Precise Scalable Static Analysis for Application-Specific Security Guarantees

Precise Scalable Static Analysis for Application-Specific Security Guarantees

A dissertation presented
by
Andrew Arthur Johnson
to
The School of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of
Computer Science

Harvard University
Cambridge, Massachusetts

May 2015

Dissertation Advisor: Professor Stephen Chong Andrew Arthur Johnson

Precise Scalable Static Analysis for Application-Specific Security Guarantees

Abstract

This dissertation presents Pidgin, a static program analysis and understanding tool that enables the specification and enforcement of precise application-specific information security guarantees. Pidgin also allows developers to interactively explore the information flows in their applications to develop policies and investigate counter-examples.

Pidgin combines *program dependence graphs* (PDGs), which precisely capture the information flows in a whole application, with a *custom PDG query language*. Queries express properties about the paths in the PDG; because paths in the PDG correspond to information flows in the application, queries can be used to specify global security policies.

The effectiveness of Pidgin depends on the precision of the static analyses used to produce program dependence graphs. In particular it depends on the precision of a points-to analysis. Points-to analysis is a foundational static analysis that estimates the memory locations pointer expressions can refer to at runtime. Points-to information is used by clients ranging from compiler optimizations to security tools like Pidgin. The precision of these client analyses relies on the precision of the points-to analysis. In this dissertation we investigate points-to analysis performance/precision trade-offs, including a novel points-to analysis for object-oriented languages designed to help establish object invariants.

This dissertation describes the design and implementation of Pidgin and the points-to analyses that allow Pidgin and other static analyses to scale to large applications. We report on using Pidgin: (1) to explore information security guarantees in legacy programs; (2) to develop and modify security policies concurrently with application development; and (3) to develop policies based on known vulnerabilities.

Contents

Listing of figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Stephen Chong. Without his hard work, guidance, and patience, this dissertation would not have been possible. I also thank committee members Greg Morrisett and Eddie Kohler for their valuable feedback. They especially helped provide perspective when it was needed.

Without my collaborators this work would have been a lot more difficult and a lot less fun. Monica Chao helped lay the foundation for our flow-sensitive points-to analysis. Scott Moore and Lucas Waye contributed to the design and implementation of Pidgin. In addition everyone in the Harvard programming languages group in some way to made my experience more enjoyable, and I learned much through osmosis, eavesdropping on (and occasionally contributing to) discussions.

My family were all extremely supportive of my decision to return to graduate school. My mom instilled in me a love of learning and the desire to never stop, and my dad's weekly conversations reminded me that there was someone else who cared about my progress.

There is not enough space to enumerate the contributions my wife, Kim, has made to my life. This work would not have been possible without her love and support. Long walks with Kingsley was the best time to work through difficult problems, and Napoleon was always ready to play, distracting me from those same problems. Lastly, I would like to thank our new son, Oliver, for giving me something new to strive for.

1

Introduction

Many applications store and compute with sensitive information, including confidential and untrusted data. Thus, application developers must be concerned with the information security guarantees their application provides, such as how public outputs may reveal confidential information and how potentially dangerous operations may be influenced by untrusted data. These guarantees will necessarily be application specific, since different applications handle different kinds of information, with different requirements for the correct handling of information. Moreover, these guarantees are properties of the entire application, rather than properties that arise from the correctness of a single component.

Current tools and techniques fall short in helping developers address information security. Testing cannot easily verify information-flow requirements such as "no information about the password is revealed except via the encryption function." Existing tools for information-flow security are inadequate for a variety of reasons, since they either unsoundly ignore important information flows, require widespread local annotations, prevent functional testing and deployment, or fail to support the specification and enforcement of application-specific policies.

The goal of this work is to enable developers to express and enforce application-specific security policies for programs written in existing languages with support for both legacy applications and new development.

This dissertation presents a methodology that makes use of static program analysis. We focus our attention on the security of imperative object-oriented languages. In these languages the precision of most static analyses, including all of those mentioned in this dissertation, depends on the precision of a points-to analysis, a fundamental static program analysis that approximates which memory locations each program expression may refer to. In Chapter 2 and Chapter 3 we present scalable multi-threaded points-to analyses and explore different trade-offs in precision and performance. In Chapter 4 we use the results of a points-to analysis and several client static analyses to construct *program dependence graphs* (PDGs) [29] that precisely and intuitively capture the information flows within an entire program.[1] We show how PDGs together with a custom PDG query language can enable the exploration, specification, and enforcement of application-specific information security guarantees. We present a tool called PIDGIN that uses the techniques described in this dissertation and demonstrate the use of PIDGIN to discover and describe security policies for both legacy and newly developed applications.

1.1 POINTS-TO ANALYSIS

In imperative languages, the precision and utility of the static analyses used to reason about application security and other important program properties rely on the results of a points-to analysis. Points-to analysis is a foundational static analysis that computes which memory locations pointers may reference.[2] In this dissertation we chose to design and implement a custom points-to analysis to easily explore the different trade-offs between pre-

[1] PDGs for a whole program are also called *system dependence graphs* [47].

[2] There are also points-to analyses that compute memory locations a pointer *must* point to. The soundness of the client analyses we use relies on a conservative overapproximation of points-to information, i.e., as computed by a *may*-point-to analysis. This dissertation focuses on may-point-to analyses.

cision and performance.

Choosing which type of *context sensitivity* to use is one of the most important performance/precision trade-offs a particular points-to analysis must make [52]. Our points-to analysis analysis is parameterized *à la* Kastrinis and Smaragdakis [52], enabling *programmable* context sensitivity. This allows us to express many different points-to analyses, including object-sensitive analyses [64, 76, 100], call-site-sensitive analyses [96, 97], and others. This programmability can also be used to define a custom context sensitivity that provides exactly the precision needed.

In general, a points-to analysis that produces more precise results (e.g., by choosing more precise context sensitivity) takes more time to compute those results. In order to effectively reason about the security properties of whole programs a points-to analysis needs to scale to hundreds of thousands of lines of code without loss of precision. To take advantage of multi-core and multi-processor architectures our analysis is multithreaded. To scale this analysis we precisely track dependencies and use a difference propagation algorithm [28, 63, 84].

1.2 Flow sensitivity and strong update

Another particularly interesting dimension of points-to analysis design is *flow sensitivity*. Flow-insensitive points-to analyses, such as that described in Chapter 2, compute points-to information that may hold at any point in the program's execution. By contrast, flow-sensitive points-to analyses compute points-to information for each program point, which is potentially more precise but less scalable. The precision of flow-sensitive points-to analyses is particularly appealing because it can enable *strong update* [15], whereby an assignment can *replace* the statically computed facts associated with a memory location. By con-

trast, in a weak update, an assignment *adds* to the statically computed facts of a memory location.

Strong update to an abstract memory location can be performed only when the abstract location corresponds to exactly one concrete memory location.[3] To ensure that there are abstract locations that usefully correspond to exactly one concrete location, Balakrishnan and Reps [6] introduce the recency abstraction, in which objects created at a given allocation site are represented by two abstract objects: one represents the object most recently allocated at the allocation site and one summarizes all other objects allocated at the allocation site. A field of the most-recently-allocated abstract object represents exactly one concrete location, and thus strong update can be performed on it.

Lhoták and Chung [62] introduce a points-to analysis for C programs that is flow-sensitive only for abstract locations on which strong update may be possible and treats other abstract locations flow insensitively, which significantly improves the performance of the analysis.

In this dissertation we combine the recency abstraction with the insights of Lhoták and Chung [62] to define a novel points-to analysis for object-oriented languages that is flow-sensitive only for abstract locations on which strong update may be possible, i.e., for fields of most-recently-allocated abstract objects and for static variables.[4] Our analysis treats other abstract locations (i.e., fields of non-most-recently-allocated abstract objects) flow insensitively. Moreover, since object construction typically takes place on the most recently allocated object, the recency abstraction enables strong update exactly where it is

[3] In object-oriented analyses, an *abstract object* represents zero or more *concrete objects*, and abstract locations include fields of abstract objects as well as local and static variables.

[4] Although abstract locations for local variables can be soundly strongly updated, we use a partial static single assignment (SSA) program representation [20], which gives many of the benefits of flow-sensitive analysis for local variables, with the efficiency of flow-insensitive analysis.

most useful for object-oriented analyses: during the establishment of object invariants.

1.3 Application specific security

This dissertation presents a novel approach to application-specific security combining program-dependence graphs with a custom graph query language. PDGs express the control and data dependencies in a program, and abstract away unimportant details such the sequential order of non-interacting statements. They are a great fit for reasoning about information security guarantees, since paths in the PDG correspond to information flows in the application. Our queries express properties of PDGs which thus correspond to information-flow guarantees about the application.

Often application-specific security guarantees are enforced with labor intensive and error-prone testing and auditing based on informal policy specifications. It is also difficult or impossible to verify certain information-flow properties with this approach.

Security-typed languages (e.g., Volpano et al. [114], Jif [78] and FlowCaml [98]) can also be used to enforce application-specific policies for programs written in full-featured programming languages. In these languages, all expressions carry a *security type* defining the allowed information flows between expressions. Global security policies are broken into many pieces and expressed via type annotations throughout the program. If an application type-checks then the security policy specified by these annotations holds. This is problematic for at least three reasons. First, it is difficult to determine from these annotations how sensitive information is handled by the system as a whole, particularly in the presence of declassification [94]. Second, changing the security policy may require modifying many program annotations. Third, supporting legacy applications using these techniques is often infeasible, as they require significant annotations and/or modifications to

the applications.

Dynamic or hybrid information-flow enforcement mechanisms (e.g., [4, 5, 14, 48, 57, 106]) are sometimes able to specify security policies separate from code, but interfere with the deployment of systems: they must be used during testing in order to ensure that enforcement does not conflict with important functionality.

Taint analysis tools (e.g., [17, 27, 65, 110, 111, 122]) are inevitably unsound because they do not account for information flow through control channels, and often do not support expressive application-specific policies.

Pre-defined policies (such as policies that might be enforced on all Android apps) can capture many security requirements of broad classes of applications. However, applications handle different types of sensitive information (e.g., bank account information, health records, school records, etc.) and what constitutes correct handling of this information differs between applications. Pre-defined policies cannot express these application-specific security requirements.

Previous PDG-based information security tools (e.g., [30, 35, 37]) have many of the same issues as security-typed languages. For all but the simplest security policies, these tools require program annotations to specify security policies, with the concomitant issues regarding legacy applications, modifying security policies, and understanding the system-wide security guarantees implied by the annotations.

Moreover, all these existing techniques focus almost exclusively on enforcement of security guarantees and do not support exploration.

Our approach as embodied in our tool, PIDGIN, addresses several of the weaknesses discussed above.

— *PIDGIN security policies are expressive, precise, and application specific*, since they are

queries in an expressive query language designed specifically for finding and describing information flows in a program. Queries can succinctly express *global* security guarantees such as noninterference [31], absence of explicit information flows, trusted declassification [42], and mediation of information-flow by access control checks.

- Developers can *interactively explore an application's information security guarantees*. If there is no predefined security specification then PIDGIN can be used to quickly explore the security-relevant information flows in a program and discover and specify the precise security policies that an application satisfies. If a policy is specified but not satisfied, then PIDGIN can help a developer understand why by finding information flows that violate the policy.
- *PIDGIN security policies are not embedded in the code.* PIDGIN policies are specified separate from the code. The code doesn't require program annotations nor does it mention or depend on PIDGIN policies. This enables the use of PIDGIN to specify security guarantees for legacy applications without requiring modification of the application.
- *Enforcement of security policies does not prevent development or testing.* Because the program code does not mention or depend on PIDGIN policies, the policies do not prevent compilation or execution. This makes it possible for developers to choose a balance between development of new functionality and maintenance of security policies.
- *PIDGIN enables regression testing of information security guarantees.* PIDGIN can be incorporated into a build process to warn developers if recent code changes violate a security policy that previously held. This includes information-flow properties that

traditional test cases can not easily detect.

1.4 Contributions and outline

This dissertation presents a new approach to application-specific security and demonstrates its utility with Pidgin, a tool that implements this methodology for Java bytecode.

Since our approach is based on static analysis we rely heavily on a precise and scalable points-to analysis. Chapter 2 describes the design and implementation of the custom multithreaded implementation of a context-sensitive flow-insensitive points-to analysis used by Pidgin. We discuss the different trade-offs that must be made when designing a points-to analysis and how we achieve scalability. We also report on the performance of our implementation for Java bytecode. To our knowledge, this is the first multithreaded points-to analysis with parameterized context sensitivity and the first multithreaded implementation of a context-sensitive points-to analysis for an object-oriented language.

Chapter 3 focuses on a particularly interesting points-to analysis trade-off, flow sensitivity, as well as exploiting the precision improvement enabled by strong update. We describe a novel analysis for object-oriented languages that is specifically designed to enable strong update. We show that this analysis scales well and improves precision in several client analyses. Note that we do not use this partially flow-sensitive analysis in our implementation of Pidgin.

Chapter 4 presents our approach to application-specific security as embodied in Pidgin. The primary contributions of this chapter are:

1. The novel insight that PDGs offer a unified approach that enables *exploration*, *specification*, and *enforcement* of security guarantees.
2. The design of an expressive language for precise, application-specific security poli-

cies, based on queries evaluated against PDGs.

3. The realization and demonstration of these insights and techniques in an effective and scalable tool.

Chapter 4 describes our methodology, the structure of the PDGs we generate, and security guarantees that can be expressed as queries in PidginQL, our custom graph query language. We also discuss our implementation for Java bytecode. This chapter culminates in Sections 4.6 and 4.7 where we relate our experience using Pidgin to discover, specify, and enforce security guarantees in several legacy applications and as part of the development process. We show that Pidgin scales to large applications and that we can find and enforce strong and precise security guarantees even for legacy applications where the security policy is not part of an a priori specification.

The work on Pidgin was done in collaboration with Lucas Waye, Scott Moore, and Stephen Chong and is adapted from Johnson et al. [49]. The flow-sensitive points-to analysis is joint work with Ling-Ya (Monica) Chao and Stephen Chong.

2

Multithreaded Points-to Analysis

2.1 Introduction and background

A *points-to analysis* or *pointer analysis* computes an approximation of the *points-to graph*, a map from the pointer expressions in a program to the set of memory locations to which they refer [26].[1] A more precise points-to analysis computes a better estimate of the locations that pointers can point to at runtime. The precision of static analyses for imperative languages often relies heavily on the precision of a points-to analysis. Client analyses that depend on the results of a points-to analysis range from simple compiler optimizations to complex analyses used to reason about the correctness and security of whole applications.

Computing an exact points-to graph is an undecidable problem so the best we can do is compute an approximation.[2] In general a more precise points-to graph takes longer to compute. There are a number of well known points-to analysis performance/precision trade-offs. We chose to implement our own points-to analysis rather than use an off-the-shelf analysis in order to more easily be able to explore these trade-offs and in order to take advantage of multi-core and multi-processor architectures.

To our knowledge, our analysis is the first multithreaded points-to analysis that supports

[1] A points-to analysis is related to but distinct from an *alias analysis* which determines which pairs of pointer expressions may refer to the same memory location [59].

[2] The undecidability of points-to analysis can be derived from the undecidability of the halting problem [55, 89].

many different types of context sensitivity and the first multithreaded context-sensitive analysis for an object-oriented language. Throughout this dissertation we restrict ourselves to analyses for object-oriented languages, although much of what we describe is not specific to these languages and is relevant to analyses for any imperative language.

2.1.1 May or must

Most points-to analyses compute the set of memory locations each pointer may point to (also called the *points-to set*). This is known as a *may* points-to analysis. A may points-to analysis is sound if it conservatively over-approximates each points-to set. This is in contrast to a *must* points-to analysis which computes an *under-approximation* of the memory locations each pointer must point to at run-time. The soundness of most client analysis, including all those in this dissertation, depend on the results of a *may* points-to analysis. [3] For the rest of this dissertation we will consider only may points-to analyses.

2.1.2 Modeling memory locations

There are a potentially unbounded number of memory locations that are dynamically allocated at runtime. In order for our points-to analysis to terminate we need some finite abstraction. We abstract heap locations in an object-oriented points-to analysis using *abstract objects*. An abstract object represents zero or more concrete objects (i.e., objects dynamically created at run-time). The most common abstraction is to model an object by its static allocation site. In this so-called *allocation-site abstraction* an abstract object represents all objects that are dynamically created at a given static allocation site. Less precise representations include using a single abstract object per type or even a single abstract

[3] In some cases a must-point-to analysis can be used to improve the precision of a may points-to analysis and client analyses [26]. This is beyond the scope of this dissertation.

object for the entire heap. In Chapter 3 we the *recency abstraction* introduced by Balakrishnan and Reps [6] which uses two abstract objects per allocation site; one representing the most-recently-allocated object and another representing all other allocations. There are also more complex *shape analyses* that base abstractions on various properties of the heap [34, 95, 117].

2.1.3 Context sensitivity

In order to provide useful results for sophisticated client analyses a points-to analysis must compute *interprocedural* points-to information, i.e., points-to information for an entire application taking into account the calling relationships between procedures. The number of methods called during the execution of a program is unbounded and, as a static analysis that must terminate, we model these using a finite abstraction. One simple choice is to analyze each method at most once, merging the results for all possible call sites. This is known as a *context-insensitive* analysis.

A *context-sensitive* analysis [26], on the other hand, differentiates multiple calls to the same procedure based on an *analysis context* (e.g., the static procedure call site). A "good" choice of analysis context for method calls groups "similar" dynamic call sites under the same context and differentiates call sites that should be kept distinct. The definitions of "good" and "similar" in the previous sentence depend on many factors including the language being analyzed, the client analysis that will use the points-to results, and even common programming idioms and design patterns. As a result, the choice of context is largely application-specific. The choice of context-sensitivity also has a large effect on performance (see e.g., [26, 52, 76, 100]). We use the abstraction of Kastrinis and Smaragdakis [52] to allow our points-to analysis to be run with many different types of context-sensitivity.

In object-oriented points-to analyses abstract objects can also include a context, called

```
1  a = new C1();
2  b = new C2();
3  foo() {
4      Object x = bar(a);
5      Object y = bar(b);
6  }
7
8  bar(Object z) {
9      return z;
10 }
```

Figure 2.1: Program where context sensitivity can increase precision

a *heap context* in Smaragdakis et al. [100] to differentiate it from the procedure analysis contexts used to distinguish different method calls and described above. Heap contexts serve the same purpose as method contexts, enabling the fine-grained partitioning of an unbounded number of memory locations into a finite number of abstract objects. This is achieved by incorporating information about the context in which an allocation occurs into these abstract objects. For example an abstract object may inherit the context of the allocating procedure.

To see why context sensitivity can produce a more precise points-to graph, consider the simple example in Figure 2.1. There are two calls to the method `bar` each with a different argument. If this code were analyzed context-insensitively then `bar` would be analyzed just once and the return value of this single method would point to `C1` and `C2`.[4] The points-to set of `x` would be similarly imprecise. If we used a *call-site sensitive analysis* [26], where analysis contexts are based on the static call site, then the calls to `bar` on lines 4 and 5 would be analyzed separately. The return from `bar` on line 4 would point only to `C1` and we

[4]We use the types `C1` and `C2` to describe the allocations on lines 1 and 2. In the analysis we use abstract objects to represent allocated objects. The use of types is to simplify the exposition.

would correctly determine that `x` also points only to `C1`. The cost of this added precision is analyzing `bar` an extra time. In general the cost and precision of a points-to analysis analysis varies dramatically with the choice of context [52, 76, 100].

Several different types of analysis context are commonly used, many of which are captured by the parameterization introduced by Smaragdakis et al. [100] and refined by Kastrinis and Smaragdakis [52].

C : contexts
CS : call-sites
AS : allocation sites
AO : abstract objects
$\textsc{Record} : AS \times C \to AO$
$\textsc{Merge} : AO \times CS \times C \to C$
$\textsc{MergeStatic} : CS \times C \to C$

Figure 2.2: Functions used to define new procedure analysis contexts and abstract objects.

The abstraction of Kastrinis and Smaragdakis [52] is defined by the three functions seen in Figure 2.2. C ranges over contexts, CS ranges over method call sites, AS ranges over allocation sites, and AO ranges over abstract objects. At a call site cs for a non-static method, the function Merge computes the new context for the callee from the abstract object ao for the receiver object and the context c of the method caller. MergeStatic similarly computes contexts for callees at call sites to static methods which have no receiver. Record is used to compute an abstract object at an allocation site as in a method with context c.[5]

These three functions can be used to instantiate a number of different context-sensitive

[5]Kastrinis and Smaragdakis [52] treat abstract objects as an allocation site and a separate heap context. We combine the two in an *abstract object* and only discuss abstract objects in this dissertation. This gives us the flexibility of ignoring the allocation site and using a coarser grained abstraction such as the type of the object being allocated. Abstract objects that do not include the allocation site tend to produce results that are too imprecise to be useful, which is why Kastrinis and Smaragdakis [52] always include it.

Call-site sensitive

$C = CS^n$	
$AO = AS$	
$\textsc{Merge}(ao, cs, c)$	$push(cs, c, n)$
$\textsc{MergeStatic}(cs, c)$	$push(cs, c, n)$
$\textsc{Record}(as, c)$	as

Full-object sensitive

$C = AS^n$	
$AO = AS^n$	
$\textsc{Merge}(ao, cs, c)$	ao
$\textsc{MergeStatic}(cs, c)$	c
$\textsc{Record}(as, c)$	$push(as, c, n)$

Type sensitive

$C = \text{ClassName}^n$	
$AO = AS \times \text{ClassName}^m$	
$\textsc{Merge}(ao, cs, c)$	$push(allocatingClass(fst(ao)), snd(ao), n)$
$\textsc{MergeStatic}(cs, c)$	c
$\textsc{Record}(as, c)$	$(as, first_m(c))$

Figure 2.3: Different context sensitivities and the merge and record functions which define their behavior.

analyses including call-site sensitive analyses, object-sensitive analyses [76], type-sensitive analyses [100], and others. We show three such instantiations in Figure 2.3. The function $push(e, s, n)$, pushes an element, e, onto a stack, s, keeping only the n most recently added elements, thus maintaining a maximum stack height of n. The function $first_n$ returns the first n elements of a stack, and *allocatingClass* extracts the class name of the allocating object from an allocation site.

In a call-site sensitive analysis the contexts correspond to the top n frames of the call stack at the time the analyzed method is called. The two context creation functions add the current call site to the top of the stack. Abstract objects are named with the object's

allocation site.

In a full-object-sensitive analysis contexts for dynamic dispatch methods are based on the receiver object abstraction. More precisely the context for a particular method call in a full-object-sensitive analysis is the abstract object for the receiver of that method call. Abstract objects are derived from the allocation site and the context of the method where the allocation occurs, i.e., combining the new allocation site with the allocation site of the object doing the allocation. A full-object-sensitive analysis is parameterized by an integer n; contexts and abstract objects are both stacks of allocation sites of height n. A larger n will result in a more precise analysis. At an allocation site, the abstract object for the newly allocated object is computed by pushing the new allocation site onto the current analysis context. The analysis context used at a non-static call site is the abstract object for the receiver. At a static call site there is no receiver so the context for the caller is used. We could also use the distinguished initial context for static call sites which would produce less precise results, but be more scalable. This analysis is extremely effective for object-oriented languages: more effective than a call-site sensitive analysis since distinguishing methods called on different receivers often produces better client analysis results than distinguishing different call sites [76].

A type-sensitive analysis is similar to a full-object-sensitive analysis except that the types of allocating objects are used in place of allocation sites. Type sensitive analyses are parameterized by n, the number of types in an analysis context stack, and m, the number of types in the abstract object stack. Note that at a call site the type of the object containing the receiver's allocation site is used rather than the type of the receiver since the method signature already contains a lot of information about the receiver type. This analysis has been shown to be more efficient than an full-object-sensitive analysis and provides many of the

same benefits [100].

The MERGE/RECORD parameterization allows a composite analysis to be built from other analyses using a simple cross product operation. For example, when using an full-object-sensitive analysis all calls to static methods are analyzed using the caller's context (since these methods have no receiver). This can lead to imprecision similar to that found in Figure 2.1. By taking the product of a standard full-object-sensitive analysis with an analysis that is call-site sensitive analysis only for static method calls, we can gain the benefits of an object-sensitive analysis for virtual calls while also being precise when analyzing static calls.

We can also use this parameterization to define custom context sensitivities that precisely meet the needs of client analyses. Smaragdakis et al. [100] recommend a type sensitive analysis with $n = 2$ and $m = 1$ (called a 2type+1H analysis). We found this analysis to be too imprecise for client analyses that need to reason about Java collections. Using an easy-to-extend context parameterization allows us to create a custom analysis by taking the cross product of the 2type+1H analysis with an analysis that is more precise only for classes in the Java collections framework.

2.1.4 FLOW SENSITIVITY AND STRONG UPDATE

A *flow-sensitive* points-to analysis computes a different points-to graph for each program point. A flow-insensitive points-to analysis computes a single points-to graph that is a conservative approximation of the points-to relations that hold across all program points. By taking control flow into account, a flow-sensitive analysis can be more precise than a flow-insensitive analysis. This additional precision can result in more opportunities for *strong update*. When a strong update is performed at an assignment to a particular abstract location, the points-to information for that location is *replaced* by new points-to information.

