Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs

MINSEOK JEON and HAKJOO OH*, Korea University, Republic of Korea

In this paper, we challenge the commonly-accepted wisdom in static analysis that object sensitivity is superior to call-site sensitivity for object-oriented programs. In static analysis of object-oriented programs, object sensitivity has been established as the dominant flavor of context sensitivity thanks to its outstanding precision. On the other hand, call-site sensitivity has been regarded as unsuitable and its use in practice has been constantly discouraged for object-oriented programs. In this paper, however, we claim that call-site sensitivity is generally a superior context abstraction because it is practically possible to transform object sensitivity into more precise call-site sensitivity. Our key insight is that the previously known superiority of object sensitivity holds only in the traditional k-limited setting, where the analysis is enforced to keep the most recent k context elements. However, it no longer holds in a recently-proposed, more general setting with context tunneling. With context tunneling, where the analysis is free to choose an arbitrary k-length subsequence of context strings, we show that call-site sensitivity can simulate object sensitivity almost completely, but not vice versa. To support the claim, we present a technique, called Obj2Cfa, for transforming arbitrary context-tunneled object sensitivity into more precise, context-tunneled call-site-sensitivity. We implemented Obj2Cfa in Doop and used it to derive a new call-site-sensitive analysis from a state-of-the-art object-sensitive pointer analysis. Experimental results confirm that the resulting call-site sensitivity outperforms object sensitivity in precision and scalability for real-world Java programs. Remarkably, our results show that even 1-call-site sensitivity can be more precise than the conventional 3-object-sensitive analysis.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Machine learning for program analysis, Pointer analysis, Context sensitivity

ACM Reference Format:
Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. *Proc. ACM Program. Lang.* 6, POPL, Article 58 (January 2022), 29 pages.
https://doi.org/10.1145/3498720

1 INTRODUCTION

> *"Since its introduction, object sensitivity has emerged as the dominant flavor of context sensitivity for object-oriented languages."*
>
> —Smaragdakis and Balatsouras [2015]

Context sensitivity is critically important for static program analysis of object-oriented programs. A context-sensitive analysis associates local variables and heap objects with context information of method calls, computing analysis results separately for different contexts. This way, context sensitivity prevents analysis information from being merged along different call chains. For object-oriented and higher-order languages, it is well-known that context sensitivity is the primary means

*Corresponding author

Authors' address: Minseok Jeon, minseok_jeon@korea.ac.kr; Hakjoo Oh, hakjoo_oh@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/1-ART58
https://doi.org/10.1145/3498720

for increasing analysis precision without blowing up analysis cost [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Li et al. 2018a; Smaragdakis and Balatsouras 2015; Smaragdakis et al. 2014; Sridharan and Bodík 2006; Thiessen and Lhoták 2017].

There have been two major flavors of context sensitivity, namely *call-site sensitivity* [Sharir and Pnueli 1981; Shivers 1988] and *object sensitivity* [Milanova et al. 2002, 2005], which differ in the choice of context information. The traditional k-call-site-sensitive analysis [Sharir and Pnueli 1981] uses a sequence of k call-sites as the context of a method. By contrast, object sensitivity uses allocation-sites as context elements: in a virtual call, e.g., `a.foo()`, an object-sensitive analysis uses the allocation-site of the receiver object (`a`) as the context of `foo`. The standard k-object-sensitive analysis [Milanova et al. 2002, 2005; Smaragdakis et al. 2011] maintains a sequence of k allocation-sites, comprising the allocation-site of the receiver object, the allocation-site of the receiver's allocator, and so on.

The Status Quo. Since its inception, object sensitivity has been established as the dominant context abstraction for object-oriented languages [Smaragdakis and Balatsouras 2015]. Ever since Milanova et al. [2002, 2005] proposed object sensitivity, its superiority over other flavors of context sensitivity has been reinforced by a large amount of research [Bravenboer and Smaragdakis 2009; Jeong et al. 2017; Lhoták and Hendren 2008; Lu and Xue 2019; Smaragdakis et al. 2011; Tan et al. 2016]. Among others, Lhoták and Hendren [2006] and Bravenboer and Smaragdakis [2009] conducted extensive experiments to conclude that object sensitivity significantly outperforms other alternatives including call-site sensitivity. As a result, object sensitivity has become an indispensable component of program analysis tools for object-oriented languages [Feng et al. 2014; Fink et al. 2008; Gordon et al. 2015; Naik et al. 2006; Xu et al. 2019; Zhang et al. 2014].

In contrast, the use of call-site sensitivity has been constantly discouraged for object-oriented programs [Jeong et al. 2017; Lhoták and Hendren 2006; Li et al. 2018a; Milanova et al. 2002, 2005; Smaragdakis et al. 2011, 2014; Tan et al. 2016]. For example, Milanova et al. [2002, 2005] judged call-site sensitivity as "ill-suited" for object-oriented programs, Kastrinis and Smaragdakis [2013] claimed that call-site sensitivity should be avoided because it is both imprecise and expensive, and Smaragdakis et al. [2014] asserted call-site sensitivity is never cost-effective. As a result, call-site sensitivity has become obsolete in practice and virtually not used anymore in program analysis tools for object-oriented programs: *"... object-sensitive analyses have almost completely supplanted traditional call-site-sensitive analyses for object-oriented languages"* [Smaragdakis et al. 2011].

This Work. We challenge this commonly-accepted wisdom by showing that call-site sensitivity is generally superior to object sensitivity even for object-oriented programs. Our key insight is that the previously established superiority of object sensitivity over call-site sensitivity is valid only when we impose a particular restriction that the analysis should keep the *most recent* k context elements, but it no longer holds in a more general setting where the restriction is eliminated. Notably, the relative superiority of object sensitivity and call-site sensitivity gets inverted when they are generalized with context tunneling [Jeon et al. 2018], where the analysis is free to use an arbitrary k-length subsequence of context strings. In this generalized setting, we show that call-site sensitivity is able to *simulate* object sensitivity, but object sensitivity is not powerful enough to simulate call-site sensitivity. We note that our aim is not to debunk the previously known result. Instead, we claim that what is currently known only persists in a limited circumstance and the converse holds when the assumption is generalized.

To support the claim, we present Obj2Cfa, a practical technique for transforming object sensitivity into more precise, context-tunneled call-site sensitivity. Our technique takes as input an arbitrary object-sensitive analysis with context tunneling and produces as output a context-tunneling policy that enables call-site sensitivity to exceed the precision limit of the baseline object sensitivity

without increasing k. Our key technical contributions to achieve this goal are the simulation and simulation-guided learning procedures. By the simulation procedure, we infer a context-tunneling policy with which call-site sensitivity can simulate the baseline object sensitivity. The resulting call-site sensitivity, however, is impractical since it requires running the baseline object-sensitive analysis as a pre-analysis. The learning phase aims to remove this burden by capturing the behavior of the simulated policy using a dataset of programs.

