whole-program points-to analyses, we subdivide this section into flow-insensitive and flow-sensitive analyses.

Flow-Insensitive Analyses

Sridharan et al. [93] present a demand-driven context-insensitive points-to analysis for Java. They introduce the CFL-reachability formulation for pointer analysis in Java: From the analyzed program a pointer assignment graph (PAG) is extracted. The graph's nodes are either variables or allocation sites, the edges between the nodes have labels. For a field-store statement $x.f \leftarrow y$, the graph contains an edge from y to x with a label f. Also for a field load $x \leftarrow y.f$ the graph lists an edge from y to x but labeled by $\overline{f}$. The graph is similar to the graph in Figure 4.7. To determine points-to relations, $flowsTo$-paths are computed within the graph. A $flowsTo$-path is valid and a points-to relation holds, if field stores and loads are properly balanced along the path. Checking for CFL-reachability is expensive and the authors further show how an over-approximation of the context-free language to a regular language computes faster points-to results. BOOMERANG is more precise, because the results are also context and flow-sensitive.

A follow-up work of Sridharan et al. [91] discusses a refinement-based context-sensitive points-to analysis. Existing points-to information is successively refined until an analysis budget is depleted. In this work, the authors also formulate call-site context-sensitivity as a CFL-reachability problem. This formulation roughly corresponds to $\mathcal{P}_{\mathbb{S}}$. The combination context- and field-sensitivity yields to an undecidable analysis [73] (see also Chapter 4.4). Sridharan et al. address undecidability by over-approximating strongly-connected components in the call graph for recursive parts of the program. For the recursive components, the CFL is transformed into a regular language resulting in context-insensitive computations. In our evaluation, we compare Boomerang to this analysis and are able to measure performance and precision improvements.

Yan et al. [104] present a demand-driven alias analysis for Java. As it solely computes information whether two pointer variable alias or not, computation of whole points-to sets is not necessary and saves computation time. The analysis pre-computes intra-procedural symbolic pointer graphs and combines them inter-procedurally to compute alias information of a query. The authors further show how to summarize alias information. Alias information is sufficient for data-race clients, however, for most other clients, pure alias information is ill-suited, e.g., for call-graph construction. We evaluate Boomerang in comparison to this analysis and report performance and precision improvements for a taint analysis client. When applied to a data-race client, the additional precision comes at the cost of efficiency, because Boomerang computes information not needed for the data-race client.

Demand-driven pointer analyses also exist for C programs. Heintze and Tardieu [37] present an Andersen-style context-insensitive pointer-to analysis. Their evaluation shows that demand-driven pointer analysis heavily varies in its computation costs. The authors report 10× speedups on some benchmarks, while for other benchmarks, the demand-driven approach is considerably slower. In particular, on some benchmarks and for some pointer queries, it is necessary to compute the complete points-to graph. This makes the computation as costly as a whole-program analysis. When analyzing programs written in Java with Boomerang, we observed similar results (see Chapter 8): the computation time of a single demand-driven query may vary heavily.

A demand-driven alias analysis for C is presented by Zheng and Rugina in [108]. Similar to Sridharan et al. [93], a CFL-formulation for the alias problem in C is described. The authors report their analysis to be 30× faster for the computation of alias information when compared to a demand-driven points-to analysis.

Flow-Sensitive Analyses

Flow-sensitivity allows performing strong updates and can thus greatly support clients such as typestate analysis [26,27,95]. Despite many applications for these clients, limited research have been published in the area of demand-driven and flow-sensitive pointer analyses.

Sui and Xue [96] propose a flow-sensitive demand-driven pointer analysis for C that is capable of strong updates. The analysis is designed in stages. It pre-computes def-use chains and performs a sparse analysis based on these. Also Shi et al. [85] present a sparse value flow analysis for C which resolves pointer relations on-demand. There analysis is called PINPOINT and is path-sensitive, flow-sensitive and context-sensitive. The idea of sparse analysis can be lifted to Java and BOOMERANG as well. A sparse analysis reduces the number of rules for the pushdown systems $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ and most certainly reduces the analysis time. An interesting open question is, whether a sparse analysis on Java has a similar benefit as on C. We plan to elaborate on sparsity in future work.

Although they do not propose a demand-driven pointer analysis, Guyer and Lin [34] present a framework for client-driven pointer analysis for C programs. The key idea is to monitor the points-to analysis and keep track of imprecision-introducing statements such as control-flow merge points, field-store statements and polymorphic call sites. Once the results are computed, the client decides if the results are precise enough or asks for more precise information on-demand, e.g., for flow- or context-sensitivity. Internally, the analysis then revises the data-flow facts at statements that introduce the imprecision. A similar approach is possible with BOOMERANG, where an imprecise points-to analysis is triggered first.

6 IDE^{al} - Weighted Pushdown Systems

Context-sensitivity, flow-sensitivity, and field-sensitivity are useful precision dimensions for data-flow analyses such as data race and taint analysis. However, there is a broader range of analyses that profit from these dimensions. For instance, while a *typestate analysis* or an analysis to *mine API uses* require flow-sensitivity, having context-sensitivity and field-sensitivity add additional precision.

In this chapter, we propose a general, efficient pointer-tracking data-flow framework called IDE^{al}, which encodes a context-sensitive, flow-sensitive, and field-sensitive data-flow analysis that automatically reasons about aliasing. This chapter focuses on instantiating IDE^{al} for a typestate analysis, however, it can also be instantiated for mining of API uses, linear constant propagation, and many more data-flow analyses.

A typestate analysis expects as input a finite state machine specifying the correct usage pattern for objects of some type. The static analysis then starts from any allocation site that allocates an object of the type. From the allocation site, IDE^{al} follows the data-flow of the object including its aliases. A sound typestate analysis requires following the data-flow of all pointers and the analysis ought to consider field stores and call sites, because they introduce flows to aliases (see Example 12 and Example 14). For these statements, IDE^{al} resorts to demand-driven Boomerang queries.

For typestate analysis [26, 27], it is common to propagate all aliasing access paths as sets within the data-flow fact. This setup enables *strong updates*, which means to simultaneously update typestate information of all aliased pointers when an update occurs on a single pointer of the set. However, this setup results in an inefficient powerset abstraction [66]. For instance, the IFDS framework has a worst-case complexity of $\mathcal{O}(ED^3)$, where E is the size of the inter-procedural control flow graph and D the size of the data-flow domain. An analysis that propagates k-limited aliasing access paths as sets results in the worst-case complexity $\mathcal{O}(E(2^{|\mathbb{V}|\times|\mathbb{F}|^k})^3)$, because a data-flow fact can be any subset of the set of all access path $\mathbb{V} \times \mathbb{F}^k$. Using IDE^{al}, we present a way to avoid this state explosion by propagating aliasing access paths individually while still performing strong updates.

IDE^{al} extends the pushdown system $\mathcal{P}_{\mathbb{S}}$ of Boomerang by weights, i.e., it lifts the pushdown system to a weighted pushdown system (WPDS). In the case of a typestate analysis, the weights encode the typestates of the pointer. IDE^{al} resorts to Boomerang to compute strong updates of typestates for all aliasing access paths.

We originally presented IDE^{al} at the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOP-

SLA) in 2017 [88]. That original version of IDE^{al} was based on the traditional IDE framework [80] and relied on access graphs. The name IDE^{al} reflects that it is based on IDE and automatically resolves *al*iasing. The version we present in this thesis relies on weighted pushdown systems and SPDS instead of IDE. Verbatim parts of our OOPSLA publication are included in this chapter.

6.1 Typestate Weight Domain

A typestate analysis in IDE^{al} is specified in the form of a finite state machine that encodes the usage pattern of the reference-type of interest. The data-flow analysis starts at every allocation site of the type of interests and associates a set of transitions of the finite state machine with each access path at each statement that can point to the allocated object. The transitions are encoded in weights and each rule of $\mathcal{P}_{\mathbb{S}}$ is labeled by a weight. IDE^{al} allows custom weights for different forms of analyses, e.g., typestate or mining API uses. We focus on typestate and formally define the weights for this analysis.

A finite state machine (FSM) that encodes the usage pattern for an object of type X has the form $(\Sigma, S, s_0, \delta, F)$. Σ is the set of methods that may be invoked on an object of type X changing the state of the object, S is the set of all possible states, s_0 is the initial state, δ is the transition function $\delta\colon S \times \Sigma \to S$, and $F \subseteq S$ is the set of accepting states of the finite state machine.

Example 21. Figure 6.1 lists an excerpt of the API of the class `java.util.Vector`. Before accessing the last element, `lastElement()` checks the existence of the element. If the vector is empty, `lastElement()` throws a `NoSuchElementException` in line 231. The API implicitly assumes the vector to be filled by calling `add()` or `insertElementAt()` before `lastElement()` is invoked.

The FSM drawn in Figure 6.1 shows a pattern for the usage of the API. The FSM encodes that a newly allocated `Vector` object resides in the *empty* state (referred to by E) encoding the vector to be empty. The out transition of this state carries the semantic that the first call on the object should be `add()`. The state of the object switches to a state in which the vector is *not empty* (N). An object in this state accepts calls to `add()`, `lastElement()`, or `clear()`. The two methods `add()` and `lastElement()` do not remove elements from the vector, i.e., calling either method cannot transfer the object into the empty state (E) which is encoded as the self-loop edges for state N. However, `clear()` changes the state of the object into E as the call removes all elements of the vector.

Definition 12. *Given a finite state machine T encoding a usage pattern and given S is the set of states of T, then the* typestate weight domain *is the tuple $(D_T, \oplus_T, \otimes_T, \overline{0}_T, \overline{1}_T)$ where each weight $w \in D_T := 2^{S\times S}$ has the form $w = \{(s_1, t_1), \ldots, (s_n, t_n) \mid s_i, t_i \in S\}$, i.e., a weight is a set of (transitive) transitions of T and we thus also write $s \mapsto t$. The binary operator $\oplus_T$ is set union. The extend-operator $\otimes_T$ is defined as follows:*

$$w_1 \otimes_T w_2 := \{s \mapsto u \mid s \mapsto t \in w_1, t \mapsto u \in w_2\}.$$

```java
public synchronized X lastElement() {
  if (this.elementCount == 0) {
    throw new NoSuchElementException();
  }
  return elementData(this.elementCount - 1);
}

//Adds the specific element and increases this.elementCount
public synchronized boolean add(X e) {...}

//Removes all elements, this.elementCount is set to 0.
public void clear() {...}
```

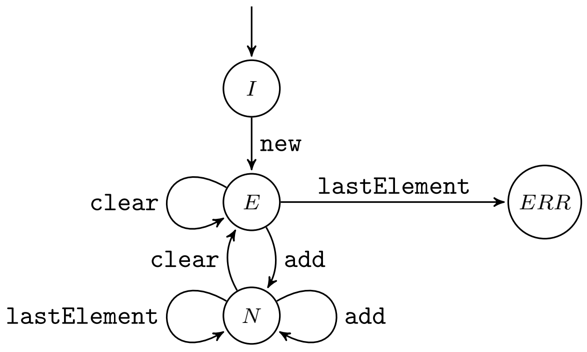
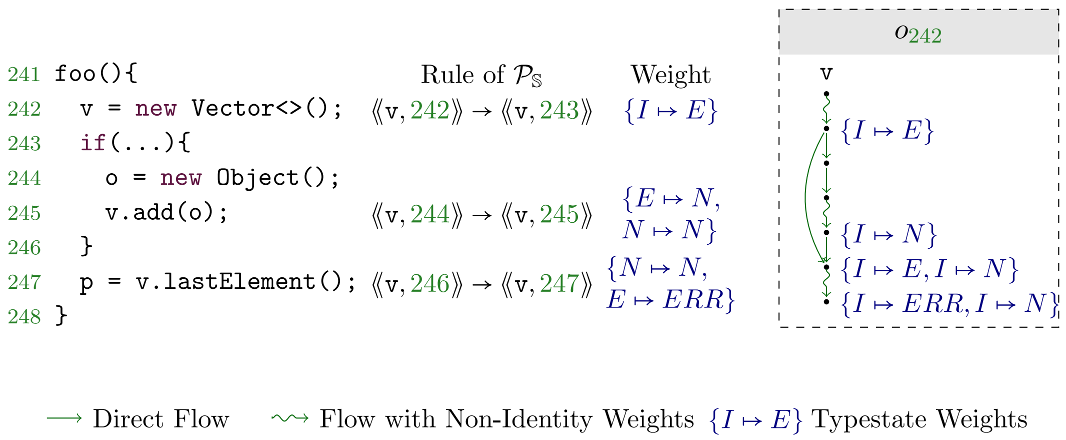


Figure 6.1: An extract of the API for `java.util.Vector` and a state machine encoding the correct usage patterns of the API.



```
241 foo(){
242   v = new Vector<>();
243   if(...){
244     o = new Object();
245     v.add(o);
246   }
247   p = v.lastElement();
248 }
```

Rule of $\mathcal{P}_{\mathbb{S}}$	Weight
$\langle\!\langle$v, 242$\rangle\!\rangle \rightarrow \langle\!\langle$v, 243$\rangle\!\rangle$	$\{I \mapsto E\}$
$\langle\!\langle$v, 244$\rangle\!\rangle \rightarrow \langle\!\langle$v, 245$\rangle\!\rangle$	$\{E \mapsto N, N \mapsto N\}$
$\langle\!\langle$v, 246$\rangle\!\rangle \rightarrow \langle\!\langle$v, 247$\rangle\!\rangle$	$\{N \mapsto N, E \mapsto ERR\}$

o_{242}
v
$\{I \mapsto E\}$
$\{I \mapsto N\}$
$\{I \mapsto E, I \mapsto N\}$
$\{I \mapsto ERR, I \mapsto N\}$
⟶ Direct Flow
⇝ Flow with Non-Identity Weights
$\{I \mapsto E\}$ Typestate Weights

Figure 6.2: An example of a typestate violation for an object of type `java.util.Vector`.

The weight $\overline{0}_T$ is defined as the empty set $\varnothing \in D_T$. Weight $\overline{1}_T$ is the element $\{s \mapsto s \mid \forall s \in S\}$.

The weights for the typestate analysis associate transitions of the FSM to each rule of $\mathcal{P}_{\mathbb{S}}$. Using only sets of states is not sufficient for the purpose of summarization. A method can transform the typestate of an object into two different target states, dependent on the state the object is in at the call site (Details in Example 23). The extend-operator ($\otimes_T$) for two weights is an one-step transitive closure over the transitions, i.e., if the target state t of a transition $s \mapsto t$ of the weight of the left operand (w_1) matches the first state of the transition of the right operand (i.e., $t \mapsto u \in w_2$), the resulting weight $w_1 \otimes_T w_2$ contains the transition $s \mapsto u$.

Example 22. Figure 6.2 shows a code snippet to be checked for the correct usage pattern of class `java.util.Vector`. Method `foo()` executes along two different control-flow paths. The concrete execution path depends on the runtime evaluation of the branch condition of the `if` statement. Both paths use the vector object that is allocated in line 242. When the branch condition evaluates to `true`, the methods `add()` and `lastElement()` are invoked in this order. When the condition is `false`, the program tries to access the vector's last element without ever adding any element to it, which throws a `NoSuchElementException`.

The typestate analysis encoded with IDE^{al} starts to track the pointer `v` pointing to the vector object allocated in line 242. Because the control flow branches, data-flow by-passes all statements within the `if` block. For this data-flow, the *ESG* in Figure 6.2 depicts the bended edge belonging to the normal rule $\langle\!\langle$v, 242$\rangle\!\rangle \rightarrow \langle\!\langle$v, 246$\rangle\!\rangle$.

The pushdown system has the three normal rules that affect the typestate of the object. These three rules[1] are listed in Figure 6.2. The rules are the only rules changing the typestate of the object stored in `v` and all other rules receive

[1]The rules would actually be inter-procedural push rules (`v` flows as `this` local variable into the methods). We model it as intra-procedural flow to simplify the presentation.

```
249 foo(){
250   v = new Vector<>();
251   v.add(new Object());
252   el = accessLastElement(v);
253   v.clear();
254   el = accessLastElement(v);
255 }
256
257 accessLastElement(Vector u){
258   o = u.lastElement();
259   return o;
260 }
```

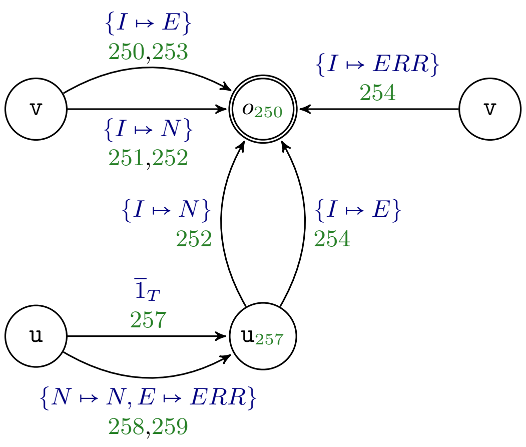


Figure 6.3: Summaries in $\mathcal{A}_{\mathbb{S}}$ require an IDE^{al}-based typestate analysis to propagate transitions as weights.

the identity weight $\overline{1}_T$. The weights that the three non-identity rules receive are given in Figure 6.2 and match transitions in the FSM of Figure 6.1.

In Figure 6.2, the final weights from the post*-saturated weighted $\mathcal{P}$−automaton (see Section 3.6) are shown next to the corresponding *ESG* nodes. The weights are already extended along the path. If a node does not have a weight, it means it has the same weight as the next predecessor node with a weight.

For instance, algorithm post* computes the weight $\{I \mapsto N\}$ for variable `v` at the statement in line 246. At this statement, the object is in state N, at the data-flow propagation start (the `new` call in line 242) the object is in state I.

In the code, the control-flow branches and is joined again (line 246). The weight inside the `if` block for `v` differs from the weight bypassing the branch block. The two weights are combined ($\oplus_T$) at the control-flow join statement in line 246 for which post* computes the weights $w_1 = \{I \mapsto E, I \mapsto N\}$ which is the result of $\{I \mapsto E\} \oplus_T \{I \mapsto N\}$.

The rule associated with the call to `lastElement` in line 247 carries the weight $w_2 = \{E \mapsto ERR, N \mapsto N\}$. Either the object is non-empty and remains non-empty, or the `Vector` is empty and accessing an element fails. The weight reaching line 247 is w_1, when extended by w_2 it is $w_1 \otimes_T w_2 = \{I \mapsto ERR, I \mapsto N\}$ and the `Vector` object may reside in the error typestate after the call to `lastElement`.

In the next example we detail why a typestate weight is a set of transitions and not a set of states.

Example 23. The code in Figure 6.3 depicts a program that first instantiates a `Vector` object (line 250) and adds an element to it (line 251). The next statement in line 252 passes the vector object as argument to the method `accessLastElement()`. Method `accessLastElement()` uses the vector object and calls the typestate changing method `lastElement()` on the object in line 258. After returning from method `accessLastElement()` the control-flow of the program continues in line 253. This statement removes all objects from the vector object. Method `accessLastElement()` is called with the same vector argument a second time in line 253. The second time the method is called, the program throws an exception in line 258 as the vector is empty.

Aside of the code, Figure 6.3 depicts the weighted $\mathcal{P}$-automaton $\mathcal{A}_{\mathbb{S}}$ by which we demonstrate how the pushdown system $\mathcal{P}_{\mathbb{S}}$ uses summaries and does not require to compute data-flow within method `accessLastElement()` twice. We switch from the *ESG* representation of IDE^{al} to the automaton representation as it eases the explanation. The upper three states of the automaton represent the data-flow of `v` within method `foo()`. The lower two states encode the data-flow within `accessLastElement()`. To unclutter the figure, we draw transitions between the same states as separate edges if the transition's weight differ. For instance in `foo()`, for all five statements there are transitions from state `v` to state o_{250}. Three different weights are associated to the five transitions and the automaton lists three edges. For instance, for lines 250 and 253 the weight is $\{I \mapsto E\}$ indicating that the vector is empty after these statements.