This is in contrast to a *weak update* where an an assignment *adds* to the points-to information for an abstract location. Strong update can be performed at an assignment any time the points-to set for the assignee corresponds to a single concrete memory location. A flow-sensitive points-to analysis can also allow client analyses to perform strong update, replacing statically computed facts at some assignments. We discuss more about flow sensitivity and strong update in Chapter 3 where we present a novel points-to analysis specifically designed to enable strong update for object-oriented programs. In this chapter we restrict our attention to a flow-insensitive analysis.

2.1.5 Static single assignment form

In *static single assignment* (SSA) form [20] each local variable is assigned to at exactly one static location in the program. When transforming to SSA form variables that are assigned to multiple times are split into multiple copies, one for each assignment, and each copy is given a different name. At control-flow join points φ assignment statements are inserted to combine different copies of the same variable. For example, an SSA variable cannot be assigned-to in two branches of a conditional, as this violates the SSA assumption. Instead assignments are made to different variables in each branch and a φ statement is created directly after the conditional that will choose which variable to use based on which branch of the conditional was taken.

Many of the benefits of flow-sensitivity mentioned in Section 2.1.4 can be achieved by transforming a program into (SSA) form. In fact much previous work on flow-sensitive analysis has involved a whole program transformation into SSA form (e.g., [15, 41, 104, 109]). While computing whole program SSA form itself requires the results of a points-to analysis (e.g., to compute SSA form for object fields), a program can be converted to *partial SSA form* efficiently and without points-to analysis results. Partial SSA form requires that

only local variables meet the SSA condition, and allows fields to be assigned to multiple times. The language we analyze throughout this dissertation is in partial SSA form. Each SSA local variable has the same points-to set throughout the program and flow sensitivity is not needed for these local variables.

2.1.6 Other trade-offs

Analysis style Most of the work on points-to analysis derives from one of two prototypes. Steensgaard-style analyses [105] are based on equivalence classes and use a union-find data structure to efficiently compute results. Andersen-style analysis [2], the style we use throughout this dissertation, is based on subset constraints and efficiently solving these constraints. Hind and Pioli [45] compares these two styles finding Steensgaard's analysis to be more efficient and Andersen's to compute more precise results. Most recent work focuses on scaling Andersen-style analyses.

Path sensitivity A path-sensitive analysis builds on flow-sensitive analysis by incorporating information learned at branch points. This can improve precision by ruling out points-to relationships that only occur on infeasible paths (e.g. a path where a number must be both positive and negative or a boolean must be both true and false). While all analyses in this dissertation are path insensitive, there is some recent work that suggests path sensitive analyses may be possible and effective [107].

Field sensitivity A field-sensitive analysis differentiates the different fields in an object or `struct`. Since our implementations are for Java bytecode, and fields in Java can be distinguished by name and cannot alias one another, we get this feature easily. Indeed, most points-to analyses for Java are field-sensitive. In C field-sensitivity is more complex;

T : class types
F : resolved instance fields
S : method signatures
M : resolved methods
V : reference variables
C : contexts
AO : abstract objects
CS : call sites
AS : allocation sites

Figure 2.4: Analysis domains

a field-sensitive analysis for C is more expensive than field-insensitive analysis but is much more precise [85]. We use a field-sensitive analysis throughout this dissertation.

2.2 Analysis

In this section we describe our points-to analysis. We present our analysis as subset relations in the style of Andersen [2]. The points-to analysis algorithm computes the smallest points-to sets that satisfy all the constraints induced by the subset relations. Our implementation for Java bytecode is discussed in Section 2.3.

Our points-to analysis assumes a class-based object oriented language with dynamic dispatch and catchable exceptions and that all allocated entities are objects. Our implementation (Section 2.3) relaxes some of these constraints (e.g., allowing the allocation of arrays), and many of the concepts presented here will be applicable to non-class based languages.

In the exposition and figures, we indicate functions with small caps (i.e., FieldsOf), variables are written italics with an initial lowercase letter (e.g., c), domains are written with italicized uppercase letters (e.g., AS), and domain elements are written using a typewriter font (e.g., `AONULL`).

2.2.1 Domains

The domains of the analysis are shown in Figure 2.4. These are the types of the values in our analysis. In addition we use *Set<D>* to indicate a finite set of domain elements of type *D*. The type of a pair of domain elements, one from *D1* and the other from *D2*, is denoted $D1 \times D2$. An element of $D1 + D2$ is either an element of *D1* or an element of *D2*.

Domain *T* is the set of class types used by the program, and Domain *F* is the set of fully resolved fields that may occur in the program. Domain *S* contains method signatures and is used to resolve dynamic dispatch. A method signature is a method name and a list of the types of the method's formal arguments. Domain *M* is the set of all fully resolved methods that may be called in the program.

Domain *V* is the set of *reference variables*, which represent program variables with reference type. Every local variable in the program with reference type (including formal method arguments and method return) has a unique $v \in V$ that represents it. Additionally, every static field of reference type is represented by a unique element of *V*.

Our analysis is context sensitive, and domain *C* is the set of contexts. A pair $(v, c) \in V \times C$ is called a *reference-variable replica* [76], a local variable *v* of a method analyzed in context *c*. Reference variables representing static fields may only appear in a distinguished initial context.

AO is the set of abstract objects each representing zero or more concrete or runtime objects. Each abstract object carries a single concrete type. *Object fields* are defined by a pair $(ao, f) \in AO \times F$, where *ao* is an abstract object and *f* is a field.

Domains *CS* and *AS* are the unique labels (i.e., program points) for call sites and allocation sites respectively.

Lookup
FieldsOf : $T \rightarrow Set\langle F\rangle$
ResolveMeth : $S \times AO \rightarrow M$
Pts : $(V \times C) + (AO \times F) \rightarrow Set\langle AO\rangle$

Context Creation
Record : $AS \times C \rightarrow AO$
Merge : $AO \times CS \times C \rightarrow C$
MergeStatic : $CS \times C \rightarrow C$

Figure 2.5: Functions used by the analysis

2.2.2 Functions

The first three functions in Figure 2.5 are functions used to lookup information needed during the analysis. FieldsOf takes a class, $\mathtt{C} \in T$, and returns the set of non-static fields on an object of type C. This includes any fields defined in a super class.

ResolveMeth takes a method signature for a dynamically dispatched method call and the abstract object for the receiver of that call and returns the fully resolved method. In our analysis we require that each abstract object has a particular concrete type, so there is always resolved method returned from this function.

Function Pts takes a reference variable replica or an object field (collectively called *points-to graph nodes*) and returns the points-to set. The points-to set is the set of all abstract objects that may be pointed to by a points-to graph node at any point in the program.

Context creation Our analysis is context sensitive and is parameterized by the three functions discussed in Section 2.1.3 and repeated in Figure 2.5. Function Record(*as, c*) provides the abstract object for an object created at allocation site *as* in context *c*. Function Merge(*ao, cs, c*) provides the callee analysis context for a dynamic-dispatch method invocation at call site *cs* in a method analyzed in context *c*, where the receiver is the abstract

object *ao*. MergeStatic(*cs, c*) provides the callee analysis context for a static method invocation at call site *cs* in a method analyzed in context *c* (i.e., there is no receiver object).

2.2.3 Points-to statement processing

We specify our analysis for a Java-bytecode-like language in static single assignment (SSA) form (see Section 2.1.5 for more on SSA form). This language includes object allocations (via `new` statements), assignments between reference variables, dynamic cast statements, φ assignment statements, and (non-static) field load and store instructions. Our language also includes exceptions and `try/catch` statements, dynamic dispatch method invocation and static method invocation.

In a pre-processing step we translate these instructions into *points-to statements*. Points-to statements represent the constraints induced by the instructions in the program. Table 2.1 presents the constraints that must be satisfied by the final points-to graph for each different type of points-to statement. Since our analysis is context sensitive a points-to statement may occur in multiple contexts.

The relations for the first points-to statement in Table 2.1 indicate that if an allocation of an object of type `C` occurs at an allocation site *as* and assigns to reference variable *to* then reference variable replica (to, c) points to the newly allocated abstract object *ao*, computed using the Record function. Also at allocation sites, the fields of a newly created abstract object point to `null`. We use a special abstract object, `AONULL`, to represent `null`.[6]

The next three relations specify constraints induced for different kinds of move statements. The first, for a simple assignment, ensures that all elements of the points-to set of

[6]Since our target language is Java bytecode fields are initially `null`. This may be different in other languages.

Table 2.1: Subset relations for points-to statements occurring in an analysis context c

Points-to statement (in context c)	Subset relations
`to = new C` at allocation site *as*	$\textsc{Pts}(to, c) \supseteq \{ao\}$ $\forall f \in \textsc{fieldsOf}(\texttt{C}).\textsc{Pts}(ao, f) \supseteq \{\texttt{AONULL}\}$ where $\textsc{Record}(as, c) = ao$
`to = from`	$\textsc{Pts}(to, c) \supseteq \textsc{Pts}(from, c)$
`to = null`	$\textsc{Pts}(to, c) \supseteq \{\texttt{AONULL}\}$
`to = (C) from`	$\forall ao \in \textsc{Pts}(from, c)$. if ao is a subtype of `C` then $ao \in \textsc{Pts}(to, c)$
`caught = (Ex) (`*notTypes*`) thrown`	$\forall ao \in \textsc{Pts}(thrown, c)$. if ao is a subtype of `Ex` $\wedge \forall T \in notTypes.ao$ is *not* a subtype of `T` then $ao \in \textsc{Pts}(caught, c)$
`to =` φ`(from`$_0$`, ..., from`$_n$`)`	for $i \in \{0, \ldots, n\}.\textsc{Pts}(to, c) \supseteq \textsc{Pts}(from_i, c)$
`to = o.f`	$\forall ao \in \textsc{Pts}(o, c).\textsc{Pts}(to, c) \supseteq \textsc{Pts}(ao, f)$
`o.f = from`	$\forall ao \in \textsc{Pts}(o, c).\textsc{Pts}(ao, f) \supseteq \textsc{Pts}(from, c)$
`to = o.m(a`$_0$`, ..., a`$_n$`) throws ex` at call site *callSite*	$\forall ao \in \textsc{Pts}(o, c)$. $\textsc{Pts}(this_t, calleeC) \supseteq \{ao\}$ for $i \in \{0, \ldots, n\}$. $\textsc{Pts}(formal_{t,i}, calleeC) \supseteq \textsc{Pts}(a_i, c)$ $\textsc{Pts}(to, c) \supseteq \textsc{Pts}(return_t, calleeC)$ $\textsc{Pts}(ex, c) \supseteq \textsc{Pts}(exception_t, calleeC)$ where $\textsc{Merge}(ao, callSite, c) = calleeC$ $\textsc{ResolveMeth}(ao, \texttt{m}) = t$
`to = C.m(a`$_0$`, ..., a`$_n$`) throws ex` at call site *callSite*	for $i \in \{0, \ldots, n\}$. $\textsc{Pts}(formal_{m,i}, calleeC) \supseteq \textsc{Pts}(a_i, c)$ $\textsc{Pts}(to, c) \supseteq \textsc{Pts}(return_m, calleeC)$ $\textsc{Pts}(ex, c) \supseteq \textsc{Pts}(exception_t, calleeC)$ where $\textsc{MergeStatic}(callSite, c) = calleeC$

the assigned reference variable replica are contained in that of the assignee. The next relation adds abstract object `AONULL` to the points-to set of replicas of a reference variable assigned the literal `null`. The relation for a checked cast, `to = (C) from`, ensures that all elements *ao* from the points-to set for replica $(from, c)$ that are subtypes of `C` are contained in the points-to set for replica (to, c).

A fourth kind of move, a filtered move, is generated from an assignment to the formal argument of a `catch` block. This relation is similar to that for a checked cast, but additionally requires that the type of elements of the points to set for replicas of *thrown* not be subtypes of any element of the set *notTypes* in order to be copied to the corresponding replica of *caught*. This additional information is used to ensure that exceptions that are definitely caught by a `catch` statement do not propagate further. The set *notTypes* generated for a particular assignment to a `catch` formal contains the types that have definitely been caught by a previous `catch` block.

The next two relations in Table 2.1 propagate points-to information at field loads and stores. The relations produced for a field load, `to = o.f`, ensures that if the receiver of the field access (o, c) points to an abstract object *ao* then (to, c) points to everything object field (ao, f) points to. The relations generated for a field store propagates points-to information from the stored reference variable replica to the appropriate object field of each abstract object the receiver of the field access can point to.

The last two entries in Table 2.1 give the relations for interprocedural propagation of points-to information at method calls. For dynamic-dispatch method invocation each receiver abstract object is used to resolve the callee method based on the type of the abstract object, and to compute the callee context, using the function Merge. In Table 2.1 the ith formal argument of method t is denoted $formal_{t,i}$ and the ith actual argument for a method

call is denoted a_i. The points-to sets for the formal arguments (excluding `this`) must be supersets of those for the actual arguments. The abstract element for the receiver needs to be added to the points-to set for the special parameter for `this`. If the method has a return value that is assigned to a reference variable *to*, then the points-to set for replica (to, c) must be a superset of that for the formal return of each resolved method. Similarly if a method throws an exception *ex* then the points-to set for replica (ex, c) must be a superset of that for the formal exception exit of each resolved method.

Static method invocation is similar, but uses function MergeStatic to compute contexts, and is not concerned with a receiver object.

At runtime, some Java bytecode instructions can throw exceptions generated by the Java virtual machine. For example a field access or dynamic-dispatch method call will throw a `NullPointerException` if the receiver is `null`. We conservatively assume that such exceptions can be thrown by all such instructions. We use a special variable to represent the generated exceptions that may be thrown by a particular instruction and create filtered move points-to statements from this variables to any `catch` blocks that may catch the exception.

2.3 Implementation

We implemented the analysis described in Section 2.2 as a multithreaded points-to analysis for Java bytecode. Our points-to analysis is built on the WAtson Library for Analysis (WALA) [13] and is written using approximately 7.5k lines of Java code. In a single-threaded preprocessing step, WALA parses and translates the stack based Java bytecode to a register based intermediate language in partial Static Single Assignment (SSA) form [20]. Our points-to analysis runs on this intermediate language.

As discussed in Section 2.1.5, by analyzing a language in SSA form we get greater pre-

cision for local variables. Since each local is assigned to at a single program point and the points-to set for each local variable can be modified only by that assignment, we get a form of flow-sensitivity for local variables even in our flow-insensitive points-to analysis. Client analyses that use the results of this flow-insensitive points-to analysis enjoy similar benefits.

Our implementation supports all Java language features except reflection. We do not directly reason about concurrency, but since our analysis is flow-insensitive for heap allocated fields, any read from the field of a shared object will be seen by every write. This means we are sound with respect to concurrency although, since we do not capture the order of reads and writes, client analyses may not be able to reason precisely about race conditions. We provide hand-written analysis signatures for some important native methods (e.g., `System.arrayCopy`). In addition, for efficiency and precision, we provide analysis signatures for a handful of Java standard library classes (e.g., `java.lang.StringBuilder`). For native methods with no hand-written analysis signature we automatically generate an analysis signature that allocates a new instance of the return type. These are potential sources of unsoundness in our analysis.

Like other practical Java points-to analyses (e.g., [11, 13]) we provide a client-selectable option to allow all instances of some common types (e.g., exceptions and strings) to be represented by a single abstract object per type, which improves performance but loses precision. Some client analyses do not rely on precision for certain types and others can regain much of this precision by other means. For example, in Section 4.5 we recover precision for `java.lang.String` objects by treating them like primitive values in the client analysis.

2.3.1 Points-to engine

The points-to analysis engine computes the least fixpoint solution to the subset constraints presented in Section 2.2. The engine is implemented using a work queue algorithm. Our algorithm is multithreaded and the queue is managed by Java's fork/join framework, which uses a work stealing algorithm [58].

As a preprocessing step a points-to statement is generated for each instruction that may modify the points-to graph. These points-to statements embody the subset relations as described in Section 2.2.3. The queue consists of these points-to statements paired with an analysis context and is initialized by adding all points-to statements for the entry-point method in the distinguished initial context. When a statement for a method invocation is processed the context for potential callees is computed using the Merge or MergeStatic function and, if a new context c is computed for a given callee, all points-to statements for that callee are paired with c and added to the work queue.

2.3.2 Dependency tracking and difference propagation

There are two classes of dependencies generated by the subset relations. There are simple subset dependencies for instruction like those for an assignment. As a reminder, the relation for an assignment `to = from` is: $\textsc{Pts}(to, c) \supseteq \textsc{Pts}(from, c)$. This means that the points-to set for (to, c) will always be a superset of the points-to set for $(from, c)$.

Because these superset constraints are so common we track them directly in the points-to engine. When the points-to set, s, for an object field or reference variable changes we propagate these changes to all the supersets of s as part of the same element of work (i.e., this propagation takes place within a single thread). We are careful to only propagate the newly added elements, a technique called difference propagation that has been used in

various forms in other points-to analyses [28, 63, 84]. Since points-to sets are monotonic (i.e., elements are only added to points-to sets during the analysis, never removed) this is a sound and efficient way to propagate changes.

We use a similar technique for checked casts and filtered moves propagating new elements only if certain subtyping relationships hold.

The second kind of dependency arises for more complex relations. Consider the relations for `o.f = from` in a context c:

$$\forall ao \in \textsc{Pts}(o, c).\textsc{Pts}(ao, f) \supseteq \textsc{Pts}(\mathit{from}, c).$$

If an element is added to the points-to set of (from, c) then the points-to engine will automatically propagate this to the points-to set for the appropriate object fields. However if an abstract object ao is added to the points-to set for (o, c) then all elements of the points-to set for (from, c) must be added to that for the object field (ao, f). We ensure this by recording a dependency from the points-to statement on the points-to set for (o, c) and reprocessing the points-to statement if that points-to set changes. Again we use a difference propagation strategy; only the new elements of the points-to set for (o, c) are used when reprocessing the points-to statement.

2.3.3 Reducing the size of the points-to graph

The complexity of inclusion-based points-to analyses based on the analysis of Andersen [2] like ours is nearly cubic in the size of the program [16, 90], although in practice this is quadratic for most Java programs [103]. These complexities were computed for a context-insensitive analysis. No formal analysis has been performed for a context-sensitive analysis; it should similarly cubic, but in the size of the program multiplied by the number of analysis contexts.

One way to improve performance is to reduce the number of elements in the domain of the points-to graph decreasing the effective program size. We can do this if we recognize when multiple points-to graph nodes (either reference variable replicas or object fields) will have identical points-to sets and collapse all these into a single node.

Some reference variables can be collapsed before the points-to analysis even begins. Any assignment to a local variable `to = from` can be removed and everywhere the points-to set for `to` is required the points-to set for `from` can be used instead. Because our language is in partial SSA form we can be sure that local variables are only assigned to once and that this is a valid and precise substitution.

Consider also the relations for load statement `to = o.f`:

$$\forall ao \in \textsc{Pts}(o, c).\textsc{Pts}(to, c) \supseteq \textsc{Pts}(ao, f).$$

Suppose we have two such load statements in the same method with the same reference variable for `o` and `f` but different reference variables for `to`. Because fields are tracked flow-insensitively (and locals are in SSA form), the points-to sets for both reference variables of `to` will be identical regardless of the order the two loads appear in the source code. This means we can remove one of the load statements and use the points-to set for one of the `to` variables in place of the other. Removing these kinds of constraints results in 30-35% less points-to statements before the analysis even starts for the benchmarks we analyze in Section 2.4.

We can also find opportunities for substitution during the analysis. If a cycle is introduced in the subset relations recorded by the points-to engine then all nodes in this cycle can be removed and represented by a single points-to graph node. A cycle is of the form $\textsc{Pts}(x, c1) \supseteq \textsc{Pts}(y, c2) \supseteq \ldots \supseteq \textsc{Pts}(z, c3) \supseteq \textsc{Pts}(x, c1)$ where the same reference variable replica appears multiple times in a chain of superset relations. If a cycle of this form

is found then all the points-to sets are identical. We identify these cycles when propagating changes from subsets to supersets and effectively replace all elements of the cycle by a single representative. This is known as *lazy cycle detection* [38] and removes 1-6% of the points-to graph nodes for the benchmarks in Section 2.4.

2.3.4 Other performance optimizations

A common optimization technique in Java is to map objects to primitive integers and use the integer representation whenever possible. This is especially effective for objects that are stored in maps or sets. We use variants of Java's `Map` and `Set` interfaces specialized for primitive integers allowing for faster insertion and containment checks as well as reducing the size of our data structures in memory.

When a points-to set is requested, an iterator for that set is returned rather than a reference to the set itself. This allows us to lazily compute elements of the sets while they are iterated. For example when a checked cast or filtered move statement is first encountered, we do not check the types of the points-to set for the replica of *from*, but instead install a *type filter*, that is only applied when elements must be computed and ensures that the correct subtype relations hold.

2.4 Evaluation

We evaluate our points-to analysis using the Dacapo benchmark suite of 11 applications [9]. These applications are described in Table 2.2. The second column gives the number of lines of code reachable from the application entry point including library code. The third column gives the number of reachable (i.e., analyzed) SSA instructions. These numbers include code from JDK version 1.6 and all other library code. The benchmarks range

Table 2.2: Applications in the Dacapo Benchmark suite [9].

Program	Lines of Code	SSA Instructions
antlr	62,750	79,395
bloat	143,649	177,942
chart	228,325	310,742
eclipse	130,236	123,860
fop	74,621	88,348
hsqldb	46,562	46,375
jython	73,378	153,496
luindex	104,306	104,774
lusearch	49,885	50,121
pmd	127,351	145,389
xalan	64,754	71,482

in size from 46k lines of code to just under 228k lines of code.

All examples in this section were run on an 32 vCPU Amazon EC2 node with 60GB of RAM and Intel Xeon E5-2680 v2 (Ivy Bridge) processors. Each vCPU is a single hyperthread and there are two hyperthreads per physical core.[7] For some of the results we present, the performance boost from hyperthreading did not outweigh the greater coordination cost when moving from 16 to 32 threads. As mentioned in Section 2.3.1 our analysis generates points-to statements as a preprocessing step. This process is single threaded and takes about ten seconds. We include this preprocessing time when comparing to WALA's analysis in Section 2.4.1, but not in the other experiments presented in this section.

Every points-to analysis chooses a different point in the complex trade-off space between performance and precision described in Section 2.1. This makes it difficult to understand exactly where in that trade-off space a particular implementation lies. One of the reasons for implementing our own analysis is to parameterize as many of these trade-offs as possible. Throughout this section we will explore different choices in this parameter space

[7] `http://aws.amazon.com/ec2/instance-types/`

and how they effect performance and scalability.

2.4.1 Comparison with WALA's points-to analysis

We compare the performance of our analysis to the points-to analysis built into the WAtson Library for Analysis (WALA) [13]. Both points-to analyses use the same front end to parse the bytecode and convert to SSA form.

In order to make this comparison as fair as possible we implemented Record, Merge, and MergeStatic functions that emulate the behavior of WALA's analysis. The Merge function computes analysis contexts based on the type of the receiver object.[8] This analysis is context-insensitive for static calls and MergeStatic always returns the same distinguished context. Abstract objects, computed by Record, are allocation sites in an analysis context (i.e., a context-sensitive allocation-site abstraction).

Exceptions, such as `NullPointerExceptions` and `ArrayIndexOutOfBoundsExceptions`, that could be generated at runtime by the JVM are treated specially in WALA's points-to analysis. Each is assigned a single abstract object and we do the same in our analysis.[9] Although WALA's points-to analysis supports reflection, we disable it for this comparison. We match our analysis signatures for native methods and other parameters to the defaults provided by WALA.

The only substantial difference between WALA's points-to analysis specification and that described in Section 2.2 is in the way we handle exception propagation. When an exception reaches a catch block and the type of the exception is a subtype of the type caught

[8] A lot of information about the type of the receiver object is inherent in the callee's method signature, meaning that this analysis context contains redundant information. In contrast, a *type-sensitive* analysis computes contexts based on the type the receiver was allocated in.

[9] This behavior is a parameter in our analysis. See Section 2.4.3 for more detail on the effect and importance of using singleton abstract objects.

by the catch block (i.e. the exception must be caught), we do not propagate the exception further. WALA, in contrast, propagates exceptions to all exceptional successors without taking into account their order. This difference is embodied by the filtered move discussed in Section 2.2.3.

Figure 2.6 shows the time our analysis takes relative to WALA's points-to analysis. For each application we scale the results by dividing the absolute time by the average time WALA's analysis takes. Each bar gives the mean of ten runs together with error bars representing one standard deviation from the mean. On the first bar we give the absolute time for WALA's analysis. This is the scaling factor. Successive bars give the relative time our analysis takes for different numbers of threads including our single threaded preprocessing step. The last group gives the relative time for running all applications.

When using one thread our analysis takes about 19% less time than WALA's on average. For this context abstraction our performance flattens out at eight threads for the smaller benchmarks. Analysis of each example programs finished in under 63 seconds when using eight threads, so there may be more contention due to lack of work. It is possible that the per-thread queues are running out of work faster with more threads, a common cause of contention in work-stealing algorithms like the one we use. On average our analysis is fastest with 16 threads when it takes about 20% as long as WALA's, a 5x speedup on average.