We implemented our technique in Doop [Bravenboer and Smaragdakis 2009], a popular pointer analysis framework for Java. We transformed a state-of-the-art object-sensitive pointer analysis into the matching call-site-sensitive analysis. Evaluation with real-world Java applications shows that the resulting call-site-sensitive analysis significantly improves the original object-sensitive analysis in terms of both precision and scalability. Remarkably, our context-tunneled 1-call-site-sensitive analysis is even more precise than the traditional 3-object-sensitive analysis with much smaller costs, which confirms our claim that call-site sensitivity can be superior to object sensitivity in the generalized setting.

Contributions. We summarize our contributions below.

- We make a novel claim that call-site sensitivity is generally superior to object sensitivity; when the notion of k-limiting is generalized with context tunneling, call-site sensitivity can simulate object sensitivity almost completely, but not vice versa.
- We present Obj2Cfa, a new technique for transforming a context-tunneled k-object-sensitive analysis into a more precise, context-tunneled k-call-site-sensitive analysis. Specifically, we make two technical contributions: the simulation (Section 4.1) and simulation-guided learning (Section 4.2) procedures, both of which are vital to achieving the goal.
- We experimentally prove our claim by applying Obj2Cfa to a state-of-the-art object-sensitive pointer analysis for Java. Our implementation and data are publicly available [1].

2 OUR CLAIM

In this section, we illustrate the main message of this paper with examples.

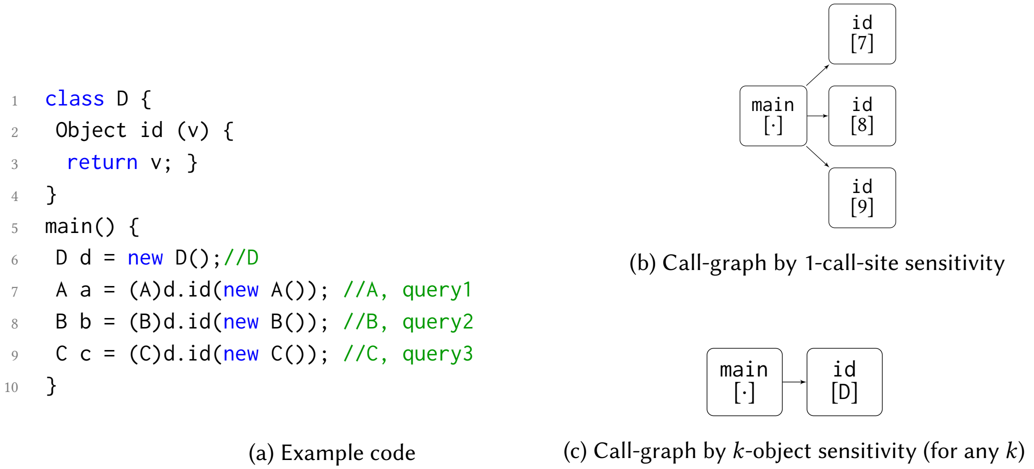
2.1 The Previously Known Superiority

First of all, we note that traditional call-site sensitivity and object sensitivity can complement each other [Liang et al. 2005].

Benefit of Call-Site Sensitivity. Figure 1 describes a typical situation where call-site sensitivity has better precision than object sensitivity. The example program has class `D` that includes the identity function `id`. The `main` method allocates an object of class `D` at line 6 and calls method `id` on it in three places at lines 7, 8, and 9 with different objects of type `A`, `B`, and `C`, respectively. Suppose pointer analysis aims to prove that the three type-casting operations at lines 7, 8, and 9 are safe. Figure 1b shows the call-graph from 1-call-site-sensitive analysis. Note that the analysis analyzes the method `id` separately for the different call-sites at lines 7, 8, and 9, and therefore is able to prove the safety of the queries.

By contrast, k-object sensitivity is unable to prove any of the queries in the program no matter what k value is used. Object sensitivity uses allocation-sites of receiver objects as calling contexts. In this example, because the three method calls share the same receiver object (i.e. the object pointed to by variable `d`), object sensitivity analyzes the method `id` with a single context element, namely the allocation-site `D`, merging the three method calls (Figure 1c).

[1] https://github.com/kupl/OBJ2CFA



(a) Example code

(b) Call-graph by 1-call-site sensitivity

(c) Call-graph by k-object sensitivity (for any k)

Fig. 1. Typical situation that benefits from call-site sensitivity

Benefit of Object Sensitivity. Figure 2 describes a representative scenario where object sensitivity is more precise than call-site sensitivity. The example code in Figure 2a has class `C` that contains $k+1$ methods ($\texttt{id}_0, \texttt{id}_1, \ldots \texttt{id}_k$), where each method $\texttt{id}_i$ is semantically equivalent to the identity function because $\texttt{id}_0$ is the identity function and $\texttt{id}_i$ ($0 < i \leq k$) calls $\texttt{id}_{i-1}$ without modifying the formal parameter `v`. The `main` method has four heap allocation-sites: namely, `C1`, `C2`, `A`, and `B`. At line 13, `main` calls $\texttt{id}_k$ with the base variable `c1` and parameter `new A()`. At line 14, $\texttt{id}_k$ is called with the base variable `c2` and argument `new B()`. Again, the goal of pointer analysis is to prove the safety of the casting operations at lines 13 and 14. For this program, a k-call-site-sensitive analysis produces the call-graph in Figure 2b. Note that the method $\texttt{id}_0$ is analyzed under the single context $[8, \ldots, 5]$, where the critical information where $\texttt{id}_k$ was originally called from is lost due to the truncation of context strings to keep their last k elements.

Object sensitivity nicely addresses this shortcoming of call-site sensitivity. It uses the allocation-sites, `C1` and `C2`, to represent the contexts of the method calls to $\texttt{id}_k$ at lines 13 and 14, respectively. Note that the receiver object remains the same in the subsequent calls to $\texttt{id}_{k-1}, \ldots \texttt{id}_0$, propagating the initial contexts down to $\texttt{id}_0$ and producing the call-graph in Figure 2c. The analysis is able to distinguish the two call chains and therefore proves the queries.

Known Superiority. Though call-site sensitivity and object sensitivity have their own strengths and weaknesses, object sensitivity is widely known to be superior to call-site sensitivity because real-world programs involve code patterns such as one in Figure 2 more often. Note that the existing superiority holds empirically, rather than theoretically, based on the experimental results of prior work (e.g., [Bravenboer and Smaragdakis 2009; Lhoták and Hendren 2008]).