We want to highlight the reuse of summaries within `accessLastElement()` which requires typestate weights to carry transitions instead of states. When the data-flow of the vector object o_{250} reaches the call site in line 252, post* creates a new state within the automaton ($\mathtt{u}_{257}$) and post* creates a transition from the newly created state into the state o_{250}. At this call site, the vector is not empty and the transition carries the weight $\{I \mapsto N\}$. This weight is call site dependent information and is not propagated from caller to callee. Therefore, within method `accessLastElement()`, the vector object (stored in `u`) receives the weight $\overline{1}_T$ in line 257. This weight $\overline{1}_T$ explicitly does not encode the state at the call site. The data-flow propagation continues within method `accessLastElement()` and associates the weight $\{N \mapsto N, E \mapsto ERR\}$ for the transition from `u` into $\mathtt{u}_{257}$ labeled by statement 258 (the call to `lastElement()`). The transition with its weight encodes that a vector object entering method `accessLastElement()` as first argument (`u`) transforms the object into the following typestate after statement 258: The object remains in typestate N if in state N at the entry to the method, or the vector object transforms into state ERR if the vector is empty (state E).

In the code example, the vector enters method `accessLastElement()` in typestate N when called under the context of call site 252 and in typestate E via call site 254. The weighted pushdown automaton does not recompute data-flow within `accessLastElement()` and instead reuses existing information for the second call site in line 254. The pushdown system $\mathcal{P}_{\mathbb{S}}$ lists a pop rule $\langle\!\langle \mathtt{u}, 259\rangle\!\rangle \to \langle\!\langle \mathtt{v}, \epsilon\rangle\!\rangle$ that post* applies for both calling contexts. When post* applies the pop rule, the weight returning from the callee is composed with the weight at the call site. One transition that the application of the pop rule constructs is the transition from `v` to o_{250} labeled by 254 and weight $\{I \mapsto ERR\}$. The weight is the composition $\{I \mapsto E\} \otimes_T \{N \mapsto N, E \mapsto ERR\}$. Therefore, IDE^{al} computes the vector object o_{250} to reside in an incorrect state after statement 254.

The change to the incorrect typestate occurs already at statement 258 when called under from statement 254. The automaton also encodes the incorrect typestate statement 258, but one needs to explicitly propagate and compose the weights along the (reverse) paths from the accepting state o_{250} to the node `u`.

The example shows, that modeling the weights as transitions allows reuse of the automaton $\mathcal{A}_{\mathbb{S}}$ and enables summarization.

A weight domain is required to be a bounded idempotent semiring and it is important to prove the four properties from Definition 5 of Section 3. If the properties are not satisfied, the termination of the data-flow analysis is not guaranteed.

Property 1. This property requires $(D_T, \oplus_T)$ to be a commutative monoid with $\overline{0}_T$ as its neutral element, and the operation $\oplus_T$ must be idempotent. Because $\oplus_T$ is set union, $(D_T, \oplus_T)$ directly defines a lattice (also a monoid) with meet operator set union. Commutativity of the lattice is given, because set union commutates. The weight $\overline{0}_T$ is the empty set and is neutral with respect to union. The lattice also requires the operator $\oplus_T$ to be idempotent, which is given, because $w \oplus_T w = w$ for any $w \in D_T$. With respect to the extend-operator $\otimes_T$, $(D_T, \otimes_T)$ is a monoid where the neutral element is $\overline{1}_T$, because it holds that $\overline{1}_T \otimes_T w = w = w \otimes_T \overline{1}_T =$ for any $w \in D_T$. It follows that Property 1 of Definition 5 is satisfied.

Property 2. A bounded idempotent semiring requires $\otimes_T$ to distribute over $\oplus_T$. Therefore, let $a, b, c \in D_T$ be arbitrary and it is:

$$\begin{aligned} a \otimes_T (b \oplus_T c) &= a \otimes_T (\{t \mapsto u \mid t \mapsto u \in b \cup c\}) \\ &= \{s \mapsto u \mid s \mapsto t \in a, t \mapsto u \in b \cup c\} \\ &= \{s \mapsto u \mid s \mapsto t \in a, t \mapsto u \in b\} \cup \{s \mapsto u \mid s \mapsto t \in a, t \mapsto u \in c\} \\ &= (a \otimes_T b) \oplus_T (a \otimes_T c) \end{aligned}$$

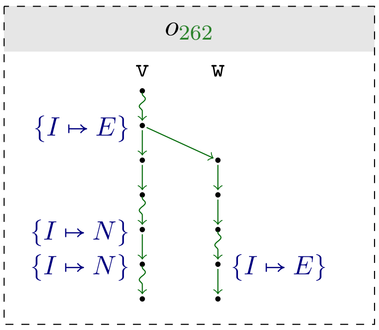
The second equation of Property 2 of Definition 5 follows equivalently.

Property 3. The weight $\overline{0}_T$ must be an annihilator with respect to $\otimes_T$, which means every element extended by $\overline{0}_T$ turns into the empty set. Because $\overline{0}_T$ is the empty set, $w \otimes_T \overline{0}_T = \overline{0}_T = \overline{0}_T \otimes_T w$ for any $w \in D_T$, which makes $\overline{0}_T$ an annihilator and Property 3 of Definition 5 is shown.

Property 4. The partial order defined by $\oplus_T$ is the partial order of set inclusion. A finite state machine has a finite set of states. Therefore, the maximal element is the weight $\top := \{(s,t) \mid \forall s,t \in S\}$. For any weight $w \in D_T$, $\top \oplus_T w = \top$. Therefore, there cannot be an infinite descending chain, which proves Property 4 of Definition 5.

All four properties are satisfied and $(D_T, \oplus_T, \otimes_T, \overline{0}_T, \overline{1}_T)$ is a weight domain for typestates, which means the weight domain can be used in IDE^{al}.

```
261 foo(){
262   v = new Vector<Object>();
263   w = v;
264   o = new Object();
265   v.add(o);
266   w.clear();
267   p = v.lastElement();
268 }
```



$\longrightarrow$ Direct Flow $\rightsquigarrow$ Flow with Non-Identity Weights $\{I \mapsto E\}$ Typestate Weights

Figure 6.4: An example illustrating the challenge that a non set-based data-flow domain encounters with aliasing: Typestate updates on aliasing variables are not reflected.

6.2 Strong Updates and Points of Aliasing

To achieve efficiency, IDE^{al} propagates access paths that point to the same objects individually. Each data-flow fact does *not* hold a set of aliasing pointers, instead, each data-flow fact is a single access graph (encoded as SPDS). This individual propagation is the same as for BOOMERANG, and IDE^{al} also requires the points of aliasing of BOOMERANG (Chapter 5). The points of aliasing transfer updates on weight to aliasing data-flow facts and enable strong updates.

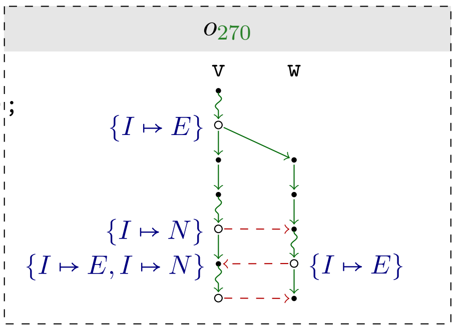
Example 24. A typestate change on one pointer can be strongly updated on a second pointer if the two pointers are must-aliases. Performing strong updates when pointers are propagated individually is non-trivial as illustrated in Figure 6.4.

The execution of the code in Figure 6.4 accesses an empty vector and terminates in an exception. The code does not satisfy the typestate usage pattern of `Vector`. The program allocates a `Vector` object in line 262. Starting from line 263, there are two aliased variables that point to the object. While variable `v` is used to add (line 265) and access the elements (line 267), between the two calls all vector's elements are removed by the call to `clear()` in line 266. However, `clear()` is invoked on the alias `w`.[2]

The call to `clear()` on variable `w` must strongly update the typestate of pointer variable `v`. IDE^{al} handles these updates, but a typestate analysis that does *not* strongly update the typestate computes unsound results. We demonstrate a typestate analysis without strong updates on the data-flow graph in Figure 6.4. Variables `v` and `w` are propagated individually, and at the call to `lastElement` in line 266, the analysis assumes variable `v` holds a non-empty vector, because the resulting weight for variable `v` is $I \mapsto N$. Consequently, at the call to `lastElement()`,

[2]This example is contrived. However, in actual code, a similar pattern occurs when an object is stored and loaded from a field; the stored and loaded variables may alias.

```
269 foo(){
270   v = new Vector<Object>();
271   w = v;
272   o = new Object();
273   v.add(o);
274   w.clear();
275   p = v.lastElement();
276 }
```



⟶ Direct Flow ⇝ Flow with Non-Identity Weights $\{I \mapsto E\}$ Typestate Weights

⇢ Indirect Flow ∘ Non-Identity Point of Aliasing

Figure 6.5: Indirect flows at non-identity weights updates reconnect data-flow when a typestate change occurred.

the analysis wrongly assumes the vector to be non-empty, and the analysis incorrectly misses to report a typestate error.

Why does the analysis not detect the error? The call to `clear()` in line 266 is invoked on pointer variable `w`, and the analysis does not update the typestate for the aliasing variable `v` at this statement. The data-flow graph shows that the propagation paths of `v` and `w` are not connected and typestate changes on `w` are not reflected to `v`, though both variables alias.

The discussed problem is a drawback of the individual, distributive propagation of aliasing access path. Existing solutions [26,27,66] that keep track of all aliasing variables in a set (powerset abstraction) can strongly update all aliasing pointers carried within the data-flow fact, however, they also cause a state explosion of the data-flow domain. To avoid the state explosion of the data-flow domain, we propose a solution that maintains the efficiency of distributive propagation but can perform strong updates at the same time.

In Chapter 5, we discussed the concept of points of aliasing. For instance, at field-store statements, indirect flow edges are introduced to model data-flow to fields of aliases. The aliases are computed for the base variable of the store statement. This concept can similarly be applied to perform strong updates of typestates. At any statement that performs a typestate update on a pointer variable, indirect flow edges to aliases of the variable that receives the update are computed and added. Weights on the indirect flow edges transport the updated typestate information to the aliases.

Apart from the call-site and field-store points of aliasing, IDE^{al} generates a point of aliasing *whenever* a weight of a rule of $\mathcal{P}_{\mathbb{S}}$ is a *non-identity weight* (a weight other than $\bar{1}$). These rules change the propagated weight and the weight update must be updated on aliases. We refer to these points of aliasing as *non-identity point of aliasing*. In the case of the typestate weights, rules associated with a statement updating the typestate of an object are labeled by a non-identity weight.

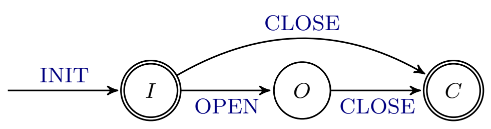


Figure 6.6: A finite state machine that encodes the correct usage of a `File` API. Any file object must be in a state marked with a double circle before it is destructed.

Example 25. Figure 6.5 shows how non-identity points-of-aliasing enhance the analysis to soundly report the typestate error. In the figure, rules of $\mathcal{P}_{\mathbb{S}}$ with non-identity weights are highlighted as meandered edges. The data-flow analysis generates three nodes that are non-identity points of aliasing: `v`@273, `w`@274 and `v`@275.

Each non-identity point of aliasing triggers a points-to query. The point of aliasing `v`@273, for instance, triggers a pointer query to Boomerang to find all aliases of variable `v` at statement in line 273. Boomerang returns that the pointer variable is allocated in line 270 and all aliasing access paths are the plain variables `v` and `w`. For each alias, an indirect flow is added. The indirect flow edges are highlighted as dashed red edges in Figure 6.5.

The additional indirect edges receive identity weights ($\overline{1}_T$), which means that the analysis propagates the weights reaching the point of aliasing to all the aliases. Hence, a node of an alias may have multiple incoming data-flow edges and their weights are combined ($\oplus$) as it is done for control-flow merge statements. For instance, node `v`@274 has the weight $\{I \mapsto E, I \mapsto N\}$. The indirect flow edge propagates the additional transition $I \mapsto E$ to the node. Because of the additional transition, the typestate analysis reports the usage of the `Vector` to violate the typestate usage pattern and the analysis reasons that `lastElement` is invoked on an empty vector.

While the analysis soundly reports the typestate violation, it also introduces an imprecision. The weight for node `v`@274 contains the transition $I \mapsto N$ which means the `Vector` object stored in `v` may not be empty. The vector is empty along all execution paths of the program as the vector reference is cleared at the instruction before. The problem is that, so far, the analysis does not strongly update the typestates of the object and thus computes typestate information that is imprecise.

In the example, the imprecision does *not* generate a false warning, because the analysis only reports when an object is in the *ERR* state. However, as we show next, for other typestate properties, this imprecision may introduce a false positive.

Example 26. Assume a `File` API with two methods `open()` and `close()`. To avoid resource leaks [99], any opened file must be closed. Figure 6.6 shows the FSM encoding the typestate problem. The FSM has two accepting states (highlighted with double circles). The accepting states mark states that the object is expected to be at destruction time, i.e., before the JVM garbage collector

```
277 foo(){
278   a = new File();
279   b = a;
280   a.open();
281   b.close();
282 }
```

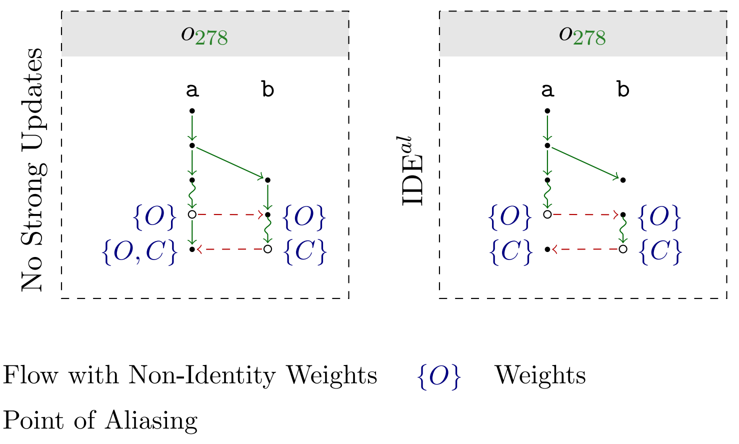


Figure 6.7: A typestate analysis without strong update delivers imprecise results (left), the object stored in `a` remains opened. IDE^{al}, which performs strong updates (right) deliver precise results. To simplify the figure, we omit the start states of all transitions (I) in the weights.

reclaims the reference. Along any execution path, the object is either closed, or it had never been opened.

Figure 6.7 depicts a minimal example for an analysis not performing strong updates reports imprecise results. First, consider the left box plotting the data-flow for o_{278} which is labeled as *No Strong Updates*. The code snippet allocates a `File` object at line 278 within method `foo()`. The object never escapes `foo()`, because it is not stored on the heap, not returned from `foo()`, and not used at a call site. The object's lifetime is bound to `foo()`, and after execution of the method, the garbage collector destroys the reference. The program correctly uses the file object, because it is eventually closed.

Similar to the `Vector` example, the methods `open()` (line 280) and `close()` (line 281) are invoked on two aliased variables `a` and `b`. The distributive propagation and re-connection via the indirect flows yields the weight $\{I \mapsto O, I \mapsto C\}$ for variable `a` at the end of method `foo()`. The typestate analysis *incorrectly* outputs that the object is in the non-accepting state O, and the analysis reports a false positive.

The imprecision is caused by the weight $I \mapsto O$ that bypasses the `close()` invocation on pointer variable `b` in line 281. The weight bypasses the call due to the direct flow edge `a`@280 to `a`@281. The edge is labeled by $\bar{1}_T$, and the weight $\{I \mapsto O\}$ propagates to the target node `a`@281. At the target node, the weight $\{I \mapsto O\}$ is combined with the weight $\{I \mapsto C\}$ that flows along the indirect flow edge `b`@281 to `a`@281.

The analysis misses to strongly update the typestate. In Figure 6.5, next to the example without strong updates, the data-flow for IDE^{al} including its strong updates mechanism is drawn. The basic idea of IDE^{al} to perform the strong update is to remove spurious data-flow rules (rules for edge `b`@280 to `a`@281 and edge `a`@281 to `a`@282).

Technical View

We now elaborate on how and which spurious pushdown rules IDE^{al} removes. The data-flow analysis of IDE^{al} (weighted post*) is a *monotonic* and *chaotic* fixed-point iteration, i.e., transitions are successively added until the weighted $\mathcal{P}$−automaton is saturated and the computation thus reaches a fixed-point. The propagation is monotone, i.e., information can only be added but not removed, because post* requires monotonicity to guarantee termination. The fixed-point iteration is chaotic, because the order of the insertion of the edges is irrelevant. However, deterministic results are guaranteed.

During the computation of the fixed-point, removing transitions from the automaton (or respective rules from the PDS) either yields non-deterministic results or even prevents termination. Therefore, IDE^{al} executes in two phases: Phase I computes the fixed-point and does not remove any pushdown rules. During this phase, a set of *removable rules* is computed. Phase II re-computes the fixed-point but omits the rules marked removable. Because the rules are not added, the weights cannot flow along the removable rules in Phase II and the fixed-point in Phase II differs from the one of Phase I.

The set of removable rules depends on the non-identity points of aliasing and the computed aliases. A point of aliasing is generated for each (normal, push, or pop) rule $\langle\!\langle v, s\rangle\!\rangle \to \langle\!\langle w, t\rangle\!\rangle$ of $\mathcal{P}_{\mathbb{S}}$ with a non-identity weight. The point of aliasing triggers a Boomerang pointer query for the data-flow fact w at statement t, because (1) statement s changes the weight for pointer w and (2) the weight change is valid for any alias of w. The Boomerang query's result is a map of allocation sites of variable w. For each allocation site a Boomerang returns the set $aliases_a(t)$ that encodes all aliasing access paths of w at statement t. Based on the latter set, IDE^{al} generates indirect-flow edges to the aliases. For any alias $\widetilde{w} \in aliases_a(t)$, with $w \neq \widetilde{w}$, $\mathcal{P}_{\mathbb{S}}$ receives the rule $\langle\!\langle w, t\rangle\!\rangle \to \langle\!\langle \widetilde{w}, t\rangle\!\rangle$. The additional rule models the indirect flows (in Figure 6.5 depicted as red, dashed edges).

The set of removable rules depends on the size of the points-to set of the Boomerang query. Only if Boomerang returns a *single* allocation site, the rules $\langle\!\langle \widetilde{w}, s\rangle\!\rangle \to \langle\!\langle \widetilde{w}, t\rangle\!\rangle$ are removable rules during Phase II.

Example 27. IDE^{al} performs a strong update only when the the points-to set for the pointer query at the point of aliasing reports a *single* allocation site. Performing a strong update when the points-to set is larger would be unsound, as the example in Figure 6.8 shows. The code snippet allocates two `File` objects, one in line 284, the other in 285. Assume an analysis that follows the object o_{285}. The typestate depends on the branch condition in line 287. The `File` object o_{285} is either in the opened state (O), or in the closed state (C).