2.4.2 Scaling context sensitivity

Kastrinis and Smaragdakis [52] exhaustively map out the precision and performance of several different kinds of context sensitivity building on similar studies carried out in previous work by Smaragdakis et al. [100] and Milanova et al. [76]. We do not attempt to

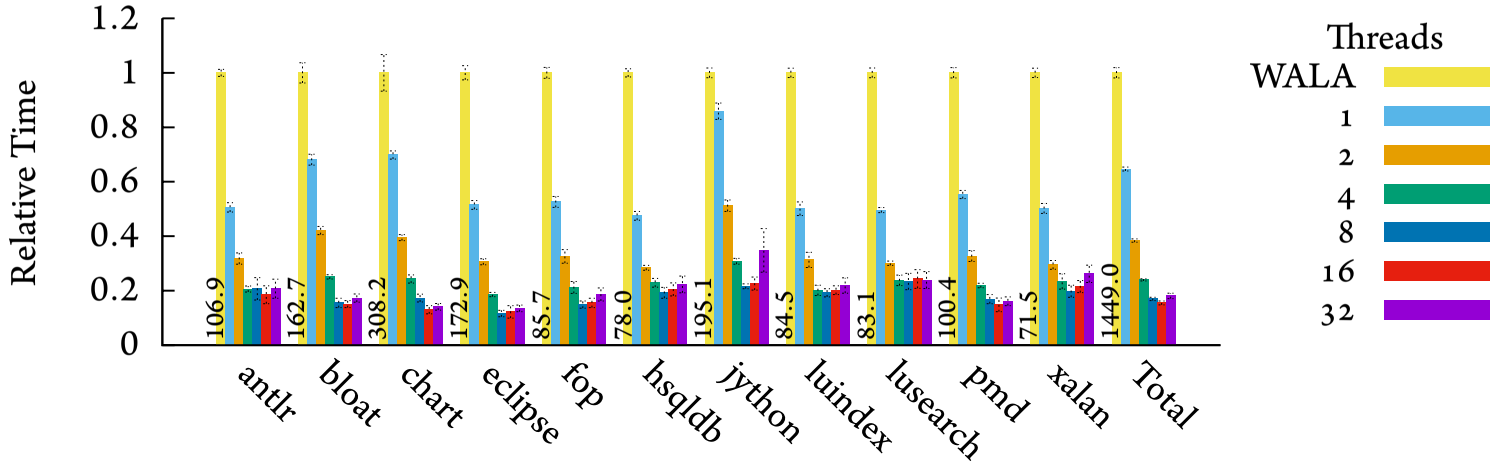


Figure 2.6: Performance comparison with the points-to analysis built into WALA [13].

replicate this here, but instead look at the scalability of a few different flavors of context sensitivity as we increase the amount of concurrency in our analysis.

For each application the bar plots in this section (Figures 2.7, 2.8, and 2.9) are scaled to the amount of time for our analysis run with a single thread. The absolute time for the single-threaded analysis (the scaling factor) is written on the first bar. In this section we do not include the preprocessing time in any of the bar charts.

For all tests in this section we use a singleton abstract object for each immutable wrapper type (`java.lang.String`, `java.lang.Integer`, etc.), each primitive array type, and each subtype of `java.lang.Throwable` (i.e., exceptions). We also use signatures for key native methods (`java.lang.System.arrayCopy`, etc.) as described in Section 2.3.

Using a work-stealing-style work queue rather than a global queue reduces coordination by providing each thread with its own queue to manage [58]. Work-queue based interactions occur only when one of the threads runs out of work. In a points-to analysis there are a number of global data structures besides the work queue that are possible sources of contention. Each element of work (i.e., a points-to statement) potentially uses and modifies the global points-to graph, the call graph, and the recorded superset relations. In addition global dependency maps are used to ensure the correct propagation of changes to both these data structures. These are the potential sources of contention in our multi-threaded analysis.

Context Insensitive Table 2.3 shows the size of the points-to graphs and call graphs when using a context-insensitive analysis. A context insensitive analysis analyzes a single copy of each method, regardless of the receiver object and call-site. This means that this

Figure 2.7: Context insensitive analysis performance relative to that for a single thread.

Table 2.3: Size of the call graph and points-to graph for a context insensitive analysis.

Program	Call Graph Nodes	Points-to Graph Nodes	Points-to Graph Edges
antlr	7,237	107,733	5,641,320
bloat	9,095	142,814	11,514,191
chart	11,832	178,161	14,195,319
eclipse	7,655	114,300	6,464,625
fop	6,755	97,760	5,212,569
hsqldb	6,497	94,433	4,951,341
jython	8,711	129,425	8,402,812
luindex	6,816	98,323	5,015,726
lusearch	6,514	93,707	4,838,386
pmd	8,250	118,031	6,334,704
xalan	6,516	93,921	4,931,801

analysis tends to produce less precise points-to information, but be more efficient to compute. Each of the examples takes less than 40.1 seconds to compute using a single thread and under 7.3 seconds when using 32 threads.

Figure 2.7 shows the scalability of the context insensitive analysis as time relative to single-threaded. In total, for all applications, two threads takes 60.8% as long as the single threaded, a 1.7x speedup. Four threads takes 36.0% as long (2.8x speedup), 8 threads 23.8% (4.2x speedup), 16 threads 22.4% (4.5x speedup), and 32 threads 21.6% (4.6x speedup). There lack of significant speedup beyond 8 threads. The absolute times are fairly small and the added coordination time washes out any performance improvement beyond 8 threads.

Type-Sensitive Figure 2.4 shows the size of the points-to graphs and call graphs when using a type-sensitive analysis. We use the language of [100] describing this particular analysis as a *2-type-sensitive points-to analysis with a 1-type-sensitive heap*. This means that analysis contexts consist of two types the first is the type that allocated the receiver object and

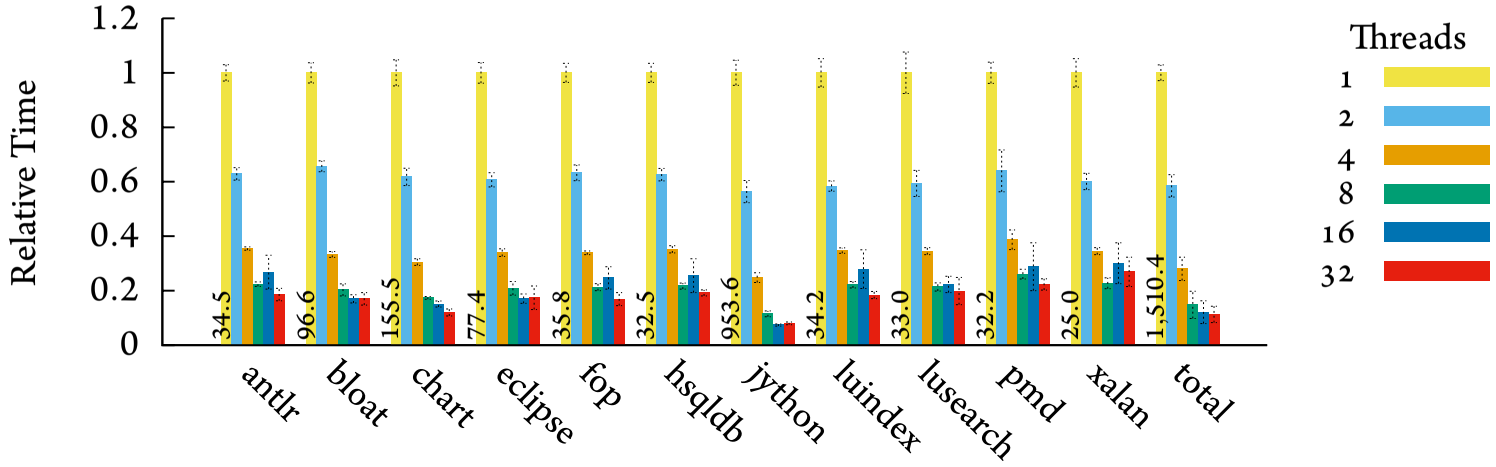


Figure 2.8: Performance relative to that for a single thread for a 2-type-sensitive analysis with a 1 type-sensitive heap.

Table 2.4: Size of the call graph and points-to graph for a 2-type-sensitive points-to analysis with a 1 type-sensitive heap.

Program	Call Graph Nodes	Points-to Graph Nodes	Points-to Graph Edges
antlr	54,725	595,517	10,216,398
bloat	79,739	856,407	55,409,374
chart	102,626	1,174,688	53,167,335
eclipse	67,472	753,795	36,174,629
fop	55,015	599,140	10,604,274
hsqldb	52,583	586,045	9,742,771
jython	134,972	1,598,154	172,342,948
luindex	54,420	591,160	10,106,414
lusearch	52,280	567,248	9,692,756
pmd	60,536	657,605	11,745,068
xalan	52,647	572,516	9,849,863

the second is the type that allocated the object that allocated the receiver. Abstract objects consist of the first element from the analysis context of the method containing the allocation site (i.e., the type that allocated the receiver object) together with the allocation site itself.

Smaragdakis et al. [100] find that a type-sensitive analysis which uses these parameters provides a good trade-off between performance and precision for many client analysis. Anecdotally we also found this to be true and use an enhanced version of this analysis for our security tool described in Chapter 4.

Even though it is more efficient than other points-to analyses [100], a type-sensitive analysis is far slower than the context-insensitive analysis we described earlier. The worst offender, jython, takes over 900 seconds single threaded and 75 seconds with 32 threads for a type-sensitive analysis, whereas the context-insensitive analysis took 24 seconds and 5 seconds respectively.

Figure 2.8 shows the scalability of a type-sensitive analysis as time relative to single-

Table 2.5: Size of the call graph and points-to graph for a 1-object-sensitive points-to analysis

Program	Call Graph Nodes	Points-to Graph Nodes	Points-to Graph Edges
antlr	45,541	447,844	4,984,359
bloat	79,385	797,831	22,918,218
chart	101,150	1,096,426	49,025,542
eclipse	51,894	528,533	8,348,777
fop	43,359	429,400	4,500,565
hsqldb	41,921	414,573	4,171,052
jython	80,140	851,614	22,419,646
luindex	42,986	425,498	4,420,326
lusearch	41,477	410,221	4,138,896
pmd	51,784	506,040	6,844,963
xalan	41,597	411,575	4,144,637

threaded. Benchmark jython has the most work and also scales the best, taking 7.4% as much time with 16 threads as with one thread a 13.5x speedup (hyperthreading 32 threads provides no benefit in this case). On average our analysis takes 12.1% of the single threaded time when using 16 threads, an over 8.2x speedup. For this analysis hyperthreading 32 threads on 16 cores takes only about 7% less time than with 16 threads. This is an 8.9x speedup over single-threaded, but some applications see little or even a negative effect moving from 16 to 32 threads.

Object Sensitive Figure 2.5 gives the size of the points-to results and Figure 2.9 shows the scalability for a full-object-sensitive analysis [100]. The particular analysis we use here is a *1-object-sensitive points-to analysis*. Abstract objects are simply the object allocation site and analysis contexts are the abstract object (i.e., the allocation site) for the receiver object. The precision of this analysis is not strictly better or worse than the type-sensitive analysis we evaluate above. Full-object-sensitive analysis contexts use the entire abstraction of the

Figure 2.9: Performance relative to that for a single thread for a 1-object-sensitive points-to analysis.

receiver object (hence *full*-object-sensitive) instead of just the type of the allocation site, but a 1-object-sensitive context contains less historical information than a 2-type-sensitive context (which tracks the type that allocated the receiver's allocator) so the precision is incomparable [100]. For these experiments the points-to graph and call graph were generally smaller for 1-object-sensitive analysis than the type-sensitive analysis, and thus performance was better. In Figure 2.9 we see about the same scalability as the type-sensitive analysis. When using 32 threads this analysis takes 12.1% of the single threaded time for an 8.3x speedup.

2.4.3 Are singletons necessary?

In Section 2.3 we briefly touched on a common technique used to achieve scalability in points-to analyses for Java. This technique is based on the *type abstraction*, a heap abstraction where each Java type is represented by a single abstract object, ignoring the particular allocation site. This is too imprecise for most client analyses and this abstraction is rarely used. However, many analyses including ours provide an option to use a single abstract object to represent all objects of *certain* types. We investigate the performance implications of using singleton abstract objects, as well as the effect on the size of the points-to graph.

The types that have the greatest affect on performance are those that are pervasive in Java applications such as `java.lang.String` and subtypes of `java.lang.Throwable`. Others, such as primitive arrays, do not have any non-primitive fields so collapsing them may have little affect on client analyses. Whether the precision lost by these optimizations is acceptable depends on the particular client analysis.

In this section we use singleton abstract objects for different sets of types in isolation to investigate the effect this has on performance. All examples use a type-sensitive analysis (2-type-sensitive points-to analysis with a 1-type-sensitive heap) and were run with 16

threads.

All experiments use single abstract object per type for JVM generated exceptions. These include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and several others. The analysis for many of the benchmarks ran out of memory if we did not use singleton abstract objects for generated exceptions. In Java, generated exceptions, in particular `NullPointerExceptions`, can be thrown by many different instructions and lead to an explosion in the size of the points-to results. By using a single abstract object we guarantee that any given points-to set will contain at most one generated `NullPointerException`, and if using an object-sensitive analysis, such as a full-object-sensitive or type-sensitive analysis, a single context will be produced for each method called on a generated exception type.

Figure 2.10 shows the effect that using a singleton abstract object for different types has on performance, while Figure 2.11 and Figure 2.12 show the effect on the size of the points-to analysis results. In general, size is inversely related to performance; bigger points-to graphs take longer to compute. As mentioned in Section 2.3.3 our analysis is nearly cubic in the size of the program, but is quadratic in practice for Java programs [103]. In each of these figures *Baseline* corresponds to using singleton abstract objects for only generated exceptions and the absolute value for the baseline for each application is shown on the first bar (this is the scaling factor). The other experiments use a singleton abstract object per type for primitive arrays, `java.lang.String`, immutable wrapper classes (including `java.lang.String`), and for subtypes of `java.lang.Throwable` (i.e., *exceptions* and *errors*) respectively. The experiment labeled "All" uses singleton abstract objects for all these types. The bars show the values relative to the baseline. We do not show results for the jython

Figure 2.10: Relative performance improvement when using singleton abstract objects for a given set of types.

Figure 2.11: Relative number of points-to graph nodes when using singleton abstract objects for a given set of types.

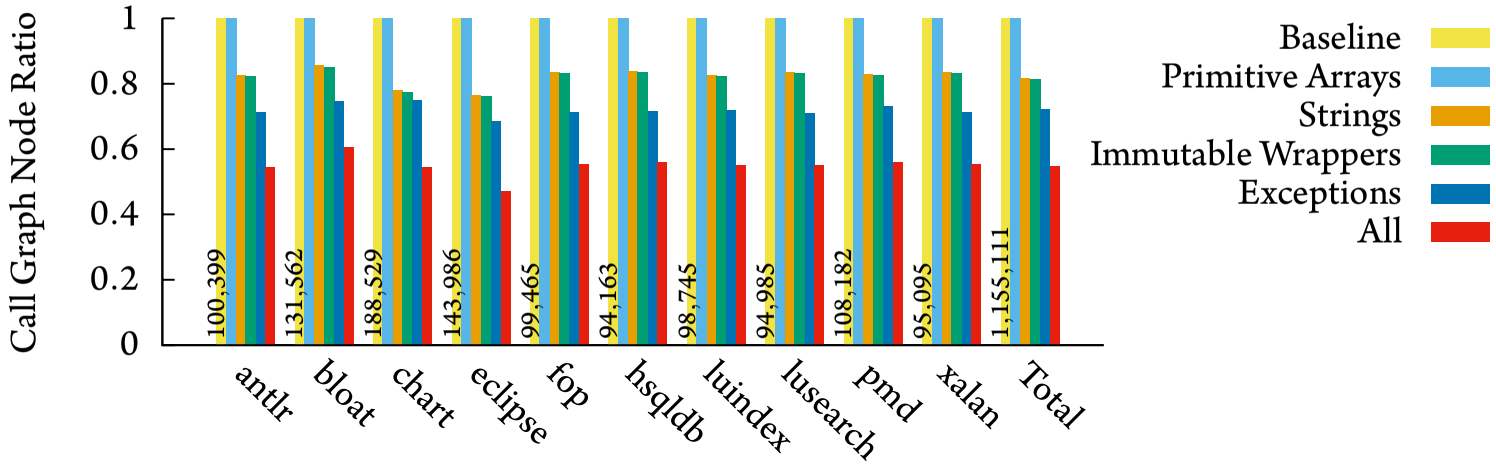


Figure 2.12: Relative number of call graph nodes when using singleton abstract objects for a given set of types.

benchmark; this benchmark runs out of memory if a singleton abstract object is not used for exceptions, even though we saw the analysis for this application complete in about 70 seconds when using singleton abstract objects for all the types described above.

As seen in Figure 2.10 using a singleton abstract object for primitive arrays does not significantly affect performance. Using a singleton abstract object for `java.lang.String` objects is almost always effective taking less than half the time in total (an exception, bloat, takes about 14% longer). Using singleton abstract objects for other immutable wrapper objects (e.g., `java.lang.Integer`) provides little additional benefit over using a singleton for just strings. For almost all the benchmarks the largest benefit comes from using a singleton abstract object per exception type. The total time for applications is 35.1% of the baseline time for a 2.8x speedup. The times range from 64.9% to 18.0% of the baseline time for a 1.5x to 5.5x speedup.

The last bar for each benchmark shows the relative time when using singleton objects for primitive arrays, immutable wrappers, and exceptions at the same time. For the eclipse benchmark the speedup was over 20x. The other benchmarks range from 41.2% to 25.1% of the baseline time for a 2.5x to 4x speedup. The total time for all benchmarks was 18.8% of the baseline time an over 5x speedup.

2.5 Related work

There has been an enormous amount of work on points-to analysis. We focus on practical points-to analyses for Java and multithreaded analyses. Hind [43] provides an excellent summary of the first 20 years worth of points-to analysis research. Most of this initial work focused on C programs. Many of the concepts and techniques are applicable to object-oriented languages, but Java in particular tends to have more address-taken variables

and abstract heap locations than C. Every reference in Java is heap-directed and every field is represented in a points-to analysis as an abstract location. This makes scaling points-to analyses for Java more difficult than scaling C points-to analyses. Smaragdakis and Balatsouras [99] present a more recent tutorial covering newer developments in points-to analysis including analyses for object-oriented languages.

Practical Java points-to analyses. Whaley [116] presents the first efficient analysis for Java that is context-sensitive. There are three commonly used Java points-to analyses. Like that of Whaley [116] they are all single threaded. These are the points-to analysis built into the WALA analysis framework [13], Doop [11] (a points-to analysis built on top of a single-threaded datalog engine), and PADDLE [60] (the points-to analysis incorporated into the Soot analysis framework [112]). These three analyses largely supersede Whaley [116] in part because they are part of larger static analysis frameworks. Accrue[10] is an interprocedural extension to the Polyglot extensible compiler framework [83], and includes an early prototype of our multithreaded points-to analysis that operates on Java source code rather than Java bytecode.[11]

The WALA points-to analysis analyzes the same intermediate language as our analysis, a static single assignment translation of Java bytecode. WALA uses their dataflow framework to compute the points-to analysis results. They keep track of previously processed points-to constraints which has a similar effect to our incremental propagation algorithm. They have a similar abstractions to the Merge and Record functions we use to define context sensitivity, although their versions (the `ContextSelector` and `InstanceKeyFactory` in-

[10] `http://people.seas.harvard.edu/~chong/accrue.html`

[11] The Accrue points-to analysis can also be extended to compute points-to results for extensions of the Java programming language expressible using Polyglot.

terfaces respectively) are not quite as cleanly defined. Similar to our analysis their default implementation of `InstanceKeyFactory` allows for the use of a single abstract object for certain types (`java.lang.String`, exceptions, and primitive arrays) as well as for types that have a large number of allocation sites.

Bravenboer and Smaragdakis [11] introduce the Doop framework. Doop is a points-to analysis that is declaratively specified and implemented in datalog, a declarative language. This means that their specification is close to their implementation, although in order for their analysis to scale, their code is still much more complex than their specification. Subsequent work [52, 100] defines the parameterization of context sensitivity we use and map out this parameter space, including the definition of several novel types of context sensitivity. Although not mentioned in any of their papers, Doop has flags allowing the use of a single abstract object for certain types (subclasses of `java.lang.AbstractStringBuilder`, exceptions, primitive arrays, and classes with only primitive fields). We allow for the use of single abstract object for some of these types, but did not investigate the impact of singleton `java.lang.AbstractStringBuilder` or classes with primitive fields.

Lhoták and Hendren [63] describes the design and implementation of SPARK. SPARK, like our analysis, is a work-list algorithm that uses difference propagation to eliminate redundant recomputation. For difference propagation they use a data structure called an *incremental set*, a set which distinguishes *new* and *old* elements. We, on the other hand, only use the *new* elements when propagating changes, which should reduce the amount of propagation, but is effectively the same. SPARK is a context-insensitive analysis; Lhoták [60] describes PADDLE, an analysis that builds on SPARK supporting context-sensitivity and using binary decision diagrams to achieve scalability. We use integer sets to represent points-to sets and a map from integers to integer sets to represent the points-to graph.

While not as memory efficient as binary decision diagrams, this representation is less abstract and is well within modern memory limits. Both PADDLE and SPARK are included in the Soot analysis framework [112].

Multithreaded points-to analyses There has been surprisingly little work to parallelize points-to analysis. To our knowledge, our analysis is the first multithreaded points-to analysis that supports a wide range of context-sensitivities and is the first multithreaded context-sensitive points-to analysis for an object-oriented language.

In 2010 Méndez-Lojo et al. [74] implemented the first multithreaded points-to analysis algorithm. Their context-insensitive analysis is for C and is based on the algorithm of Hardekopf and Lin [38]. They pose the solving of the points-to graph as a graph rewriting problem. Edges in the graph are points-to relations and nodes are pointer variables. The existence of certain edges results in the addition of new edges or modification of existing edges. They do not allow for modifications to the set of nodes in the graph, which precludes function pointers and some forms of context-sensitivity. Their algorithm would also not directly support a language with dynamic dispatch. They use a form of optimistic speculative execution in a system that automatically rolls back failed operations. In order to limit the number of expensive rollbacks they use a work-stealing algorithm similar to the one that we use. Their algorithm also supports a form of cycle detection similar to that described in Section 2.3.3, although they identify possible cycles before running their analysis to help speed up detections, while we detect cycles on the fly with no pre-computed data (which they call online cycle detection). Mendez-Lojo et al. [75] port this algorithm to the GPU while reducing the amount of synchronization needed for their graph rewriting rules, and Nagaraj and Govindarajan [79] implement a flow-sensitive (but still context-insensitive)

version of the analysis of Méndez-Lojo et al. [74] using a work-list to manage the graph rewrites. Some of the graph rewriting rules in this line of work are analogous to the propagation of changes for superset relations described in Section 2.3.2. The more complex rules for load and store are handled in our analysis by carefully tracking dependencies and reprocessing points-to statements as necessary. It is not clear which approach is more expensive, but ours is applicable to arbitrarily complex points-to statements, such as those for virtual method calls.

Nasre [81] use a two-dimensional variation of a Bloom filter to represent points-to sets and other data structures in a flow-sensitive context-insensitive analysis. Collisions in the hash functions used for the Bloom filters can cause unrelated pointers to share points-to information. They bootstrap their GPU based flow-sensitive analysis with a single-threaded flow-insensitive analysis used to generate constraints for load statements.

Prabhu et al. [87] use a GPU to accelerate a 0CFA analysis (analogous to a context-insensitive analysis) for the lambda calculus and report speedups of up to 72 times for an artificial benchmark suite. They suggest that their work could be adapted to compute points-to information, but have not yet pursued this goal.

Putta and Nasre [88] use a replication-based approach creating a separate points-to set for each points-to statement as it is processed. This may lead to a given points-to graph node mapping to two different points-to sets. These sets are merged after iterating through all statements. After merging, all points-to statements are reprocessed and this repeats until the analysis reaches a fixpoint. Their analysis supports a form of call-site sensitivity that uses a stack to represent method contexts, but does not support arbitrary analysis contexts.

Edvinsson et al. [25] implement a context-insensitive analysis for object-oriented languages using a coarse grained parallelism. They parallelize the different method targets of

dynamic dispatch sites and independent control-flow branches. They propose, but do not implement, a similar technique for parallelizing different contexts in a context-sensitive analysis.

3

Flow Sensitive Points-to Analysis

3.1 Introduction and background

Flow-insensitive points-to analyses, such as the one described in Chapter 2, compute points-to information that may hold at any point in the program's execution. By contrast, flow-sensitive points-to analyses compute points-to information for each program point, which is potentially more precise but less scalable. The precision of flow-sensitive points-to analyses is particularly appealing because it can enable *strong update* [15], whereby an assignment can *replace* the statically computed facts associated with a memory location. By contrast, in a weak update, an assignment *adds* to the statically computed facts of a memory location.

In this chapter, we present a points-to analysis for object-oriented programs that enables strong update in both the points-to analysis and subsequent client analyses.Strong update is particularly important in object-oriented program analyses because the correctness, security, and performance of object-oriented programs often rely on object invariants established during object construction. Without the ability to perform strong updates in object-oriented program analysis, it is difficult to reason about the establishment of object invariants. For example, in Java, many objects have as part of their invariant that some field is non-null, but since the initial value of all fields is null, a flow-insensitive analysis, or an analysis with only weak update, is unable to reason that the field is always non-null after

object construction.