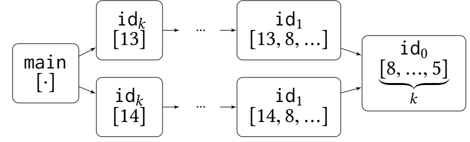
2.2 Revisiting the Superiority in Generalized k-Limited Setting

Our new claim is that the known superiority of object sensitivity over call-site sensitivity no longer holds when the notion of k-limiting is generalized with context tunneling.

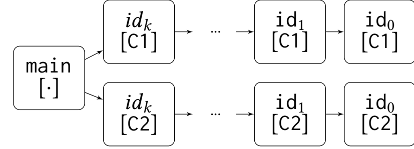
Context Tunneling. Context tunneling [Jeon et al. 2018] allows an analysis to maintain an arbitrary k-length subsequence of context strings. For example, when $s = [C_1, C_2, C_3, C_4, C_5]$ is a sequence of context elements that may appear in an unbounded ($k = \infty$) context-sensitive analysis, the traditional 3-limited analysis abstracts the context string into its suffix $[C_3, C_4, C_5]$. With context

```
1  class C {
2   Object id0(v) {
3    return v; }
4   Object id1(v) {
5    return this.id0(v); }
6    ...
7   Object idk(v) {
8    return this.idk-1(v); }
9  }
10 main() {
11  C c1 = new C();//C1
12  C c2 = new C();//C2
13  A a = (A)c1.idk(new A()); //A, query1
14  B b = (B)c2.idk(new B()); //B, query2
15 }
```

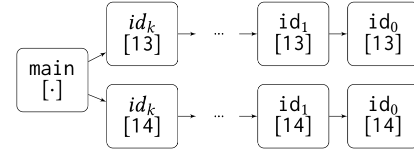
(a) Example code



(b) Call-graph by k-call-site sensitivity



(c) Call-graph by 1-object sensitivity



(d) Call-graph by 1-call-site sensitivity with tunneling

Fig. 2. Typical situation that benefits from object sensitivity

tunneling, however, the analysis is free to use any subsequence of s such as $[C_1, C_3, C_5]$ and $[C_2, C_4, C_5]$, as a k-limited abstraction of the original context string. Note that the traditional k-limited approach is a special case of the generalized approach with context tunneling.

Key Insight. Our key insight is summarized as follows:

- The major weakness of call-site sensitivity in the traditional setting is no longer a weakness in the generalized k-limited setting with context tunneling.
- By contrast, object sensitivity still suffers from its limitation even with the generalization.

With context tunneling, call-site sensitivity does not suffer from its shortcoming and can now prove the queries in Figure 2a. Suppose that we use a context-tunneling policy that chooses the first k elements of a context string rather than the last k ones. Then, the resulting 1-call-site-sensitive analysis produces the call-graph in Figure 2d, which is exactly the same as the call-graph of the 1-object-sensitive analysis in Figure 2c. Because the call-graphs are equivalent, the call-site-sensitive analysis is as precise as the object-sensitive analysis, successfully proving the queries.

On the other hand, object sensitivity fails to simulate call-site sensitivity even with context tunneling. Consider the program in Figure 1 where call-site sensitivity is typically more precise than object sensitivity. For this program, object sensitivity cannot prove all of the queries no matter what context-tunneling policy and k value are used. In Figure 1, object sensitivity can use the allocation-site `D` as context elements. Thus, only the two context subsequences, i.e., `[D]` and `[·]`, are possible for the method `id` with context tunneling, all of which fail to analyze `id` separately for the three different call-sites.

We clarify that, like the previously known superiority, our claim in this paper is empirical. On the theoretical side, we do not know yet whether or not call-site sensitivity can always simulate object sensitivity in the general setting with context tunneling, which we leave as an open question for future work. We discuss this issue in more detail in Section 7.

Obj2Cfa. Based on the insight, we develop Obj2Cfa, a practical technique for transforming a given k-object-sensitive analysis into a more precise, context-tunneled k-call-site-sensitive analysis (Section 4). The resulting analysis is (empirically) more precise than the baseline object sensitivity, as it enjoys the benefits of both object sensitivity and call-site sensitivity. For example, it produces precise results for both cases in Figure 1 and 2.

3 SETUP: POINTER ANALYSIS WITH CONTEXT TUNNELING

In this section, we define a pointer analysis with context tunneling.

Program. We assume a program is a sequence of instructions, where each instruction is associated with a distinct label. An instruction is either heap allocation ($x = new()$), move ($x = y$), store ($x.f = y$), load ($x = y.f$), or virtual method call ($x = y.m_r^p(a)$). We assume that every method (m) has a single formal parameter (p) and return variable (r). Given a program P to analyze, we assume the following:

- Var_P: the set of program variables.
- fld_P: the set of field signatures.
- $Label_P$: the set of instruction labels of the program.
- $Mthd_P$: the set of methods of the program.
- $Mthdof_P$: the mapping from labels to the methods containing them (i.e. $Label_P \to Mthd_P$).
- $Invo_P$: invocation sites (i.e. call sites, $Invo_P \subseteq Label_P$).
- $Heap_P$: heap allocation sites (i.e. $Heap_P \subseteq Label_P$).
- Ctx_P: the set of calling contexts (i.e. $Ctx_P = Label_P^*$).
- $Hctx_P$: the set of heap contexts (i.e. $Hctx_P = Label_P^*$).
- $main_P$: the entry method of the program.

Notation. For function $X : A \to B$, where A and B are sets, we write $X[a \mapsto b]$ (where $a \in A, b \in B$) for the function X that is extended to map a to b. For function $X : A \to \wp(B)$, $X[a \overset{w}{\mapsto} b]$ (where $a \in A, b \subseteq B$) denotes $X[a \mapsto X(a) \cup b]$ (i.e. weak update). Given $X, Y : A \to \wp(B)$, $X \sqcup Y$ denotes $\lambda a.X(a) \cup Y(a)$. Given a sequence $s = a_1a_2 \dots a_{n-1}$ and an element a_n, we write $\lceil s +\!\!+ a_n \rceil_k$ for $a_1a_2 \dots a_{n-1}a_n$ if $n < k$. If $n \geq k$, the result is $a_{n-k+1} \dots a_{n-1}a_n$.