Next to the code, Figure 6.8 depicts the data-flow graph that IDE^{al} computes. The graph that belongs to o_{285} marks the edge belonging to the normal rule $\langle\!\langle \mathtt{a}, 289\rangle\!\rangle \to \langle\!\langle \mathtt{a}, 290\rangle\!\rangle$ as weak update. If IDE^{al} would strongly update the typestate and remove the edge. it would incorrectly compute that the object must be in a closed state. However, when the program executes and the code within the if block is not executed, the variables `a` and `b` do not aliased and the object pointed-to by `a` remains open.

```
283 foo(){
284   b = new File();
285   a = new File();
286   a.open();
287   if(...){
288     b = a;
289   }
290   b.close();
291 }
```

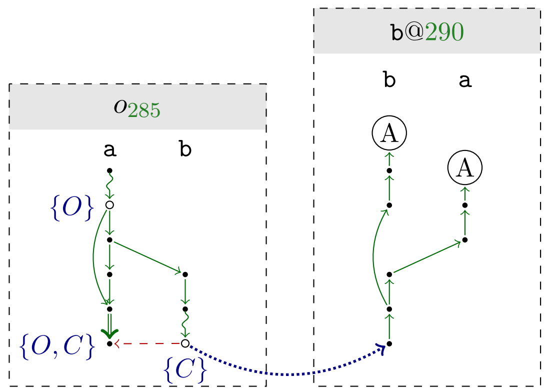


Figure 6.8: A typestate update on a variable that has two allocation sites cannot be strongly updated for aliasing variables. To simplify the figure, we omit the start states of all transitions (I) in the weights.

In this example, IDE^{al} does *not* perform a strong update. This is because the Boomerang-query for the point of aliasing triggered in line 290 returns that variable `b` may point to two objects, o_{285} and o_{284}, depending on which path is executed. When `b` points to the object o_{284}, the call to `close()` is not made on object o_{285} but on object o_{284}. Therefore, the typestate for o_{285} when `a` bypasses the call to `close` in 290 *cannot* be strongly updated. The data-flow graph for the backward analysis of the Boomerang query is also shown in Figure 6.8. The data-flow graph shows that `b` may point to two different allocated objects.

Loops, Recursion and Arrays

There are several cases in which IDE^{al} cannot perform strong updates because IDE^{al} would compute unsound results otherwise. An object that is propagated within a loop or a recursive part of the program cannot receive a strong update as the same variable may point to an object of an earlier iteration of the repeated execution.

Example 28. Figure 6.9 depicts a minimal code example of a program for which a strong update of typestate information is not possible in line 300.

Method `foo()` first creates a new `File` object in line 293 by calling the factory method `createFile()`. The newly created file object is stored within variable `b`. The program then executes a `while` loop which, per iteration, creates another `File` object by calling the factory method again (line 295). The instantiated object is stored in variable `a`. On variable `a` method `open()` is invoked in line 296. Dependent on the branch condition of statement 297, statement 298 re-assign variable `b` to point-to the object of `a`. Eventually, the loop closes the object

```
292 foo(){
293   b = createFile()
294   while(...){
295     a = createFile();
296     a.open();
297     if(...){
298       b = a;
299     }
300     b.close();
301 }
```

```
302 createFile(){
303   x = new File();
304   return x;
305 }
```

Figure 6.9: Typestate of object residing in loops cannot be strongly updated.

stored in variable `b` in line 300. When the code within the `if` block is not executed during runtime, the `File` object stored in variable `a` will remain open and a typestate analysis is expected to produce a warning.

IDE^{al} produces a warning as the analysis does not perform a strong update and the file may remain open. At the call to `close()` in line 300, IDE^{al} computes the allocation sites of `b` and BOOMERANG finds the allocation site in line 303. The allocation site returns unbalanced to the call sites of the factory method in lines 293 and 295 within method `foo()`. In the context of `foo()` BOOMERANG computes two allocation sites to be reachable, one via each call site of the factory (see Section 5.4). Therefore, the size of the points-to set is larger than 1 and the analysis does not perform a strong update. IDE^{al} correctly approximates that the `File` object stored in variable `a` may remain open.

Similarly, a strong update can also not be performed, when the object data-flow propagates through an array. BOOMERANG and IDE^{al} are array-insensitive and merge all objects that flow into an array. Array stores and loads are modeled as flows through the synthetic field *ARRAY*. Before IDE^{al} performs a strong update it checks if the data-flow was propagated through this specific field.

6.3 Weight Domain for API Usage Pattern Mining

Typestate analysis is an instance of the weights that IDE^{al} can be instantiated with. The general concept of IDE^{al}, the indirect flows and the respective strong updates, also apply to any other feasible weight domain in the literature. For instance, earlier work discussed the weight domain for linear-constant propagation and a domain for the shortest witness path of a data-flow connection [76]. IDE^{al} can instantiate data-flow problems with these weight domains and track data-flows through the heap automatically.

Another example weight domain for IDE^{al} is the weight domain for *API usage pattern mining*. To the best of our knowledge, we are the first to present this weight domain. Mining the specification of an API is helpful to automatically detect bugs [71]. The specification is also helpful to support developers unfamiliar with an API [109] or to derive the FSM for a typestate analysis.

Therefore, we present the design of a weight domain for usage pattern mining in this section. Starting from the allocation statements, the usage pattern analysis traces the object along all potential execution paths and collects the *method call sequence* for each object. The call sequence is the sequence of calls that an object receives. Technically, a method call sequence is a sequence $m_1, \ldots, m_n$ with $m_i \in M$, where M is the set of declared methods invocable on the tracked object.

Statically, loops and recursive program structures generate an execution path of infinite length and infinitely long call sequences may be inferred by the analysis. As a finite representation is required to guarantee termination of the analysis, when a method invocation occurs within a loop, the static analysis over-approximates the number of times the method is invoked, by switching from call sequences to *call sets*. A call set is a subset $S \subset M$ and over-approximates a call sequence. A call set does not preserve the order in which the invocations occur along the control-flow. The set-based representation also ignores the multiplicity of each method call, i.e., it does not track the times each invocation occurred along an execution path.

In the following, we use the notation $s_{\{\}}$ to refer to a call set, an individual element is denoted as $s_{\{i\}} \in M$. A call sequence is marked by the subscript $s_{[]}$, to refer to an individual element we use $s_{[i]} \in M$ accordingly.

Definition 13. *Assume M to be the set of all declared methods for some object. The* pattern inference weight domain *is the tuple $(D_I, \oplus_I, \otimes_I, \overline{0}_I, \overline{1}_I)$. Each weight $w = \{x^1_{[]}, \ldots, x^n_{[]}, y^1_{\{\}}, \ldots, y^m_{\{\}}\} \in D_I$ is a set of call sequences $x^j_{[]}$ and call sets $y^j_{\{\}}$. The binary operator $\oplus_I$ is set union. The extend operator $\otimes_I$ is defined as follows:*

$$w_1 \otimes_I w_2 := \{w_1^i \otimes_I^e w_2^j \mid w_1^i \in w_1, w_2^j \in w_2\}$$

where

$$x \otimes_I^e y := \begin{cases} (x_{[1]}, \ldots, x_{[n]}, y_{[1]}, \ldots, y_{[m]}) & \text{if } x, y \text{ call sequences and } \forall i, j : x_{[i]} \neq y_{[j]} \\ \{x_{[1]}, \ldots, x_{[n]}\} \cup \{y_{[1]}, \ldots, y_{[m]}\} & \text{if } x, y \text{ call sequences and } \exists i, j : x_{[i]} = y_{[j]} \\ \{x_1, \ldots, x_n\} \cup \{y_1, \ldots y_m\} & \text{otherwise} \end{cases}$$

The weight $\overline{0}_I$ is defined as the empty set $\emptyset \in D_I$. Weight $\overline{1}_I$ is the empty call sequence *ϵ, a sequence of length* 0*.*

Example 29. Figure 6.10 outlines the weight domain on an example. The code snippet instantiates a `Vector` object (o_{307}) in line 307 and invokes the methods `add()`, `clear()`, and `get()` on the object in line 309, line 311, and line 314 respectively. The code contains two control-flow branching statements. The `if` statement in line 310 and the loop construct in line 313.

Figure 6.10 illustrates the analysis results of API usage pattern mining based on IDE^{al}. For instance, the weight at the control-flow join statement in line 312

```
306 foo(){
307   v = new Vector<Object>();
308   o = new Object();
309   v.add(o);
310   if(...){
311     v.clear();
312   }
313   for(i < ...){
314     v.get(i);
315   }
316 }
```

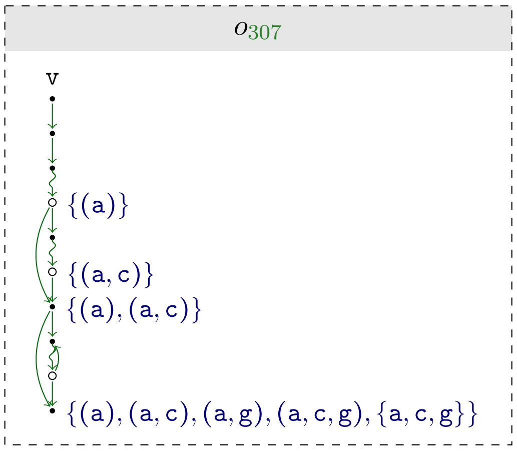


Figure 6.10: Illustrating the pattern inference domain.

for variable `v` is $\{(\underline{\texttt{a}}\texttt{dd}), (\underline{\texttt{a}}\texttt{dd}, \underline{\texttt{c}}\texttt{lear})\}$[3]. The weight consists of two call sequences. The first call sequence lists the method `add()`. The second call sequence lists the methods `add()` and `clear()`. Semantically, the weight summarizes the methods that are invoked from the allocation site until the respective statement. The object o_{307} flows along two paths to statement 312. Along one path, the only method invoked on o_{307} is `add()`, along the second path, the call to `add()` is followed by a call to `clear()`.

At the statement in line 316, the weight additionally consists of the elements $\{\ldots, (\texttt{a}, \texttt{g}), (\texttt{a}, \texttt{c}, \texttt{g}), \{\texttt{a}, \texttt{c}, \texttt{g}\}\}$. All elements of the weight end in a method call to $\underline{\texttt{g}}\texttt{et}$.

The call sequence $(\texttt{a}, \texttt{c}, \texttt{g})$ results from the execution path such that the branch condition at statement 310 evaluates to *true*, and the `for` loop is executed exactly once.

The call set $\{\texttt{a}, \texttt{c}, \texttt{g}\}$ is the result of the loop construct between lines 313 and 315. Due to the loop, the control-flow graph contains a backward edge from statement 314 to 313. The backward edge is also generated in the data-flow graph for o_{307} and the graph contains a cycle. Within this cycle, the object receives a call to `get()` and the call sequence that the data-flow algorithm constructs for one iteration of the loop $(\texttt{a}, \texttt{c}, \texttt{g})$ is propagated a second time to the loop start. The call sequence is extended $(\otimes_I^e)$ by the call sequence $(\texttt{g})$. Because both call sequences contain the method invocation of `g`, the call sequence is over-approximated by the call set $\{\texttt{a}, \texttt{c}, \texttt{g}\}$.

In the following, we prove the properties for a weight domain given in Definition 5.

[3]To keep the weights in Figure 6.10 readable, the called methods listed in the weights are abbreviated by their first letters.

Property 1. The operator $\oplus_I$ is set union and $\oplus_I$ inherits commutativity. The operator $\oplus_I$ is also idempotent as $w \oplus_I w = w$ for any $w \in D_I$. It follows that the pair $(D_I, \oplus_I)$ defines a commutative monoid with neutral element $\overline{0}_I$. The pair $(D_I, \otimes_I)$ is also a monoid with neutral element $\overline{1}_I$ as it is $\overline{1}_I \otimes_I w = w = w \otimes_I \overline{1}_I =$ for any $w \in D_I$.

Property 2. Let $a, b, c \in D_I$ be arbitrary, then

$$\begin{aligned} a \otimes_I (b \oplus_I c) &= a \otimes_I (\{y \mid y \in b \cup c\}) \\ &= \{x \otimes_I y \mid x \in a, y \in b \cup c\} = (a \otimes_I b) \oplus_I (a \otimes_I c) \end{aligned}$$

because $\otimes_I$ is defined element-wise (see Definition 13). Similarly, $(a \oplus_I b) \otimes_I c = (a \otimes_I b) \otimes_I (a \otimes_I c)$.

Property 3. For every element $w \in D_I$, $w \otimes_I \overline{0}_I = \overline{0}_I = \overline{0}_I \otimes_I w$ and $\overline{0}_I$ is an annihilator.

Property 4. Every program has a finite amount of statements, any call sequence may only be as long as the longest acyclic path in the program. Therefore, the call sequences cannot introduce infinite descending chains. The size of each call set is limited, because the set representation lists each invocation at most once. Therefore, the weight domain does fulfill the infinite descending chain property.

All four properties are satisfied, and the API usage pattern mining weights form a weight domain.

6.4 Evaluation

We extended the implementation of Boomerang (Section 5.5) to support non-identity points of aliasing and strong updates, and call this extension IDE^{al}. The extension is also publicly available[4] as part of the implementation of Boomerang. In this evaluation, we assess IDE^{al} based on this implementation. We instantiate IDE^{al} based on a typestate analysis problem and compare the analysis (TS^{al}) to an existing typestate analysis with a similar feature set.

Fink et al. [27] present a typestate analysis (TS^{f}) that verifies typestate properties including aliasing. TS^{f} runs in multiple stages. The early stages are less precise and prune out impossible data-flows such that analysis time is saved in later, less efficient stages. All stages of the analysis TS^{f} are based on the IFDS framework. The last stage of TS^{f} is expected to have comparable precision to TS^{al}. An important difference between TS^{f} and TS^{al} is their heap model. To group aliased pointers and track their shared typestate, TS^{f} uses a powerset abstraction within their data-flow domain. IDE^{al} is designed to propagate aliasing pointers individually and avoids the powerset abstraction.

[4] `https://github.com/CROSSINGTUD/WPDS`

Table 6.1: Typestate properties used for this evaluation.

Name	Description
`Vector`	Never try to retrieve an element from an empty `Vector`.
`Iterator`	Always call `hasNext()` before `next()` on an `Iterator`.
`URL`	Never set options on an already connected `URLConnection`.
`IO`	Do not read from or write to a closed `Stream` or `Writer`.
`KeyStore`	Always initialize a `KeyStore` before using it.
`Signature`	Always follow the phases of initialization of a `Signature`.

The analysis TS^f is publicly available[5] and is based on the program analysis framework WALA[6]. The default configuration of TS^f performs three analysis stages. During our initial experiments, we discovered that the first stage does not report any typestate violation, preventing any computation of subsequent stages. We consulted with the authors of TS^f who confirmed this behaviour and were unable to fix the problem. Instead of using the staged solver, we compare to the second and third stages directly.

Both typestate analyses, TS^f and TS^{al}, take finite state machines as input to check for typestate violations. Table 6.1 list all typestate properties we used for this evaluation.

Based on the typestate analysis, we assess IDE^{al} and ask the following research questions:

- **RQ1**: What is the effect of the difference in the heap models TS^{al} and TS^f on a micro-benchmark?
- **RQ2**: How does TS^{al} perform on large programs when compared to TS^f in terms of computation time?
- **RQ3**: How precise are the analysis results reported by TS^f and TS^{al}?
- **RQ4**: What impact do aliasing and strong updates have on TS^{al}?

6.4.1 Heap Model Performance on a Micro-Benchmark

In this section, we compare the two analyses TS^{al} and TS^f. Both analyses are field-sensitive and model flows through the heap, i.e., precisely model field-store and field-load statements. Yet, both analyses model the heap flows entirely differently. We compare the models in terms of their precision and recall, but also in terms of their efficiency, i.e., how many data-flow propagations are required.

The heap model of TS^{al} is encoded as the field automaton $\mathcal{A}_{\mathbb{F}}$ computed based on the pushdown system $\mathcal{P}_{\mathbb{F}}$. The automaton concisely represents all access paths at each statement. The access path encodes how the data-flow object is referenced.

[5]`https://github.com/tech-srl/safe`
[6]`http://wala.sourceforge.net/`

The analysis TS^f uses a different heap model in each stage. The second stage *APUnique* of TS^f, hereafter referred to by TS^f_2, computes objects that are *uniquely* allocated, i.e., the allocation site does not reside within a loop or a recursive invocation and during execution the program allocates a unique object at this site. For these allocations, any typestate update is always strongly updated [27].

The third stage *APFocus*, hereafter referred to by TS^f_3 [27], is more precise than TS^f_2 and uses the following data-flow abstraction to model the heap: Each data-flow element consists of (1) the *allocation site* of the tracked instance, (2) a *set of must-aliased pointers* (k−limited access paths) to that allocation site, (3) a *completeness* flag indicating whether the set of must-aliased pointers is complete, and (4) the *state* that the object is currently in. The set of must-aliased pointers enables performing strong updates, because all aliasing pointers are kept in one data-flow element. If one of the pointers typestate is updated, it reflects to all other pointers. The data-flow domain used by TS^f_2 is a powerset abstraction, because each data-flow fact contains a set of access paths. Therefore, the number of created data-flow facts grows exponentially. In IDE^{al}, aliases are propagated individually, however, strong updates for IDE^{al} require additional Boomerang queries.

Experimental Setup. To inspect the differences between the heap models of TS^f and TS^{al}, we ran both analyses (for TS^f both stages TS^f_2 and TS^f_3) on a set of micro-benchmarks that consists of 72 sample programs. These programs ship with the implementation of TS^f. The micro-benchmarks contain typestate violations with aliasing and strong update scenarios that challenge typestate analysis. Hence, it is a good baseline for comparison. We have applied both TS^f and TS^{al} to check for the typestate properties in Table 6.1 in these micro-benchmark programs. We evaluate the heap models efficiency and count how many methods are *visited* by the data-flow analyses. A method is visited, if at least one data-flow fact is generated within the method. We consider an analysis to be more efficient if less methods are visited. TS^f and TS^{al} are based on different static analysis frameworks, WALA and Soot, and also rely on different data-flow frameworks, IFDS and synchronized pushdown systems. Hence, we find that a one-to-one comparison based on the visited methods of the data-flows is the most objective metric. We also carefully configured Soot and WALA to enable a fair comparison for both analyses, including using call graphs of similar precision.

In this experiment, we also compare the precision and recall of all analyses. The micro-benchmark contains labels indicating the ground truth for the checked typestate properties on the programs. For each program, the number of typestate violations are specified explicitly. Based on this specification, we evaluate the analyses precision and recall.

Performance Results. Table 6.2 summarizes our findings. In the table, the columns for *Visited Methods* gather the number of methods the underlying solvers

Table 6.2: Comparing the efficiency of TS^f_2, TS^f_3, and TS^{al} in terms of the number of visited methods.

		Visited Methods		
Typestate	**# Programs**	TS^f_2	TS^f_3	TS^{al}
`KeyStore`	3	461	345	3
`Iterator`	17	44	38	6
`URL`	2	587	514	2
`Vector`	30	98	63	15
`IO`	14	333	155	12
`Signature`	6	2,901	2,817	8

have propagations in. The numbers are geometric means taken over all input programs (column *# Programs*) of the micro-benchmark. Table 6.2 shows the statistics for TS^f according to the two analysis stages TS^f_2 and TS^f_3.

Since TS^f_3 is the most precise stage in TS^f, we only compare TS^{al} to this stage in the following discussion. The numbers and arguments for TS^f_2 are similar. The analysis stage TS^f_3 has a noticeable difference in the amount of visited methods. Across all micro-benchmark programs, TS^f_3 requires visiting a geometric mean of 38.1× more methods compared to TS^{al}.