Overview of Analysis Strong update to an abstract memory location can only be performed when the abstract location corresponds to exactly one concrete memory location. To ensure that there are abstract locations that usefully correspond to exactly one concrete location, we use the *recency abstraction* of Balakrishnan and Reps [6], in which objects created at a given allocation site are represented by two abstract objects: one represents the object most recently allocated at the allocation site and one summarizes all other objects allocated at the allocation site.[1] By definition, a field of the most-recently-allocated abstract object represents exactly one concrete location, and thus strong update can be performed on it. Moreover, since object construction typically takes place on the most recently allocated object, the recency abstraction enables strong update exactly where it is most useful for object-oriented analyses: during the establishment of object invariants. (Balakrishnan and Reps use the recency abstraction to reason precisely about the establishment of virtual-function tables in stripped executables, which is similar to reasoning about the establishment of a specific object invariant.)

Inspired by Lhoták and Chung [62], our points-to analysis is flow-sensitive only for abstract locations on which strong update may be possible, i.e., for fields of most-recently-allocated abstract objects, and for static variables. We treat other abstract locations (e.g., fields of non-most-recently-allocated abstract objects) flow insensitively, which improves the performance of the analysis significantly. For local variables we use a partial static single assignment (SSA) program representation [20], which gives many of the benefits of flow-sensitive analysis for local variables, with the efficiency of flow-insensitive analysis.

[1] As described in Section 2.1, an *abstract object* represents zero or more *concrete objects*, and in an object-oriented analysis abstract locations include fields of abstract objects as well as local and static variables.

```
1  A a = new A();
2  a.f = new B();
3  a.f.foo(); // Can a.f be null?
4
5  Collection<C> bag = ...
6  while(...) {
7      C c = new C();
8      c.g = new D();
9      c.g.bar(); // Can c.g be null?
10     bag.add(c);
11 }
12
13 for (C o : bag) {
14     o.g.bar(); // Can o.g be null?
15 }
```

Figure 3.1: Strong update example

Therefore we do not track local variables flow-sensitively even though abstract locations for local variables can be soundly strongly updated.

As discussed in Section 2.1.3, our flow-insensitive points-to analysis is parameterized in the same way as the framework of Kastrinis and Smaragdakis [52]. That is, the names of abstract objects and the analysis contexts for code are determined by three functions, Record, Merge, and MergeStatic and different instantiations of these functions enable many different points-to analysis. Because we use a similar parameterization for our flow-sensitive analysis, this can be seen as a transformation that enables useful, efficient strong update by adding the recency abstraction and limited flow sensitivity to any existing flow-insensitive points-to analysis (that is expressible using the framework of Kastrinis and Smaragdakis).

EXAMPLE To see the benefits of flow sensitivity and the recency abstraction, consider the Java-like code in Figure 3.1 and a client analysis that attempts to identify field accesses and method calls where the target is guaranteed to be non-null (and thus the field access or method call cannot throw a `NullPointerException`).

Note that the target of the method call to `foo()` on line 3 is never null. However, in Java, all fields with reference type are initialized to null so field `a.f` will be null immediately after the allocation on line 1. Thus, a flow-insensitive points-to analysis (which conservatively approximates what `a.f` may point at *any* program point) must include null in the points-to set for `a.f`, and is not able to conclude that the target of the method call will be non-null. Flow sensitivity can help precision here, by distinguishing the points-to set of `a.f` immediately after line 1 (where it might be null) from the points-to set of `a.f` immediately after line 2 (where it is definitely not null).

Now consider a points-to analysis that is flow sensitive, but uses just a single abstract object to represent all objects created at an allocation site (this is called the *allocation-site abstraction* in contrast with the recency abstraction). Immediately following line 7, this analysis would know that local variable `c` points to the (abstract) object allocated at line 7, and that the field `g` of that object may point to null. Since this allocation occurs inside a loop, `c` points to one of potentially many concrete objects created at that allocation site. Thus, we are not able to use strong update for the assignment `c.g = new D()` on line 8 (since the abstract location referred to by `c.g` represents potentially many concrete locations). We thus can only *add* to the points-to set of `c.g`, instead of replacing it, and we must conservatively assume that the target of the method call at line 9 may be null, even though it is never null at run time.

Now consider a flow-sensitive points-to analysis that uses the recency abstraction. Since

the abstract object pointed to by c at line 8 represents the most recent object allocated at line 7, we know that abstract location c.g represents exactly one concrete location, and so can perform strong update, thus concluding that immediately after line 8 the points-to set of c.g cannot contain null. Thus, the target of the method call on line 9 cannot be null.

Moreover, since we have established that field g of the most recent object created at line 7 cannot be null after execution of the while loop, the points-to analysis can conclude that this holds true of *all* objects created at line 7, whether they are the most recent or not. Thus, at line 14, where o may refer to either the most-recently-allocated abstract object or the non-most-recently-allocated abstract object, the points-to analysis is able to conclude that o.g, the target of the method call, is non-null.

Contributions This chapter makes the following contributions.

1. A novel efficient points-to analysis for object-oriented programs that enables strong update in both the points-to analysis and subsequent client analyses. This analysis builds on the recency abstraction of Balakrishnan and Reps [6] to enable strong update, and, for efficiency, uses flow sensitivity for only a subset of the abstract locations [62]. The analysis is parameterized *à la* Kastrinis and Smaragdakis [52], thus providing efficient strong update for a wide variety of points-to analyses.
2. An implementation of this analysis for Java bytecode that scales up to 130k lines of code in 92 seconds. Although our analysis limits flow-sensitivity just to the locations where it is most useful, the implementation required careful engineering to ensure efficiency. Our implementation is multi-threaded, and uses a novel representation of flow-sensitive points-to sets that is amenable to efficient computation.
3. A demonstration of the usefulness of our points-to analysis, and of strong update for

T : class types
F : resolved instance fields
S : method signatures
M : resolved methods
V : reference variables
C : contexts
AO : abstract object

IPP : inter-program points
PP : program points

Figure 3.2: Analysis domains for our flow-sensitive points-to analysis

object-oriented languages, by comparing the performance of several client analyses using our points-to analysis versus our flow-insensitive points-to analysis. For example, we are able remove 93% of the null-pointer exceptions in a common benchmark suite compared with 86% removed by a flow-insensitive analysis.

3.2 Analysis

We present our points-to analysis as points-to statements and subset constraints analogous to those seen in Section 2.2. Our points-to analysis engine finds the smallest points-to sets that satisfy these constraints. Again our points-to analysis assumes a class-based object oriented language with dynamic dispatch and that all allocated entities are objects, and our implementation relaxes the latter assumption to include the allocation of arrays.

This analysis is an extension of the analysis from Section 2.2, and we use the same typefaces and notation as that section. This analysis adds selective flow sensitivity and incorporates the recency abstraction [6] into our heap object abstraction. The primary goal of this section is to emphasize changes to a flow-insensitive analysis needed to support flow-sensitivity, strong update, and the recency abstraction.

3.2.1 Domains

The domains of our analysis are shown in Figure 3.2. These are the types of values used by our analysis. The first seven domains are the same as those from in our flow-insensitive analysis and are described in Section 2.2.1.

The first difference between this analysis and the flow-insensitive analysis of Section 2.2, is that all abstract objects (elements of domain AO) contain a parameter that indicates whether the abstract object represents a most-recently-allocated object or non-most-recently-allocated object. This is how we implement the recency abstraction of Balakrishnan and Reps [6] in our analysis.

The last two domains PP and IPP are needed to support flow sensitivity. Domain PP is the set of all program points in the program. In our flow-insensitive analysis we only needed the program points for allocation sites and call sites, here we need program points for any pointer-relevant statement. We use inter-program points IPP to describe analysis facts immediately before and after program points. Specifically, for program point $pp \in PP$, there is one inter-program point immediately before pp, and one inter-program point immediately after pp. Because program points occur in methods, and methods may be analyzed in multiple contexts, a *program point replica* $(pp, c) \in PP \times C$ represents program point pp of method m being analyzed in context c. Similarly, *inter-program point replica* $(ipp, c) \in IPP \times C$ represents inter-program point ipp of method m being analyzed in context c.

3.2.2 Functions

Figure 3.3 lists the functions used to specify our analysis. FieldsOf and ResolveMeth are the same as for our flow-insensitive analysis and are described in Section 2.2.2. TypeOf

Lookup
FieldsOf : $T \rightarrow Set\langle F\rangle$
ResolveMeth : $S \times AO \rightarrow M$
TypeOf : $AO \rightarrow T$
PtsFI : $(V \times C) + (AO \times F) \rightarrow Set\langle AO\rangle$
PtsFS : $AO \times F \times IPP \times C \rightarrow Set\langle AO\rangle$
PtsInit : $AO \times F \rightarrow Set\langle AO \times IPP \times C\rangle$
IsRecent : $AO \rightarrow Boolean$
NonMostRecent : $AO \rightarrow AO$

Context Creation
Record : $PP \times C \rightarrow AO$
Merge : $AO \times PP \times C \rightarrow C$
MergeStatic : $PP \times C \rightarrow C$

Program Point Graph
MethEntry : $M \rightarrow PP$
MethExit : $M \rightarrow PP$
Succs : $PP \rightarrow Set\langle PP\rangle$
Succs : $PP \rightarrow Set\langle PP\rangle$
Before : $PP \rightarrow IPP$
After : $PP \rightarrow IPP$
ValidPathExists : $IPP \times C \times IPP \times C \rightarrow Boolean$
KilledOnAllPaths : $IPP \times C \times IPP \times C \times AO \times F \rightarrow Boolean$
AllocOnAllPaths : $IPP \times C \times IPP \times C \times AO \rightarrow Boolean$
Killed : $PP \times C \rightarrow Set\langle AO \times F\rangle$

Figure 3.3: Flow-sensitive analysis functions

gives the type of the abstract object (in our analysis all abstract object are always of a particular type).

The next two functions, PtsFI and PtsFS, are used to lookup the points-to sets for flow-insensitive and flow-sensitive points-to graph nodes, respectively. A points-to graph node is either a reference variable replica (in $V \times C$) or an object field (in $AO \times F$). As discussed in Section 3.1 local variables and fields of non-most-recent abstract objects are tracked flow-insensitively; flow-insensitive points-to sets are looked up using PtsFI. Fields of most-recent abstract objects are tracked flow-sensitively and their points-to sets can be looked up using PtsFS.[2] Note that flow-sensitive points-to sets are computed for an object field at a particular inter-program point replica replica in $IPP \times C$.

PtsInit is used to lookup the inter-program points at which elements of the flow-sensitive points-to set are first added. PtsInit(ao, f) returns a set of abstract objects together with the inter-program point replicas at which they were added to the flow-sensitive points-to set for (ao, f). For example, if there is an assignment to field f of abstract object ao at inter-program point replica (ipp, c) that may cause that field to point to abstract object ao' then (ao', ipp, c) is in PtsInit(ao, f). Note that the same abstract object may be added at different inter-program point replicas, in which case it will appear multiple times in PtsInit(ao, f).

The function IsRecent(ao) returns true if ao is a most-recent abstract object and false if it is a non-most-recent abstract object. NonMostRecent takes a most-recent abstract object and provides the non-most-recent version of the same abstract object. If ao is already a non-most-recent abstract object then NonMostRecent(ao) returns ao unchanged.

[2]We also track points-to sets for static fields flow sensitively in our implementation. The modification to the analysis is straightforward, but we elide it for simplicity.

As in our flow-insensitive analysis, this analysis is parameterized by three functions used for context creation: RECORD, MERGE, and MERGESTATIC. These functions are analogous to their counterparts introduced in Section 2.1.3, but we use the program points for call sites and allocation sites in place of the specialized types, *CS* and *AS*, seen there. This parameterization is essentially that of Smaragdakis et al. [100] and Kastrinis and Smaragdakis [52], and this analysis still supports any context sensitivity expressible using these functions (e.g., all those described in Section 2.1.3 and any hybrid analysis [52] seen elsewhere in this dissertation).

The last eight functions in Figure 3.3 are used to query the program-point graph and inter-program-point graph. For each method the program-point graph has a distinguished method entry and exit program point retrieved using METHENTRY and METHEXIT respectively. SUCCS(*pp*) gives the intraprocedural successors to program point *pp*, defining the *intra*procedural program-point graph. These three functions combined with the call graph, computed by the points-to analysis, define the *inter*procedural program point graph. BEFORE(*pp*) and AFTER(*pp*) give, respectively, the *inter*-program point that is immediately before and after program point *pp*.

Function VALIDPATHEXISTS(*ipp′, c′, ipp, c*) determines whether there is a valid path in the inter-program-point graph from (ipp', c') to (ipp, c). We give several definitions of *valid path* in Section 3.2.5. KILLEDONALLPATHS(*ipp′, c′, ipp, c, ao, f*) returns true if and only if object field (ao, f) is killed on all valid paths in the inter-program-point graph from (ipp', c') to (ipp, c). An object field is *killed* when it is strongly updated. Similarly, ALLOCONALLPATHS(*ipp′, c′, ipp, c, ao*) returns true if abstract object *ao* is allocated on all valid paths from (ipp', c') to (ipp, c).

KILLED(*pp, c*) returns the set of object fields that are *killed* at a particular program point

$$\textsc{Killed}(pp, c) = \begin{cases} \{\} & |\textsc{PtsFI}(o, c)| > 1 \\ \{\} & \textsc{PtsFI}(o, c) = \{ao\} \\ & \wedge\ \neg\textsc{IsRecent}(ao) \\ \{(ao, fld)\} & \textsc{PtsFI}(o, c) = \{ao\} \\ & \wedge\ \textsc{IsRecent}(ao) \\ \{(ao, f) \mid f = fld\} & \textsc{PtsFI}(o, c) = \{\} \end{cases}$$

where `o.fld = x` is the points-to statement at *pp*

Figure 3.4: Definition of the KILLED function.

replica. An object field (ao, f) is killed at (pp, c) if the instruction at (pp, c) strongly updates (ao, f). This function is defined in Figure 3.4, described in the next section.

3.2.3 Defining strong update

Figure 3.4 gives the definition of the KILLED function. KILLED describes which object fields are strongly updated at a particular program point and is used to define the KILLEDONALLPATHS function.

The KILLED function returns an empty set for all program points except those corresponding to field store instructions. If KILLED(*pp, c*) returns a non-empty set then the elements of that set are strongly updated at (pp, c) and elements of the points-to set for the object field are *not* preserved across (pp, c). If a program point replica (pp, c) does correspond to a field store `o.fld = from`, there are four cases based on the size of the points-to set for (o, c) (and the recency of its elements). Let *pts* be the points-to set for (o, c). First, if the size of *pts* is larger than one we do not know which concrete memory location is updated by the field store. In this case strong update cannot be performed and no object field is killed. Second, if the size of *pts* is one but the element of *pts* is a non-most-recent abstract object then no strong update can be performed because non-most-recent abstract objects

can correspond to more than one concrete object. Third, if the size of *pts* is one and the element, *ao*, of *pts* is a most-recent abstract object, then strong update can be performed and the field (ao, fld) is killed.

Lastly we consider the case when the points-to set for (o, c) is empty. Note that while this can occur during the analysis, the final solution will not have empty points-to sets, at least not on any valid path. While solving the constraints elements are added to points-to sets but never removed. Conservatively, we assume that any abstract object containing the appropriate field could be added to the points-to set for (o, c). This means that any object field with that field could be killed at (pp, c).

At a particular program point replica the kill set can only shrink as more elements are added to the points-to set for (o, c). This is necessary to ensure the monotonicity of all points-to sets, otherwise a points-to set element that was previously preserved at (pp, c) could later be killed at (pp, c). If monotonicity were violated this could cause the analysis to enter an infinite loop when attempting to find the least solution to the subset constraints.

3.2.4 Flow-sensitive subset constraints

We specify our analysis using the same points-to statements from Section 2.2.3. Since we track reference variables and fields of non-most-recent abstract objects flow-insensitively many of the points-to statements represent the same, flow-insensitive, subset relations seen in that section. These are shown in Table 3.1 for completeness, but are the same as those presented in Section 2.2.3 and are not described here.

The flow-sensitive points-to sets are only used in the relations produced for a field load and field store when the receiver is a most-recent abstract object and for new allocations. Table 3.2 shows the subset relations that the points-to graph must satisfy for these state-

Table 3.1: Subset relations for points-to statements (in an analysis context c) that only involve flow-insensitive points-to information.

Instruction in context c	Subset relations
`to = from`	$\textsc{PtsFI}(to, c) \supseteq \textsc{PtsFI}(from, c)$
`to = null`	$\textsc{PtsFI}(to, c) \supseteq \{\texttt{AONULL}\}$
`to = (C) from`	$\textsc{PtsFI}(to, c) \supseteq_{\texttt{C}} \textsc{PtsFI}(from, c)$
`caught = (Ex) (notTypes) thrown`	$\forall ao \in \textsc{Pts}(thrown, c)$. if ao is a subtype of `Ex` $\wedge \forall T \in notTypes.ao$ is *not* a subtype of `T` then $ao \in \textsc{Pts}(caught, c)$
`to = φ(from`$_0$`, ..., from`$_n$`)`	for $i \in \{0, \ldots, n\}$.$\textsc{Pts}(to, c) \supseteq \textsc{Pts}(from_i, c)$
`to = o.m(a`$_0$`, ..., a`$_n$`)`	$\forall ao \in \textsc{PtsFI}(o, c)$. $\textsc{PtsFI}(this_t, newC) \supseteq \{ao\}$ for $i \in \{0, \ldots, n\}$. $\textsc{PtsFI}(formal_{t,i}, newC) \supseteq \textsc{PtsFI}(a_i, c)$ $\textsc{PtsFI}(to, c) \supseteq \textsc{PtsFI}(return_t, newC)$ $\textsc{Pts}(ex, c) \supseteq \textsc{PtsFI}(exception_t, calleeC)$ where $\textsc{Merge}(ao, pp, c) = newC$ $\textsc{ResolveMeth}(ao, \texttt{m}) = t$
`to = C.m(a`$_0$`, ..., a`$_n$`)`	for $i \in \{0, \ldots, n\}$. $\textsc{PtsFI}(formal_{m,i}, newC) \supseteq \textsc{PtsFI}(a_i, c)$ $\textsc{PtsFI}(to, c) \supseteq \textsc{PtsFI}(return_m, newC)$ $\textsc{Pts}(ex, c) \supseteq \textsc{PtsFI}(exception_t, calleeC)$ where $\textsc{MergeStatic}(pp, c) = newC$

Table 3.2: Subset relations for points-to statements that may use or modify the flow-sensitive points-to graph.

Instruction in context c at program point pp	Subset relations
`to = new C`	let $ao = \textsc{Record}(pp, c)$ $ao \in \textsc{PtsFI}(to, c)$ $\forall f \in \textsc{FieldsOf}(\texttt{C}).(\texttt{AONULL}, \textit{After}(pp), c) \in \textsc{PtsInit}(ao, f)$ $\forall f \in \textsc{FieldsOf}(\texttt{C})$. $\quad \forall\, ao' \in \textsc{PtsFS}(ao, f, \textsc{Before}(pp), c)$. $\quad\quad ao' \neq ao \Rightarrow ao' \in \textsc{PtsFI}(\textit{NonMostRecent}(ao), f)$ $\forall f \in \textsc{FieldsOf}(\texttt{C}).\forall\, ao' \in \textsc{PtsFS}(ao, f, \textsc{Before}(pp), c)$. $\quad ao' = ao$ $\quad\Rightarrow \textsc{NonMostRecent}(ao') \in \textsc{PtsFI}(\textit{NonMostRecent}(ao), f)$ $\forall f \in \textsc{FieldsOf}(\texttt{C}).\forall ao' \in AO$. $\quad ao \in \textsc{PtsFS}(ao', f, \textsc{Before}(pp), c) \wedge ao' \neq ao$ $\quad\quad \Rightarrow (\textit{NonMostRecent}(ao), \textit{After}(pp), c) \in \textsc{PtsInit}(ao', f)$
`to = o.f`	$\forall ao \in \textsc{PtsFI}(o, c)$. $\quad \textsc{IsRecent}(ao) \Rightarrow \textsc{PtsFI}(to, c) \supseteq \textsc{PtsFS}(ao, f, \textsc{Before}(pp), c)$ $\quad \neg\, \textsc{IsRecent}(ao) \Rightarrow \textsc{PtsFI}(to, c) \supseteq \textsc{PtsFI}(ao, f)$
`o.f = from`	$\forall ao \in \textsc{PtsFI}(o, c)$. $\quad \textsc{IsRecent}(ao) \Rightarrow$ $\quad\quad \forall ao' \in \textsc{PtsFI}(from, c).(ao', \textit{After}(pp), c) \in \textsc{PtsInit}(ao, f)$ $\quad \neg\, \textsc{IsRecent}(ao) \Rightarrow \textsc{PtsFI}(ao, f) \supseteq \textsc{PtsFI}(from, c)$

ments. Each statement is labeled with a program point, and since our analysis is context-sensitive each statement is analyzed in a particular context. PtsFS is used to get the flow-sensitive points-to set at a particular program point. PtsInit records the inter-program point replicas at which elements are added to a flow-sensitive points-to set. We discuss the *propagation* of flow-sensitive points-to information from this initial inter-program point in Section 3.2.5.

An allocation statement `to = new C` uses the Record function to compute a new abstract object *ao* and ensures that *ao* is in the points-to set for (to, c). The abstract objects produced by the Record function are always most-recent abstract objects as expected for a newly allocated object. The fields of the new abstract object point to `null` at the inter-program point immediately after the allocation, so we add `AONULL` at this inter-program point to PtsInit for these fields.

In addition, because *ao* (a most-recent abstract object) has just been allocated, what was previously the most-recent version of *ao* is now the non-most-recent. Any elements (not equal to *ao*) that were in the points-to set for (ao, f) before the allocation are added to the points-to set for $(\mathit{NonMostRecent}(ao), f)$ which is tracked flow-insensitively. If an element of the points-to set (ao, f) is equal to *ao* then, in this special case, NonMostRecent(*ao*) is added to the points-to set for $(\mathit{NonMostRecent}(ao), f)$. Lastly, any flow-sensitive points-to set that contained *ao* before the allocation contains the non-most-recent version of *ao* after the allocation.[3]

The constraints for a field load, `to = o.f`, ensure that for every abstract object $ao \in$ PtsFI(*o, c*) everything that the field f of *ao* pointed to before the assignment is pointed to

[3]Via the constraint in Figure 3.5 flow-insensitive points-to sets already contain the non-most-recent version if they contain the most-recent version and do not need this update at allocations.

$$ao \in \textsc{PtsFI}(x, c) \wedge \textsc{IsRecent}(ao) \Rightarrow \textsc{NonMostRecent}(ao) \in \textsc{PtsFI}(x, c)$$

Figure 3.5: Interaction between the recency abstraction and flow-insensitive points-to sets.

by (to, c). If *ao* is a most-recent abstract object (i.e., $\textsc{IsRecent}(ao)$ is true) then we track (ao, f) flow-sensitively, the points-to set before the assignment is $\textsc{PtsFS}(ao, f, \textsc{Before}(pp), c)$, and these elements are added to the points-to set for (to, c). If *ao* is a non-most-recent abstract object then the points-to set for (ao, f) is $\textsc{PtsFI}(ao, f)$ which must be a subset of the points-to set for (to, c).

The constraints for field store, `o.f = from`, are also split into two cases. If *ao* is an element of the points-to set for (o, c) and *ao* is a most-recent abstract object then all elements of the points-to set for $(from, c)$ are added to the flow-sensitive points-to set for (ao, f) at the inter-program point immediately following the store instruction (i.e., these elements are added to PtsInit together with this inter-program point replica). If *ao* is a non-most-recent abstract object we track (ao, f) flow-insensitively and the constraint ensures that the flow-insensitive points-to set for (ao, f) contains all the elements of the points-to set for $(from, c)$.

The constraint in Figure 3.5 holds for all points-to statements and describes how the recency abstraction interacts with flow-insensitive points-to sets. If a flow-insensitive points-to set points to a most-recently-allocated abstract object *ao*, then it must also point to the corresponding non-most-recently allocated abstract object $\textsc{NonMostRecent}(ao)$. Intuitively, this is because flow-insensitive points-to sets describe facts that may hold at any program point and if a field points to the most-recently allocated object just before the allocation site that allocated the object, then immediately after the allocation it points to the corresponding non-most-recently allocated object.

$$\textsc{PtsFS}(ao, f, ipp, c) = \left\{ ao' \;\middle|\; \begin{array}{l} (ao', ipp', c') \in \textsc{PtsInit}(ao, f) \\ \wedge\, \textsc{ValidPathExists}(ipp', c', ipp, c) \\ \wedge\neg\, \textsc{KilledOnAllPaths}(ipp', c', ipp, c, ao, f) \\ \wedge\neg\, \textsc{AllocOnAllPaths}(ipp', c', ipp, c, ao) \\ \wedge\, (\neg\, \textsc{IsRecent}(ao') \\ \quad \vee\, \neg\, \textsc{AllocOnAllPaths}(ipp', c', ipp, c, ao')) \end{array} \right\}$$

Figure 3.6: Flow-sensitive points-to set definition.