Pointer Analysis. We consider a subset-based pointer analysis with on-the-fly call-graph construction [Smaragdakis and Balatsouras 2015], which computes four pieces of information: points-to, field points-to, reachability, and call-graph information. The points-to information, $X \in Var_P \times Ctx_P \to \wp(Heap_P \times Hctx_P)$, maps variables with contexts to sets of heaps with heap contexts. The field points-to information, $Y \in Heap_P \times Hctx_P \times fld_P \to \wp(Heap_P \times Hctx_P)$, maps fields of heaps with their heap contexts to sets of heaps with heap contexts. The reachability information, $R \in Mthd \to \wp(Ctx)$, maps methods to their reachable contexts. A call-graph, $G \in \wp(Invo_P \times Ctx_P \times Mthd_P \times Ctx_P)$, is a set of context-sensitive call edges, where $(l, ctx_1, m, ctx_2) \in G$ indicates that method m is called from the invocation-site l and the caller and callee contexts are ctx_1 and ctx_2, respectively. The analysis is flow-insensitive and aims to compute the least fixed point of the semantic function F defined as follows:

$$F_{P,k}^{T,U} = \lambda(X, Y, R, G). \bigsqcup_{l \in Label_P} f_{l,k}^{T,U}(X, Y, R, G)$$

where $k \in \mathbb{N}$ is the maximum context depth to distinguish and $f_{l,k}^{T,U}$ is the transfer function for the instruction whose label is l. T and U are context-tunneling abstraction and context-update function, respectively, which will be explained shortly. Running the analysis is to compute $fixF$:

$$fixF_{P,k}^{T,U} = F_{P,k}^{T,U}(\bot_X, \bot_Y, \bot_R, \bot_G) \sqcup F_{P,k}^{T,U}(F_{P,k}^{T,U}(\bot_X, \bot_Y, \bot_R, \bot_G)) \sqcup \dots$$

where the bottom elements are defined as follows:

$$\bot_X = \lambda(x, c).\emptyset, \qquad \bot_Y = \lambda(l, hc, f).\emptyset, \qquad \bot_R = \lambda m. \begin{cases} \{\epsilon\} & \text{if } m = main_P \\ \emptyset & \text{otherwise} \end{cases}, \qquad \bot_G = \emptyset.$$

Transfer Function. The transfer function for the allocation, move, store, and load instructions is standard. Let (X', Y', R', G') be $f^{T,U}_{l,k}(X, Y, R, G)$. When the command is allocation ($x = new()$), it extends the points-to map so that the variable x points-to the heap l under the reachable context $ctx \in R(Mthdof(l))$:

$$X' = \bigsqcup_{ctx \in R(Mthdof(l))} X[(x, ctx) \overset{w}{\mapsto} \{(l, ctx)\}].$$

For a store command ($x.f = y$), the field points-to information is updated as follows:

$$Y' = \bigsqcup_{ctx \in R(Mthdof(l))} Y[(l, hctx, f) \overset{w}{\mapsto} \{(l', hctx')\}]$$

where $(l, hctx) \in X(x, ctx)$ and $(l', hctx') \in X(y, ctx)$. The points-to information is updated as follows for a load ($x = y.f$) instruction:

$$X' = \bigsqcup_{ctx \in R(Mthdof(l))} X[(x, ctx) \overset{w}{\mapsto} \{(l, hctx)\}]$$

where $(l, hctx) \in Y(l', hctx', f)$ and $(l', hctx') \in X(y, ctx)$. The reachability map and call-graph remain the same for the above instructions (i.e. $R' = R$ and $G' = G$). The transfer function for move instruction ($x = y$) combines the points-to set of y into that of x without modifying R and G.

The transfer function for method calls is less standard as it should account for context tunneling. To support context tunneling [Jeon et al. 2018], we assume a context-tunneling space $\mathbb{S}$ is given. The space $\mathbb{S}$ can be defined in various ways and the choice does not affect the soundness of the analysis. In this paper, we simply define the space to be the set of all invocation sites, i.e., $\mathbb{S} = Invo_P$ and let $T \subseteq \mathbb{S}$ be a tunneling abstraction given before the analysis. For method call $x = y.m^p_r(a)$, the transfer function, $f^{T,U}_{l,k}$, first generates the callee's context ctx' using the context-update function U, $ctx' = U(ctx, T, X, l, y, k)$, which takes information available at the call-site. The output ctx' of U will be shortly defined for each context sensitivity flavor. Once ctx' is computed, the analysis makes the formal parameter p under ctx' have the points-to set of the actual parameter a under ctx and the points-to set of the return variable r is transferred to the variable x. The resulting X' is defined as follows:

$$\bigsqcup_{ctx \in R(Mthdof(l))} X[(p, ctx') \overset{w}{\mapsto} X(a, ctx), (x, ctx) \overset{w}{\mapsto} X(r, ctx')]$$

and the reachability and call-graph are updated accordingly: $R' = R[m \overset{w}{\mapsto} \{ctx'\}]$ and $G' = G \cup \{(l, ctx, m, ctx')\}$.

Context Update. Let us define the context-update function U. For object sensitivity, it is defined as follows:

$$U(ctx, T, X, l, y, k) = \begin{cases} \lceil hctx +\!\!+ h \rceil_k & l \notin T, \\ hctx & l \in T \end{cases} \tag{1}$$

where $(h, hctx) \in X(y, ctx)$. When a method is called from an invocation site l with the base variable y and caller's context ctx, the analysis first looks at the heap h (and its context $hctx$) that the variable y under ctx points to, and creates the callee's context by appending the heap (h) to its heap context ($hctx$). The context may be truncated to keep the last k elements at most (i.e. $\lceil hctx +\!\!+ h \rceil_k$). Note that U creates the new context only when $l \notin T$ (i.e. no context tunneling). Otherwise ($l \in T$), it applies context tunneling and propagates the existing context ($hctx$).

```
1  class D {
2   Object id(v) {
3    return v;}
4   Object id1(v) {
5    return this.id(v);}
6   void m() {
7    A a = (A)this.id(new A());//A1, query1
8    B b = (B)this.id(new B());//B1, query2
9   }
10 }
```

```
11 void main() {
12  D d1 = new D();//D1
13  D d2 = new D();//D2
14  D d3 = new D();//D3
15
16  A a3 = (A)d1.id1(new A());//A2, query3
17  B b3 = (B)d2.id1(new B());//B2, query4
18  d3.m();
19 }
```

Fig. 3. Running example

For call-site sensitivity, U is defined as follows:

$$U(ctx, T, X, l, y, k) = \begin{cases} \lceil ctx ++ l \rceil_k & l \notin T, \\ ctx & l \in T \end{cases} \tag{2}$$

When $l \notin T$, the analysis appends the current invocation site l to ctx. With context tunneling ($l \in T$), it uses the caller's context ctx for the callee's.

Instance Analyses. Given a program P and its tunneling abstraction $T \subseteq Invo_P$, we write $\text{call}_k(P, T)$ and $\text{obj}_k(P, T)$ for the k-call-site- and k-object-sensitive analyses, respectively. In the rest of the paper, we fix k and omit the subscript k from call_k and obj_k. These instance analyses are used with a *context-tunneling policy*. A context-tunneling policy $\mathcal{T}$ is a function that maps a program P into a tunneling abstraction for P:

$$\mathcal{T}(P) \subseteq Invo_P.$$

With a policy $\mathcal{T}$, we perform the analysis for a program P as follows: $\text{call}(P, \mathcal{T}(P))$ or $\text{obj}(P, \mathcal{T}(P))$.