The typestate properties `Signature` and `URL` show the most significant difference in the number of visited methods. TS^f_3 visits 2,817 methods for `Signature`, and 514 methods for `URL`. In contrast to that, TS^{al} requires only to visit 8 and 2 methods for the same typestate properties, respectively. The extreme difference is explained through the following. For `URL`, TS^{al} starts from the call sites to the method `connect()` of any `URLConnection` object and reports an error once a method that sets an option on the object is invoked. In contrast, TS^f_3 starts earlier at the allocation site of the `URLConnection` itself. TS^f_3 records aliases only during forward propagation, while TS^{al} gets the automatic support from IDE^{al} to detect aliases before the seeds by issuing the appropriate alias queries to Boomerang. IDE^{al} evaluates those queries on demand, which requires visiting fewer methods in the forward propagation. The same reasoning applies to the typestate property `Signature`.

For the remaining typestate properties, both analyses, TS^f and TS^{al} start at the same allocation sites and one could expect them to visit the same number of methods. However, we noticed that the underlying heap models constitute to a drastic difference. For TS^{al}, each data-flow propagation is bound to a local variable. With synchronized pushdown systems, each data-flow fact consists of a local variable plus a field automaton that represents the field accesses that the tracked object resides in. A local variable is restricted to the method that it is declared in and any data-flow associated with that local variable must only be propagated within that particular method. The data-flow abstraction used by TS^f cannot make use of this additional information.

		TS_3^f
317	`foo(){`	
318	`  A a = new A();`	
319	`  b = a;`	
320	`  B c = new B();`	$\langle 320, \{\texttt{c}\}, \mathit{true}\rangle$
321	`  a.f = c;`	$\langle 320, \{\texttt{c}, \texttt{a.f}\}, \mathit{false}\rangle$
322	`  B d = b.g;`	$\langle 320, \{\texttt{c}, \texttt{a.f}\}, \mathit{false}\rangle$
323	`  bar(d);`	$\langle 320, \{\texttt{c}, \texttt{a.f}\}, \mathit{false}\rangle$
324	`}`	

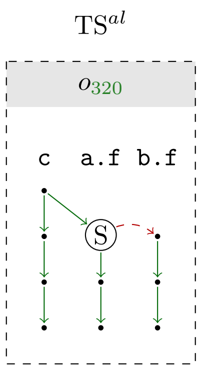


Figure 6.11: An example illustrating the differences between TS_3^f and TS^{al} with respect to the structure of data-flow facts.

We illustrate this behaviour through the example in Figure 6.11. We ignore the propagated typestate property to simplify the example. Assume both TS^{al} and TS_3^f to track the object that is created at line 320. After line 321, TS_3^f propagates the abstraction $\langle 320, \{\texttt{c}, \texttt{a.f}\}, \mathit{false}\rangle$. The *completeness* flag is set to *false*, because after the field-store statement, the tracked object is also accessible via the pointer `b.f`, which is not in the set of must-aliased pointers $\{\texttt{c}, \texttt{a.f}\}$. The representation does not explicitly store `b.f`, and TS_3^f has to assume that the tracked object could also be accessed in method `bar` (called at line 323), although no appropriate pointer ever escapes to the method. Therefore, TS_3^f propagates the data-flow fact $\langle 320, \{\texttt{c}, \texttt{a.f}\}, \mathit{false}\rangle$ into `bar`, needlessly increasing the number of propagations. On the other hand, TS^{al} does not propagate any data-flow facts into `bar()`, because the object of interest cannot be accessed from variable `d`, the only variable that escapes to `bar()`. The graph drawn for TS^{al} in Figure 6.11 shows that no node for variable `d` is ever created. TS^{al} completely skips the analysis of `bar()`. For the example, TS^{al} visits only method `foo()`, whereas TS_3^f visits at least two methods (`foo()`, `bar()`, and any other method transitively reachable within the call graph).

Precision and Soundness Results. We have shown that TS^{al} analyzes fewer methods in comparison to TS^f. It directly follows the question whether it changes the precision or soundness values of the analyses. Table 6.3 lists the true positives, false positives, and false negatives for all three typestate analysis configuration: TS_2^f, TS_3^f, and TS^{al}. The numbers are listed per typestate property.

A first important observation is that the configuration TS_2^f is less precise than the other two configurations. The precision of TS_2^f is significantly lower: 0.61 compared to 0.84 for TS_3^f and 0.92 for TS^{al}. For the `Vector` typestate property, for example, TS_2^f lists 10 false positives, whereas the others only have 2 and 1 false positives, respectively. A false positive that TS_2^f has, but none of the other analysis configuration produces, occurs on a test case that requires flow-sensitive

Table 6.3: Comparing precision and recall in terms of true positives (TP,✓), false positive (FP,⊕) and false negatives (FN,⊖) between the stages TS_2^f and TS_3^f to the IDE^{al}-based typestate analysis TS^{al}.

	$\textbf{TS}_2^f$			$\textbf{TS}_3^f$			$\textbf{TS}^{al}$		
Typestate	TP	FP	FN	TP	FP	FN	TP	FP	FN
`KeyStore`	✓			✓			✓	⊕	
`Iterator`	7×✓	4×⊕		6×✓	⊕	⊖	6×✓		⊖
`URL`	✓			✓			✓		
`Vector`	24×✓	10×⊕	⊖	22×✓	⊕⊕	⊖⊖⊖	24×✓	⊕	⊖
`IO`	4×✓	4×⊕	6×⊖	4×✓		6×⊖	10×✓	⊕⊕⊕	
`Signature`	4×✓	8×⊕		4×✓	4×⊕		4×✓		
Precision		0.61			0.84			0.90	
Recall		0.85			0.79			0.96	

alias information. The stage TS_2^f does not track flow-sensitive alias information and outputs a false warning.

The false negative for TS^{al} for the typestate property `Vector` is because IDE^{al} does not model exceptional flows. On the test program, a vector object flows via an exception to a catch block. The vector object is stored as a field of the exception and unwrapped within the catch block. Within the catch block, the program erroneously accesses the first element of the empty vector. Both stages TS_2^f and TS_3^f also experience these false negatives.

Summary. On the micro-benchmark, TS^{al} requires fewer propagations than TS_3^f due to the individual propagations of aliases. At the same time, TS^{al} is slightly more precise than TS_3^f.

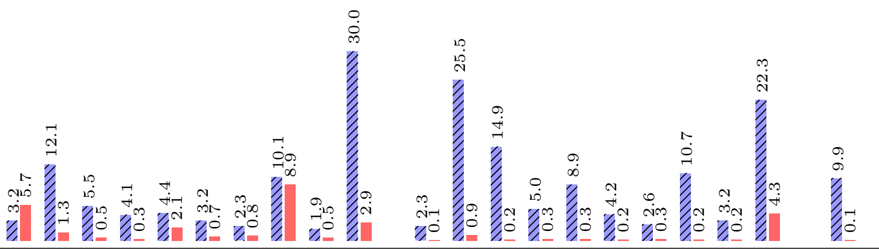
6.4.2 Typestate Analysis on DaCapo

In a second experiment, we evaluate the typestate analysis based on real-world programs from the DaCapo benchmark suite. In this section, we concentrate on the actual performance, measured in actual analysis time of TS^f and TS^{al} and not in the metric of visited methods. Additionally, we provide precision results for the discovered typestate violations.

Experimental Setup. For this experiment, we executed the typestate analyses TS^f and TS^{al} on all programs of the DaCapo 2006 benchmark suite [7]. During the experiments on the micro-benchmark, we observed that the stage TS_3^f has a precision comparable to TS^{al}. Hence, for this experiments we only compare to stage TS_3^f of TS^f. For the micro-benchmark, it is easy to control the object allocations, i.e., seeds, that the analysis is triggered at. On the DaCapo benchmark, we noticed that both analyses do not compute the same seeds to start off the

Table 6.4: Analysis time for running TS^{al} and TS_3^f on the DaCapo benchmark programs.

	Vector										IO										It.
	ANTLR	BLOAT	CHART	ECLIPSE	FOP	LUINDEX	LUSEARCH	PMD	XALAN	HSQLDB	ANTLR	BLOAT	CHART	ECLIPSE	FOP	LUINDEX	LUSEARCH	PMD	XALAN	HSQLDB	BLOAT
Seeds	2	8	3	7	3	14	3	12	2	1	15	30	16	20	14	12	13	16	12	36	13
Errors	0 1	3 5	1 1	2 2	2 2	12 12	2 2	5 9	1 1	1 0	3 2	18 13	7 2	3 2	3 2	3 2	3 2	6 2	3 2	21 14	0 4
🟥 TS^{al} ▨ TS_3^f **Avg. Analysis Time** (in s)	3.2 5.7	12.1 1.3	5.5 0.5	4.1 0.3	4.4 2.1	3.2 0.7	2.3 0.8	10.1 8.9	1.9 0.5	30.0 2.9	2.3 0.1	25.5 0.9	14.9 0.2	5.0 0.3	8.9 0.3	4.2 0.2	2.6 0.3	10.7 0.2	3.2 0.2	22.3 4.3	9.9 0.1
Timeouts	0 1	3 2	0 0	0 0	0 1	0 0	0 0	0 7	0 0	1 0	0 0	16 9	4 0	0 0	0 0	0 0	0 0	0 0	0 0	19 14	0 0



typestate analysis, a symptom of relying on different analysis frameworks (Soot and WALA) which do not produce identical call graphs. The seeds are computed based on the call graph reachable methods. For an objective comparison of the analysis times, we take the intersection of the seeds which are call graph reachable given both call graphs and we report all results based on the set of seeds that are consistent in both analyses.

For all programs, we use a 0-1-CFA call graph (`ZeroOneCFA` for WALA and the standard Spark call graph in Soot). This call graph is context-insensitive and distinguishes objects by their allocation sites. We do not include times for the call graph construction in our evaluation, because call graph construction is an orthogonal problem and the construction times vary due to the different analysis frameworks and implementations. Further on, we report the analysis times individually per object instead of reporting the accumulated analysis times of all seeds. This allows limiting the execution time of an analysis of one object, which for this experiment is 30 seconds. We analyze the complete program as well as all dependencies, including the complete the Java Runtime Environment. As a consequence, data-flows through complex-to-analyse code patterns occurred, for example, `HashMaps`, `TreeSets`, or general visitor patterns. For these data-flows, a larger analysis budget is required, and we constitute another experiment in Section 8 particular to these complex data-flows.

We run typestate analysis for all properties listed in Table 6.1. In Subsection 6.4.1, we report that it actually suffices to start at call sites such as `connect()` for the typestate properties `URL` and `Signature`. To avoid introducing any bias to the performance numbers, we configure TS^{al} for these typestate properties to start at allocation sites of objects instead.

Performance Results. Table 6.4 reports the summary of the performance result of this experiment. We cannot report any results for the properties `URL`, `KeyStore`, and `Signature`, because the DaCapo benchmark has no uses of `KeyStore`, occurrences of `Signature`, or seeds to `URL`. The table shows the total number of seeds per benchmark program per typestate property. The table contains bar plots, above its bars, we depict the geometric means over the analysis times per seed. Below the bars, we report the number of seeds that timed out within the time budget of 30 second. A seed that timed out is not excluded from the average and penalizes with the timeout budget.

The analysis times in Table 6.4 show that TS^{al} outperforms the analysis TS$_3^f$ by an order of magnitude. For the typestate property `IO`, we measured a performance improvement of a factor of 21.3× in comparison to TS$_3^f$. For the typestate property `Vector`, we measured a speed-up of 3.9×. For the typestate property `Iterator`, we noted a speed-up of 99×. The results for `IO` and `Iterator` are a similar order of magnitude to the result of the micro-benchmark experiment, where we observed that TS^{al} requires to visit 38.1× fewer methods than TS$_3^f$. `Vector` objects have a longer lifetime and are reachable in more methods on average, which reduces the impact of the difference in heap model (details in Section 8.1.3). The analysis time of TS^{al} also includes the execution times for the additional Boomerang queries that IDE^{al} triggers to compute strong updates and alias information on-demand. Opposed to that, the analysis TS$_3^f$ inquires a pre-computed whole-program points-to analysis for alias information. The computation time for this whole-program points-to analysis is part of the call graph construction and not included in the measured analysis time of TS$_3^f$.

Precision Results. For an evaluation of the precision on the DaCapo benchmark suite we manually inspected the reported errors from the analyses. We restricted our inspection to the seeds for which TS^{al} reported errors, but also to those that finished within the given time budget of 30 seconds. In total, 24 typestate violations are reported for the typestate property `Vector`. Further investigation shows that all of the identified and tracked objects *may* be in an error state, i.e., an element of the `Vector` may be accessed before any element was added to the `Vector`. However, our manual inspection unveiled that the accesses to the elements are guarded by appropriate size checks (e.g., branching based on `isEmpty()`) on the `Vector`. This means, at runtime, a typestate violation cannot occur. This shortcoming is due to the weakness of the expressiveness of the typestate pattern: the size checks cannot be modeled within the typestate machine, and is not an artifact of the over-approximations of IDE^{al} nor its data-flow domains. In other words, there are potentially valid data-flow connections that lead to those reports, but they are overcome by additional information that the analysis is not designed to track. Therefore, we classify the reported errors as true positives (with respect to the tracked typestate pattern).

Summary. TS^{al} outperforms TS$_3^f$ on the DaCapo benchmark suite where TS^{al} reduces the analysis times by a factors ranging from 3.9× to 99×, even though IDE^{al}'s analysis times include the time for demand-driven points-to queries to

Table 6.5: Comparison of the typestate results on the micro-benchmark for TS^{al}, when strong updates are disabled (TS^{-SU}) and aliasing is disabled (TS^{-al}).

Configuration	True Positives	False Positives	False Negatives	Precision	Recall
TS^{al}	46×✓	5×⊕	⊖⊖	0.9	0.96
TS^{-SU}	45×✓	6×⊕	⊖⊖⊖	0.88	0.94
TS^{-al}	41×✓	6×⊕	7×⊖	0.87	0.85

Boomerang. A manual inspection of the findings shows that the analysis is highly precise.

6.4.3 The Impact of Aliasing and Strong Updates

In the last two subsections, we compared the two analyses TS^{f} and TS^{al}. In this subsection, we focus on IDE^{al} and discuss the impact of handling aliasing and strong updates for the typestate client analysis TS^{al}.

Experimental Setup. Based on the same setup used for the experiments on the micro-benchmark programs and the DaCapo benchmark suite, we run TS^{al} in two additional configurations. For one run, we ignore the strong updates, every typestate update is performed weakly. This configuration is denoted by TS^{-SU}. In the other configuration, aliasing information is completely ignored. The latter configuration cannot perform strong updates, because strong updates also require aliasing information. This configuration is denoted by TS^{-al}. Both configurations allow us to compare the effect of aliasing and strong updates on TS^{al}.

Results. Table 6.5 depicts the analysis when run in the standard configuration, TS^{al} and in the two configurations TS^{-al} and TS^{-SU}. For each configuration, the table lists the columns *true positives*, *false positives*, and *false negatives* used for the computation of *Precision* and *Recall*. The precision and recall values are smaller in the configurations disabling strong updates and aliasing than for the standard configuration of IDE^{al}.

For TS^{-SU}, an additional false positive is reported on one of the `Vector` programs. The program allocates a `Vector` object, adds an element to it, stores the vector object to some field and loads the same vector object from that field. Next, the program performs an operation that is illegal for a non-empty vector. Due to the field store and load statement, there are two local variables pointing to the same object. For the typestate analysis, one of the two local variables holds a vector in an empty state, due to a missing strong update. As a consequence, the analysis reports a false positive. The additional false negatives for TS^{-SU} occurs on a program with a typestate violation. The program stores and loads a vector object to a static field. A missing strong update of typestate information leads to a missing true warning.

In the configuration TS^{-al}, the typestate analysis reports an additional false positive and five more false negatives. The false positive and one of the five false negatives are the same as for TS^{-SU}. The remaining four false negatives occur on programs in which the tracked object is stored and loaded from a field of another (parent) object. This code pattern requires computation of aliasing information for the parent object.

We run the TS^{al} in the three configurations on DaCapo and the results of this experiment on DaCapo differ slightly. When we disable aliasing for TS^{al}, more than 50% of the errors are not reported anymore. For all of those cases, the tracked objects are stored inside fields of other objects and are accessed indirectly within other methods through the fields. Such data-flows require aliasing information about the parent object, information that is missing when aliasing is disabled. On the other hand, disabling strong updates on DaCapo programs does not report any false positives on the inspected seeds.

Summary. While strong updates marginally improve precision and recall on the micro-benchmarks, when aliasing is disabled, the precision and recall values drop more significantly. We observed the same on the DaCapo programs, handling aliases has a higher influence and is required to obtain sound results.

6.5 Related Work

In this section, we discuss existing *data-flow frameworks*, most of which expose the problem of aliasing to the client analysis, as well as *solutions to aliasing* that client analyses may apply. For a more extensive discussion of the state-of-the-art alias analyses for object-oriented programs, we refer the reader to this survey [92].

6.5.1 Data-flow Analysis Frameworks

Apart from IFDS [74] and IDE [80], there exists a wide range of data-flow frameworks that simplify the implementation of interprocedural static data-flow analyses. For example, TVLA [81] uses abstract predicates that evaluate to a three-valued logic. In addition to 0 (*false*) and 1 (*true*), three-valued logic maintains a third value (1/2) that represents *unknown* or *maybe* evaluations. Using predicates enables TVLA to infer aliasing automatically. TVLA has been extended later to support interprocedural analysis [30, 40]. While TVLA propagates sets of aliasing pointers, IDE^{al} propagates aliasing pointers independently, which drastically reduces the size of the analysis domain.

Separation logic defines another technique for data-flow analysis [78]. In separation logic the heap is modeled explicitly. Each instruction directly operates on the model of the heap, analog to an actual execution. Separation logic extends standard Hoare logic by adding a separating conjunction. The conjunction splits the heap into disjunct regions. At a call site, for instance, the heap can be divided into the region accessed within the callee and the region's complement. With an extension called bi-abduction, Calcagno et al. [13] managed to design a scalable shape analysis based on separation logic. IDE^{al} uses a storeless model of the heap which does not require a splitting of the heap as abstract pointers instead of concrete allocation sites are propagated. Calcagno et al. [13] achieve scalability by performing a compositional analysis. Similar benefits are expected for IDE^{al} by pre-computing summaries, as described by Arzt and Bodden [4].

Blackshear et al. [8] propose a goal-directed approach, called Hopper, to analyze programs that are based on event-driven systems, such as Android. The novelty lies in jumping along control-flow feasible paths, once a data-flow flows into system code, e.g., Android. Instead of analyzing the system's code, the flow directly jumps to the respective point in the non-system part of the program. Hopper relies on separation logic, i.e., explicitly models the heap in a store-based abstraction. The authors report a significant performance boost through jumping. In future work, we want to investigate how IDE^{al} can make use of a similar functionality.

Ferrara [25] proposes an abstract-interpretation-based generic framework to value-flow analysis that includes heap-reasoning. The authors formally prove that heap and value analyses can be combined into one analysis, similar to IDE^{al}'s Phase I and Phase II. Two types of analyses that can be instantiated within their framework are numerical and shape analysis. Opposed to their work, IDE^{al} computes context-sensitive results. This, however, comes by the cost of restricting the value domain from a infinite to a finite height lattice, as IDE^{al} does not support a widening operator.

Madsen and Møller [60] describe a sparse data-flow analysis framework for JavaScript code. A *sparse* data-flow analysis operates on def-use chains that it constructs from the control-flow graph of a given program. This approach differs from a traditional data-flow analysis that processes every statement in the program. Sparse data-flow analyses leverage the fact that def-use chains are typically more sparse than the control-flow graph, thus the analysis requires fewer propagations of data-flow facts. The pushdown systems of IDE^{al} are not sparse, they generate identity rules for statements that do not use variables. Encoding $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ sparsely could further improvement the efficiency of IDE^{al}.