3.2.5 Flow-sensitive points-to set propagation

Figure 3.6 describes the elements in the flow-sensitive points-to set for (ao, f) at a particular inter-program point replica, (ipp, c). In order to compute this set we use PtsInit, which gives the inter-program point replicas at which an element is added to a flow-sensitive points-to set, together with reachability queries on the inter-program-point graph.

If (ao', ipp', c') is an element of $\textsc{PtsInit}(ao, f)$ this means that ao' was first added to the points-to set for (ao, f) at inter-program point replica (ipp', c'). In order for ao' to be in the points-to set at a different inter-program point replica, (ipp, c), there must be a valid path in the inter-program-point graph from (ipp', c') to (ipp, c). In addition, at least one of these valid paths must not *kill* the field (ao, f). If (ao, f) is strongly updated then the points-to set is replaced (i.e., elements of the point-to set for (ao, f) before the strong update are no longer in the points-to set after the strong update). If (ao, f) is strongly updated on every valid path from (ipp', c') to (ipp, c) then $\textsc{KilledOnAllPaths}(ipp', c', ipp, c, ao, f)$ is true and ao' is not in the points-to set at (ipp, c).

We also have to consider the allocations that occur on valid paths from (ipp', c') to (ipp, c). An allocation of *ao* will result in a new most-recent *ao* and fields of that new object will be initialized to `null` as seen in Table 3.2. If an allocation of *ao* occurs on *all* valid

paths from (ipp', c') to (ipp, c) (i.e., AllocOnAllPaths(*ipp', c', ipp, c, ao*) is true) then elements that were added at (ipp', c') will not remain at (ipp, c).

Assume ao' is a most-recent abstract object in the points-to set for (ao, f) before ao' is allocated. Then NonMostRecent(ao') replaces it in the points-to set for (ao, f) after the allocation (as seen in Table 3.2). If ao' is added to the points-to set for (ao, f) at (ipp', c') and ao' is allocated on all valid paths from (ipp', c') to (ipp, c) (i.e., AllocOnAllPaths(*ipp', c', ipp, c, ao'*) is true) then ao' will not be in the points-to set for (ao, f) at (ipp, c).

In summary if ao' is added to the points-to set for (ao, f) at inter-program point replica (ipp', c') then it is in the points-to set at (ipp, c) if all the following conditions hold:

1. There is a valid path from (ipp', c') to (ipp, c) in the inter-program-point graph;
2. There is at least one valid path from (ipp', c') to (ipp, c) where (ao, f) is not strongly updated;
3. There is at least one valid path from (ipp', c') to (ipp, c) where ao is not allocated;
4. If ao' is a most-recent abstract object then ao' is also not allocated on all paths from (ipp', c') to (ipp, c).

The definitions of *valid path* in ValidPathExists, KilledOnAllPaths, and AllocOnAllPaths has intentionally been left opaque. The definition of *valid* produces sound results as long as we do not rule out paths that may occur at runtime during a particular execution. Different definitions of what constitutes a valid path lead to differing levels of precision in our analysis.

At one extreme we could declare that there is a valid path between every pair of inter-program point replicas and ignore strong updates on that path. To do this we can define ValidPathExists to return true for any two inter-program point replicas and define the functions KilledOnAllPaths and AllocOnAllPaths to always return false. In this

case, an element added to a points-to set at any program point will be contained in that points-to set at any other program point. In other words the analysis is fully flow-insensitive.

Another option is to define the path predicates using graph reachability in the inter-program-point graph. In this case, ValidPathExists checks for the existence of a path in the graph theoretic sense, KilledOnAllPaths$(ipp', c', ipp, c, ao, f)$ returns false if (ipp, c) is reachable from (ipp', c') on some path in the inter-program-point graph that does not strongly update (ao, f), and AllocOnAllPaths is defined in a similar way.

The graph reachability definitions above produce flow-sensitive points-to results, but we can be more precise if we rule out paths that are infeasible at runtime. For example, we can only consider paths with matching call and return sites, a form of CFL reachability [90]. Paths in the inter-program-point graph that enter via a particular call site and exit via a different return site do not correspond to any actual execution and can safely be ignored. This improves precision eliminating some spurious elements in our flow-sensitive points-to sets.

There are other paths in the inter-program-point graph that do not correspond to a possible program execution. For example, there may be a path which passes through the true branch of the conditional `x > 42` and the false branch of the conditional `x > 32` in a program where this is impossible for any real execution. By gathering such conditions and determining their satisfiability (e.g., by using an SMT solver such as Z3 [22]) we can further eliminate infeasible paths, improving the precision of our analysis.

In our implementation, we do not take into account the compatibility of a path's conditionals, but do rule out paths based on incompatible call and return sites.

3.3 Implementation

This analysis uses a modified version of the framework used for our flow-insensitive analysis, the implementation of which is discussed in Section 2.3. As a result they share many of the same features. The implementation of our flow-sensitive points-to analysis consists of approximately 14.5k lines of Java code of requiring about 7k more lines of code than our flow-insensitive analysis (much of the remaining 7.5k is shared between the two analyses). We again implement our analysis for Java bytecode. As in our flow-insensitive analysis we use WALA [13] to parse and translate Java bytecode to a register-based intermediate language in partial Static Single Assignment (SSA) form [20] and our points-to analysis runs on this intermediate language.

We handle almost all features of Java. Like our flow-insensitive analysis we do not handle reflection. Unlike in our flow-insensitive analysis, reads and writes are ordered, and we do not reason soundly about the order of concurrent accesses to shared data. We handle native methods in the same way as our flow-sensitive analysis, writing analysis signatures in Java for key methods and generating an analysis signature for other methods. We also use analysis signatures to improve precision and performance for some key Java standard library classes. These are sources of unsoundness in our analysis.

Another feature inherited from our flow-insensitive analysis, we allow client analyses to use a single abstract object for some common types. Choosing this option will improve performance, but lose precision for those types.

Our points-to analysis engine solves the constraints described in Section 3.2 and is implemented as a multi-threaded work queue algorithm, using Java's work-stealing fork/join framework [58]. To efficiently propagate changes, we carefully track dependencies and use a difference propagation algorithm [28, 63, 84]. The basic points-to analysis engine and

optimizations carry over from our flow-insensitive analysis; more detail on these can be found in Section 2.3. The rest of this section is dedicated to describing further optimizations specifically in support of flow-sensitivity and strong update.

3.3.1 Points-to set representation

We represent the points-to facts as a graph, where edges go from points-to graph nodes to abstract objects (to reiterate, a points-to graph node is either a reference variable replica or an object field). We represent the flow-insensitive points-to graph simply as a map from points-to graph nodes to sets of abstract objects.

For flow-sensitive points-to graph nodes (i.e., fields of most-recently-allocated heap objects), we associate with each flow-sensitive points-to graph node n and abstract object o a set of inter-program point replicas $s_{n,o}$, such that if inter-program point replica (ipp, c) is in $s_{n,o}$, then n points to o at (ipp, c). We do not explicitly represent each member of this set. Instead we use a novel compact representation of the set of inter-program point replicas, called *program-point closures*.

A program-point closure is specific to a particular points-to graph node n and abstract object o, and contains an explicit set of *sources*: inter-program point replicas at which the points-to relation between n and o is established (e.g., immediately following field stores that add o to the points-to set of n). These sources are precisely the elements of $\textsc{PtsInit}(n)$ (described in Section 3.2.2) where the first element of the returned tuple is o. A program-point closure implicitly represents the set of all inter-program point replicas for which there is a valid path from a source that does not go through a program point replica that allocates either o or n, nor goes through a program point replica that performs a strong update to object field n. (This corresponds to finding an appropriate path as defined in Figure 3.6.)

Thus, determining whether an inter-program point replica is in a set represented by a

program-point closure is reduced to a graph reachability problem in the inter-procedural control-flow graph. We are able to efficiently and precisely answer these graph reachability questions by computing a *method summary* for each (method, context) pair

Specifically, if method m is analyzed in context c, then we compute (and memoize) which abstract objects are allocated and which object fields are killed on all paths from the entry program point replica to the exit program point replica. That is, if abstract object o is allocated on all paths from the method entry to the method exit, then o will be in the summary; if there exists some path from the method entry to the method exit where o is not allocated, then o will not be in the summary. Since the method may call other methods, computation of the method summary may require using the method summaries of the callees. In addition, we also compute (and memoize) what method-context pairs are reachable from m in context c, and which abstract objects are allocated and object fields killed on all paths to that method. That is, if there is a path from the entry of method m in context c to the entry of method m' in context c', then the method summary for m in context c will describe which objects are allocated and which object fields are killed on all paths from the entry of m in context c to the entry of m' in context c'.

Method summaries greatly improve the efficiency of determining whether some source inter-program point replica can reach some target inter-program point replica without going through a program point replica that allocates a relevant abstract object or kills a relevant object field. In particular, whenever a method call is encountered during this search, method summaries efficiently summarize the potential results of exploring the callee, including allowing the search to jump directly to the entry of the method containing the target inter-program point replica.

Computation of the method summaries must be iterated until a fixed point is reached,

and must be updated as the call graph is discovered. However, many inter-program point reachability queries use the method summaries, and the computational effort required for method summaries may be paid off by increased efficiency of inter-program point reachability queries. For the three applications in the DaCapo benchmark suite [9] not analyzed in Section 3.4 this approach did not scale. We hypothesize that this is due to large connected components in the call graph for those applications, which could inflate the cost of the iteration and re-computation of these method summaries.

In addition, the use of method summaries also improves the precision of the inter-program point reachability queries, since it ensures that only paths the inter-procedural control flow graph with matching call and return sites are explored. That is, it does not consider paths that enter a callee method from one caller, and return to a different caller. We thus implement a form of CFL reachability [90]. Other possible alternative definitions of *reachability* are discussed in Section 3.2.5.

3.3.2 Context sensitivity

The context sensitivity of our analysis is parameterized in the context by the Merge, MergeStatic, and Record functions. State-of-the-art abstractions for object-oriented programs include type-sensitivity [100] and hybrid analysis [52] (which combine object-sensitive and non-object-sensitive approaches), and we have implemented several of these abstractions in our analysis.

However, in order for our analysis to employ strong update at a field store $o.f = from$, the replica of reference variable o must point to a single abstract object which is a most-recently-allocated object. In particular, since this is most useful in establishing object invariants, we need the receiver in constructor methods to be a single most-recently-allocated object. Unfortunately, this is generally not the case for a type-sensitive analysis. While it is

true for a full-object sensitive analysis [76], we found that this abstraction does not scale well.

We therefore use a novel analysis that provides full-object sensitivity for constructor calls, and provides type sensitivity everywhere else, ensuring scalability. This novel analysis is simply the cross-product of a type sensitive analysis with an analysis that uses a full-object-sensitive context to analyze constructors and a single other context to analyze all other methods.

In a full-object sensitive analysis the context is based on the abstract object for the receiver. Because, in Java bytecode, constructors are always called immediately after the allocation of the object being constructed, the receiver for constructor calls is always a most-recent abstract object. This means that if we use a full-object sensitive analysis for these constructor calls, the `this` variable inside constructor methods will (almost[4]) always be a single most-recent abstract object, and updates to fields on this object can be strong updates, enabling us to establish object invariants that hold after object initialization in the constructor.

3.4 Evaluation

In this section we present the results of applying our analysis to several applications in the Dacapo Benchmark suite (version 2006-10-MR2) [9]. We present and discuss the performance of the points-to analysis. For three of the applications, bloat, chart, and jython, our flow-sensitive analysis did not terminate after an hour so these do not appear in our results. We compare the precision of our flow-sensitive analysis to a flow-insensitive analysis

[4]It is possible to contrive an example where, in a constructor, an allocation of the same abstract object that the receiver pointed to occurs before a field update and prevents us from strongly updating this field. We did not encounter this type of code in our evaluation and suspect that it is rare in practice.

by examining the results of three client analyses: a non-null analysis, a cast removal analysis, and a numeric interval analysis. We also empirically demonstrate the importance of strong update by running each client analysis with and without strong update and comparing the results. All evaluation was performed on a 16 virtual-CPU Amazon EC2 instance (8 hyper-threaded cores) using Intel Xeon E5-2666 processors with 30GB of RAM (although none of our tests required more than 4GB of RAM).

3.4.1 Performance

Table 3.3: Flow-sensitive points-to analysis performance.

Program	Lines of code	SSA Instructions	Mean (s)	Std Dev
antlr	62,750	79,395	55.11	4.08
eclipse	130,236	123,860	91.67	3.21
fop	74,621	88,348	62.13	3.44
hsqldb	46,562	46,375	10.25	0.72
luindex	104,306	104,774	19.22	2.64
lusearch	49,885	50,121	9.61	1.31
pmd	127,351	145,389	25.17	1.10
xalan	64,754	71,482	14.76	1.11

Table 3.3 shows the mean performance of our flow-sensitive analysis. We analyzed each application together with all library code including JDK version 1.6; the number of lines of code and WALA SSA instructions express the size of all methods that are reachable from the program entry point including library code. To compute the average and standard deviation we ran each analysis 10 times. While the performance is significantly worse than our flow-insensitive points-to analysis (taking on average over 40 times longer), the absolute time taken for the analysis is still reasonable: the slowest example takes just over a minute and a half. For some client applications, the additional gain in precision may make this a

reasonable trade off.

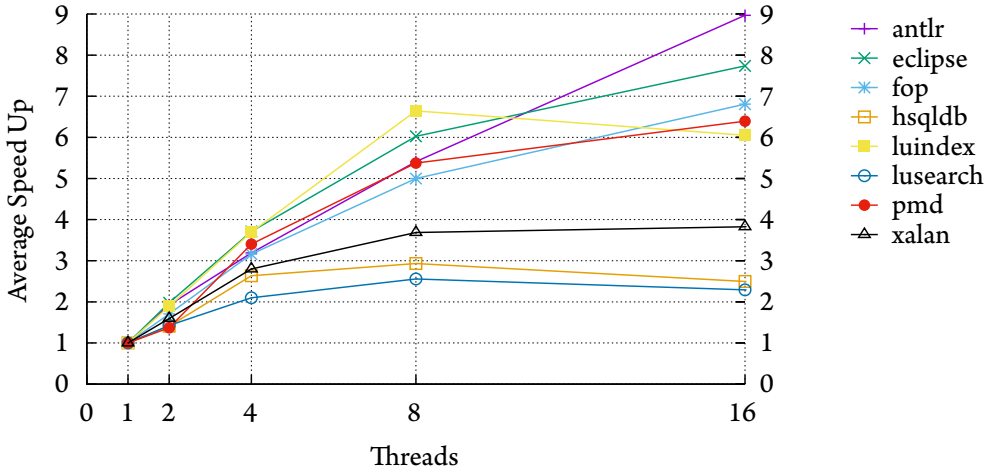


Figure 3.7: Performance of the flow-sensitive points-to analysis vs. number of processing threads.

Figure 3.7 shows how our analysis scales with the number of processing threads. All tests were run on a single 16 vCPU instance, but using differing numbers of worker threads in Java's fork/join framework. We see a general upward trend, with some of faster benchmarks trending down with increasing thread count as contention for shared data structures including contention on the work queues takes proportionally longer.

3.4.2 Client analyses

To test the precision of our points-to analysis we implemented three useful and common client analyses. These analyses rely on the results of the points-to analysis for their precision.

Non-null analysis A non-null analysis is a data-flow analysis that determines the set of variables and memory locations that are definitely non-null or may-be-null before and af-

ter every instruction in an application. In Java every non-static method call, non-static field access, array access, and even `throw` statement may potentially dereference a null pointer. For safety reasons, a Java Virtual Machine (JVM) dynamically checks these operations before executing them. If a dereferenced pointer (e.g., through a field access or method call) is null a run-time `NullPointerException` is thrown and, if it is not handled, the JVM will exit. A non-null analysis can be used to statically detect both possible null-pointer accesses, which may indicate buggy code, and places where it is impossible for a given pointer to be null, which may allow for the removal of dynamic checks.

Table 3.4: The precision of the non-null analysis as measured by the percent of possible `NullPointerExceptions` (e.g. non-static method calls) that can be proved impossible by the analysis.

Program	Percent of possible NPE proved impossible		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	91.95%	84.93%	84.85%
eclipse	91.00%	86.96%	83.51%
fop	93.39%	88.21%	86.90%
hsqldb	94.24%	89.97%	89.15%
luindex	92.25%	87.36%	83.39%
lusearch	95.16%	90.47%	89.18%
pmd	92.82%	88.24%	85.63%
xalan	94.55%	88.24%	88.12%
TOTAL	92.88%	87.95%	85.72%

Table 3.4 shows the difference in precision of the non-null analysis in three different configurations. The first column shows the percent of `NullPointerExceptions` that can be proven impossible using the results of our flow-sensitive points-to analysis. The second column shows the percentages using the results of a flow-sensitive points-to analysis that does not perform strong update, i.e., in the points-to analysis all object fields are weakly updated. The third column shows the results when the non-null analysis uses the results of

a flow-*insensitive* points-to analysis.

The flow-sensitive analysis without strong update enables the removal of 87.95% of the possible null-pointer exceptions in the program, more than the flow-insensitive analysis which removes 85.72%. This is due to a more precise points-to graph, but the real benefit is seen when using the results of our flow-sensitive points-to analysis with strong update. Here, 92.88% of the exceptions are proven impossible. Compared to the flow-sensitive analysis with strong update, the flow-insensitive analysis leaves more than twice as many locations in the code that the programmer needs to manually check for errors and/or that the run-time system needs to dynamically inspect. The difference between the flow-sensitive analysis with and without strong update shows the benefit of the recency abstraction, which, as discussed in Section 3.1, is what enables client analyses to usefully employ strong update.

Cast removal The cast removal analysis makes a single pass over the code, inspecting each `checkcast` instruction. If a given instruction casts program variable `v` to type T and if the points-to set for (all replicas of) `v` contains only abstract objects that are subtypes T then the cast can never fail. In this case the `checkcast` instruction is unnecessary and can safely be removed.

Table 3.5 shows the percentage of casts that can be removed using the results of our points-to analysis, a flow-sensitive points-to analysis without strong update, and a flow-insensitive points-to analysis. Our analysis removes 60.72% of the casts across all programs, which is over 35% more than the flow-insensitive analysis and 17% more than the flow-sensitive analysis without strong update.[5] This increase in precision is due to the more pre-

[5]For two of the programs, the flow-sensitive analysis without strong update performs slightly worse

Table 3.5: The precision of a cast removal analysis measured by the percent of dynamic casts that are always allowed.

Program	Percent of dynamic casts that can be proven unnecessary		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	58.55%	52.71%	47.27%
eclipse	60.39%	47.99%	45.51%
fop	57.03%	49.81%	50.61%
hsqldb	59.69%	49.64%	52.26%
luindex	61.57%	53.72%	48.69%
lusearch	62.20%	54.01%	50.00%
pmd	66.54%	52.88%	35.44%
xalan	58.79%	58.12%	50.23%
TOTAL	60.72%	51.65%	44.73%

cise points-to sets.

Interval analysis A numeric interval analysis conservatively approximates the value of program variables and fields with numeric type. Like the non-null analysis, the interval analysis is a data-flow analysis.

Figure 3.6 gives the percentage of intervals that the analysis determines cannot contain zero. Using the results of a flow-sensitive points-to analysis with strong update results in 9% more intervals that cannot contain zero than the analysis that uses flow-insensitive points-to results. The performance of the interval analysis using a flow-sensitive points-to analysis with and without strong update is essentially equivalent.

---

percentage-wise than the flow-insensitive analysis. This is due to a more precise call graph (i.e., fewer methods are reachable), which means there are fewer class casts that could potentially be removed.

Table 3.6: The precision of the interval analysis as measured by the percent of intervals that contain zero.

Program	Percent of intervals that do not contain zero		
	Flow-sensitive w/ Strong Update	Flow-sensitive no Strong Update	Flow-insensitive
antlr	62.19%	62.02%	59.15%
eclipse	62.98%	62.45%	58.11%
fop	58.83%	58.08%	57.40%
hsqldb	63.22%	63.14%	55.40%
luindex	58.51%	57.94%	55.20%
lusearch	61.55%	60.89%	55.62%
pmd	64.81%	64.14%	53.70%
xalan	57.06%	54.75%	53.88%
TOTAL	61.06%	60.64%	56.06%

3.5 Related work

Our work focuses on providing strong update in analyses of object-oriented programs, which we achieve via a points-to analysis that uses Balakrishnan and Reps' recency abstraction [6] with flow-sensitivity for singleton abstract locations. We focus here on previous work related to strong update and flow-sensitive points-to analysis. Discussion on multi-threaded points-to analysis can be found in Section 2.5.

Strong update Lhoták and Chung [62] present an efficient points-to analysis with strong update for C/C++ programs. The analysis is flow sensitive only for singleton abstract locations, and treats an abstract location flow insensitively when its points-to set contains two or more elements. The analysis is cubic in the size of the program, although context insensitive. Our approach to strong update and flow sensitivity is based on their work. However, their techniques are not directly applicable to object-oriented programs [61], since without careful context sensitivity, there are few singleton abstract locations in object-

oriented programs. To achieve efficient strong update in Java-like languages, we needed to incorporate context sensitivity and the recency abstraction [6] in order to usefully increase the number of singleton abstract locations.

Balakrishnan and Reps [6] introduce the recency abstraction that we use in this paper. They use it to resolve virtual function calls in executables. The recency abstraction is effective in their setting because function dispatch tables are typically established immediately after creating an object, and thus gaining additional precision on the most recently created object enables precision exactly where it is most useful. It is effective in our setting, for both points-to analysis and client analyses, for similar reasons: because object invariants are established during object construction, the recency abstraction enables precision exactly where it is most useful.

Balakrishnan and Reps also point out that many points-to analyses for C unsoundly assume that the initial points-to set for a pointer is the empty set when, in fact, all pointer variables initially point to uninitialized memory (i.e., they can take on any address). Similarly in Java, it is unsound to assume that the initial points-to set for fields with reference type is the empty set, since the initial points-set is actually {`null`}. To our knowledge all of the flow-sensitive points-to analyses we discuss in this section that perform strong update make this unsound assumption in order for their analyses to scale and perform strong update. Our analysis does not make this assumption. Instead we soundly rely on the recency abstraction to strongly update fields when they are first assigned to.

Sagiv et al. [95] use a 3-valued logic to reason about shape analysis, and introduce a "focus" operation, which permits the extraction of a single location from a summary node (i.e., a node in a shape graph that represents multiple locations), and thus enables precise reasoning and strong update for the single location. Hackett and Rugina [34] introduce a

novel shape abstraction where each shape configuration is concerned with reasoning precisely about a single heap location, and thus strong update can be performed on the single heap location. Yavuz-Kahveci and Bultan [119] use a shape abstraction that explicitly counts the number of concrete locations represented by an abstraction location, thus enabling strong update.

Dillig et al. [24] generalize strong and weak update to reasoning about under- and over-approximations of the locations that an update might modify. Specifically, they consider updates to array elements, where approximating the locations that may be updated corresponds to approximating the array index.

Flow-sensitive points-to analysis Most work on flow-sensitive points-to analysis casts it as a data-flow problem on an interprocedural control flow graph or a similar data structure. Choi et al. [18] present a flow-sensitive analysis for C that uses an iterative traversal of the call graph where intraprocedural aliasing information is computed with a data-flow on a *sparse evaluation graph*: a control flow graph that elides all basic blocks that cannot affect points-to information. Hind and Pioli [44] improve Choi et al.'s performance by using a worklist algorithm and only propagating reachable alias relations to callees, and Goyal [32] improves the computational complexity using finite differencing and dominated convergence optimizations.

Using a *static single assignment* (SSA) program representation [20] can improve the propagation of points-to information, by making def-use chains more easily available. In addition, SSA form can provide a form of flow sensitivity even for a flow-insensitive analysis (which we benefit from for local variables in our analysis, using WALA's partial SSA form). Chase et al. [15] propose an algorithm for a Lisp-like language that dynamically

transforms a whole program into SSA form during the points-to analysis. Tok et al. [109] implement a similar algorithm that recomputes both client analysis results and points-to results as new dependencies in the points-to graph are discovered. Hasti and Horwitz [41] propose a flow-insensitive analysis that iteratively transforms a program into SSA form, with each iteration providing more precise results. They speculate that in the limit their analysis approaches the same precision as a flow-sensitive analysis. Staiger-Stöhr [104] also computes a whole-program SSA, but does so incrementally during a flow-sensitive points-to analysis.

Hardekopf and Lin [39] present an analysis for C that uses a partial SSA form (for local variables) and an iterative data-flow analysis to propagate points-to information. They note that SSA form implies that flow sensitivity gives additional precision only for address-taken variables, which greatly reduces the size of the propagated flow-sensitive points-to information. This observation does not carry over to object-oriented languages, where almost all locations are "address taken", since they are fields of dynamically created objects. They further improve their analysis by using def-use information for heap allocated variables computed using an auxiliary flow-insensitive points-to analysis [40].

Yu et al. [121] present a flow-sensitive context-sensitive points-to analysis for C programs, also using an auxiliary flow-insensitive points-to analysis. They partition variables into *levels* and use points-to information at higher levels to transform variables at lower levels into SSA form.

Nagaraj and Govindarajan [79] implement a flow-sensitive context-insensitive version of the graph-rewriting-based analysis of Méndez-Lojo et al. [74] using a work-list to manage the graph rewrites.