4 OBJ2CFA: TRANSFORMING OBJECT SENSITIVITY TO CALL-SITE SENSITIVITY

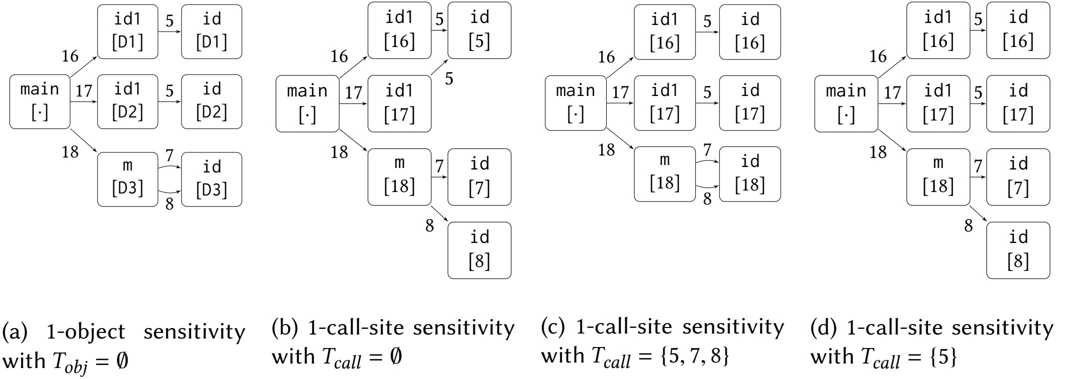
We now present our technique, Obj2Cfa. Given an object-sensitive analysis specified by an arbitrary tunneling policy $\mathcal{T}_{\text{obj}}$, our technique transforms it to another tunneling policy $\mathcal{T}_{\text{call}}$ such that the call-site-sensitive analysis with $\mathcal{T}_{\text{call}}$ becomes more precise than the baseline object-sensitive analysis with $\mathcal{T}_{\text{obj}}$.

To achieve this, Obj2Cfa works in the two steps: simulation and learning. It first converts $\mathcal{T}_{\text{obj}}$ into a *simulated policy* $\mathcal{T}_{\text{sim}}$. With $\mathcal{T}_{\text{sim}}$, call-site sensitivity becomes more precise than object sensitivity with $\mathcal{T}_{\text{obj}}$ but running the analysis with $\mathcal{T}_{\text{sim}}$ is expensive as it uses the baseline object-sensitive analysis as a pre-analysis. The purpose of the second step is to remove this overhead by learning the behavior of $\mathcal{T}_{\text{sim}}$ from training data. The learned policy $\mathcal{T}_{\text{call}}$ is as precise as $\mathcal{T}_{\text{sim}}$ but more efficient as it does not rely on the simulation procedure.

4.1 Simulation

The first technical contribution of this paper is the simulation procedure. Given a program P and its tunneling abstraction $T_{obj} \subseteq Invo_P$, where T_{obj} is given by $\mathcal{T}_{\text{obj}}$, i.e., $T_{obj} = \mathcal{T}_{\text{obj}}(P)$, the goal of simulation is to infer a tunneling abstraction $T_{call} \subseteq Invo_P$ such that $\text{call}(P, T_{call})$ becomes more precise than $\text{obj}(P, T_{obj})$.

Running Example. We illustrate the simulation procedure with the example program in Figure 3. The code contains class `D` that has three methods `id`, `id1`, and `m`. Methods `id` and `id1` are identity functions. The method `m` contains two method invocations at lines 7 and 8, which call `id` with new `A` and `B` objects. The `main` method creates three objects at the allocation-sites `D1`, `D2`, and `D3`, and



(a) 1-object sensitivity with $T_{obj} = \emptyset$

(b) 1-call-site sensitivity with $T_{call} = \emptyset$

(c) 1-call-site sensitivity with $T_{call} = \{5, 7, 8\}$

(d) 1-call-site sensitivity with $T_{call} = \{5\}$

Fig. 4. Call-graphs of running example produced by object sensitivity and call-site sensitivity.

stores them in variables `d1`, `d2` and `d3`, respectively. At line 16, `main` calls `id1` with a new object of type `A` and the base variable `d1`. At line 17, `id1` is called with a new object with type `B` and base variable `d2`. At line 18, `main` also calls `m` with base variable `d3`. We assume that the code has four queries, which ask the safety of casting operations at lines 7, 8, 16, and 17. Note that all of these are safe since `id` and `id1` are identity functions.

In this example, for simplicity, we assume an 1-object-sensitive analysis without context tunneling (i.e. $T_{obj} = \emptyset$) is given but our technique is applicable to object sensitivity with arbitrary k and tunneling abstraction $T_{obj} \subseteq \mathit{Invo}_P$. Figure 4a shows the call-graph produced by the baseline 1-object-senstivie analysis, where a call-graph edge is represented by invocation-site, caller method, caller context, callee method, and callee context. For example, the edge `id1[D1]` $\xrightarrow{5}$ `id[D1]` indicates that method `id` is called from `id1` at invocation-site 5, where the callee and caller contexts are `D1`. Note that this object-sensitive analysis is not precise enough to prove all queries. Although it can prove queries at lines 16 and 17 as it distinguishes the two different contexts of `id1`, it fails to prove queries at lines 7 and 8 because it uses the same context `[D3]` for `id` at both call-sites.

Figure 4b shows the call-graph obtained by the ordinary 1-call-site-sensitive analysis without context tunneling. Note that the precision of the analysis is incomparable to that of the baseline 1-object sensitivity. Because the analysis uses the call-site as the calling context, it is able to prove the queries at lines 7 and 8 by separately analyzing the two calls to `id`. However, it fails to prove the queries at lines 16 and 17 as the variable `v` in `id` under context `[5]` points-to both heap objects `A2` and `B2` that in turn propagates back to the variables `a` and `b` in the `main` method.

Inferring Tunneling Abstraction. Simulation is a two-step process. It first runs the baseline object-sensitive analysis (i.e. $\mathrm{obj}(P, T_{obj})$) to obtain its call-graph $G \subseteq \mathit{Invo}_P \times \mathit{Ctx}_P \times \mathit{Mthd}_P \times \mathit{Ctx}_P$. Next, it analyzes the structure of G and infers a tunneling abstraction T_{call} that makes call-site sensitivity to simulate G. At a high-level, we infer three kinds of invocation sites and define

$$T_{call} = (I_1 \cup I_2) \setminus I_3$$

where I_1, I_2, and I_3 are invocations sites in P. Intuitively, I_1 and I_2 denote the invocation sites that require context tunneling in order for call-site sensitivity to simulate object sensitivity. On the other hand, I_3 is the invocation sites where context tunneling must be avoided to preserve the original precision of call-site sensitivity.