6.5.2 Solutions to Aliasing

We have already compared to the typestate analysis by Fink et al. [27] in detail and skip its discussion here.

Yahav and Ramalingam [103] propose a typestate analysis on top of TVLA, but the authors report later that TVLA does not scale well to large programs [26]. An interesting contribution of their work, however, is *separation*, as they report a huge benefit in separating the typestate analysis into sub-problems. We use a simple version of separation by invoking IDE^{al} per tracked object.

Naeem and Lhoták [66] show how to perform a typestate analysis (TS^{n}), using property specifications called tracematches [10]. Tracematches are a language extension to AspectJ [7], and allow the analysis to select program points using

[7]`https://eclipse.org/aspectj/`

declarative patterns called *pointcuts.* TS^{n} uses a coarse-grained field-insensitive abstraction for the objects that are allocated on the heap. In contrast to TS^{f} and TS^{al}, TS^{n} can check for patterns that detect buggy interactions of multiple objects (e.g., updating a list while an iterator iterates over it). TS^{n} implements this by tracking all objects that are allocated in the input program, which is a significant limitation to efficiency. Naeem and Lhoták [67] later overcome this limitation by generating flow-insensitive callee and caller summaries. The summaries are constructed by pre-analyzing the methods where pointcuts have no matches. TS^{n} then plugs in those summaries at the appropriate call sites. We plan to extend IDE^{al} to use a similar approach to synchronize information about multiple interacting objects.

Tripp et al. [100] propose Andromeda, an IFDS-based taint analysis that handles aliasing by propagating access paths individually. Similar to IDE^{al}, Andromeda resolves aliases in a context-sensitive and flow-sensitive fashion through an on-demand backward analysis. Unlike IDE^{al}, due to propagating aliases individually, Andromeda does not support strong updates. Once tainted, Andromeda does not remove the taint if an alias is sanitized. FlowDroid [5] takes a similar approach to alias resolution.

7 Detection of Cryptographic API Misuses on a Large Scale

Almost any software system processes, stores, or interacts with sensitive data. Such data typically includes user credentials in the form of e-mail addresses and passwords, as well as company data such as the company's income, employee's health, and medical data. Cryptography is the field of computer science that develops solutions to protect the privacy of data and to avoid malicious tampering.

Software developers should have a basic understanding of key concepts in cryptography to build secure software systems. Prior studies [20,65] have shown that software developers commonly struggle to do so and as a result fail to implement cryptographic[1] tasks securely. While cryptography is a complex and difficult to understand area, it also evolves quickly and software developers must continuously remain informed about broken and out-dated cryptographic algorithms and configurations.

But it is not only the lack of education on the developer's side, common crypto APIs are also difficult to use correctly and securely. For instance, implementing a data encryption with the Java Cryptographic Architecture[2] (JCA), the standard crypto API in Java, requires the developer to combine multiple low-level crypto tasks such as secure key generation, choosing between symmetric or asymmetric crypto algorithms in combination with matching block schemes and padding modes. While the JCA design is flexible to accommodate any potential combination, it yields to developers implementing crypto tasks insecurely by misusing the API.

Example 30. Figure 7.1 demonstrates an example code that incorrectly uses some of the JCA's classes for encryption. At instantiation time of an `Encrypter` object, the constructor generates a `SecretKey` for algorithm `"Blowfish"` (parameter to the call to `getInstance()` in line 330) of size 448 (parameter to call in line 331). In line 332, the key is stored to field `key` of the constructed `Encrypter` instance. The `Encrypter` object's public API offers a method `encrypt()`, which, when called, creates a `Cipher` object in line 336. The `Cipher` object is configured to encrypt data using the `"AES"` algorithm (parameter to the call to `getInstance()` in line 336). The developer commented out line 337 that (1) initializes the algorithm's mode and (2) passes the `SecretKey` stored in field `key` to the `Cipher` object. The call to `doFinal()` in line 338 performs the encryption operation and encrypts the content of the `plainText` and stores it in the `byte` array `encText`.

[1]Hereafter, used interchangeably with crypto.

[2]`https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`

```
325 public class Encrypter{
326   private SecretKey key;
327   private int keyLength = 448;
328 
329   public Encrypter(){
330     KeyGenerator keygen = KeyGenerator.getInstance("Blowfish");
331     keygen.init(this.keyLength);
332     this.key = keygen.generateKey();
333   }
334 
335   public byte[] encrypt(String plainText){
336     Cipher cipher = Cipher.getInstance("AES");
337     //cipher.init(Cipher.ENCRYPT_MODE, this.key);
338     byte[] encText = cipher.doFinal(plainText.getBytes());
339     return encText;
340   }
341 }
```

Figure 7.1: An example of a misuse of a cryptographic API.

There are four API misuses in this code example. First, the developer commented-out a required call in line 337. Second, if the developer includes the line in the comment, the generated key (`"Blowfish"`) and the encryption cipher (`"AES"`) do not match. Third, and related, the key length of 448 is not suitable for the algorithm AES that expects a size of 128, 192, or 256. Fourth, depending on the crypto provider, AES is used with electronic codebook mode (ECB). Using ECB results in low entropy within the bytes of `encText`. The first three API misuses throw exceptions at runtime that, using static analysis, could already be detected at compile time. Using ECB, however, does not throw an exception and silently leads to insecure code.

Such examples are found in real-world software artifacts and motivate the research that we have conducted [47]. Our work presents (1) a domain-specific language (DSL), called CrySL, for the specification of API usage *rules* and (2) *CryptoAnalysis*, a tool that compiles the rules into an automated static code analysis. Stefan Krüger (my collaborator) designed CrySL and wrote rules that specify the API of the JCA.

This chapter summarizes our work and sketches the design of the static analysis *CryptoAnalysis* that combines results of several IDE^{al}-based typestate analyses, connects interaction between API objects and triggers Boomerang queries to detect `String` and `Integer` constants. Using *CryptoAnalysis* we present a large scale study of *CryptoAnalysis* on artifacts from Maven Central[3].

7.1 The CrySL language

To detect such API misuses, Krüger et al. [47] designed CrySL, a domain-specific language that allows the specification of crypto API uses. CrySL defines a

[3] `https://mvnrepository.com/repos/central`

whitelist approach that specifies correct uses of an API, and *CryptoAnalysis* reports code that deviates from the specification. We briefly introduce the main semantics of the language in this section and discuss the basic design of *CryptoAnalysis*. The language definition and the carefully written CrySL specifications[4] for the JCA are not a contribution of this thesis.

With the CrySL specifications for the JCA, *CryptoAnalysis* is able to detect all four crypto related issues showcased in Example 30. We discuss the important syntax elements of CrySL based on a minimal CrySL specification covering the misuses in Example 30. We refer to the original work [47] for the definition of all syntax elements of CrySL.

A CrySL specification is composed of multiple CrySL *rules*. Each CrySL rule starts with `SPEC` clause specifying the type of the class that the CrySL rule is defined for. Figure 7.2 depicts two CrySL rules for the classes `javax.crypto.Cipher` and `javax.crypto.KeyGenerator`. The `SPEC` clause is followed by an `OBJECTS` block that defines a set of *rule members*. The values of the rule members are then constraint on within the `CONSTRAINTS` block. For instance, the `CONSTRAINTS` for the rule to `KeyGenerator` restricts the rule member `keySize` in line 355 to the values `128`, `192`, or `256`. When using a `KeyGenerator`, the integer value for `keySize` must be one of the listed values.

The `EVENTS` block defines labels (e.g., `Get` in line 348 and `Inits` in line 349), each label is a set of events. An event is an invocation of a method and is defined via the method signature. For example, label `Inits` is defined as the event of calling the method with signature `init(int keySize)` (line 349). The parameter name (`keySize`) matches the name of a rule member, and when the program calls the event's method, the value of the parameter of the call is bound to the rule member `keySize`, which means that the parameter must satisfy the given constraint.

The labels defined within the `EVENTS` block are used in the `ORDER` block. The `ORDER` clause lists a regular expression (inducing a finite state machine) over the labels and defines the usage pattern (i.e., typestate property) of the specified type. Each object of the specification is required to follow the defined usage pattern. For instance, the specification for `KeyGenerator` expects each object of its type to call any method of the label `GetInstance` prior to any of the `Inits` call followed by a `GenerateKey` call. The `ORDER` specification for `Cipher` uses a `+` for the label `doFinal`, indicating that the method `doFinal()` must be called at least once and arbitrary additional calls of the method can follow.

The remaining two blocks are the `REQUIRES` and `ENSURES` block of a rule. Each line of these blocks lists a *predicate*. A predicate is defined by a name followed by a list of parameters. CrySL predicates cover the specification of the interaction of multiple objects of different types. The `KeyGenerator` rule lists a predicate `generatedKey` with two parameters `key` and `algorithm` in the `ENSURES` block in line 357. When an object of type `KeyGenerator` is used according to the specification in the `CONSTRAINTS`, `ORDER`, and `REQUIRES` block, the predicate listed in the `ENSURES` block is generated for the object. Other CrySL rules that interact with

[4] `https://github.com/CROSSINGTUD/Crypto-API-Rules`

```
SPEC javax.crypto.KeyGenerator
OBJECTS
  int keySize;
  javax.crypto.SecretKey key;
  java.lang.String algorithm;
EVENTS
  Get: getInstance(algorithm);
  Inits: init(keySize);
  GenerateKey: key = generateKey();
ORDER
  Gets, Inits, GenerateKey
CONSTRAINTS
  algorithm in {"AES", "Blowfish", ...};
  keySize in {128, 192, 256};
ENSURES
  generatedKey[key, algorithm];
```

(a) CrySL rule for `javax.crypto.KeyGenerator`.

```
SPEC  javax.crypto.Cipher
OBJECTS
  java.lang.String trans;
  byte[] plainText;
  java.security.Key key;
  byte[] cipherText;
EVENTS
  Get: getInstance(trans);
  Init: init(encmode, key);
  doFinal: cipherText = doFinal(plainText);
ORDER
  Get, Init, (doFinal)+
CONSTRAINTS
  encmode in {1,2,3,4};
  part(0, "/", trans) in {"AES", "Blowfish", "DESede", ..., "RSA"};
  part(0, "/", trans) in {"AES"} => part(1, "/", trans) in {"CBC"};
REQUIRES
  generatedKey[key, part(0, "/", trans)];
ENSURES
  encrypted[cipherText, plainText];
```

(b) CrySL rule for `javax.crypto.Cipher`.

Figure 7.2: Two simplified CrySL rules for the JCA.

KeyGenerator objects can list the predicate in their REQUIRES block. For instance, the CrySL rule Cipher lists the predicate generatedKey as a required predicate in line 375.

7.2 Compiling CrySL to a Static Analysis

CryptoAnalysis is a static analysis compiler that transforms CrySL rules into a static analysis. Internally, *CryptoAnalysis* is composed of three static sub-analyses: (1) an IDE^{al}-based typestate analysis, (2) a Boomerang instance with extensions to extract String and int parameters on-the-fly and (3) an IDE^{al}-based taint analysis (i.e., all weights are identity). The three static analyses deliver input to a constraint solver that warns if any part of the CrySL specification is violated.

Example 31. We discuss a walk-through of *CryptoAnalysis* based on the CrySL specification defined in Figure 7.2 and the code snippet provided in Figure 7.1. *CryptoAnalysis* first constructs a call graph (CHA, VTA, or SPARK) and computes call-graph reachable allocation sites for in CrySL specified types. Factory methods can also serve as allocation sites. For example, the factory methods getInstance() of Cipher and KeyGenerator internally create objects of the respective type and *CryptoAnalysis* considers these calls as allocations sites. In the code example in Figure 7.1 the allocations sites are the objects o_{330} and o_{336}.

Starting at the allocation sites, *CryptoAnalysis* conducts a context-sensitive, flow-sensitive, and field-sensitive typestate analysis (TS^{al}) and checks if the object satisfies the ORDER clause of the rule. The call sequence on the KeyGenerator object o_{330} satisfies the required typestate automaton defined as regular expression in the ORDER block. Opposed to that, the Cipher object o_{336} does not satisfy the ORDER clause, because the developer commented out line 337. *CryptoAnalysis* warns the developer about the violation of this clause (line 369).

Apart from the analysis TS^{al}, *CryptoAnalysis* also extracts String and int parameters of events (statement that change the typestate) to bind the actual values to the rule members of a CrySL rule. For instance, the getInstance("Blowfish") call in line 330 binds the value "Blowfish" to the rule member algorithm of the CrySL rule for KeyGenerator. In this example, the String value is easy to extract statically, but it might also be defined elsewhere in the program. For example, the value binding for the rule member keySize is the actual int value flowing to the init call in line 331 as a parameter. The actual value is loaded from the heap, because it is the value of the instance field keyLength of the Encrypter object. Therefore, *CryptoAnalysis* triggers a Boomerang query for this.keyLength@331 to find the actual int value of the field. It is straightforward to extend Boomerang to also trace primitive types throughout the program. Since Boomerang uses a storeless heap model, one simply defines assignment statements that assign primitive values (e.g., int x = 1) as "allocation sites". With this extended notion of allocation sites, the forward analysis of Boomerang propagates the data-flow of x through the program.

To conclude, *CryptoAnalysis* infers that object o_{330} tries to generate a `SecretKey` for the algorithm `"Blowfish"` with a key length of 448 in line 332. The `KeyGenerator` rule disallows the chosen key length (`CONSTRAINTS` in line 355), and *CryptoAnalysis* warns the developer to choose an appropriate `keySize`.

Assume the developer to change the code to use an appropriate value for `keySize`, and the `KeyGenerator` is used in compliance to its CrySL specification, then *CryptoAnalysis* generates the predicate `generatedKey` for the `SecretKey` object stored to field `key` of the `Encrypter` instance as expected.

If additionally, the developer includes the `init` call on the cipher object in line 337, (1) the `ORDER` clause of the CrySL rule for `Cipher` is satisfied and (2) the `generatedKey` predicate flows via the field `this.key` to the `Cipher` object o_{336}. As the `Cipher` rule `REQUIRES` the predicate (line 375), the `ORDER` and `REQUIRES` blocks for the object o_{336} are satisfied.

However, the `CONSTRAINTS` for object o_{336} are still not satisfied. *CryptoAnalysis* reports that (1) the key is generated for algorithm `"Blowfish"`, and this selection does not fit the algorithm chosen for `Cipher` (`"AES"`) and (2) when using algorithm `"AES"`, one should use it in `"CBC"` mode (`CONSTRAINTS` in line 373). When the developer fixes these two mistakes, *CryptoAnalysis* reports the code to correctly use the JCA with respect to the CrySL rule.

7.3 Evaluation on Maven Central

The CrySL specifications for the JCA cover a total of 23 interfaces and classes. In general, the CrySL rules cover different standard crypto tasks such as hashing and signing of data, asymmetric as well as symmetric encryption, and decryption tasks. For the evaluation of *CryptoAnalysis*, we use the Maven Central Repository[5] and consider it an appropriate source of software artifacts. Maven Central is a popular software repository where developers can publish their software artifacts. Publishing allows other developers to easily access and include the software into their own projects. At the time of writing, over 2.7 million software artifacts are published at Maven Central.

We decided to evaluate *CryptoAnalysis* on software artifacts from Maven Central mostly as we consider the data set to be representative. Additionally, it is difficult to find good benchmark suites of Java programs for crypto uses, for instance only few programs of the DaCapo 2006 benchmark suite use crypto. The large number of artifacts on Maven Central guarantees a variety of uses of crypto and is representative for real-world software at the same time.

Experimental Setup. The Maven Central repository maintains different versions of software artifacts. We restrict our analysis to the latest version of each software artifact as of July 2018. A total of 152,996 artifacts remain that we run *CryptoAnalysis* on. These experiments are run on an Intel Xeon E5-2680, 2.40 GHz machine with 16 processors and 128 GB of memory.

[5] `https://mvnrepository.com/repos/central`

The implementation of *CryptoAnalysis* is single-threaded and we concurrently analyse 10 artifacts at a time, each in its own Java virtual machine. We granted each virtual machine a maximum of 12 GB of heap memory and limited the analysis of each artifact to 1 hour.

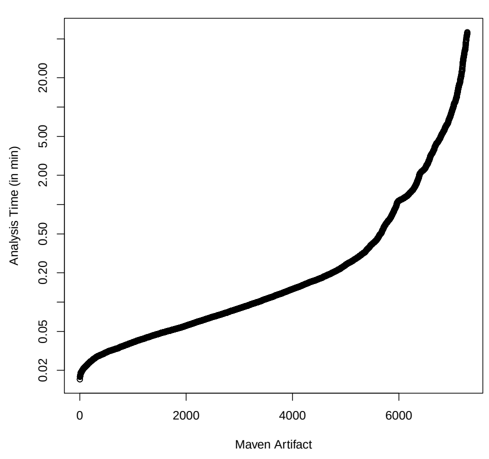
Most artifacts on Maven Central are libraries. Opposed to standalone applications, libraries frequently do not contain a `main()` method. Efficient computation of a sound and precise call graph for a library is challenging [72]. One has to make assumptions about the use of the library classes within an unknown application. A library frequently does not allocate its own library objects and the standard points-to based call graph algorithms fail to compute a sound call graph, because many edges ares missing. SPARK has a mode (option `library`[6]) particular crafted for the library-analysis case. In this mode, dummy allocation sites for all public classes of the library are instantiated and lead to non-empty points-to sets for variables within the library. After first experiments with this call graph mode on Maven Central, we observed that even call graph construction failed to terminate within the budget of 1 hour for most artifacts. Eventually, we thus chose CHA, which is highly imprecise but reasonably efficient to compute.

Performance. During the analysis, we record several performance metrics such as the time to setup SOOT and construct the CHA call graph, which we simply refer to as *setup time*. Additionally, we record the number of call-graph reachable methods and the number of CrySL objects that these methods allocate. We also measure the *analysis time* for *CryptoAnalysis*. This time includes the computation of the object-allocation statements (iterating over all reachable methods in the call graph) as well as the actual data-flow analysis times for these objects.

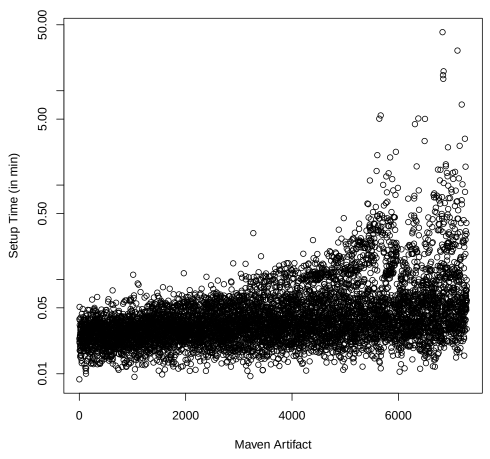
For each of the 152,996 artifacts, the analysis constructs a call graph to compute the reachable object allocations. In a total of 8,193 artifacts (5.3% of all artifacts), CryptoAnalysis found at least one use of some class of the JCA. For these 8,193 artifacts, the data-flow analyses (IDE^{al} and BOOMERANG) are executed based on the CrySL specifications. For 7,287 artifacts, the analysis finishes in under an hour each. For this experiment, we set a query budget for IDE^{al} and BOOMERANG of 5 seconds and we encountered 16.0% of timeouts, which lead to partially unsound results. We discuss factors for these query timeouts in Chapter 8. The query timeouts also cause the timeouts of the remaining 906 artifacts, and we exclude reporting any statistics across the timed-out artifacts in the following discussion.