Kahlon [50] also uses an auxiliary flow-insensitive points-to analysis to partition a pro-

gram into parts on which the analysis can run independently and in parallel, thus improving the efficiency of the flow-sensitive analysis (note that they simulate parallelism and do not implement a multithreaded version of their analysis). Ye et al. [120] also partition the program, and their analysis is flow sensitive only between partitions, using a flow insensitive analysis within a partition. Nasre [81] bootstraps a context-insensitive GPU implementation with a flow-insensitive analysis and uses a two-dimensional variation of a bloom filter efficiently represent large data structures. Unrelated pointers can imprecisely share information due to collisions in the hash functions used for the bloom filters.

Li et al. [66] formulate the results of a flow-sensitive points-to analysis for C as a graph reachability problem on a *value flow graph*. Traditional points-to sets are computed using the transitive closure of the value flow graph.

Several of the flow-sensitive points-to analyses described above scale to millions of lines of C code. However, in general, C code typically has fewer address-taken variables and abstract heap locations than Java code, where every field is represented in a points-to analysis as an abstract location. De and D'Souza [21] present a scalable flow-sensitive points-to analysis for Java that computes the points-to sets of *access paths* (i.e., sequences of fields accesses), instead of points-to sets of variables and object fields. This provides more opportunities for strong update, as often the points-to set of an access path in a given context is a singleton. They use an auxiliary flow-insensitive points-to analysis to bootstrap their analysis and reduce the time needed to reach a fix point. Scalability is achieved by limiting access paths to length 2. While they reduce the size of the call graph compared to a flow-insensitive analysis, they do not report whether their points-to analysis provides additional precision for client analyses, so the relative precision with respect to our analysis is unclear.

4

Pidgin

4.1 Introduction

What security guarantees a program provides is a property of the whole program and these guarantees are application-specific. Applications deal with different types of sensitive data as diverse as personal photos, credit card numbers, and medical records. What constitutes correct handling of this information varies greatly between applications: a shopping website may reveal the last four digits of your credit card number and a message to a friend may include a photo (but only the specific photo and only to the specific recipient).

This chapter presents a new approach to achieving application-specific security guarantees. We combine a whole-application program-dependence graph (PDG) [29, 47], which precisely captures the information flows in a program, with a domain-specific graph query language. Paths in a PDG correspond to information flows in an application. Our queries express properties of these paths which therefore correspond to information-flow security guarantees provided by the application. This methodology as implemented in our tool, Pidgin, has several benefits:

— *Expressive policies* Because policies in Pidgin are queries in a language specifically built to describe how information flows through a program, Pidgin can be used to express complex and precise whole-program security policies. These include noninterference [31], absence of explicit information flows (taint-tracking), trusted de-

classification [42], and mediation of information-flow by access control checks.

- *Support for exploration and discovery* PIDGIN provides a unified approach that supports the exploration, expression, and enforcement of information security policies. PIDGIN can be used interactively, using successive queries and their results to explore how information propagates through a program and what security guarantees that program provides. This is especially effective for legacy applications that were not developed with a specific a priori security policy. Exploration can help discover what security guarantee an application provides and decide whether that guarantee is sufficient. If an appropriate security guarantee is identified, PIDGIN can be used to succinctly express the policy corresponding to that guarantee and check whether it holds.
- *Policies separate from code* PIDGIN security policies do not require any modification to application code and enforcement of PIDGIN policies does not require any runtime support. This means that PIDGIN does not interfere with other development and testing efforts, and developers can choose how to balance the development of new functionality with the maintenance of security policies.
- *Support for regression testing* PIDGIN policies can be used to check security guarantees during development, perhaps as part of a build process. If a policy fails, PIDGIN can be used to help understand why by identifying information flows in the program that violate the policy.

These benefits stand in contrast to existing tools and techniques. Security-type systems (e.g., Volpano et al. [114], Jif [78] and FlowCaml [98]) and previous work that uses PDGs to enforce information security (e.g., [30, 35, 37]) specify polices as annotations scattered throughout the program. This can make it hard to identify the security guarantee provided

by the application especially in the presence of declassification [94]. In addition, modifying the application or security policy may require the modification of many program annotations. Support for legacy applications is difficult or impossible since it requires the addition of many annotations and possibly other modifications.

Dynamic information-flow enforcement techniques (e.g., [4, 5, 14, 48, 57, 106]) can sometimes express policies separate from code, but interfere with development and must be used during testing to ensure that the enforcement mechanism does not prevent intended functionality.

Static and dynamic taint-tracking tools (e.g., [3, 17, 27, 65, 110, 111, 122]) do not consider implicit information flows [53], support a limited class of policies, and often do not support application-specific policies. One of the most recent, FlowDroid [3], works with a pre-defined (i.e., not application-specific) set of sources and sinks and does not support sanitization, declassification, or access control policies. Because PIDGIN supports more expressive policies, we detect 159 of the 163 (=98%) vulnerabilities in the SecuriBench Micro [70] 1.08 test suite compared to Flowdroid's 117 (=72%).

In addition, previous work focuses almost exclusively on the enforcement of security policies with no support for exploration and discovery.

The rest of this chapter describes the design and implementation of PIDGIN. PIDGIN combines PDGs with an expressive policy language to provide a unified approach enabling exploration, specification, and enforcement of application-specific security guarantees.

PIDGIN produces PDGs for Java bytecode and evaluates queries against these PDGs, either interactively or in batch mode. Our techniques are applicable to other languages.[1]

[1]We have generated PDGs for C/C++ programs by analyzing LLVM bitcode [56] produced by the clang compiler (`http://clang.llvm.org/`), and explored information security in these programs using the same query language and query evaluation engine. This dissertation focuses on our Java tool.

PIDGIN is both useful and scalable. We have used PIDGIN to discover diverse information security guarantees in legacy Java applications, and to specify and enforce information security policies as part of the development process for two new applications. We have analyzed programs ranging in size up to 330,000 lines of code (including library code); even for the largest program, construction of the PDG (including our flow-insensitive points-to analysis, and dataflow analyses to improve precision) takes 90 seconds, and checking each of our policies on the PDG takes less than 14 seconds.

Security guarantees we have established using PIDGIN include: in a password manager, the master password is not improperly leaked; in a chat server application, punished users are restricted to certain kinds of messages; and in a course management system, the class list is correctly protected by access control checks. Moreover, we have developed security guarantees based on reported vulnerabilities in Apache Tomcat, and PIDGIN verifies that the security guarantees hold after the vulnerability is patched and fail to hold in earlier versions.

4.2 PIDGIN BY EXAMPLE

Consider the Guessing Game program presented in Figure 4.1. This program randomly chooses a secret number from 1 to 10, prompts the user for a guess, and then prints a message indicating whether the guess was correct.

A program dependence graph (PDG) representation of this program is shown in Figure 4.2. Shaded nodes are *program-counter nodes*, representing the control flow of the program. All other nodes represent the value of an expression or variable at a certain program point. There is a single summary node representing the formal argument to the `output`

```
1  secret = getRandom(1, 10);
2
3  output(“Guess a number between 1 and 10”);
4  int guess = getInput();
5
6  bool correctGuess = (secret == guess);
7  if (correctGuess) {
8     output(“Congratulations! ”+ guess + “ was right”);
9  }
10 else {
11    output(“Sorry, your guess was incorrect”);
12 }
```

Figure 4.1: Guessing Game program

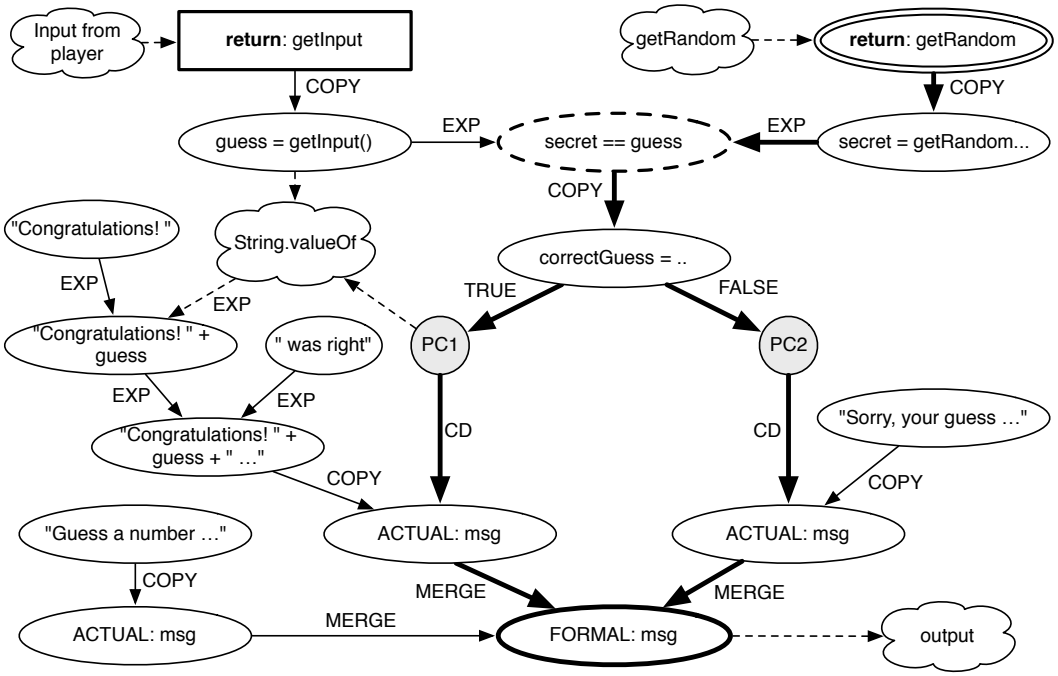


Figure 4.2: PDG for Guessing Game program

function. There are three nodes representing actual arguments, one for each call to `output`, and an edge from each to the formal argument. Edges labeled CD indicate control dependencies and other edges indicate data dependencies. Dashed edges and clouds show where we have elided parts of the PDG for clarity. (All other emphasis is for the exposition below. Program-counter nodes that are not relevant to the discussion have been removed for simplicity.)

Although the Guessing Game program is simple, it has interesting security properties that can be expressed as queries on the PDG.

NO CHEATING! The program should not be able to cheat by choosing a secret value that is deliberately different from the user's guess. That is, the choice of the secret should be independent of the user's input. This policy holds if the following PIDGINQL query returns an empty graph:

```
let input = pgm.returnsOf("getInput") in
let secret = pgm.returnsOf("getRandom") in
pgm.forwardSlice(input) ∩ pgm.backwardSlice(secret)
```

PIDGINQL is a domain specific graph query language that enables exploration of a program's information flows, and specification of information security policies. Constant pgm, short for *program*, is bound to the PDG of the program. Primitive expressions (such as forwardSlice) compute a subgraph of the graph to the left of the dot. Query expression pgm.returnsOf("getInput") evaluates to the node in the program PDG that represents the value returned from function `getInput` (shown in a rectangle in Figure 4.2). This is the user's input. Similarly, the second line identifies the node representing the value returned from function `getRandom` as the secret. This node is outlined with a double circle in the PDG.

Query expression pgm.forwardSlice(input) evaluates to the subgraph of the PDG that is reachable by a path starting from the query variable input. This is the subgraph that depends on the user input, either via control dependency, data dependency, or some combination thereof. Similarly, pgm.backwardSlice(secret) is the subgraph of the PDG that can reach the node representing the secret value. The entire query evaluates to the intersection of the subgraphs that depend on the user input and on which the secret depends, i.e., all paths from the user input to the secret.

For the PDG in Figure 4.2, this query evaluates to an empty subgraph. This means that there are no paths from the input to the secret, and thus the secret does not depend in any way on the user input.

Finding all nodes in the PDG that lie on a path between two sets of nodes is a common query, and we can define it as a reusable function in PidginQL as follows:

```
let between(G, from, to) = G.forwardSlice(from) ∩ G.backwardSlice(to)
```

This allows us to simplify our query. We can also turn our PidginQL query into a security policy (i.e., a statement of the security guarantee offered by the program) by asserting that the result of this query should be an empty graph. This is done in PidginQL by appending "is empty" to the query.

Noninterference Noninterference [31, 93] requires that information does not flow from confidential inputs to public outputs. For our purposes, the secret number (line 1 in Figure 4.1) is a confidential input, and output statements (lines 3, 8, and 11) are publicly observable.

We can check whether noninterference holds between the secret and the outputs using a query similar to the one above:

```
let secret = pgm.returnsOf("getRandom") in
let outputs = pgm.formalsOf("output") in
pgm.between(secret, outputs)
```

Unlike our previous example the query does not result in an empty subgraph. Indeed, this program does not satisfy noninterference: there are two paths from the secret to the output (marked in Figure 4.2 with bold lines). This is not surprising, as the functionality of this program requires that some information about the secret is released.

FROM SECRET TO OUTPUT By characterizing all paths from the secret to the output we can provide a guarantee about what the program's public output may reveal about the secret. Inspecting the result of the noninterference query above, we see there are two paths from the secret to the public outputs. (If there were many paths, we could have isolated one path to examine, by changing the last line to pgm.shortestPath(secret, outputs).) Both paths pass through the node for the value of expression "`secret == guess`". This means that the public output depends on the secret only via the comparison between the secret and the user's guess. We can confirm this by removing this node from the graph and checking whether any paths remain between the secret and the outputs. This can be expressed in PIDGINQL as:

```
1 let secret = pgm.returnsOf("getRandom") in
2 let outputs = pgm.formalsOf("output") in
3 let check = pgm.forExpression("secret == guess") in
4 pgm.removeNodes(check).between(secret, outputs)
5 is empty
```

Expression pgm.forExpression("secret == guess")[2] evaluates to the node for the conditional expression (outlined in Figure 4.2 with a dotted line). The fourth line removes this node from the PDG then computes the subgraph of paths from secret to outputs.

This query results in an empty subgraph, meaning we have described all paths from secret to outputs. Thus the program satisfies the policy: *The secret does not influence the output except by comparison with the user's guess.*

This is an example of trusted declassification [42] and is a pattern found in many applications. We capture this with a user-defined policy function asserting that all flows from srcs to sinks pass through a node in declassifiers.

```
let declassifies(G, declassifiers, srcs, sinks) =
G.removeNodes(declassifiers).between(srcs, sinks)
is empty
```

Using this function we change our policy to:

```
let secret = pgm.returnsOf("getRandom") in
let outputs = pgm.formalsOf("output") in
let check = pgm.forExpression("secret == guess") in
pgm.declassifies(check, secret, outputs)
```

Note that our policy is weaker than noninterference: the output does depend on the secret. Noninterference is too strong to hold in many real programs, and weaker, application-specific guarantees are common. PDGs often contain enough structure to characterize these (potentially complex) security guarantees, which can be stated succinctly and intuitively given an expressive language to describe and restrict permitted information flows.

[2]For presentation reasons we refer to the specific Java expression "secret == guess". In a more realistic example, a policy would likely refer instead to a function or class, which is less brittle with respect to code changes. However, the ability to refer to specific expressions allows developers to precisely specify queries and policies if needed.

4.3 Program dependence graphs (PDGs) and security guarantees

Pidgin allows programmers to explore a program's information flows and to express and enforce security policies that restrict permitted information flows. We achieve this using program dependence graphs (PDGs) [29] to explicitly represent the data and control dependencies within a program. Pidgin's PDGs represent control and data dependencies within a whole program. Annotations and meta-information encoded in Pidgin PDGs enable precise and useful queries and security policies. In this section, we describe the structure of Pidgin's PDGs and the different kinds of security guarantees that can be obtained from them.

4.3.1 Structure of Pidgin PDGs

There are several kinds of nodes in Pidgin PDGs. *Expression nodes* represent the value of an expression, variable, or heap location at a program point. *Program-counter nodes* represent the control flow of a program, and can be thought of as boolean expressions that are true exactly when program execution is at the program point represented by the node. In addition, *procedure summary nodes* facilitate the interprocedural construction of the PDG by summarizing a procedure's entry point, arguments, return value, etc. Finally, *merge nodes* represent merging from different control flow branches, similar to the use of phi nodes in static single assignment form [20]. Nodes also contain metadata, such as the position in the source code of the expression a node represents.

Pidgin PDGs are context sensitive, object sensitive, and field sensitive. A context-sensitive PDG will have a copy of the PDG for a method for each context in which that method appears. Methods in our PDG analysis inherit contexts from the points-to analysis used by the PDG analysis. In this work they are flow sensitive for local variables and flow insensitive

for heap locations. We discuss more about PDG flow sensitivity in Section 4.5.

Edges of the PDG indicate data and control dependencies between nodes. To improve precision and enable more complex queries, edges in PIDGIN PDGs have labels that indicate *how* the target node of the edge depends on the value represented by the source node of the edge. Examples of these edge labels can be seen in Figure 4.2. COPY indicates that the value represented by the target is a copy of the source. EXP indicates that the target is the result of some computation involving the source. Edges labeled MERGE are used for all edges whose target is a merge or summary node.

Label CD indicates a control dependency from a program-counter node to an expression node. An expression is control dependent on a program-counter node if it is evaluated only when control flow reaches the corresponding program point. An edge labeled TRUE or FALSE from an expression node to a program-counter node indicates that control flow depends on the boolean value represented by the expression node.

4.3.2 SECURITY GUARANTEES FROM PDGS

As Section 4.2 demonstrated, paths in a PDG can correspond to information flows in a program, and PIDGIN allows developers to discover, specify, and enforce security guarantees.

Information security guarantees are application specific, since what is regarded as sensitive information and what is regarded as correct handling of that information varies greatly between applications. The query language PIDGINQL (described in Section 4.4) provides several convenient ways for developers to indicate sources and sinks, such as queries that select the values returned from a particular function. The ability for PIDGINQL to specify relevant parts of the graph means that the program does not require annotations for security policies. PIDGIN can be used to describe many complex policies. We next describe

several kinds of security guarantees that developers can express using PidginQL.

Noninterference The absence of a path in a PDG from a source to a sink indicates that noninterference holds between the source and the sink. This result was proved formally by Wasserrab et al. [115]. As seen in Section 4.2, this is equivalent to the PidginQL query pgm.between(source, sink) evaluating to an empty graph:

```
let noninterference(G, source, sink) = G.between(source, sink) is empty
```

Noninterference is a strong guarantee, and many applications that handle sensitive information will not satisfy it: the query pgm.between(source,sink) will result in a non-empty graph. For example, an authentication module doesn't satisfy noninterference because it needs to reveal some information about passwords (specifically, whether a user's guess matches the password).

Even when noninterference does not hold, developers need assurance that the program handles sensitive information correctly. For example, a developer may want the result of the authentication module to depend on the password *only* via an equality test with the guess. In the remainder of this section, we describe security guarantees that are weaker than noninterference and can be expressed as queries on PDGs.

No explicit flows A coarse-grained notion of information-flow control considers only *explicit* information flows and ignores *implicit* information flows [23]. This is also known as *taint tracking* and corresponds to considering only data dependencies and ignoring control dependencies.

Although arbitrary information may flow due to control dependencies, it can be useful and important to show that there are no explicit information flows from sensitive sources

to dangerous sinks. Indeed, the prevalence of taint-tracking mechanisms (e.g., Perl's taint mode, and numerous systems [3, 65, 111, 122]) show that it is intuitive and appealing for developers to consider just explicit flows. Moreover, tracking only explicit flows can lead to fewer false positives (albeit at the cost of more false negatives) [27, 53].

Restricting attention to data dependencies is straightforward with a PDG. Specifically, if all paths from sensitive sources to sensitive sinks have at least one edge labeled CD (i.e., a control dependency from a program-counter node to an expression node), then there are no explicit flows from the source to the sink. This can be expressed by the following PidginQL policy function:

```
let noExplicitFlows(sources, sinks) = pgm.removeEdges(pgm.selectEdges(CD))
                                        .between(sources, sinks) is empty
```

Expression pgm.removeEdges(pgm.selectEdges(CD)) selects all edges labeled CD in the PDG and removes them from the graph. Using this graph, between(sources, sinks) finds the subgraph containing all paths between sources and sinks. If this results in an empty graph the policy holds, and there are no explicit flows from the sources to the sinks.

Often a program intentionally contains explicit flows (e.g., a program that prints the last four digits of a credit card number). To obtain guarantees in this case, a more precise policy is needed.

Describe all information flows In general, a developer can specify a security policy by describing all permitted paths from sensitive sources to dangerous sinks. This is because paths in the PDG correspond to information flows in the program. Using the query language, the developer can enumerate the ways in which information is permitted to flow. If, after removing paths corresponding to these permitted information flows, only an empty graph remains then all information flows in the program are permitted, and the program

satisfies the security policy. The "no explicit flows" example can be viewed in this light (i.e., the policy requires that all paths from a source to a sink must involve a control dependency), but more expressive characterizations of paths are often necessary, useful, and interesting.

For example, consider a program that takes a (secret) credit card number and prints the last four digits. This is an intentional explicit flow, though most taint analysis frameworks would mark it as a security violation. The following policy requires that all paths from the credit card number to the output go through the return value of method `lastFour`.

```
let ccNum = ... in
let output = pgm.formalsOf("output") in
let lastFourRet = pgm.returnsOf("lastFour") in
pgm.declassifies(lastFourRet, ccNum, output)
```

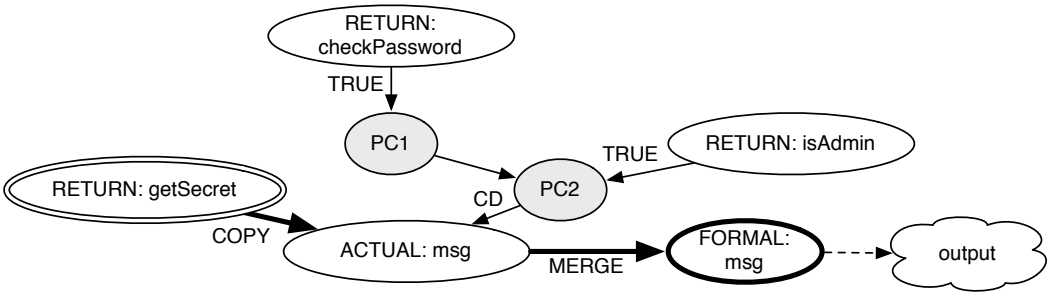
Recall that pgm.declassifies(lastFourRet, ccNum, output) (defined in Section 4.2) removes the nodes lastFourRet from graph pgm, and asserts that in the resulting graph there are no paths from ccNum to output.

This policy treats method `lastFour` as a *trusted declassifier* [42]: information is allowed to flow from ccNum to output provided it goes through the return value of `lastFour` because `lastFour` is trusted to release only limited information about credit card numbers. Determining whether `lastFour` is in fact *trustworthy* is beyond the scope of this work. Trustworthiness of `lastFour` could, for example, be achieved through a code review, or through formal verification of its correctness. Nonetheless, this PidginQL policy provides a strong security guarantee, and reduces the question of correct information flow in the entire program to the trustworthiness of one specific method.

Describe conditions for information flow In some cases it is important to know not just the flows from sensitive sources to dangerous sinks, but also under what conditions

```
1 if (checkPassword(pwd))
2     if (user.isAdmin())
3         output(getSecret());
```

(a) Access control program



(b) Relevant fragment of PDG

Figure 4.3: Access control example

these flows occur. Using PDGs, we can extract this information by considering control dependencies of nodes within a path. This is difficult for most existing information-flow analyses, as the conditions under which a flow occurs are not properties of the paths from sources to sinks.

For example, consider the program in Figure 4.3a, which is a simple model of an access control check guarding information flow. Secret information is output at line 3, but only if the user provided the correct password (line 1) and the user is the administrator (line 2). If we look at the relevant fragment of the PDG for this program (Figure 4.3b) we see that there is a single path from a sensitive source (the double-circled node for the return from the `getSecret` function) to a dangerous sink (the bold node representing the formal argument to output). By examining the control dependencies for one of the nodes on this path, we can determine that this flow happens only if both access control checks pass. That is, all paths from the sensitive source to the dangerous sink are control dependent on both

"`checkPassword`" and "`isAdmin`" returning true.