Our key idea to infer I_1 and I_2 is to assume that G was produced by a context-tunneled call-site-sensitive analysis and infer backward its tunneling abstraction. To this end, we identify and exploit two fundamental properties of context-tunneled call-site sensitivity.

The first property is that the callee method's context becomes equivalent to the caller's context when context tunneling is applied during call-site sensitivity. This is because, in call-site sensitivity, applying context tunneling at an invocation site always makes the called method inherit the caller's context. Thus, we scan each call-graph edge (l, c, m, c') of G and identify those that have this property ($c = c'$). We define I_1 to be the set of invocation sites of all such edges:

$$I_1 = \{l \in \mathit{Invo}_P \mid (l, c, m, c') \in G, c = c'\}.$$

For example, I_1 is $\{5, 7, 8\}$ for the call-graph in Figure 4a, where the invocation-site 5 comes from the call-graph edges $(5, \texttt{D1}, \texttt{id}, \texttt{D1})$ and $(5, \texttt{D2}, \texttt{id}, \texttt{D2})$, 7 comes from $(7, \texttt{D3}, \texttt{id}, \texttt{D3})$, and 8 comes from $(8, \texttt{D3}, \texttt{id}, \texttt{D3})$.

In practice, applying context tunneling at I_1 gives call-site sensitivity immunity against nested call chains that are popular in object-oriented programs. For example, Figure 4a shows that object sensitivity precisely distinguishes two invocations of `id` according to their base objects $D1$ and $D2$. In contrast, conventional call-site sensitivity must use larger k to precisely analyze those nested call chains.

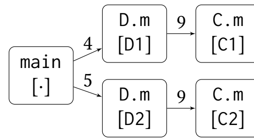
The second property of context-tunneled call-site sensitivity is that different caller contexts imply different callee contexts. Suppose two different call-graph edges (l, c_1, m, c_1') and (l, c_2, m, c_2'), where the last (i.e., kth) context element of c_1 is different from that of c_2, are generated in call-site sensitivity. If context tunneling was applied at l, then the last context elements of c_1' and c_2' are certainly different because the callee should inherit the caller's contexts (i.e. $c_1 = c_1'$ and $c_2 = c_2'$). We collect invocation sites in G with this property:

$$I_2 = \{l \in S \mid \forall (l, c_1, m, c_1'), (l, c_2, m, c_2') \in G. \mathit{last}(c_1) \neq \mathit{last}(c_2) \implies \mathit{last}(c_1') \neq \mathit{last}(c_2')\}$$

where S denotes the invocation sites where a method is called under two different contexts:

$$S = \{l \in \mathit{Invo}_P \mid \exists (l, c1, m, c1'), (l, c2, m, c2') \in G, \mathit{last}(c1) \neq \mathit{last}(c2)\},$$

and the function *last* takes a context and returns its last context element, i.e., $\mathit{last}(a_1 a_2 \dots a_k) = a_k$. I_2 denotes a sound and complete property of context-tunneled call-site sensitivity. That is, if context tunneling is not applied to l, the callee methods' contexts inevitably share the last context element l. In Figure 4a, I_2 is $\{5\}$ because the invocation-site 5 has two outgoing edges $(5, \texttt{D1}, \texttt{id}, \texttt{D1})$ and $(5, \texttt{D2}, \texttt{id}, \texttt{D2})$, where $\texttt{D1} \neq \texttt{D2} \implies \texttt{D1} \neq \texttt{D2}$ holds. The invocation-sites 7 and 8 do not belong to I_2 because they have only one call-graph edge. In Figure 4a, I_1 includes I_2, but in general, I_2 may be distinct from I_1 as shown in the following call-graph:



where $I_1 = \emptyset$ and $I_2 = \{9\}$. A detail example is described in Section 1.2 of our supplementary material [2].

Note that applying context tunneling at $I_1 \cup I_2 = \{5, 7, 8\}$ makes call-site sensitivity simulate the baseline object sensitivity (Figure 4c and Figure 4a are equivalent). However, it loses the precision benefit of the original call-site sensitivity (compare Figure 4c vs. Figure 4b). The purpose of I_3 is to

[2] https://zenodo.org/record/5560499

```
1  class Container {
2    Object elem;
3    void add(Object el) {
4      this.elem = el;}
5    Itr itr() {
6      Object e = this.elem;
7      Itr itr = new Itr(e);//It
8      return itr;}
9  }
10 class Itr {
11   Object next;
12   Itr(Object obj) {
13     this.next = obj;}
14   Object next() {
15     return this.next;}
16 }
17 void main() {
18   Container c1 = new Container();//C1
19   c1.add(new A());//A
20   Itr i1 = c1.itr();
21   object o1 = i1.next();
22
23   Container c2 = new Container();//C2
24   c2.add(new B());//B
25   Itr i2 = c2.itr();
26   object o2 = i2.next();
27 }
```

Fig. 5. Example code

avoid this precision loss. We define I_3 to be the set of invocation-sites where the called method has a single context:

$$I_3 = \{l \in Invo_P \mid \forall (l, c_1, m, c_1'), (l, c_2, m, c_2') \in G.\ c_1' = c_2'\}.$$

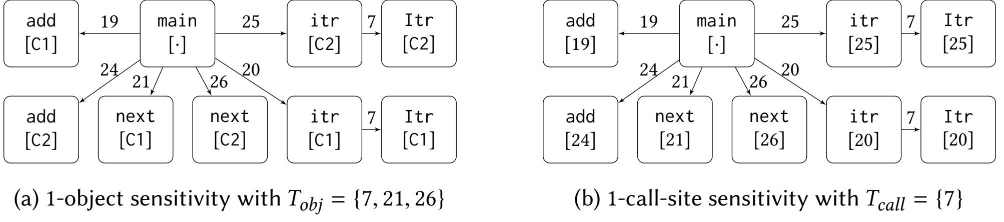
In Figure 4a, I_3 equals to $\{7, 8, 16, 17, 18\}$. For example, I_3 includes 7 because only one call-graph edge exists out of that invocation-site and I_3 does not include 5 because it has two outgoing edges to the same method (`id`) under different contexts (`[D1]` and `[D2]`). Intuitively, I_3 represents the set of invocation-sites that make conventional call-site sensitivity at least as precise as the given object sensitivity; avoiding context tunneling at I_3 would make call-site sensitivity analyze the called method more precisely than (or at least equal to) object sensitivity. It is because updating context with the invocation sites ensures that the method invocation is not conflated with another invocation from different invocation-sites which is not the case for object sensitivity. In summary, we infer $T_{call} = \{5\}$ from Figure 4a: $T_{call} = (I_1 \cup I_2) \setminus I_3 = (\{5, 7, 8\} \cup \{5\}) \setminus \{7, 8, 16, 17, 18\} = \{5\}$.