Figure 7.3 shows the measured performance metrics for the 7,287 successfully analyzed artifacts. Figure 7.3a plots the analysis times of the artifacts in minutes in ascending order. The order of artifacts is the same for Figure 7.3b, Figure 7.3c, and Figure 7.3d. On average, across all artifacts, the analysis time is 88 seconds per artifact. The distribution of the times vary highly, and the plots show that for more than 85.7% of the crypto-using artifacts (7,023 of 8,193), the analysis time remains under 10 minutes. For the artifacts that take more than 10 minutes

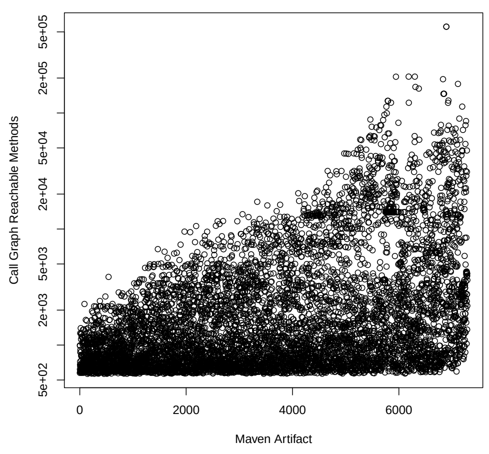
[6]`https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm`



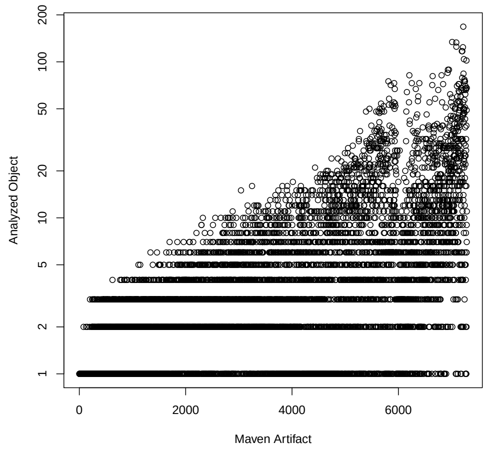
(a) Analysis times for *CryptoAnalysis* (excluding call graph construction time).



(b) Call graph construction times for each artifact.



(c) Call graph reachable methods for each artifact.



(d) Number of analyzed objects, i.e., objects a CrySL specification exists for.

Figure 7.3: Different metrics computed for the artifacts of Maven Central. The artifacts in subfigures Figure 7.3b, Figure 7.3c and Figure 7.3d are ordered based on the artifact's analysis times in Figure 7.3a.

Table 7.1: Correlation coefficient between analysis time of *CryptoAnalysis* and the setup time, the call graph reachable methods and analyzed objects.

Correlation Coefficient with Analysis Time	
Setup Time	0.06
Reachable Methods	0.13
Analyzed Objects	0.45

to analyze, the growth of the analysis time is drastic. For 264 artifacts (3.2%), the analysis time ranges between 10 minutes and 60 minutes. For the remaining 906 crypto-using artifacts (11.1%), the analysis does not finish within the 1 hour time limit.

Given that *CryptoAnalysis* performs a highly precise and sophisticated static analysis (flow-sensitive, field-sensitive, and context-sensitive), these results are promising, and for many artifacts, the analysis terminates in a reasonable time. To provide more insights and discuss factors that can be used as predictors of the analysis times, we correlate the times to metrics that are pre-computable, i.e., computed ahead of any data-flow computations. We find the *setup time*, the number of *call graph reachable methods*, and the number of *analyzed CrySL-objects* are reasonable pre-computable metrics.

We compute the correlation coefficients between the analysis times and the setup time, reachable methods, and analyzed objects. Each correlation coefficient is a numeric value between -1 and 1 and measures how two data sets correlate. The closer the value to 1, the higher the correlation between the two sets. A value of 0 indicates no correlation between the data sets. Table 7.1 shows that the coefficient is smaller for the two metrics Setup Time (0.06) and Reachable Methods (0.13), whereas it is higher for the analyzed objects (0.45). Therefore, the number of analyzed objects is the best of the three metrics to predict the analysis time of *CryptoAnalysis*.

Findings. Of the 7,287 artifacts, only 2,308 artifacts (31.7%) use the JCA according to the CrySL specification and can be considered secure. The remaining 68.3% of all artifacts contain at least misuse the JCA.

For the evaluation of the reported findings of *CryptoAnalysis*, we report which of the CrySL blocks (`CONSTRAINTS`, `REQUIRES`, `ORDER`) is violated and which class the error relates to. Table 7.2 provides an overview of all findings. The table lists the CrySL specified classes and the number of violations for each block. All of the 7,287 artifacts contain a total of 45,917 instances of the 23 CrySL-specified JCA types, i.e., each artifact instantiates an average of 6 objects of the JCA. For all CrySL rules and all artifacts, *CryptoAnalysis* finds a total of 22,664 violations.

The violations are distributed roughly equally between `CONSTRAINTS`, `ORDER`, and `REQUIRES` violations. There are 7,030 `CONSTRAINTS` errors, 8,860 `ORDER` errors, and 6,774 missing `REQUIRES` predicates.

The class with the most violations of the CrySL specifications is the class

Table 7.2: CrySL violations of the JCA on all Maven Central Artifacts.

	CrySL Violations			
Specified Type	`CONSTRAINTS`	`ORDER`	`REQUIRES`	**Total**
`AlgorithmParameters`	32	-	6	38
`Cipher`	1,301	2,193	2,142	5,636
`DHGenParameterSpec`	-	-	-	-
`DHParameterSpec`	-	-	-	-
`DSAGenParameterSpec`	-	-	-	-
`DSAParameterSpec`	-	-	-	-
`GCMParameterSpec`	-	-	55	55
`HMACParameterSpec`	-	-	-	-
`IvParameterSpec`	-	-	774	774
`KeyGenerator`	43	48	67	158
`KeyPair`	-	1	-	1
`KeyPairGenerator`	94	112	-	206
`KeyStore`	388	588	-	976
`Mac`	34	534	3	571
`MessageDigest`	4,462	4,491	-	8,953
`PBEKeySpec`	249	422	169	840
`PBEParameterSpec`	81	-	85	166
`RSAKeyGenParameterSpec`	-	-	-	-
`SecretKey`	-	-	-	-
`SecretKeyFactory`	-	41	-	41
`SecretKeySpec`	-	-	2,788	2,788
`SecureRandom`	8	99	-	107
`Signature`	338	331	685	1,354
Total	7,030	8,860	6,774	22,664

`MessageDigest` followed by the `Cipher` class. Many software artifacts still use the outdated `MessageDigests` algorithms `"MD5"` and `"SHA1"`. The CrySL specification for `MessageDigest` does not allow these as hashing algorithms, and *CryptoAnalysis* reports `CONSTRAINTS` violations. `MessageDigest` also has a rather strict usage pattern: when a hash is computed by a `digest` call, the interface expects a call to `reset` before using the object for the computation of a second hash. This pattern is frequently misused, and *CryptoAnalysis* reports 4,491 `ORDER` errors of this type.

For the `Cipher` class, *CryptoAnalysis* reports violations for all three blocks (`CONSTRAINTS`, `ORDER`, and `REQUIRES`). The `CONSTRAINTS` errors are related to misconfiguration of the encryption algorithm. For instance, we found 154 artifacts that use the insecure `"DES"` algorithm and 513 artifacts that use the algorithm `"AES"` with the insecure block mode `"ECB"`. The `ORDER` errors are reported along data-flow paths of `Cipher` objects that are not in an accepting state when the objects are destroyed. The CrySL rule expects the last invocation on any `Cipher` object to be an invocation of `doFinal()` to encrypt/decrypt the actual data. Without calling `doFinal()`, no data is actually processed. For 2,142 of all `Cipher` objects, a predicate listed in the `REQUIRES` blocks is missing. The `Cipher` rule expects two predicates in the `REQUIRES` block. The keys for encryption must be securely generated (by using `SecretKeyFactory` or `KeyGenerator` according to its CrySL rule) or the class `AlgorithmParameters` must be correctly used.

Example 32. We want to elaborate on one finding more closely, because it shows the capability of our analysis. Listing 7.1 shows a code excerpt of an artifact that uses a `KeyStore` object. A `KeyStore` stores certificates and is protected with a password. A `KeyStore` object has a method `load()` whose second parameter is a password. The API expects the password to be handed over as a `char[]` array. The `KeyStore` API explicitly uses the primitive type instead of a `String`, because `Strings` are immutable and cannot be cleared.[7] However, many implementations convert the password from a `String` and hereby introduce a security vulnerability; when not yet garbage collected, the actual password can be extracted from memory, e.g., via a memory dump.

The code in Listing 7.1 contains two security vulnerabilities that *CryptoAnalysis* detects. First, the password is converted from a `String` object via a call to `toCharArray()` to the actual array (line 418), i.e., during the execution of the code the password is maintained in memory as `String`. Second, under some conditions (lines 380, 384, 391, 393 and 402 must evaluate to *true*), the password is hard-coded.

CryptoAnalysis reports a `CONSTRAINTS` error on this example, because the `String pass` (highlighted by the green box) in line 418 may contain the `String` `"changeit"` as it is defined in line 381 (also highlighted). The data flow corresponding to the finding is non-trivial to detect manually, however, *CryptoAnalysis* is able to do so by support of BOOMERANG. *CryptoAnalysis* triggers a BOOMERANG query for the second parameter of the `load()` call in line 418 and finds the

[7] `https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx`

`toCharArray()` call. From that call the analysis traces the variable `pass` in method `getStore()` and finds it to be a parameter of `getStore()`, and the data-flow propagation continues at invocations of the method. The method `getStore()` is called in line 403 where BOOMERANG data-flow propagation follows the variable `truststorePassword`. This variable is assigned the return value of the call site in line 401. The backward data-flow analysis continues in line 397 and finds the variable to be assigned from the return value of the method call in line 394. Within the callee `getKeyStorePassword()`, BOOMERANG traces the variable `keystorePass` and eventually finds the allocation site `"changeit"` in the highlighted line with the line number 381. Eventually, *CryptoAnalysis* reports that variable `pass` is of type `String` and that it may contain the hard-coded password `"changeit"`.

Summary. *CryptoAnalysis* is an efficient and precise static data-flow analysis tool that is fully based on IDE^{al} and BOOMERANG. For over 85.7% of all crypto-using Maven artifacts, the analysis terminates in under 10 minutes. The tool reports 68.7% of all crypto-using Maven artifacts to contain at least one misuse.

```
protected String getKeystorePassword(){
  String keyPass = (String)this.attributes.get("keypass");
  if (keyPass == null) {
    keyPass = "changeit";
  }
  String keystorePass = (String)this.attributes.get("keystorePass");
  if (keystorePass == null) {
    keystorePass = keyPass;
  }
  return keystorePass;
}
protected String getTruststorePassword(){
  String truststorePassword = (String)this.attributes.get("...");
  if (truststorePassword == null){
    truststorePassword = System.getProperty("...");
    if (truststorePassword == null) {
      truststorePassword = getKeystorePassword();
    }
  }
  return truststorePassword;
}
protected KeyStore getTrustStore(){
  ...
  String truststorePassword = getTruststorePassword();
  if ((truststore != null) && (truststorePassword != null)) {
    ts = getStore(truststoreType, truststore, truststorePassword);
  }
  return ts;
}
private KeyStore getStore(String type, String path, String pass){
  KeyStore ks = null;
  InputStream istream = null;
  ks = KeyStore.getInstance(type);
  if ((!"PKCS11".equalsIgnoreCase(type)) && ...){
    File keyStoreFile = new File(path);
    if (!keyStoreFile.isAbsolute()) {
      keyStoreFile = new File(System.getProperty("..."), path);
    }
    istream = new FileInputStream(keyStoreFile);
  }
  ks.load(istream, pass.toCharArray());
  return ks;
}
```

Listing 7.1: Code example with a hard-coded password.

8 Influencing Factors on Analysis Performance

In this chapter, we further investigate the factors influencing the efficiency of the queries to IDE^{al} and Boomerang. First, we elaborate on the impact on the analysis times when exchanging the two (almost) equally precise heap models: access paths and synchronized pushdown systems. Second, we relate other algorithm-specific factors to the analysis times to motivate further paths for exploration.

8.1 Access Paths and Synchronized Pushdown Systems

In Chapter 4, we discuss synchronized pushdown systems (SPDS) and the theoretical difference compared to the access path model. Based on IDE^{al} (Chapter 6) and Boomerang (Chapter 5), this section complements the theoretical worst-case complexity analysis by empirical performance results.

The results of the worst-case complexity analysis of SPDS indicate that SPDS is more efficient than k-limited access path (AP^k) for data-flows spanning only few statements and storing the data to many fields. Based on experiments with Boomerang and IDE^{al}, we elaborate on how the two heap models compare in practice. Additionally, we show that the hypothesis SPDS are build on is valid and that SPDS do not introduce false positives in comparison to access paths. Therefore, we reason that an improperly matching of call sites does not induces a properly matching of fields.

Before we designed and implemented Boomerang and IDE^{al} based on SPDS, we used access paths and access graphs to represent field accesses in both analyses [88, 90]. We observed these heap models as one factor hindering efficient data-flow analysis. In the following experiments, we confirm this observation and compare the prior implementations of both analyses to the implementations based on SPDS. To distinguish the access-path based analyses from ones using SPDS, hereafter, we refer to the access-path based implementations as IDE$^{al}_{AP}$ and Boomerang$_{AP}$. The implementations for IDE$^{al}_{AP}$ and Boomerang$_{AP}$ are also publicly available[1].

Based on these two different implementations, we address the following research questions:

- **RQ1**: How does the number of field accesses on the data-flow paths of a Boomerang query relate to the time taken to compute the query's results?

[1]IDE$^{al}_{AP}$: `https://github.com/secure-software-engineering/ideal` and Boomerang$_{AP}$: `https://github.com/secure-software-engineering/boomerang`

```
stateExplosion() {
  Node x = new Node();
  Node p = new Node();
  Node t = new Node();
  while(...){
    if(...){
      x.a1 = p;
    }
    p = x;
  }
  if(...){
    t = x.a1;
  }
  queryFor( t );
}
```

Figure 8.1: The example code EXPL_1 that provokes state explosions for access-path based domains. This code contains a single field-store and a single field-load of the field `a1`.

- **RQ2**: How does a typestate analysis based on IDE^{al} compare to the same analysis based on IDE^{al}_{AP}?
- **RQ3**: How does the number of methods and field-stores, i.e., push-rule applications, along a data-flow path influence the typestate analysis time for an abstract object?

8.1.1 Micro-Experiment: Controlled Field Explosion

In this micro-experiment, we compare access path, access graphs, and SPDS based Boomerang queries in a controlled lab environment. Boomerang_{AP} can be configured to use either k-limiting or access graphs as its field abstraction. We designed a target analysis program for which we can control the number of field accesses along a data-flow path for a pointer query. We run the same Boomerang query configured with different heap models on a target program which is parametrizable in number of field accesses. Dependent on the parameter, the amount of field accesses in the target program increases. Hereby, we demonstrate the differences between the field abstractions without changing the other dimensions (e.g., number of visited methods). To complement the worst-case complexity analysis of Section 4.4.2, we measure how an increase in the number of field accesses affects the query's analysis time in practice.

Experimental Setup. In Chapter 2, we show an example using `java.util.TreeMap`, in which a data-flow analysis generates all combination of access paths

$$T = \{\texttt{this}.f_1.f_2.\cdots.f_n.\texttt{value} \mid f_i \in \{\texttt{right}, \texttt{left}, \texttt{parent}\}, n \in \mathbb{N}\}.$$

Based on the `TreeMap` implementation, we extracted a minimal code example that provokes a similar state explosion and generates a similar sized set of access paths. We also made the number of field accesses along the data-flow parametrizable and arrived at the code shown in Figure 8.1. The code is designed to provoke a state explosion for the static data-flow analysis when points-to information for variable `t` is queried at the call to `queryFor()` in line 434 (variable highlighted green). The backward analysis computes the three allocations in lines 422, 423, and 424 as allocation sites for `t`. From there, Boomerang's forward analysis requires generating all access paths in the set $\{\texttt{x}.f_1.f_2.\cdots.f_m \mid f_i \in \{\texttt{a1}\}, m \in \mathbb{N}\}$ at each statement. This result is due to the fact that the `while`-loop assigns `x` to `p` but also stores `p` in a field of `x`.

The code snippet allows one to parametrize the number of field accesses. By duplicating both `if` blocks (lines 426–428 and lines 431–433) and replacing the field `a1` of the field-store and load by another field, for example `a2`, the complexity of the data-flow increases as data flows to all access paths within the set $\{\texttt{x}.f_1.f_2.\cdots.f_m \mid f_i \in \{\texttt{a1}, \dots, \texttt{an}\}, m \in \mathbb{N}\}$. We call the program with n field accesses EXPL_n.

For this performance experiment, we stepwise scale the number of fields accesses n in the program EXPL_n and trigger points-to queries to Boomerang as well as to Boomerang$_{AP}$. For Boomerang$_{AP}$ we run queries configured using access graphs and k-limited access paths with values $k = 1, \dots, 5$. For each query and each program EXPL_n, we measure the analysis time which we report on.

Results. Figure 8.2 plots the number of fields n of EXPL_n on the x-axis against the query's analysis time on the y-axis. The chart depicts 7 line plots, one for Boomerang, one for Boomerang$_{AP}$ using access graphs, and 5 lines for AP^k with $k = 1, \dots, 5$.

A first observation is that the analysis times of access graphs increase exponentially when more than 5 fields occur along the data-flow. With $n = 5$, the access-graph-based analysis already takes 17 seconds to terminate. For values larger $n = 6$, the queries hit the budget of 50 seconds.

We next compare access-path-based Boomerang$_{AP}$ to SPDS. $AP^{k=1}$ is more efficient than SPDS but also expected to be much less precise. SPDS is slightly less efficient compared to $AP^{k=1}$, but the latter is imprecise as soon as the data-flow path involves more than a single field store. Lastly, Boomerang$_{AP}$ queries in the configuration $AP^{k=3}$ are roughly as performant as the query configured to use access graphs. However, an increase of k to 4 and 5 yields to analyses with worse performance.

Although the heap models are equally precise, Boomerang based on SPDS clearly outperforms the access-graph based version of Boomerang in this experiment. The line for SPDS shows a quadratic growth when using SPDS, and, even

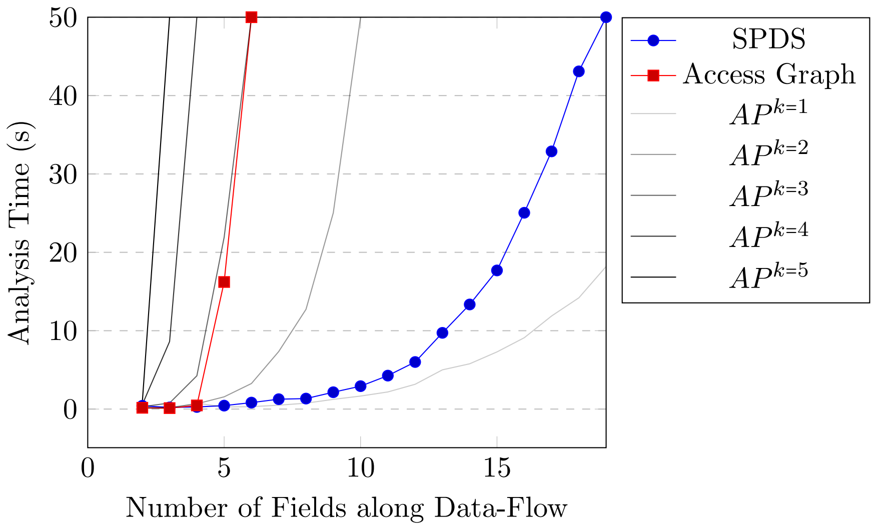


Figure 8.2: The number of relevant field accesses for a data-flow analysis and its effect on the analysis time.

on EXPL$_{18}$, which nests the query object into 18 different fields, the data-flow query finishes in 43 seconds.