We can describe this with the policy:

```
1 let sec = pgm.returnsOf("getSecret") in
2 let out = pgm.formalsOf("output") in
3 let isPassRet = pgm.returnsOf("checkPassword") in
4 let isAdRet = pgm.returnsOf("isAdmin") in
5 let guards = pgm.findPCNodes(isPassRet, TRUE) ∩
6                pgm.findPCNodes(isAdRet, TRUE) in
7 pgm.removeControlDeps(guards).between(sec, out)
8 is empty
```

Lines 1 and 2 find the appropriate PDG nodes for the secret and output functions, respectively. The expressions on lines 5 and 6 finds any program counter nodes in the PDG corresponding to program points that can be reached only when `checkPassword` and `isAdmin` return true. The primitive removeControlDeps(*E*) removes nodes from the graph that are control dependent on any program counter node in *E*. Intuitively, the graph computed by pgm.removeControlDeps(guards) is the result of removing all nodes that are reachable only when the password is correct and the user is the admin. The following policy function captures this pattern:

```
let flowAccessControlled(G, checks, srcs, sinks)  =
  G.removeControlDeps(checks).between(srcs, sinks)
  is empty
```

In the example above, access control checks protect information flow from a confidential source to a public output. A simpler pattern is when access control checks guard execution of a sensitive operation. The following policy function asserts that execution of sensitiveOps (representing some sensitive operation, such as calls to a dangerous procedure) occurs only when access control checks represented by checks succeed:

```
let accessControlled(G, checks, sensitiveOps)  =
  G.removeControlDeps(checks) ∩ sensitiveOps
  is empty
```

Query	$Q ::= F\ Q \mid E$
Policy	$P ::= F\ P \mid E$ is empty $\mid p(A_0, \ldots, A_n)$
Function Definition	$F ::=$ let $f(x_0, \ldots, x_n) = E$; $\mid$ let $p(x_0, \ldots, x_n) = E$ is empty;
Expression	$E ::=$ pgm $\mid E.PE \mid E_1 \cup E_2 \mid E_1 \cap E_2$ $\mid$ let $x = E_1$ in $E_2 \mid x \mid f(A_0, \ldots, A_n)$
Argument	$A ::= E \mid$ *EdgeType* $\mid$ *NodeType* $\mid$ *JavaExpression* $\mid$ *ProcedureName*
Primitive Expression	$PE ::=$ forwardSlice(E) $\mid$ backwardSlice(E) $\mid$ shortestPath(E_1, E_2) $\mid$ removeNodes(E) $\mid$ removeEdges(E) $\mid$ selectEdges$(EdgeType)$ $\mid$ selectNodes$(NodeType)$ $\mid$ forExpression$(JavaExpression)$ $\mid$ forProcedure$(ProcedureName)$ $\mid$ findPCNodes$(E, EdgeType)$ $\mid$ removeControlDeps(E)
EdgeType	::= CD $\mid$ MERGE $\mid$ COPY $\mid$ EXP $\mid$ TRUE $\mid$ FALSE $\mid$ ...
NodeType	::= PC $\mid$ ENTRY_PC $\mid$ FORMAL $\mid$ ABSTRACT_LOC $\mid$ ...

Figure 4.4: PidginQL grammar

4.4 Querying PDGs with PidginQL

We have developed PidginQL, a domain-specific language that allows a developer to explore information flows in a program, and to specify security policies that restrict information flows. PidginQL is a graph query language, specialized to express readable and intuitive queries relevant to information security. The grammar for PidginQL is shown in Figure 4.4. The grammar includes let statements, functions, graph composition operations, and primitives that are useful for expressing information security conditions.

QUERIES AND EXPRESSIONS A query Q is a sequence of function definitions followed by a single expression. Expressions evaluate to graphs. There is a single constant expression, pgm (short for *program*), which always evaluates to the original program dependence graph for the program under consideration. A primitive expression PE is a function on a graph: $E_0.PE$ evaluates expression E_0 to a graph G_0 and then the primitive expression returns a subgraph of G_0, computed according to the semantics of the specified operation (which we describe in more detail below). Expression $E_1 \cup E_2$ evaluates E_1 and E_2 to graphs G_1 and G_2 respectively and returns the union of G_1 and G_2. Similarly, $E_1 \cap E_2$ evaluates both E_1 and E_2 and returns the intersection of the results. Expressions also include let bindings, variable uses, and invocations of user-defined functions.

POLICIES A policy P is a sequence of function definitions followed either by an assertion that expression E evaluates to an empty graph (E is empty) or an invocation of a user-defined policy function (which will assert that some expression evaluates to an empty graph). As discussed in Sections 4.2 and 4.3, if a query, Q, considers all information flows from sources to sinks, and removes only permitted flows, *and* Q results in an empty graph when evaluated on a program's PDG, the program contains only permitted information flows. Evaluating a policy results in an error if the assertion fails, i.e., if the query does not evaluate to an empty graph.

Queries are typically used when interactively exploring information flows, since non-empty query results can be examined and further explored to understand the information flows present in a program and discover security violations. Policies are useful for enforcement and regression testing (i.e., determining whether a modified program still satisfies a security guarantee).

PRIMITIVE EXPRESSIONS PIDGINQL contains several primitive operations for exploring information flows in programs and specifying restrictions on permitted information flows. Some of these are described throughout this chapter; for completeness we describe them all below.

Expression forwardSlice is useful for selecting everything *influenced by* sensitive sources and backwardSlice for selecting everything that *influences* critical sinks. Both forwardSlice and backwardSlice may take another argument (not shown in the grammar) that controls the depth of the slice, for example to select immediate successors of a node.

For example, expression E_0.forwardSlice(E_1) evaluates the subexpressions to graphs G_0 and G_1 and computes the subgraph of G_0 that is reachable from any node in G_1. We improve the precision of slicing by including only nodes of G_0 that are reachable from a node in G_1 by a *feasible path* (i.e., a path where method calls and returns are appropriately matched). The call graph we use to construct the PDG is necessarily a finite approximation of the actual control flow of the program. Removing infeasible paths from slices, an example of CFL-reachability [90], greatly improves the precision of queries and policies, as it helps mitigate the imprecision that arises from this finite approximation.[3]

Expression E_0.shortestPath(E_1,E_2) is useful during exploration to find a simple (feasible) path remaining after executing a query. This helps identify vulnerabilities or missing security conditions.

Expression E_0.removeNodes(E_1) evaluates the subexpressions to graphs G_0 and G_1 and returns the subgraph of G_0 with all nodes in G_1 removed (and all edges to or from these nodes removed). Expression E_0.removeEdges(E_1) evaluates the subexpressions to graphs

[3]Faster, but less precise primitive expressions (not shown in the grammar) are also provided that compute slices that may include infeasible paths.

G_0 and G_1 and returns a subgraph of G_0 with all edges in G_1 removed.

Operations selectEdges and selectNodes take, respectively, an edge type or node type, and return the subgraph that contains all edges or nodes of that type.

Expression E_0.forExpression(*JavaExpression*) evaluates E_0 to G_0 and selects nodes in G_0 that correspond to the given Java expression. Expression E_0.forProcedure(*ProcedureName*) evaluates to a graph containing the subgraphs representing the bodies of matching procedures. For example, E_0.forProcedure("java.io.PrintStream.print*") finds the nodes in G_0 for all methods of the class `java.io.PrintStream` whose name begins with `print` (e.g., `print` and `println`).

Expression E_0.findPCNodes(E_1, *EdgeType*) is used to find program counter nodes in E_0 that correspond to control-flow decisions based on expressions in E_1. Edge type *EdgeType* must be either TRUE or FALSE. If E_0 and E_1 evaluate to graphs G_0 and G_1 respectively, E_0.findPCNodes(E_1, TRUE) evaluates to the program counter nodes in G_0 that are reachable only by a TRUE edge from some expression node in G_1. That is, a program point corresponding to a program counter node in E_0.findPCNodes(E_1, TRUE) will be reached only if some expression in G_1 evaluates to true.

It is useful to combine the results of different findPCNodes queries. For example, query pgm.findPCNodes(a, TRUE) ∩ pgm.findPCNodes(b, FALSE) will evaluate to the program counter nodes for program points that are reached only when an expression denoted by a evaluates to true *and* an expression denoted by b evaluates to false. We add syntactic sugar to describe these combinations using boolean expressions in square brackets, simplifying the example to pgm.[a && !b].

Expression E_0.removeControlDeps(E_1) can be used in combination with findPCNodes, for removing nodes that are control dependent on a boolean expression. For example,

findPCNodes can find the program counter nodes that correspond to program points that are reached only when a particular expression is true and removeControlDeps can remove any nodes control dependent on those program counter nodes. These are the nodes that represent expressions that only evaluate when program points represented by those program counter nodes are reached. In this way we can compute a graph that does not contain any nodes that correspond to parts of the program that are only reached when a particular expression is true. In Section 4.3, we use removeControlDeps to define access control policies. Using the syntactic sugar defined in the previous paragraph together with removeControlDeps makes it easy to define access control policies with complex access requirements (for examples, see the policies presented in Section 4.7.2).

Any primitive expression that takes a *ProcedureName* or *JavaExpression* as an argument will raise an error if it evaluates to an empty graph. This ensures that API changes, such as changing a method name, will trigger an error until a corresponding change is made to the PidginQL policy.

User-defined functions PidginQL functions are defined with let $f(x_0, \ldots, x_n) = E$ and let $p(x_0, \ldots, x_n) = E$ is empty. Function definitions are either graph functions (which will evaluate to a graph) or policy functions (which assert that some expression evaluates to an empty graph).[4] Functions are invoked with syntax $f(A_0, \ldots, A_n)$. We also support $A_0.f(A_1, \ldots, A_n)$ as alternative syntax to allow user-defined functions to be easily composed with other operations.

Examples of user-defined functions in Sections 4.2 and 4.3 are between, formalsOf, and

[4]For presentation purposes, we syntactically distinguish graph and policy functions; in the implementation using a policy function where a graph function is expected will result in an evaluation error not a parsing error.

returnsOf. For example, function entriesOf, which finds program-counter nodes representing method-entry program points in G for procedures matching *ProcedureName*, is defined as:

```
let entriesOf(G, ProcedureName) =
  G.forProcedure(ProcedureName).selectNodes(ENTRYPC)
```

User-defined functions are a powerful tool for building complex queries and policies. We have identified useful (non-primitive) operations and defined them as functions. In our query evaluation tool, these definitions are included by default, providing a rich library of useful query and policy functions, including between, formalsOf, returnsOf, entriesOf, declassifies, noExplicitFlows, and flowAccessControlled.

4.5 Implementation

Pidgin has two distinct components. The first component analyzes Java bytecode, including the JDK (up to 1.6) and library code, and produces PDGs. The second component evaluates queries against a PDG, and can be used either interactively or in "batch mode." Interactive mode displays results of queries in a variety of formats and is useful to explore the information flows in a program, for example to explore security guarantees in legacy programs or to find information flows that violate a given policy. The ability to interactively query a program to discover and describe information flows is a novel contribution of this work. Batch mode simply evaluates PidginQL queries and policies and is useful for checking that a program enforces a previously specified policy (e.g., as part of a nightly build process).

PDG construction Our implementation, which uses the WALA framework [13], is approximately 15,720 lines of code, not including the flow-insensitive points-to analyses

described in Chapter 2. A scalable points-to analysis is key to the scalability of PIDGIN. As we saw in Section 2.4, the flow-insensitive multi-threaded engine significantly outperforms WALA's points-to analysis. The remaining code implements the PDG construction, including various dataflow analyses to improve the precision of the PDG.

After the points-to analysis we run a non-null analysis, which determines which fields and variables may be null at a particular program point. The results of the non-null and points-to analyses are then used to determine the precise types of exceptions that can be thrown. The precise-exception results are used to improve control-flow analysis allowing the PDG to be computed using a more precise control-flow graph.

We construct a PDG for all code reachable from a specified `main` method via an interprocedural dataflow analysis. We use a (flow-insensitive) type-sensitive points-to analysis (a 2-type-sensitive analysis with a 1-type-sensitive heap [100]). We use additional precision for Java standard library container classes (3-type-sensitive with a 2-type-sensitive heap) and string builders (1-full-object-sensitive [76, 100]) to reduce false dependencies in these commonly used classes. This custom analysis is simple to implement using the abstraction described in Section 2.1.3.

The PDG construction analysis is context-sensitive, object-sensitive, field-sensitive, but flow-insensitive (we discuss flow sensitivity below) just like the points-to analysis we use. Context sensitivity and object sensitivity, in particular, is essential in order to create a PDG with enough precision to limit false failures of PIDGIN policies. Context sensitivity allows us to differentiate calls to the same method. If a particular method is called when processing sensitive and non-sensitive information, it is important to differentiate the two uses. For example, without context sensitivity every call to `add` in `java.util.HashSet` would result in a path to the same abstract heap location and every call to `get` would have a path from

that same location. This means that if sensitive information were stored in a `HashSet`, one of the most commonly used classes in Java applications, we would not be able to precisely reason about how it is handled upon retrieval. This simple example elucidates the need for a precise context-sensitive points-to analysis. The more precise the points-to analysis the less false paths arise in the PDG and the more likely a security guarantee that is provided by an application can be enforced using PIDGIN.

We handle all Java language features except reflection. The PDG captures all control and data dependencies, but not dependencies due to concurrent races. Because our analysis is flow-insensitive for heap locations, all reads of a given heap location depend on all writes to that location, which soundly approximates concurrent access to shared data.

As described in Section 2.3 we have the option in the points-to analysis to use a single abstract object to represent all instances of certain common types. For PIDGIN, in order to scale we use a single abstract object to represent all `java.lang.String`s. For increased precision in the PDG, we (soundly) treat methods on `String` objects and objects of immutable primitive wrapper classes (`java.lang.Integer`, etc.) as primitive operations by replacing method calls with edges describing their effects. This is key to PIDGIN's scalability and precision. Different `String`s contain different information, and must be distinguished to enforce realistic security policies. Treating `String`s like primitive values in the PDG provides sufficient precision while permitting a scalable points-to analysis. In addition, we provide analysis result signatures for some native methods. For native methods without signatures, we assume that the return values of the methods depend only on the arguments and the receiver, and that the methods have no heap side effects. These assumptions are potential sources of unsoundness in our analysis.

Flow-sensitivity Although Pidgin uses a flow-insensitive points-to analysis and constructs a flow-insensitive PDG, we saw in Section 3.4 that a flow-sensitive analysis can be significantly more precise than a flow-insensitive analysis. The PDG analysis in particular can benefit from flow sensitivity by ensuring that heap reads only depend on prior writes (e.g., reads do not depend on writes to the same location that occur later in the program).

To implement a flow-sensitive PDG analysis the procedure summary nodes described in Section 4.3.1 would be expanded to include summaries representing the procedure-entry value for heap locations that may be read by a procedure and the procedure-exit value for heap locations that may be written by a procedure. In contrast, a flow-insensitive PDG analysis can use a single PDG node for each heap location. This node will depend on values written to that location at any point in the program. This means that any read from that location will depend on all writes regardless of where they occur.

For the applications and policies described in Section 4.6, our PDG analysis is flow insensitive for heap locations, but achieve a form of flow sensitivity for local variables due to WALA's static single assignment representation [20].

PidginQL Query Engine We implemented a custom query engine for PidginQL that evaluates queries against PDGs. Although PidginQL could be implemented using an existing graph query language and engine (such as Cypher [82] or Gremlin [46]), we used a custom engine for flexibility and fast prototyping.

The query evaluator is 8,700 lines of Java code. It implements call-by-need semantics and caches subquery results. Call-by-need reduces the graph expressions that must be evaluated. Caching improves performance, particularly when used interactively, since parts of queries are often reused. When exploring information flows with Pidgin, a user typically

submits a sequence of similar queries.

4.6 Case Studies

In this section we present the results of applying Pidgin. For three legacy applications there was no predefined specification and we used Pidgin to explore the information flows and discover precise security policies that these applications satisfy. These were a web-based Course Management System (CMS); and two open-source applications, Free Chat-Server (FreeCS) and Universal Password Manager (UPM). For a fourth legacy application, the Apache Tomcat web server, we developed policies based on reported vulnerabilities and confirmed that the policies hold after patching, but fail on the unpatched version. We used our system to support simultaneous application and policy development for a small tax application we wrote ourselves, PTax. The diversity and specificity of these policies demonstrate the flexibility and expressivity of PidginQL.

In Section 4.6.7 we apply Pidgin to the SecuriBench Micro benchmark [70], and in Sections 4.7.1 and 4.7.2 we discuss using Pidgin both to explore security guarantees of a legacy application and to specify and enforce policies during development of an application.

4.6.1 Analysis performance

The first column of Table 4.1 presents the lines of code analyzed, i.e., lines reachable from the specified `main` method, including JDK 1.6 and library code. For each Tomcat vulnerability, we wrote a test harness that exercises the component(s) containing the vulnerability, and ran Pidgin on the harness. Pidgin thus analyzes all code reachable from the test harness, which does not include all Tomcat components. Table 4.1 shows results for

Table 4.1: Program sizes and analysis results

Program	Size (LoC)	Points-to Analysis				PDG Construction			
		Time (s)		Nodes	Edges	Time (s)		Nodes	Edges
		Avg	SD			Avg	SD		
CMS	161,597	2.0	0.2	333,741	2,557,316	13.1	0.1	1,812,263	3,540,271
FreeCS	102,842	0.8	0.1	135,489	545,853	6.0	0.1	742,860	1,407,576
UPM	333,896	5.7	1.1	637,348	17,255,214	24.8	0.1	3,544,271	7,016,244
Tomcat	160,432	6.6	0.2	508,227	11,474,544	14.3	0.1	1,973,632	4,048,266
PTax	65,165	1.1	0.2	92,527	2,088,865	2.8	0.2	280,633	539,253

Table 4.2: Policy evaluation times

Program	Policy	Time (s)		Policy LoC
		Mean	SD	
CMS	A1	5.5	0.06	3
	A2	3.9	0.06	5
FreeCS	B1	1.3	0.03	10
	B2	4.2	0.13	31
UPM	C1	11.7	0.12	7
	C2	13.3	0.13	12
Tomcat	D1	<0.1	<0.01	4
	D2	5.9	0.12	10
	D3	0.1	<0.01	3
	D4	5.9	0.03	4
PTax	E1	0.4	<0.01	4
	E2	1.3	0.01	14

only the largest harness.

Table 4.1 also presents the performance of the flow-insensitive points-to analysis and PDG construction analyses for each program, giving the mean and standard deviation (SD) of ten runs. For the programs and policies in this section the precision of the flow-insensitive points-to analysis was sufficient when combined with the custom context sensitivity described in Section 4.5. Analyses were performed on a 16 vCPU Amazon EC2 instance using Intel Xeon E5-2666 processors with 30GB of RAM.

Table 4.2 summarizes policy evaluation times for all policies discussed in this section, based on ten evaluations. Policy times are reported for a cold cache (i.e., with no previously cached results for subqueries). All policies evaluated in under 14 seconds. The last column in Table 4.2 gives the number of lines for each policy.

4.6.2 Course Management System (CMS)

CMS [10] is a J2EE web application for course management that has been used at Cornell University since 2005. We used a version of CMS that replaces the relational database backend with an in-memory object database. This version has previously been used to test performance of a distributed computing system [67]. CMS uses the model/view/controller design pattern. We examined the security of the model and controller logic; views simply display the final computed results.

Policy A1. *Only CMS administrators can send a message to all CMS users.*

This is a typical access control policy, ensuring that the function used to send messages to all users, is called only when the current user is an administrator.

```
let addNotice = pgm.entriesOf("addNotice") in
let isAdmin = pgm.returnsOf("isCMSAdmin") in
let isAdminTrue = pgm.findPCNodes(isAdmin,TRUE) in
pgm.accessControlled(isAdminTrue, addNotice)
```

Policy A2. *Only users with correct privileges can add students to a course.*

This five line policy is similar to Policy A1.

4.6.3 Free Chat-Server

Free Chat-Server is an open-source Java chat server that has been downloaded nearly 100,000 times.[5] Once the chat server has started, users can send messages, maintain friend

[5] http://sourceforge.net/projects/freecs/

lists, create, join and manage group chat sessions, etc. Administrators can ban, kick, and punish misbehaving users.

Policy B1. *Only superusers can send broadcast messages.*

We used PIDGIN to confirm that the ability to send messages to all users is available only to users with the right `ROLE_GOD`. This can be described with an access control policy similar to Policy A1. However, while exploring the information flows present in this program, we realized that our initial definition of what constituted a "broadcast message" was imprecise. PIDGIN enabled us to quickly find this apparent violation of the policy and refine our security policy appropriately.

Policy B2. *Punished users may perform limited actions.*

Misbehaving users can be disciplined by setting a `punished` flag in the object representing the user. In the PDG for Free Chat-Server, there are 357 sites where actions can be performed, all of which are invocations of the same method. We developed a PIDGINQL policy that precisely describes which actions a punished user may perform by using PIDGIN to interactively explore information flows, focusing on calls to the "perform action" method that were not access controlled by the `punished` flag. The final policy is the largest we have developed, 31 lines of PIDGINQL.

4.6.4 UNIVERSAL PASSWORD MANAGER (UPM)

UPM is an open-source password manager. Users store encrypted account and password information in the application's database and decrypt them by entering a single mas-

ter password. It has been downloaded over 90,000 times.[6]

Policy C1. *The user's master password entry does not explicitly flow to the GUI, console, or network except through trusted cryptographic operations.*

When we consider only the data dependencies in the program, the user's master password entry flows to public outputs only via the encryption and decryption operations in the trusted Bouncy Castle cryptography library.

Policy C2. *The user's master password entry does not influence the GUI, console, or network inappropriately.*

When we consider control dependencies, we find that the user's master password entry may influence public outputs, but only in appropriate ways (through trusted declassifiers). For example, an incorrect or invalid password triggers an error dialog box, and our policy accounts for this flow.

4.6.5 Apache Tomcat

Apache Tomcat[7] is a popular open source web server. Tomcat provides application developers with Java Servlet and Java Server Pages APIs, an HTTP server, and tools and management interfaces for server administrators. For several reported Tomcat vulnerabilities from the CVE database,[8] we developed PidginQL policies and confirmed that the policies fail to hold on vulnerable versions of Tomcat, and successfully hold on patched versions.

Note that the use of Pidgin on a test harness provides stronger guarantees than a simple test case. Most importantly, Pidgin can test information-flow properties (such as nonin-

[6] http://upm.sourceforge.net/

[7] http://tomcat.apache.org/

[8] http://cve.mitre.org/

terference) which are not testable by a single test case. In addition, a single PidginQL policy on a test harness provides guarantees on many possible executions. For the Tomcat test harnesses, we effectively test all possible parameters of server requests, because PidginQL policies and the PDG construction do not examine specific string values.

Policy D1. *CVE-2010-1157: The BASIC and DIGEST authentication HTTP headers do not leak the local host name or IP address of the machine running Tomcat.*

The PidginQL policy asserts that there are no paths from the sources of the host name and IP address to the authentication headers. This is a standard noninterference policy and ensures the completeness of the fix.

Policy D2. *CVE-2011-0013: Data from web applications are be properly sanitized before being displayed in the HTML Manager.*

It should not be possible for client web applications to run arbitrary scripts in the HTML Manager, a component for use by Tomcat administrators. This vulnerability arose because some data from client web applications was not properly sanitized. The PidginQL policy identifies the sanitization functions and asserts that all data from client applications that is displayed by the HTML manager passes through a sanitization function. Note that the policy does not ensure the proper *implementation* of the sanitization functions, but identifies them as trusted code that needs to be inspected.

Policy D3. *CVE-2011-2204: A user's password does not flow into an exception which gets written to the log file.*

The PidginQL policy is a noninterference policy asserting that the password does not influence the arguments to any exception method. This includes the creation of exceptions that leaked the password prior to the fix for CVE-2011-2204, but also ensures that there

were no similar leaks elsewhere in the code.

Policy D4. *CVE-2014-0033: Session IDs provided in the URL are be ignored when URL rewriting is disabled.*

The session ID from the request should not be used if URL rewriting is explicitly disabled. The PidginQL policy is a flow access controlled policy asserting that, if URL rewriting is disabled, then the session ID in the URL does not influence the session to which a request is associated.

4.6.6 PTax

PTax is a toy tax computation application that we wrote. PTax supports multiple users who login with a username and password and input their tax information. This sensitive information is stored in a file to be accessed later by the user, provided the user supplies the correct password. Before development, we defined a number of PidginQL policies we expected to hold. As development progressed, the policies were iteratively refined to reflect implementation choices (e.g., names of methods, signature of the authentication module), although the intent of the policies remained the same.

Policy E1. *Public outputs do not depend on a user's password, unless it has been cryptographically hashed.*

This can be expressed as the PidginQL policy:

```
let passwords = pgm.returnsOf("getPassword") in
let outputs = pgm.formalsOf("writeToStorage") ∪ pgm.formalsOf("print") in
let hashFormals = pgm.formalsOf("computeHash") in
pgm.declassifies(hashFormals, passwords, outputs)
```

This is a trusted-declassification policy. The declassifies function ensures that the only information flows from the user's password to public outputs are through the argument to

Table 4.3: SecuriBench Micro results

Test Group	Detected	False Positives
Aliasing	12/12	0
Arrays	9/9	5
Basic	63/63	0
Collections	14/14	5
Data Structures	5/5	0
Factories	3/3	0
Inter	16/16	0
Pred	5/5	2
Reflection	1/4	0
Sanitizers	3/4	0
Session	3/3	1
Strong Update	1/1	2
Total	159/163	15

the hash function.

Policy E2. *Tax information is encrypted before being written to disk and decrypted only when the user's password is entered correctly.*

Policy E2 is a combined declassification policy and access control policy, whose exact statement depends on the specification of the `userLogin` method, which allows for users to a limited number of guesses and returns the credentials upon successful login and `null` otherwise.

4.6.7 Micro-benchmark Results

To compare with other Java analysis tools, we evaluated Pidgin using the SecuriBench Micro [70] 1.08 suite of 123 small test cases that embody various types of vulnerabilities that can be used to test the effectiveness and breath of security tools. We present the results in Table 4.3. We developed Pidgin policies for each test and detect 159 out of a total of 163 vulnerabilities. We do not detect vulnerabilities due to reflection. We also miss an in-

correctly written sanitization function, though our policy marks it as a trusted declassifier, and thus indicates it should be inspected or otherwise verified.

For many tests the policy is a simply noninterference, requiring that sensitive values from an HTTP request do not affect public output. For some tests there is an allowed implicit flow, and we developed appropriate policies. Some tests require domain-specific policies (e.g., the Sanitizers tests required application-specific declassification policies).