Additionally, we apply context tunneling to the invocation-sites if all their parameters are passed from those of the caller methods. For example, $\{5\}$ in our example belongs to this case because all the parameters (i.e., `this` and `v`) come from the caller. Such invocations need context tunneling because caller methods' contexts determine the value of the parameters. For the same reason, we avoid context tunneling if all the parameters are allocated just before the invocations (e.g., $\{16, 17\}$ in our example). The invocation-sites are useful because using them as context elements determines the value of the invocations' parameters.

Effectiveness in the Real-World. To show the effectiveness of simulation more clearly, we provide a representative real-world example. Also, this example shows that call-site sensitivity can simulate object sensitivity with a nontrivial tunneling abstraction ($T_{obj} \neq \emptyset$).

Containers and iterators have been popular to exemplify the strength of object sensitivity (e.g. [Jeon et al. 2018; Milanova et al. 2005; Tan et al. 2016]), as they are prevalent in object-oriented programs. Figure 5 shows a typical example. In Figure 5, the `Container` class has its iterator of which the field wraps the container's data, and the data is obtained by the calling iterator's method.

In this case, 1-object sensitivity needs context tunneling to distinguish all the method calls. The conventional 1-object-sensitive analysis is not precise as the two iterators share the same



(a) 1-object sensitivity with $T_{obj} = \{7, 21, 26\}$

(b) 1-call-site sensitivity with $T_{call} = \{7\}$

Fig. 6. call-graphs produced by 1-object sensitivity with context-tunneling and 1-call-site sensitivity

heap allocation-site `It`. Consider a tunneling abstraction $T_{obj} = \{7, 21, 26\}$. With this tunneling abstraction, 1-object sensitivity produces precise results with the call-graph shown in Figure 6a.

With our technique, call-site-sensitivity can simulate the call-graph produced by the context-tunneled object-sensitive analysis. From the call-graph in Figure 6a, our technique finds out

$$I_1 = \{7\}, \quad I_2 = \{7\}, \quad I_3 = \{19, 20, 21, 24, 25, 26\}.$$

and produces a tunneling abstraction $T_{call} = \{7\}$. Figure 6b shows the call-graph of 1-call-site sensitivity with the tunneling abstraction, which is exactly the same as that of the baseline object-sensitive analysis in Figure 6a. Note that 1-object sensitivity and 1-call-site sensitivity use different tunneling abstractions, i.e., $T_{call} = \{7\}$ vs. $T_{obj} = \{7, 21, 26\}$.

In the supplementary material, we show that our technique can handle more real-world cases, namely *precision-critical* patterns [Li et al. 2018a]. Li et al. [2018a] identified fairly representative and exhaustive patterns that require object sensitivity in real-world Java programs; applying 2-object sensitivity only to the methods with those patterns is sufficient to achieve 98.8% of the full precision [Li et al. 2018a]. Conventional call-site-sensitive analysis is ineffective to handle such patterns as they require the analysis to maintain deep calling contexts. With context tunneling, however, our technique enables call-site sensitivity to simulate object sensitivity in all cases. We provide detailed description of the patterns and how our technique works in Section 1 of the supplementary material.

Simulated Policy. Now we define the simulated policy $\mathcal{T}_{\mathsf{sim}}$. Let simulate be the simulation process described above. As input, simulate takes a program P and a tunneling abstraction $T_{obj} \subseteq \mathit{Invo}_P$ of baseline object sensitivity. Running the simulation procedure, denoted simulate(P, T_{obj}), produces a tunneling abstraction $T_{call} \subseteq \mathit{Invo}_P$ for call-site sensitivity. With simulate, we can transform $\mathcal{T}_{\mathsf{obj}}$ into $\mathcal{T}_{\mathsf{sim}}$ as follows:

$$\mathcal{T}_{\mathsf{sim}} = \lambda P.\ \mathsf{simulate}(P, \mathcal{T}_{\mathsf{obj}}(P)). \tag{3}$$

Although the simulated policy $\mathcal{T}_{\mathsf{sim}}$ makes call-site sensitivity more precise than the baseline object sensitivity, using $\mathcal{T}_{\mathsf{sim}}$ is impractical because the simulation procedure incurs the overhead of running the baseline object-sensitive analysis.

4.2 Simulation-Guided Learning

The second technical contribution of this paper is simulation-guided learning that aims to remove the overhead of $\mathcal{T}_{\mathsf{sim}}$ while maintaining its precision. To do so, from a dataset of programs $\mathbf{P} = \{P_1, P_2, \ldots, P_n\}$, we learn the final policy, i.e., $\mathcal{T}_{\mathsf{call}}$, that captures the behavior of $\mathcal{T}_{\mathsf{sim}}$ without invoking the expensive simulation procedure.

Parameterized Policy. To learn a policy from data, we need to define a *parameterized policy*, denoted $\mathcal{T}_f$, whose behavior is fully controlled by the parameter f. The goal of learning then is to find an appropriate parameter from data.

For parameterization, we adapt the idea of prior work [Jeong et al. 2017], where the parameter f is a boolean formula over atomic features. We assume that a set $\mathbb{A} = \{a_1, a_2, \ldots, a_m\}$ of atomic features is given (we explain them shortly). Formally, a feature a_i is a function from programs to predicates on invocation sites, i.e., $a_i(P) : Invo_P \rightarrow \{true, false\}$. That is, an invocation site l in a program P has feature a_i iff $a_i(P)(l)$ is $true$. In prior work [Jeon et al. 2018], atomic features are combined by a boolean formula f to express complex, in particular disjunctive, properties:

$$f \rightarrow true \mid false \mid a_i \in \mathbb{A} \mid \neg f \mid f_1 \wedge f_2 \mid f_1 \vee f_2$$

A boolean formula f denotes a set of invocation sites. We write $[\![f]\!]_P$ for the denotation of f with respect to P: $[\![true]\!]_P = Invo_P$, $[\![false]\!]_P = \emptyset$, $[\![a_i]\!]_P = \{l \in Invo_P \mid a_i(P)(l)\}$, $[\![\neg f]\!]_P = Invo_P \setminus [\![f]\!]_P$, $[\![f_1 \wedge f_2]\!]_P = [\![f_1]\!]_P \cap [\![f_2]\!]_P$, $[\![f_1 \vee f_2]\!]_P = [\![f_1]\!]_P \cup [\![f_2]\!]_P$. Then, with a boolean formula f, we define the parameterized policy $\mathcal{T}_f$ as follows:

$$\mathcal{T}_f(P) = [\![f]\!]_P.$$

Learning Objective. The goal of learning is to find a formula f that enables $\mathcal{T}_f$ to capture the behavior of $\mathcal{T}_{\mathsf{sim}}$ on the training programs; we aim to find a formula f such that

$$\sum_{P \in \mathbf{P}} \mathsf{call}(P, \mathcal{T}_f(P)) \approx \sum_{P \in \mathbf{P}} \mathsf{call}(P, \mathcal{T}_{\mathsf{sim}}(P)) \tag{4}$$

where we assume call returns the number of unproved queries (e.g., #may-fail casts, where lower is more precise). Note that our learning objective is more challenging than those considered in prior work [Jeon et al. 2018; Jeong et al. 2017]. In prior work, the objective was typically to find a "good-enough" policy but, in our case, the learned policy should capture the specific behavior of the simulated policy.