This experiment shows the benefit of using SPDS opposed to access graphs or k-limiting when data flows through a sequence of field-stores. This experiment showcases that the explosion of the size of the data-flow domain directly relates to the analysis time.

Summary. When data flows through five or more nested field-stores, BOOMERANG based on SPDS is more efficient than BOOMERANG$_{AP}$ using access paths or access graphs. SPDS show a performance close to k-limited access paths with $k = 1$ although their precision corresponds to $k = \infty$.

8.1.2 Precision and Performance of a Typestate Analysis

In this experiment, we evaluate the precision and performance of SPDS in comparison to the access graph heap model on real-world programs. We re-use the IDE^{al}-based typestate analysis discussed in Section 6.4 and run the same analysis implemented in IDE$^{al}_{AP}$.

Experimental Setup. We use the typestate specifications detailed in Table 6.1 and applied both versions IDE^{al} and IDE$^{al}_{AP}$ to each of the 11 programs of the DaCapo benchmark suite. At first, the analyses pre-compute a context-insensitive SPARK-based call graph [55], determine all abstract objects, for instance, allocation sites of type `Vector` or `Iterator`, and execute a static analysis per abstract object.

Table 8.1: Statistics of a typestate analysis performed in IDE^{al} (▨) and IDE$^{al}_{\mathcal{P}}$ (🟥) on the DaCapo 2006 benchmark programs. A row containing a dash in column *Objects* means no object (allocation site) was found in the program and no statistics are collected. Note: the analysis includes all libraries.

	IO						**Vector**						**Iterator**					
	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)	Objects	Visited Methods	Nesting Depth	Total Time (s)	Timeouts	Rel. Timeouts (%)
ANTLR	17	12	4	4,023 15	5 0	▨ 29.4 🟥 0	2	178	3	150 601	0 1	▨ 0 🟥 50	-	-	-	- -	- -	▨ 0 🟥 0
BLOAT	36	234	9	15,604 3,491	26 3	▨ 72.2 🟥 8.3	9	182	6	1,203 119	2 0	▨ 22.2 🟥 0	288	23	9	72,062 6,104	105 3	▨ 36.5 🟥 1.0
CHART	18	10	4	3,602 22	6 0	▨ 33.3 🟥 0	4	18	1	1 5	0 0	▨ 0 🟥 0	21	5	4	6,601 4	11 0	▨ 52.4 🟥 0
ECLIPSE	24	14	5	6,363 40	10 0	▨ 41.7 🟥 0	24	62	10	6,001 1,210	10 2	▨ 41.7 🟥 8.3	5	7	8	660 2	1 0	▨ 20 🟥 0
FOP	16	14	4	4,218 24	7 0	▨ 43.8 🟥 0	4	29	6	601 143	1 0	▨ 25.0 🟥 0	4	5	1	1 2	0 0	▨ 0 🟥 0
HSQLDB	54	248	9	27,483 10,502	45 15	▨ 83.3 🟥 27.8	1	17	1	1 2	0 0	▨ 0 🟥 0	-	-	-	- -	- -	▨ 0 🟥 0
JYTHON	69	157	11	25,593 6,184	41 10	▨ 59.4 🟥 14.5	36	659	13	15,338 13,370	24 14	▨ 66.7 🟥 38.9	5	7	9	180 3	0 0	▨ 0 🟥 0
LUINDEX	15	12	3	3,604 14	6 0	▨ 40 🟥 0	15	30	4	2,193 28	3 0	▨ 20 🟥 0	6	5	1	31 2	0 0	▨ 0 🟥 0
LUSEARCH	15	13	5	3,243 19	5 0	▨ 33.3 🟥 0	14	134	13	4,471 3,019	7 5	▨ 50 🟥 35.7	11	5	10	2,044 3	3 0	▨ 27.3 🟥 0
PMD	18	12	4	3,736 18	5 0	▨ 27.8 🟥 0	13	282	5	6,000 2,094	10 3	▨ 76.9 🟥 23.1	54	5	8	10,631 7	17 0	▨ 31.5 🟥 0
XALAN	16	11	3	2,582 18	4 0	▨ 25.0 🟥 0	3	20	1	0 3	0 0	▨ 0 🟥 0	2	5	7	62 1	0 0	▨ 0 🟥 0

We ran this experiment on a 2.3 GHz Intel Core i7 machine, and we granted 12 GB of heap memory to the JVM. During the computation, we record two statistics of IDE^{al} about the data-flow of each abstract object. First, the number of *Visited Methods*, which counts the methods that IDE^{al} propagates data-flow facts within. Second, we compute the *Nesting Depth* of an object which is the length of the longest acyclic path contained in any of the analysis' automata $\mathcal{A}_{\mathbb{F}}$. In the case all paths are acyclic, the nesting depths reflects the minimal value for k-limiting to avoid approximation. For EXPL$_n$, this value is equal to n. To limit the total analysis time to an acceptable time budget for programs using many abstract objects, we limit the computation of the data-flow for each abstract object to 10 minutes.

Precision Results SPDS is based on the hypothesis that an improperly matched call site does not induce a properly matched field access and vice versa. SPDS over-approximates when the target program contains two distinct paths d_1 and d_2 such that d_1 properly matches one language ($L_{\mathbb{F}}$ or $L_{\mathbb{S}}$) but does match in the second language and conversely for path d_2 (see Section 4.4.1). For the typestate analysis IDE^{al}, this over-approximation would lead to an additionally reported finding (false positive) in comparison to IDE$^{al}_{AP}$, because it would cause the analysis to construct an invalid data-flow path. Yet, for all objects for which IDE^{al} and IDE$^{al}_{AP}$ terminate (464 out of 819 objects), both analyses report the *same results*. This evidence shows that our hypothesis is true in practice.

Performance Results. Table 8.1 lists the results of the typestate analysis grouped by the three typestate properties (`IO`, `Vector`, and `Iterator`) on the DaCapo benchmark suite. Each row of the table corresponds to one program. For each property, column *Objects* lists how many `Vector`, `Iterator`, or `IO` allocation sites the program contains in the pre-computed call-graph.[2] Column *Visited Methods* shows the average number of methods visited when computing the data-flows for all objects. The column *Nesting Depth* represents the average nesting depth of all objects. The last three columns reflect the analysis time: The column *Total Time* lists the accumulated analysis time of all objects on the benchmark (excluding call-graph construction time). The column *Timeouts* shows the *number* of objects for which the data-flow analysis exceeded the budget of 10 minutes, the last column, *Rel. Timeouts* shows the *fraction* of those objects over all analyzed ones. The last three columns, *Total Time*, *Timeouts*, and *Rel. Timeouts* are split horizontally into two rows per program. For each program, the upper row contains data for the access-graph based implementation IDE$^{al}_{AP}$, and the lower row contains data for IDE^{al}.

For example, the program ANTLR allocates a total of 17 objects related to `IO`, e.g., of type `FileInputStream` or `FileOutputStream`. Across these 17 objects, on average, the data-flow path visits 12 methods and stores the object within 4 unique fields, indicating that an access path of length at least 4 is required for a precise analysis. For 5 of the 17 analyzed objects, IDE$^{al}_{AP}$ times out, whereas IDE^{al} does not time out on any object. The analysis time for IDE$^{al}_{AP}$ totals to 4,023 seconds, whereas IDE$^{al}_{AP}$ computes the same data-flows in only 15 seconds.

The results show that IDE^{al} outperforms IDE$^{al}_{AP}$ for the typestate properties `IO` and `Iterator` on all DaCapo benchmarks in terms of the total analysis time and the number of timeouts. On geometric average IDE^{al} is 83× more efficient than IDE$^{al}_{AP}$ for the property `Iterator`. For the `IO` property, IDE^{al} is 64× more efficient. For the typestate property `Vector`, IDE^{al} outperforms IDE$^{al}_{AP}$ only by a factor of 1.8×.

The timeouts dominate the overall analysis time. Switching from IDE$^{al}_{AP}$ to IDE^{al}, i.e., from access graphs to SPDS, reduces the timeouts from 160 to 28 for all 298 `IO` objects, and from 57 to 25 for the 125 `Vector` objects. For the 396 `Iterator` object data-flows, the difference is most significant: with IDE$^{al}_{AP}$, a total of 137 data-flows time out, while only 3 time out with IDE^{al}.

Table 8.1 also lists the maximal nesting depth for each analysis. The values indicate the minimal k-limit for a precise analysis using k-limited access paths. The values range up to $k = 13$ and indicate that the program EXPL$_{13}$ of the micro-experiment that uses 13 different fields along a data-flow is not an unreal-

[2]The allocation sites, i.e., the number of analyzed objects in this experiments differs from the results reported in Table 6.4. In Table 6.4 restricts the objects to the objects that are call graph reachable in both SOOT and WALA.

istic scenario of a target program for a typestate analysis.

Summary. On most of the DaCapo benchmark programs, the typestate analysis IDE^{al} outperforms the access-graph-based analysis IDE$^{al}_{AP}$ in terms of analysis time. While IDE^{al} times out on only 6.8% (56 out of 819) object data-flows, IDE$^{al}_{AP}$ times out on 252 data-flows, i.e., on 37% of the objects.

8.1.3 Visited Methods and Nesting Depth

We further seek to relate the theoretic worst-case complexity results of Section 4.4.2 to the practical performance results we obtained. For access-path and access-graph-based analyses, the more deeply a traced object is nested in another object, the longer access path are constructed and the higher the expected analysis time for the object is. While the time also increases for SPDS, due to the concise representation of $\mathcal{A}_{\mathbb{F}}$, it is expected that the time is affected less heavily. On the other hand, for data-flows that reach more statements, i.e., have a high number of visited methods, SPDS is expected to have a higher complexity.

Experimental Setup For the two typestate analysis based on IDE^{al} and IDE$^{al}_{AP}$, we measured the nesting depth, the visited methods, and the analysis time per object data-flow. These statistics give a detailed view on the analysis.

Results First, we explain the variance in the performance reported in Table 8.1 across the typestate properties by relating the values in the *Timeout* column to the values of *Visited Methods* and *Nesting Depth.*

IDE^{al} times out for only 3 of the 396 `Iterator` objects, and, in total, times out in only 52 of all 819 objects. This is a significant reduction, which we explain as follows. For all benchmarks, the number of visited methods for the typestate `Iterator` is relatively low. On average, an `Iterator` object is alive across 5–23 methods. These methods include the factory calls where the `Iterator` is allocated, as well as their constructors. Ignoring loops, `Iterator` objects are nested in other object's fields in a depth between 1 and 10. The access paths required for `Iterator` are also cyclic. For example, for the benchmark program Eclipse, the following access path is required to be tracked

```
config.this$0.ig.nodes.map.tail.parent.next.prev.right
```

where the part `tail.parent.next.prev.right` can occur in any arbitrary order. IDE$^{al}_{AP}$, which uses access graphs, must store these access graphs for each statement individually, because they might slightly vary. On the other hand, in the case of IDE^{al}, the automaton $\mathcal{A}_{\mathbb{F}}$ represents all access paths at every statement concisely in a single automaton.

The `Vector` typestate property is the other extreme. The performance gains in IDE^{al} through SPDS are negligible. As data containers, `Vector` objects are expected to have a longer lifetime than `Iterator` objects. In Table 8.1, the number of visited methods for `Vector` objects ranges from 17 to 659. The longer

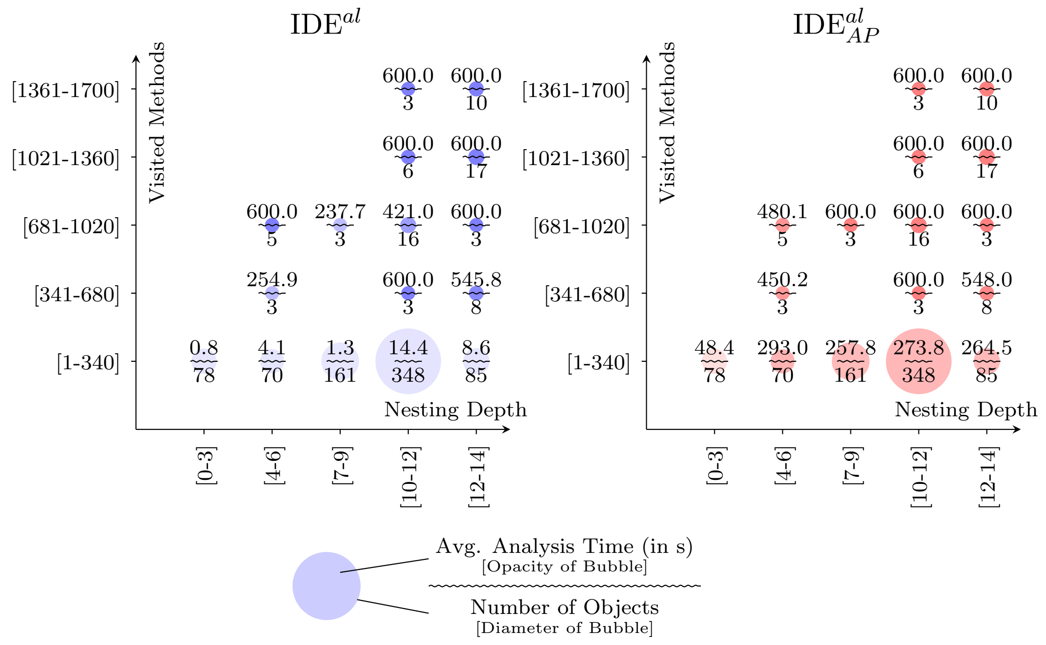


Figure 8.3: Relating analysis times to the number of visited methods and the nesting depth along objects data-flow for IDE^{al} and IDE$^{al}_{AP}$.

lifetime of the objects reduces the benefit of SPDS which is designed to improve scalability in the dimension of the nesting depth.

The number of visited methods of a data-flow as a metric for the lifetime of an object motivates our second representation of the data set. In Figure 8.3, we group all objects, regardless of their type, into buckets, depending on the nesting depth and the number of visited methods along their data flows. The visited methods are plotted along the y-axis, and the number of fields along the x-axis. We subdivide both axes into five equally long ranges which generates a total of 25 buckets. For each bucket, we report two statistics in Figure 8.3. The first is the number of objects contained in the bucket, indicated by the value below the meandered line in the diagram. The second statistic is the average analysis time of the objects within this bucket, which is indicated by the number above the meandered line. Timeouts are included in the average with their 10 minutes. We also visualize these statistics as circles associated to the bucket. The diameter of the circle corresponds to the number of objects contained in a bucket. The more opaque a circle is, the more time the analysis took on average across the objects. Figure 8.3 shows a diagram for IDE^{al} on the left and for IDE$^{al}_{AP}$ on the right.

The diagram shows two important characteristics. First, for the vast majority of objects, IDE^{al} significantly reduces the analysis times compared to IDE$^{al}_{AP}$. Second, the more methods a data-flow visits, the larger the analysis time. While the latter holds for IDE$^{al}_{AP}$ and IDE^{al}, the number of field-stores along the data-flow paths does influence the analysis times more heavily for IDE$^{al}_{AP}$.

IDE^{al} shows the largest speedups for the bottom part of Figure 8.3. In other words, switching from access graphs to SPDS benefits data-flows which visits few methods, no matter how deeply the object is nested. IDE^{al} effectively reduces the analysis time for all buckets whose visited method range is [1-340]. Figure 8.3 shows that the majority of data-flows fall into this range, which contains 743 out of all 819 objects and require only a fraction of all call-graph reachable methods for the analysis. Additionally, Figure 8.3 shows that IDE^{al} times out slightly more often when the number of visited methods increases. This observation aligns with the worst-case complexity analysis.

Summary. The number of visited methods and the number of fields participating in the data-flow are two influencing factors for the analysis times of IDE^{al}. However, the number of visited methods has a higher impact and SPDS is most advantageous in situations where the data-flow spans few methods but flows through many fields.

8.2 Factors on Maven Central

While Boomerang and IDE^{al} compute results efficiently in many analysis scenarios, across all our evaluations we also face analysis timeouts. Those hinder the computation of results, because only partial information is returned. For the typestate analysis on the DaCapo benchmark suite (Section 6.4.2), we observed 12.5% timeouts with a query budget of 30 seconds which reduces to 6.8% (Section 8.1.2) when a timeout of 10 minutes is selected. For the datarace client (Section 5.5.4) and a tight budget of 1 second, we report 96.6% of timeouts on the same benchmarks. For the analysis of *CryptoAnalysis* on the artifacts of Maven Central (Chapter 7) and a time budget of 5 seconds, 16.0% of all IDE^{al} and Boomerang queries time out. For a better understanding of the occurrence of timeouts, we discuss influencing factors and their correlation to the analysis time of Boomerang and IDE^{al}.

Experimental Setup. The discussion is based on data recorded during the large scale experiment of *CryptoAnalysis* on Maven Central (Chapter 7). The static analysis for *CryptoAnalysis* composes of multiple Boomerang and IDE^{al}-based sub-queries. The IDE^{al}-based queries perform a typestate analysis and start forward propagating at allocation sites of object of interest. The Boomerang queries start backward propagating and search for variables at selected call sites. For each query, we limited the analysis time to a maximum of 5 seconds.

A Boomerang and IDE^{al} query constructs multiple forward and backward-directed SPDS, each consisting of two PDS ($\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$) and two $\mathcal{P}$-automata ($\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{F}}$). The backward-directed SPDS searches for allocation sites of a query variable (at a statement), while the forward-directed SPDS start propagation in these allocation sites. During the data-flow computation for each query, we record the *field transitions* and *call transitions*, which represents the number of (unique) transitions of all automata $\mathcal{A}_{\mathbb{F}}$, $\mathcal{A}_{\mathbb{S}}$ respectively. The number of all (unique) *call rules* and *field rules* is the count of all rules of all $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$. The

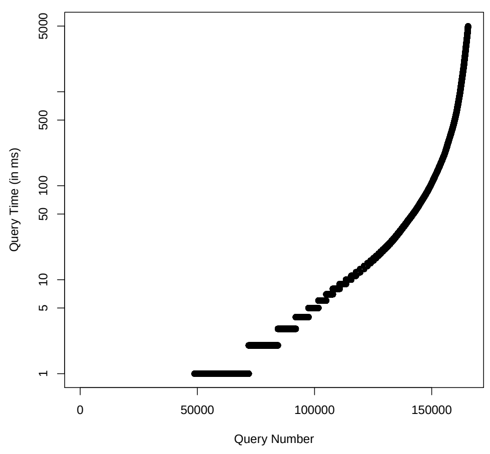


Figure 8.4: Query times for the BOOMERANG and IDE^{al} queries in *CryptoAnalysis*.

number of *forward nodes* is the size of the union of the sets $post_{\mathbb{SF}}$ across all forward-directed SPDS. This metric corresponds to the number of nodes of the *ESG* of IFDS/IDE. Likewise, the number of *backward nodes* is the count for all backward-directed SPDS. We also record how many data-flows escape to *static fields* or to *arrays*.

Result. Across all artifacts, *CryptoAnalysis* triggers a total of 182,825 queries to BOOMERANG and IDE^{al}. Of these queries, 165,521 (90.5%) terminate within the time budget of 5 seconds. Figure 8.4 shows the distribution of the analysis times of all terminated queries. The queries are sorted with respect to their analysis time. For 97.8% of the terminating queries (162,144 of 165,521), the query time remains below a second, then the analysis time grows exponentially. This trend is similar to the trend of the complete analysis time of *CryptoAnalysis* of all artifacts (Figure 7.3a). The observed exponential growth makes it difficult to predict the analysis times, and we agree with recent work that found context-sensitive points-to analysis (as BOOMERANG) to have unpredictable performance [58,87].