False positives (i.e., cases where Pidgin mistakenly declares a program insecure) were caused by known limitations of our tool, including imprecise reasoning about individual array elements, dead code elimination that required arithmetic reasoning (Pred), and flow-insensitive tracking of heap locations (Strong Update).

4.7 Using Pidgin

Pidgin is a flexible tool. In this section we discuss two Pidgin use cases in more depth. We describe the exploration of information flows in a legacy application, detailing the policy discovery process for one of the policies discussed in Section 4.6. In addition, we describe using Pidgin throughout the development of a new application and how Pidgin can be used as a security regression testing tool during development.

4.7.1 Legacy applications

The interactivity of Pidgin was essential to understanding the security guarantees provided by legacy case-study programs and writing queries that describe these guarantees. As we did not write these programs, we first familiarized ourselves with the source code, and then attempted to develop queries that described security guarantees that the programs offered. We were occasionally surprised when a relatively simple policy failed. We would then inspect the paths that remained (often with the shortestPath operation), which helped us

to understand the information flows in the program and refine the query until we had a policy that the program satisfied.

In this section, we illustrate the interactive query and policy generation process by describing how we developed Policy C1 for Universal Password Manager (UPM). This policy states that: *The user's master password entry does not explicitly flow to the GUI, console, or network except through trusted cryptographic operations.*

Generate a program dependence graph Before querying, we first generated a program dependence graph for UPM. As discussed in Section 4.6, this took less than six seconds for the points-to analysis and 25 seconds to construct the PDG. We do this analysis once, serialize the PDG to disk, and use the same PDG for many queries.

Find sources and sinks UPM protects a user's passwords by encrypting them with a single master password. The application prompts the user for the master password, and then uses this to decrypt the database containing the user's passwords. If the master password is incorrect, the decryption will fail.

We decided to investigate confidentiality guarantees regarding the master password that is entered by the user. Inspecting the application code, we found that the master password is returned from the `askUserForPassword` method. The return values of this method are *sources*.

```
let sources = pgm.returnsOf("askUserForPassword") in ...
```

Note that we chose to regard the return values of this method as sensitive sources. In doing so, we are trusting the implementation of `askUserForPassword` to correctly handle data received from the input: a Java Swing widget. We could use Pidgin to learn more about

how the password is handled by this method, but `askUserForPassword` is 11 lines of code and uses standard Java Swing API calls to create a dialog box with a password field, so we inspected it by hand. We also trust the Swing library. This is a common way to use PIDGIN: to reduce trust in an entire application to trust in well-designed and well-maintained libraries, and a small amount of application code.

We identified three different places that data may leave the application: 1) the GUI (via the Swing API); 2) the console (via `java.io.PrintStream`); and 3) the network (via a custom `java.net.HTTPTransport` class). Formal arguments to methods in these three locations are *sinks*.

```
let sinks = pgm.formalsOf("javax.swing.*")
          ∪ pgm.formalsOf("sun.swing")
          ∪ pgm.formalsOf("PrintStream.print.*")
          ∪ pgm.formalsOf(".*HTTPTransport.*") in ...
```

TRY SIMPLE QUERIES We first checked if there are *any* paths between the sources and sinks with:

```
let ... // define sources and sinks
pgm.between(sources, sinks)
```

This results in a subgraph of about 3,000,000 nodes, more than three quarters of the original PDG. Unsurprisingly, most of the interesting work is done by the application after asking for a password and before presenting or sending results. (When graphs are too large, the PIDGINQL user interface automatically presents a textual summary of the graph, rather than a more data-rich presentation.)

We narrowed our focus to just data dependencies, and tried another simple query.

```
let ... // define sources and sinks
pgm.noExplicitFlows(sources, sinks)
```

Investigate counterexamples This policy failed, revealing that there are some paths via only data dependencies. We investigated the graph by excluding control dependencies with the standard library function:

```
let ... // define sources and sinks
let explicit(G) = G.removeEdges(pgm.selectEdges(CD)) in
pgm.explicit.between(sources, sinks)
```

The resulting graph is smaller, around 700,000 nodes, but still too large to inspect by hand. We wanted to find a strong policy explaining how the data dependencies allow information about the master password to leak from the application. To begin this process, we found a counterexample by using the shortestPath operation:

```
let ... // define sources and sinks
pgm.explicit.shortestPath(sources, sinks)
```

The resulting path converts the password to bytes and uses those bytes in a decryption function in the Bouncy Castle cryptography library to decrypt the password database. An entry from this database is then used to form an HTTP request. This is expected behavior. The password manager constructs these requests in order to allow users to easily login into websites using passwords stored in its encrypted database. Bouncy Castle[9] is one of the most widely used open source Java cryptography libraries and is clearly trusted by the UPM code. Therefore we can trust the password to not leak (except via cryptographic computations) once it enters the Bouncy Castle decryption function. Updating our query gives:

```
let ... // define sources and sinks
let decrypt = pgm.forProcedure("CBCBlockCipher.decryptBlock") in
pgm.explicit.removeNodes(decrypt).shortestPath(sources, sinks)
```

[9] https://www.bouncycastle.org/

```
let sources = pgm.returnsOf("askUserForPassword") in
let sinks = pgm.formalsOf("javax.swing*")
          ∪ pgm.formalsOf("sun.swing")
          ∪ pgm.formalsOf("PrintStream.print*")
          ∪ pgm.formalsOf("HTTPTransport*") in
let declassifiers =
  // Bouncy Castle encryption and decryption functions
  pgm.forProcedure("CBCBlockCipher.decryptBlock")
  ∪ pgm.forProcedure("AESEngine.encryptBlock")
  ∪ pgm.forProcedure("AESEngine.decryptBlock")
  // AES byte packing function
  ∪ pgm.forProcedure("AESEngine.packBlock") in
pgm.explicit.declassifies(declassifiers, sources, sinks)
```

Figure 4.5: PidginQL policy expressing Policy C2

This resulted in a very similar path into one of the Bouncy Castle *encryption* functions. Repeating the same process revealed paths through another decryption function and a byte-packing function only called within Bouncy Castle.

Create a PidginQL policy The final policy is shown in Figure 4.5. This policy was developed incrementally and interactively. Whereas the informal description of Policy C1 is vague, the PidginQL policy is a strong, precise, checkable policy that clarifies which data flows from the master password to public output are appropriate.

4.7.2 New development

Often new development begins with an incomplete and imprecise security specification that evolves as development progresses. Pidgin policies are flexible and, because they are not embedded in the program text, can be easily modified along with the informal security specification and the code itself. Pidgin policies can be used for regression testing to ensure that changes to the code do not cause policy violations.

We illustrate this process by describing the use of Pidgin throughout the development of a conference management system that we wrote, PChair. PChair is a toy conference management system that handles the submission, revision, and reviewing of papers. There are five user roles: author, reviewer, program committee member, program chair, and system administrator. A user may have multiple roles. We analyzed only the backend which maintains and controls access to the review and paper databases.

Access control policies in conference management systems can be intricate and complex. For example, several information leaks have been found and fixed in HotCRP [54]. In the end we developed fourteen separate Pidgin security policies for PChair. These policies either restrict access to sensitive data (author names, paper content, reviews, etc.) or ensure proper permissions for sensitive operations (e.g., accepting a paper or moving a deadline), along with a single trusted declassification policy (PC members can learn *whether* they have a conflict even if they cannot see the conflicting paper). This exposition focuses on Policy F1, seen below, which specifies when information about paper reviews may be revealed.

Policy F1. *A review can be viewed only by an authorized user.*

Define an informal policy. Before beginning development we wrote down the policies we desired informally. Policy F1 was initially: *Only authors of a paper, reviewers of a paper, and PC members can see a paper's reviews.*

Implement initial version of the application and Pidgin policy. We implemented PChair using role-based access control, as this closely mirrored our informal specification. We used simple functions to check whether the current user has a particular role, and then referred to these functions in our policies. Thus, our policies rely on the correct-

ness of these functions, which were deliberately designed to be simple and easy to verify.

For the initial version of Policy F1 we used syntactic sugar seen in Section 4.4 allowing us to find program-counter nodes for program points that are reached only when a particular boolean expression holds. Using this short hand, the initial version of the policy directly implements the informal specification.

```
... // output = errors or responses sent to the client
let isAuthorOf = pgm.returnsOf("isAuthorOf") in
let isPC = pgm.returnsOf("isPCMember") in
let isReviewer = pgm.returnsOf("isReviewer") in
let getReview = pgm.returnsOf("getReview") in
let check = pgm.[isAuthor || isPC || isReviewer] in
pgm.flowAccessControlled(check, getReview, output)
```

Recall that user-defined function flowAccessControlled is defined as follows:

```
let flowAccessControlled(G, checks, sources, sinks)  =
   G.removeControlDeps(checks).between(sources, sinks)
   is empty
```

We first gather the possible outputs, messages sent to the client. Then find the return values for the access checks and ensure that at least one of them is true on all flows from `getReview` to the client. The only way to access a review is by calling `getReview` (a simple Pidgin policy ensures that this is the case).

Update policies when the specification is modified. We iteratively added new features to PChair during development. As the functionality of the application evolved, the security policies also evolved. For example, we added a system administrator role. System administrators have superuser-like abilities, which required changes to many of our informal specifications and Pidgin policies: the informal specification for Policy F1 became: *Only authors of a paper, reviewers of a paper, PC members, and systems administrators can see a paper's reviews.* The access control check in the PidginQL policy added isAdmin as an

acceptable condition.

Because PIDGIN policies are not spread out throughout the code base (as, e.g., security-type annotations) updating the policies was straightforward, and accomplished easily.

REGRESSION TESTING SECURITY POLICIES. We used PIDGIN to check enforcement of security policies whenever new code was committed to our source repository. The commit would fail unless all security policy checks succeeded. Thus, as functionality evolved, the PIDGINQL policies were required to evolve with them.

This automated regression testing of security policies was useful several times. In one case, due to incorrect refactoring of a security-relevant piece of functionality (a missing negation), a security policy failed. Timely notification of the security policy failure allowed us to easily identify and fix the security violation. In another case, changing the name of a method in the code but not the security policy caused a security policy to fail with an evaluation error (when a returnsOf operation evaluated to an empty graph), requiring us to ensure that the security policy was up to date with respect to the code.

The final PIDGINQL policy for Policy F1 is shown below, and accounts for additional functionality added to the application, including notification deadlines and recording reviewer/paper conflicts.

```
... // output = errors or responses sent to the client
let isAuthorOf = pgm.returnsOf("isAuthorOf") in
let isPC = pgm.returnsOf("isPCMember") in
let isReviewerOf = pgm.returnsOf("isReviewerOf") in
let isAdmin = pgm.returnsOf("isAdmin") in
let notifyDeadlinePast = pgm.returnsOf("notifyDeadlinePassed") in
let reviewDeadlinePast = pgm.returnsOf("reviewDeadlinePassed") in
let hasConflict = pgm.returnsOf("hasConflict") in
let getReview = pgm.entriesOf("getReview") in
let check = pgm.[isAdmin ||
        (isAuthorOf && notifyDeadlinePast) ||
        (isPC && reviewDeadlinePast && !hasConflict ||
```

```
        isReviewerOf)] in
pgm.flowAccessControlled(check, getReview, output)
```

An interesting failure happened for another of our PChair policies, Policy F2, shown below.

Policy F2. *A paper's acceptance status can be released only to an author of the paper after the notification deadline, or to PC members without conflicts.*

The following PIDGIN policy ensures that all flows from return values of `isAccepted` to the client are protected by the correct access check.

```
... // output = errors or responses sent to the client
... // define deadline, role, and conflict checks
let isAccepted = pgm.returnsOf("isAccepted") in
let check = pgm.[(isAuthorOf && notifyDeadlinePast) || (isPC && !hasConflict)] in
pgm.flowAccessControlled(check, isAccepted, output)
```

During development, we discovered that this policy was not enforced. After the notification deadline, only accepted papers can be updated. If a user tries to update a rejected paper or update a paper before the deadline, an error message is displayed. However, which error message was displayed revealed information about whether or not the paper had been accepted. This implicit information flow leaked information about the paper's acceptance status. PIDGIN provided enough information to identify this subtle violation and find the bug, which was easily fixed by checking the notification deadline before checking acceptance status.

4.8 Related work

PDGs for security In a series of papers, Snelting and Hammer (and collaborators) argue for the use of PDGs for information-flow control, due to the precision and scalability of PDGs. They have developed JOANA [30], an object sensitive and context sensi-

tive tool for checking noninterference in Java bytecode [35], shown their techniques to be sound [115], and considered information flow in concurrent programs [102]. They also use path conditions to improve precision by ruling out impossible paths [108]. Hammer et al. [36] consider enforcement of a form of *where* declassification [94].

The key differences between our work and previous work using PDGs for information-flow control is that (1) our query language allows for expressive, application-specific policies that are separate from code, whereas JOANA requires program annotations and supports a limited class of policies; (2) we seek to use the PDG to enable *exploration* of security guarantees of programs in addition to *enforcement* of explicitly specified security guarantees; and (3) PIDGIN scales to larger programs. The largest reported use of JOANA is on a program with about 63,000 lines of code (excluding the JDK 1.4 library, which is approximately 100k lines of code total) for a scalability test where no security policy is specified. For this example JOANA is only able to generate a context-insensitive PDG and this takes about a day [33, 102].

Program dependence graphs were introduced by Ferrante et al. [29], along with an algorithm to produce them. PDGs were presented as an ideal data structure for certain intraprocedural optimizations. Program slicing for an interprocedural extension to PDGs is introduced by Horwitz et al. [47] and made more precise by Reps [90] using *CFL reachability*. Program slicing is useful for describing security guarantees and is built into PIDGINQL as primitive expressions forwardSlice and backwardSlice. Reps and Rosay [91] define *program chopping*, of which the PIDGINQL function between, defined in Section 4.2, is an example. Abadi et al. [1] develop a core calculus of dependency. Although they do not directly consider program dependence graphs, they show that program slicing and information flow type systems can be translated to this calculus. Cartwright and Felleisen [12]

give a denotational semantics to PDGs derived from the semantics of the original program. Bergeretti and Carré [8] use structures similar to PDGs to automatically find bugs in *while* programs and increase program understanding.

Yamaguchi et al. [118] use intraprocedural PDGs together with abstract syntax trees to detect vulnerabilities in C code. Vulnerabilities (e.g., buffer overflows) are identified using *graph traversals*, which are similar to some of our graph queries. Unlike Pidgin, the vulnerabilities their tool found were each contained within a single function, and their tool does not support whole program security policies. Furthermore, they consider only properties of a single program execution rather than application-specific information-flow properties such as those described in Sections 4.3 and 4.6. As common with bug-finding tools, their tool does not attempt to guarantee the absence of vulnerabilities even if none are found.

Kashyap and Hardekopf [51] use PDGs to infer *security signatures* describing how information flows within small (under 5k lines) JavaScript browser add-ons. These signatures can then by used by an auditor to decide whether an add-on should be accepted. Pidgin is similarly focused on increasing program understanding. Unlike our work, where policies can be application specific, they use a predefined set of sources and sinks. In addition to distinguishing control and data dependencies, their PDG edges contain annotations to indicate which edges may be more likely to carry relevant information. These additional annotations could also benefit Pidgin, for example to help prioritize potential policy violations to present to the user.

Legacy applications and policy inference Pidgin supports discovery of information security guarantees for legacy applications. Rocha et al. [92] present a framework that allows declassification policies to be specified for legacy applications. Policies are sep-

arate from code and enforcement of policies is checked using *expression graphs*, which, like PDGs, capture data and control dependencies. Policies are specified as graphs that describe which expression graphs can be declassified. Unlike the framework of Rocha et al., PIDGIN supports a rich class of policies and allows developers to *explore* the information flows in an application, and thus provides support for deciding what policy is appropriate for an application. By contrast, Rocha et al. only discuss declassification and do not consider how developers produce policies. Moreover, we have implemented our approach for Java bytecode; to the best of our knowledge, Rocha et al. do not implement their framework, nor consider how to extend to a full-fledged programming language.

Other work seeks to infer security policies for existing programs. Vaughan and Chong [113] use a data-flow analysis to infer expressive information security policies that describe what sensitive information may be revealed by a program. King et al. [53], Pottier and Conchon [86], Smith and Thober [101], and the Jif compiler [77, 78] all perform various forms of type inference for security-typed languages. Mastroeni and Banerjee [73] use refinement to derive a program's semantic declassification policy. We do not currently support automatic inference of security policies from a PDG. We instead provide the developer with tools and abstractions to help them explore the information flows in a program.

Several analyses infer explicit information flows (e.g., [68, 69, 71]). While efficient and practical, these analyses do not track implicit flows and may be inadequate in settings where strong information security is required. As described in Section 4.3, PIDGIN also supports exploration of explicit information flows, and policies for explicit information flows.

ENFORCEMENT OF EXPRESSIVE POLICIES Many tools and techniques seek to *enforce* expressive and strong information security policies. Security-type systems (e.g., [78, 98])

are the main technique used to enforce such policies. The survey by Sabelfeld and Myers [93] provides an overview of these security policies and enforcement techniques. More recently, Banerjee et al. [7] combine security-types with an expressive logic for describing a program's declassification policy and Nanevski et al. [80] use an expressive type-theoretic verification framework to specify and enforce rich information-flow properties. The security guarantees we consider in Section 4.3.2 are related to the security policies considered in these previous works. The absence of paths from sources to sinks corresponds to noninterference. Requiring all paths to go through certain nodes (such as the formal argument of a sanitization function) is a form of trusted declassification (e.g. [42, 72]). Reasoning about the conditions under which potentially dangerous information flows occur is similar to reasoning about *when* declassification is permitted [19, 94]. Restricting attention to only explicit information flows is equivalent to a static taint analysis (e.g., [3, 68, 69, 71, 111]).

5

Conclusion

We have developed a tool, Pidgin, that uses state-of-the-art static analysis techniques to help developers reason about the application-specific security guarantees their programs provide.

The precision, scalability, and utility of Pidgin and many other static analyses depends on a points-to analysis. We have designed and implemented a multithreaded points-to analysis that scales to large applications and makes it easy to specify the point in the precision/performance trade-off space required by client analyses.

One of the points-to analysis trade-offs that is most difficult to scale is flow sensitivity. We have developed a flow-sensitive analysis for object-oriented programs that is specifically designed to enable strong update in client analyses. We have shown that this analysis is scalable and produces better results than a flow-insensitive analysis. Since it enables strong update during object construction, it is particular useful for client analyses that benefit from precise reasoning about object invariants established during object construction.

Using the results of our flow-insensitive points-to analysis and several intermediate analyses, Pidgin generates a program dependence graph (PDG). Program dependence graphs precisely capture the information flows within programs.

Pidgin combines program dependence graphs with an expressive query language. Because individual paths within a PDG correspond to particular information flows within

a program, queries on PDGs offer a unified approach for the exploration, specification, and enforcement of security guarantees. By using the query language to describe paths in the PDG, developers can understand how information flows within a program and express precise, application-specific security guarantees including noninterference, trusted declassification, and access-controlled information flows.

PIDGIN is a practical tool. We have used PIDGIN to explore the information security of legacy applications, to specify and enforce information security during development, and to extract policies from known vulnerabilities. PIDGIN scales to Java applications with over 300k lines. Our case studies demonstrate that PIDGIN can express (and verify enforcement of) interesting application-specific security policies, some of which are difficult or impossible to express using existing tools and techniques.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[3] 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 *Proceedings of the ACM Conference on Program Language Design and Implementation*, 2014.

[4] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009.

[5] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[6] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, 2006.

[7] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[8] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 1985.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2006.

[10] Chavdar Botev, Hubert Chao, Theodore Chao, Yim Cheng, Raymond Doyle, Sergey Grankin, Jon Guarino, Saikat Guha, Pei-Chen Lee, Dan Perry, Christopher Re, Ilya Rifkin, Tingyan Yuan, Dora Abdullah, Kathy Carpenter, David Gries, Dexter Kozen, Andrew Myers, David Schwartz, and Jayavel Shanmugasundaram. Supporting workflow in a course management system. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 2005.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2009.

[12] Robert Cartwright and Mattias Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[13] IBM T. J. Watson Research Center. T. J. WAtson Library for Analysis (WALA). `http://wala.sf.net`, 2006-2015.

[14] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[15] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[16] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. *ACM SIGPLAN Notices*, 43(1):159–169, 2008.

[17] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the ACM Workshop on Secure Web Services*, 2009.

[18] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[19] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 1991.

[21] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[23] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[24] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, 2010.

[25] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.

[26] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Notices*, 1994.

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Usenix Conference on Operating Systems Design and Implementation*, 2010.

[28] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 1998.

[29] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[30] Dennis Giffhorn and Christian Hammer. Precise analysis of Java programs using JOANA (tool demonstration). In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[31] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[32] Deepak Goyal. Transformational derivation of an improved alias analysis algorithm. In *Automatic Program Development*. Springer Netherlands, 2008.

[33] Jürgen Graf. Speeding up context-, object- and field-sensitive SDG generation. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[34] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[35] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 2009.

[36] Christian Hammer, Jens Krinke, and Frank Nodes. Intransitive noninterference in dependence graphs. In *Proceedings of the 2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, 2006.

[37] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[38] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[39] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[40] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011.

[41] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[42] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006.

[43] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[44] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, 1998.

[45] Michael Hind and Anthony Pioli. Which pointer analysis should I use? *SIGSOFT Software Engineering Notes*, 2000.

[46] Florian Holzschuher and René Peinl. Performance of graph query languages: Comparison of Cypher, Gremlin and native access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013.

[47] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, 1988.

[48] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your IFCException are belong to us. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[49] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[50] Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[51] Vineeth Kashyap and Ben Hardekopf. Security signature inference for JavaScript-based browser addons. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[52] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[53] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the International Conference on Information Systems Security*, 2008.

[54] Eddie Kohler. Hot Crap! In *Proceedings Conference on Organizing Workshops, Conferences, and Symposia for Computer Systems*, 2008.

[55] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1992.

[56] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.

[57] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Annual Asian Computing Science Conference*, 2006.

[58] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[59] Thomas Lenherr. *Taxonomy and applications of alias analysis*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2008.

[60] Ondrej Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 2006.

[61] Ondrej Lhoták. Post on the Soot mailing list. `https://mailman.cs.mcgill.ca/pipermail/soot-list/2012-March/004154.html`, 2012.

[62] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[63] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[64] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 2008.

[65] Du Li. Dynamic tainting for deployed Java programs. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming, Systems, Languages, and Applications*, 2010.

[66] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[67] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS Symposium on Operating systems principles*, 2009.

[68] Yin Liu and Ana Milanova. Static analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008.

[69] Yin Liu and Ana Milanova. Practical static analysis for inference of security-related program properties. In *Proceedings of the IEEE 17th International Conference on Program Comprehension*, 2009.

[70] Benjamin Livshits. Securibench Micro. `http://suif.stanford.edu/~livshits/work/securibench-micro/`, 2006.

[71] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[72] Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, 2004.

[73] Isabella Mastroeni and Anindya Banerjee. Modelling declassification policies using abstract domain completeness. *Mathematical Structures in Computer Science*, 2011.

[74] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[75] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[76] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 2005.

[77] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, 1999.

[78] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, Nathaniel Nystrom, Danfeng Zhang, Owen Arden, Jed Liu, and K. Vikram. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001-2015.

[79] Vaivaswatha Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[80] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 2013.

[81] Rupesh Nasre. Time- and space-efficient flow-sensitive points-to analysis. *ACM Transactions on Architecture and Code Optimization*, 2013.

[82] neo4j. Intro to Cypher. `http://neo4j.com/developer/cypher-query-language/`, 2015.

[83] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Proceedings of the 12th international conference on Compiler construction*, 2003.

[84] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Control*, 2004.

[85] David J Pearce, Paul HJ Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 2007.

[86] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, 2000.

[87] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[88] Sandeep Putta and Rupesh Nasre. Parallel replication-based points-to analysis. In *Proceedings of the 21st international conference on Compiler Construction*, 2012.

[89] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 1994.

[90] Thomas Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, 1997.

[91] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, 1995.

[92] B.P.S. Rocha, S. Bandhakavi, J. den Hartog, W.H. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[93] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[94] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, 2005.

[95] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

[96] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Application*. Prentice-Hall, 1981.

[97] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[98] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003.

[99] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2015.

[100] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011.

[101] Scott F. Smith and Mark Thober. Improving usability of information flow security in Java. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2007.

[102] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 2015.

[103] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[104] Stefan Staiger-Stöhr. Implementing sparse flow-sensitive andersen analysis. Technical report, Universität Stuttgart, 2009.

[105] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, 1996.

[106] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, 2011.

[107] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, 2011.

[108] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In *Proceedings of the International Workshop on Formal Aspects of Security and Trust*, 2010.

[109] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, 2006.

[110] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[111] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. ANDROMEDA: accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*, 2013.

[112] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[113] Jeffrey A. Vaughan and Stephen Chong. Inference of expressive declassification policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[114] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 1996.

[115] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009.

[116] John Whaley. *Context-sensitive pointer analysis using binary decision diagrams*. PhD thesis, Stanford University, 2007.

[117] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, 2000.

[118] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[119] Tuba Yavuz-Kahveci and Tevfik Bultan. Automated verification of concurrent linked lists with counters. In *Proceedings of the 9th International Symposium on Static Analysis*, 2002.

[120] Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the 21th International Static Analysis Symposium*, 2014.

[121] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[122] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM Operating Systems Review*, 2011.