Table 8.2 provides further details on main factors influencing the analysis times. We compute the correlation coefficient between the query time and other factors, e.g., number of forward nodes. The closer to 1 the correlation coefficient is for two data rows, the higher their correlation. Table 8.2 is ordered by the coefficient, and it shows that the number of call transitions correlates most (0.67) to the query time. The more call transitions the automata $\mathcal{A}_{\mathbb{S}}$ contain, the longer the query times. The same holds for the number of field transitions for which the correlation coefficient has a value of 0.63. The field rules and the call rules influence the amount of transitions within the respective automata, hence it is not surprising that the sizes of the rule sets correlate to the query time.

Table 8.2: Correlation coefficient related to the query time.

Correlation coefficient ordered by significance	
Call Transitions	0.67
Call Rules	0.66
Field Rules	0.64
Field Transitions	0.63
Forward Nodes	0.53
Backward Nodes	0.49
Visited Methods	0.47
Array Flows	0.12
Static-Field Flows	0.04

The metric of visited methods correlates with value 0.47 to the query times. While the correlation is not as high as for call transitions or field transitions, the data-flow visited methods have an influence on the query times. Opposed to this, the metrics array flows and static-field flows do not correlate (coefficient of 0.04 and 0.12) to the query time. We assumed these two metrics to be relevant because static field and array flows are difficult to control: A static field can be accessed anywhere in the code and a static field propagates almost through the whole program. *CryptoAnalysis* is array-insensitive and all elements stored to an array alias. Our hypothesis that propagation of static fields as well as arrays easily prevent an analysis from being efficient is inconclusive.

Summary. The distribution of the query times is exponential and hence difficult to predict. The query times correlates most to the shapes of the two pushdown systems $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ that reflect to the call and field transitions of the automaton $\mathcal{A}_{\mathbb{F}}$ and $\mathcal{A}_{\mathbb{S}}$. However, flows to arrays and across static fields are not likely to impact the query time significantly.

8.3 Future Work

Based on the results of this chapter, we see a few possibilities of future work to improve the analysis times and reduce the timeouts.

- Pushdown systems enable a range of optimizations, for example *summarization* [49] where sub-automata of $\mathcal{A}_{\mathbb{S}}$ (or $\mathcal{A}_{\mathbb{F}}$) are shared across multiple post*-computations across different object data-flows. Sharing results between multiple $\mathcal{A}_{\mathbb{S}}$ is certainly possible by means of Lal's proposed techniques. For automaton $\mathcal{A}_{\mathbb{F}}$, it is more difficult: at points of aliasing, the field automata are explicitly manipulated, which might become difficult to coordinate when parts of the automata are shared.
- The pushdown systems contain many rules that propagate the same information from statement to statement (identity rules), we see a potential to

make the pushdown system sparse and save on rules and, in turn, transitions. A reduction of the rules of $\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ is expected to reduce the memory consumption and the analysis time as less rules need to be stored and applied.

- Orthogonal to the optimizations on the pushdown systems themselves, one can (and we currently are) investigating demand-driven call-graph refinement [91] and its impact on the query time. In particular, for analyses that use an imprecise call graph, e.g., *CryptoAnalysis* that uses CHA for the experiments in Chapter 7, a reduction in the number of callees at call sites can be expected. This in turn, may lower the analysis time.

9 Conclusion

Finding an acceptable balance between precision, recall, and performance of a static analysis is a tedious task when designing and implementing a static analysis. Efficient but imprecise analyses frequently produce an unacceptable amount of false positives, conversely, precise analyses need to encode a drastically larger - even infinite - data-flow domain that leads to analyses that are difficult to scale.

Synchronized pushdown systems (SPDS), the first contribution of this thesis (Chapter 4) presents a precise and efficient solution to a known to be undecidable problem [73]. SPDS compute a field- and context-sensitive data-flow analysis and reduce the over-approximations to corner cases that are non-existent in practice, corner cases for which an improperly matched call site does not induce a properly matched field access and vice versa. SPDS synchronizes the results of two pushdown systems, one that models field-sensitivity and a second that models context-sensitivity. The experiments performed within this thesis show that SPDS are an efficient replacement for the standard storeless k-limited access path model. In our evaluation of precision of SPDS compared to access paths, we could not identify any of the above mentioned corner case to occur meaning that SPDS, in our experiments, incur no false positives. In summary, SPDS are as precise as $k = \infty$ while being as efficient as $k = 1$.

The demand-driven pointer analysis Boomerang, presented in Chapter 5, addresses a fundamental problem in data-flow analysis. Boomerang computes the objects (or memory locations) a pointer variable may point to at runtime. Boomerang coordinates multiple SPDS. On its own, each of the SPDS efficiently solves the distributive part of the pointer relations and hands over the non-distributive part to the points of aliasing of Boomerang. The data-flow framework IDE^{al} that we present in Chapter 6 extends the ideas of the distributive propagations of Boomerang and additionally propagates weights along the data-flow. The weights allow data-flow analysis such as a typestate analysis or an analysis for API usage pattern mining. The experiments performed with IDE^{al}, in which we compare an IDE^{al}-based typestate analysis to a state-of-the-art typestate analysis, advocate that distributive propagation pays-off as a whole.

In Chapter 7, we also specifically apply our data-flow frameworks within an analysis that detects security vulnerabilities and weaknesses. Through an execution of the analysis on a large software repository for Java libraries, we confirm once again the efficiency of synchronized pushdown systems.

We hope that the presented algorithms, frameworks, and their implementations help static analysis designers in the future to lower their burden of finding the right approximation and the correct aliases for their analysis problem at hand.

Publications and Contributions

Over the course of this thesis I authored and co-authored multiple publications. The list below states these publications and describes my contributions.

- *Johannes Lerch, **Johannes Späth**, Eric Bodden, and Mira Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In International Conference on Automated Software Engineering (ASE), 2015.* [53]

  Johannes Lerch came up with the initial concept and an implementation, furthermore he evaluated the approach by himself. I contributed to conceptual and technical discussions, and helped Johannes Lerch to sharpen the formalism. I also supported the technical writing of the publication. The concept inspired my work on SPDS as presented in Chapter 4.

- ***Johannes Späth**, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In European Conference on Object-Oriented Programming (ECOOP), 2016.* [90]

  Parts of this publication are reused in Chapter 5. I am the main author of this publication. The concept of BOOMERANG was inspired by other publications [5, 100]. I implemented the analysis and performed the evaluation myself. The benchmark suite POINTERBENCH was co-authored by Lisa Nguyen Quang Go. Eric Bodden and Karim Ali joined technical discussions and co-authored the publication.

- ***Johannes Späth**, Karim Ali, and Eric Bodden. IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis. In Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2017.* [88]

  Parts of this publication are reused in Chapter 6. I am the main author of this publication. The concept for IDE^{al} was developed in technical discussion with Karim Ali and Eric Bodden. I implemented the framework and conducted the evaluation myself. The co-authors helped with designing the experiments and writing the publication.

- *Stefan Krüger, **Johannes Späth**, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In European Conference on Object-Oriented Programming (ECOOP), 2018.* [47]

  This publication presents parts of the work discussed in Chapter 7. I am a co-author of the publication and contributed to designing and writing a compiler that transforms CrySL into its static analysis. I developed and implemented large parts of the compiler. I formalised CrySL's Formal Semantics (Section 5). In collaboration with Stefan Krüger, I setup and conducted the evaluation, furthermore I contributed to the technical writing of the publication.

- ***Johannes Späth***, *Karim Ali, and Eric Bodden. Context-, Flow- and Field- Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. In Symposium on Principles of Programming Languages (POPL), 2019.* [89]

  The contents of this publication are reused in Chapter 4. I am the main author of this publication. I developed the concept of SPDS myself. I designed, setup and performed the evaluation which incorporated feedback from my co-authors. Karim Ali and Eric Bodden helped me writing the publication and sharpened the presentation.

Bibliography

[1] L.O. Andersen and Københavns Universitet. Datalogisk Institut. *Program Analysis and Specialization for the C Programming Language*. DIKU rapport. Datalogisk Institut, 1994.

[2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting Doop to Soufflé: A Tale of Inter-Engine Portability for Datalog-Based Analyses. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*, pages 25–30, 2017.

[3] Steven Arzt. *Static Data Flow Analysis for Android Applications*. PhD thesis, Darmstadt University of Technology, Germany, 2017.

[4] Steven Arzt and Eric Bodden. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *International Conference on Software Engineering (ICSE)*, pages 725–735, 2016.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[6] George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. A Datalog Model of Must-Alias Analysis. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*, pages 7–12, 2017.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[8] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective Control-Flow Abstraction via Jumping. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 163–182, 2015.

[9] Eric Bodden. Interprocedural Data-Flow Analysis with IFDS/IDE and Soot. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*, pages 3–8, 2012.

[10] Eric Bodden, Reehan Shaikh, and Laurie J. Hendren. Relational Aspects as Tracematches. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 84–95, 2008.

[11] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory (CONCUR)*, pages 135–150, 1997.

[12] Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 243–262, 2009.

[13] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Symposium on Principles of Programming Languages (POPL)*, pages 289–300, 2009.

[14] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 30:1–30:30, 2018.

[15] Ben-Chung Cheng and Wen-mei W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In *Programming Language Design and Implementation (PLDI)*, pages 57–69, 2000.

[16] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *International Symposium on Static Analysis (SAS)*, pages 260–278, 2001.

[17] Arnab De and Deepak D'Souza. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 665–687, 2012.

[18] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Programming Language Design and Implementation (PLDI)*, pages 230–241, 1994.

[19] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *Programming Language Design and Implementation (PLDI)*, pages 567–577, 2011.

[20] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *International Conference on Computer and Communications Security (CCS)*, pages 73–84, 2013.

[21] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.

[22] Javier Esparza and Andreas Podelski. Efficient Algorithms for Pre* and Post* on Interprocedural Parallel Flow Graphs. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–11, 2000.

[23] Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-Supported Refactoring for JavaScript. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 119–138, 2011.

[24] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 465–484, 2015.

[25] Pietro Ferrara. Generic Combination of Heap and Value Analyses in Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 302–321, 2014.

[26] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–144, 2006.

[27] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.

[28] Alain Finkel, Bernard Willems, and Pierre Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. *Electronic Notes in Theoretical Computer Science*, pages 27–37, 1997.

[29] Manuel Geffken, Hannes Saffrich, and Peter Thiemann. Precise Interprocedural Side-Effect Analysis. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 188–205, 2014.

[30] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In *International Symposium on Static Analysis (SAS)*, pages 240–260, 2006.

[31] Neville Grech and Yannis Smaragdakis. P/Taint: Unified Points-To and Taint Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 102:1–102:28, 2017.

[32] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 108–124, 1997.

[33] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-Up Shape Analysis. In *International Symposium on Static Analysis (SAS)*, pages 188–204, 2009.

[34] Samuel Z. Guyer and Calvin Lin. Client-Driven Pointer Analysis. In *International Symposium on Static Analysis (SAS)*, pages 214–236, 2003.

[35] Ben Hardekopf and Calvin Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *International Symposium on Code Generation and Optimization (CGO)*, pages 289–298, 2011.

[36] David Hauzar, Jan Kofron, and Pavel Bastecký. Data-Flow Analysis of Programs with Associative Arrays. In *International Workshop on Engineering Safety and Security Systems (ESSS)*, pages 56–70, 2014.

[37] Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. In *Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001.

[38] Michael Hind. Pointer Analysis: Haven't we Solved this Problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[39] David Hovemeyer and William Pugh. Finding Bugs is Easy. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 132–136, 2004.

[40] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. A Relational Approach to Interprocedural Shape Analysis. In *International Symposium on Static Analysis (SAS)*, pages 246–264, 2004.

[41] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 13–22, 2009.

[42] Vini Kanvar and Uday P. Khedker. Heap Abstractions for Static Analysis. *ACM Computing Surveys (CSUR)*, pages 29:1–29:47, 2016.

[43] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An Efficient Data Structure for Must-Analysis. In *Compiler Construction (CC)*, pages 48–58, 2018.

[44] George Kastrinis and Yannis Smaragdakis. Hybrid Context-Sensitivity for Points-To Analysis. In *Programming Language Design and Implementation (PLDI)*, pages 423–434, 2013.

[45] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap Reference Analysis Using Access Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007.

[46] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages (POPL)*, pages 194–206, 1973.

[47] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 10:1–10:27, 2018.

[48] Akash Lal and Thomas W. Reps. Improving Pushdown System Model Checking. In *International Conference on Computer Aided Verification (CAV)*, pages 343–357, 2006.

[49] Akash Lal and Thomas W. Reps. Solving Multiple Dataflow Queries Using WPDSs. In *International Symposium on Static Analysis (SAS)*, pages 93–109, 2008.

[50] Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended Weighted Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*, pages 434–448, 2005.

[51] Johannes Lerch. *On the Scalability of Static Program Analysis to Detect Vulnerabilities in the Java Platform.* PhD thesis, Darmstadt University of Technology, Germany, 2016.

[52] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 98–108, 2014.

[53] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths. In *International Conference on Automated Software Engineering (ASE)*, pages 619–629, 2015.

[54] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-To Analysis with Efficient Strong Updates. In *Symposium on Principles of Programming Languages (POPL)*, pages 3–16, 2011.

[55] Ondrej Lhoták and Laurie J. Hendren. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction (CC)*, pages 153–169, 2003.

[56] Ondrej Lhoták and Laurie J. Hendren. Context-Sensitive Points-to Analysis: Is it Worth it? In *International Conference on Compiler Construction (CC)*, pages 47–64, 2006.

[57] Xin Li and Mizuhito Ogawa. Stacking-Based Context-Sensitive Points-To Analysis for Java. In *Hardware and Software: Verification and Testing - International Haifa Verification Conference (HVC)*, pages 133–149, 2009.

[58] Yue Li, Tian Tan, Anders Møler, and Yannis Smaragdakis. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *International Symposium on Foundations of Software Engineering (FSE)*, November 2018.

[59] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.

[60] Magnus Madsen and Anders Møller. Sparse Dataflow Analysis with Pointers and Reachability. In *International Symposium on Static Analysis (SAS)*, pages 201–218, 2014.

[61] Ravi Mangal, Mayur Naik, and Hongseok Yang. A Correspondence Between Two Approaches to Interprocedural Analysis in the Presence of Join. In *European Symposium on Programming (ESOP)*, pages 513–533, 2014.

[62] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 365–383, 2005.

[63] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-To and Side-Effect Analyses for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, 2002.

[64] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions on Software Engineering and Methodology*, pages 1–41, 2005.

[65] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping Through Hoops: Why do Java Developers Struggle with Cryptography APIs? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[66] Nomair A. Naeem and Ondrej Lhoták. Typestate-Like Analysis of Multiple Interacting Objects. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 347–366, 2008.

[67] Nomair A. Naeem and Ondrej Lhoták. Faster Alias Set Analysis Using Summaries. In *Compiler Construction (CC)*, pages 82–103, 2011.

[68] Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. In *Compiler Construction (CC)*, pages 124–144, 2010.

[69] Mangala Gowri Nanda and Saurabh Sinha. Accurate Interprocedural Null-Dereference Analysis for Java. In *International Conference on Software Engineering (ICSE)*, pages 133–143, 2009.

[70] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *International Symposium on Static Analysis (SAS)*, pages 165–180, 2004.

[71] Michael Pradel and Thomas R. Gross. Leveraging Test generation and Specification Mining for Automated Bug Detection Without False Positives. In *International Conference on Software Engineering (ICSE)*, pages 288–298, 2012.

[72] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call Graph Construction for Java Libraries. In *FSE*, pages 474–486, 2016.

[73] Thomas W. Reps. Undecidability of Context-Sensitive Data-Independence Analysis. *ACM Transactions on Programming Languages and Systems*, pages 162–186, 2000.

[74] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[75] Thomas W. Reps, Akash Lal, and Nicholas Kidd. Program Analysis Using Weighted Pushdown Systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 23–51, 2007.

[76] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. *Science of Computer Programming*, pages 206–263, 2005.

[77] Thomas W. Reps, Emma Turetsky, and Prathmesh Prabhu. Newtonian Program Analysis via Tensor Product. In *Symposium on Principles of Programming Languages (POPL)*, pages 663–677, 2016.

[78] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[79] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[80] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, pages 131–170, 1996.

[81] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.

[82] Stefan Schwoon. *Model checking pushdown systems.* PhD thesis, Technical University Munich, Germany, 2002.

[83] Stefan Schwoon, Somesh Jha, Thomas W. Reps, and Stuart G. Stubblebine. On Generalized Authorization Problems. In *International Workshop on Computer Security Foundations (CSFW)*, page 202, 2003.

[84] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* Computational Risk Management. 1978.

[85] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Programming Language Design and Implementation (PLDI)*, pages 693–706, 2018.

[86] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your Contexts well: Understanding Object-Sensitivity. In *Symposium on Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[87] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective Analysis: Context-Sensitivity, Across the Board. In *Programming Language Design and Implementation (PLDI)*, pages 485–495, 2014.

[88] Johannes Späth, Karim Ali, and Eric Bodden. IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99:1–99:27, 2017.

[89] Johannes Späth, Karim Ali, and Eric Bodden. Context-, Flow- and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. In *Symposium on Principles of Programming Languages (POPL)*, 2019.

[90] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 22:1–22:26, 2016.

[91] Manu Sridharan and Rastislav Bodík. Refinement-Based Context-Sensitive Points-To Analysis for Java. In *Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[92] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. 2013.

[93] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-Driven Points-To Analysis for Java. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 59–76, 2005.

[94] Bjarne Steensgaard. Points-To Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[95] Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *Transactions on Software Engineering (TSE)*, pages 157–171, 1986.

[96] Yulei Sui and Jingling Xue. Demand-Driven Pointer Analysis with Strong Updates via Value-Flow Refinement. *Technical Report*, 2017.

[97] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 264–280, 2000.

[98] Tian Tan, Yue Li, and Jingling Xue. Efficient and Precise Points-To Analysis: Modeling the Heap by Merging Equivalent Automata. In *Programming Language Design and Implementation (PLDI)*, pages 278–291, 2017.

[99] Emina Torlak and Satish Chandra. Effective Interprocedural Resource Leak Detection. In *International Conference on Software Engineering (ICSE)*, pages 535–544, 2010.

[100] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 210–225, 2013.

[101] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Programming Language Design and Implementation (PLDI)*, pages 87–97, 2009.

[102] Guoqing (Harry) Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-Reachability-Based Points-to Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 98–122, 2009.

[103] Eran Yahav and G. Ramalingam. Verifying Safety Properties Using Separation and Heterogeneous Abstractions. In *Programming Language Design and Implementation (PLDI)*, pages 25–34, 2004.

[104] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Demand-Driven Context-Sensitive Alias Analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165, 2011.

[105] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Programming Language Design and Implementation (PLDI)*, pages 435–446, 2013.

[106] Qirun Zhang and Zhendong Su. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability. In *Symposium on Principles of Programming Languages (POPL)*, pages 344–358, 2017.

[107] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid Top-Down and Bottom-Up Interprocedural Analysis. In *Programming Language Design and Implementation (PLDI)*, pages 249–258, 2014.

[108] Xin Zheng and Radu Rugina. Demand-Driven Alias Analysis for C. In *Symposium on Principles of Programming Languages (POPL)*, pages 197–208, 2008.

[109] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.