Learning Algorithm. We present a new, simulation-guided learning algorithm that can effectively solve the problem in Eq. (4). In Section 5.2, we show that the existing, unguided learning algorithm [Jeon et al. 2018] is not powerful enough to solve the problem of capturing the behavior of the simulated policy ($\mathcal{T}_{\mathsf{sim}}$).

The overall structure of our algorithm is given in Algorithm 1. We invoke the algorithm by Learn($\mathbf{P}, \mathbb{A} \cup \neg\mathbb{A}, \mathcal{T}_{\mathsf{sim}}$), where $\neg\mathbb{A} = \{\neg a \mid a \in \mathbb{A}\}$, so that the formula f is initially set to the disjunction of all atomic features and their negation: $f = a_1 \vee \neg a_1 \vee a_2 \vee \neg a_2 \cdots \vee a_m \vee \neg a_m$, where $a_1, a_2, \ldots, a_m \in \mathbb{A}$ (line 2). Then, we repeat the loop at lines 4–14. At the beginning of each iteration, the formula f is in disjunctive normal form; f is of the form $c_1 \vee c_2 \vee \cdots \vee c_k$, where c_i is a conjunctive clause. A single refinement step for f is done by choosing a clause c in f (via ChooseClause at line 5), choosing an atomic feature (via ChooseAtom at line 6), and replacing c in f by $c \wedge a$ (line 8). If the quality of the refined formula f' has been improved over the original formula f (line 9), we set f to f' and otherwise discard f'. This process is repeated until no more refinement is possible. To check this termination condition, the algorithm maintains Γ that maps clauses to available refiners (line 3, 10, and 12).

The key feature of our algorithm is to steer the search toward the desired solutions by receiving guidance from the simulated policy ($\mathcal{T}_{\mathsf{sim}}$). The guidance happens in the three components, i.e., ChooseClause, ChooseAtom, and Improved. Given a formula $f = c_1 \vee c_2 \vee \cdots \vee c_k$, we choose a clause whose behavior most deviates from $\mathcal{T}_{\mathsf{sim}}$ over the training programs; that is, ChooseClause($f, \mathcal{T}_{\mathsf{sim}}, \mathbf{P}$) chooses c in f that maximizes the difference between $[\![c]\!]_P$ and $\mathcal{T}_{\mathsf{sim}}$:

$$\mathsf{ChooseClause}(f, \mathcal{T}_{\mathsf{sim}}, \mathbf{P}) = \mathrm{argmax}_{c \in f} \textstyle\sum_{P \in \mathbf{P}} |[\![c]\!]_P \setminus \mathcal{T}_{\mathsf{sim}}(P)|.$$

Algorithm 1 Overall learning algorithm

Input: Training programs $\mathbf{P}$, features A, and simulated policy $\mathcal{T}_{\mathsf{sim}}$
Output: A boolean formula f

```
 1: procedure LEARN(P, A, T_sim)
 2:     f ← a_1 ∨ a_2 ∨ ··· ∨ a_k (A = {a_1, a_2, ..., a_k})
 3:     Γ ← λc ∈ f.A \ {a | a ∈ c}
 4:     repeat
 5:         c ← ChooseClause(f, T_sim, P)
 6:         a ← ChooseAtom(c, Γ, T_sim, P)
 7:         c′ ← c ∧ a
 8:         f′ ← Replace c in f by c′
 9:         if Improved(f′, f, T_sim, P) then
10:             Γ ← Γ[c′ ↦ Γ(c) \ {a}], f ← f′
11:         else
12:             Γ ← Γ[c ↦ Γ(c) \ {a}]
13:         end if
14:     until no more refinement is possible (∀c. Γ(c) = ∅)
15:     return f
16: end procedure
```

To refine the chosen clause c, $\mathsf{ChooseAtom}(c, \Gamma, \mathcal{T}_{\mathsf{sim}}, \mathbf{P})$ chooses an atom $a \in \Gamma(c)$ that maximizes the following:

$$\textstyle\sum_{P \in \mathbf{P}} |\mathcal{T}_{\mathsf{sim}}(P) \cap [\![c]\!]_P \cap [\![a]\!]_P|.$$

Intuitively, it chooses the atom that most conservatively refines the clause toward the simulation result. To this end, it selects a that maximizes $\mathcal{T}_{\mathsf{sim}}(P) \cap [\![a]\!]_P$ (refining toward the simulation result) and $[\![c]\!]_P \cap [\![a]\!]_P$ (conservatively refining c). With the chosen c and a, the formula f is specified to f' by replacing c in f by $c \wedge a$. To check whether f' improves over f, we evaluate the formulas with the following objective function:

$$O(f, \mathcal{T}_{\mathsf{sim}}, \mathbf{P}) = \textstyle\sum_{P \in \mathbf{P}} \mathsf{call}(P, \mathcal{T}_f(P) \cap \mathcal{T}_{\mathsf{sim}}(P)).$$

Given a formula, the objective function runs the analysis over the training programs with the intersection of the current tunneling policy ($\mathcal{T}_f$) and the simulated policy ($\mathcal{T}_{\mathsf{sim}}$). The condition $\mathsf{Improved}(f', f, \mathcal{T}_{\mathsf{sim}}, \mathbf{P})$ is true iff $O(f', \mathcal{T}_{\mathsf{sim}}, \mathbf{P}) \leq O(f, \mathcal{T}_{\mathsf{sim}}, \mathbf{P})$. In the objective function, note that we evaluate the performance of $\mathcal{T}_f(P) \cap \mathcal{T}_{\mathsf{sim}}(P)$ instead of $\mathcal{T}_f(P)$. This is a critical step to avoid local minima. For example, suppose we have formulas f_1 and f_2, where f_2 is obtained by refining f_1, and that both $[\![f_1]\!]_P$ and $[\![f_2]\!]_P$ are supersets of $\mathcal{T}_{\mathsf{sim}}(P)$ (i.e. $\mathcal{T}_{\mathsf{sim}}(P) \subseteq [\![f_2]\!]_P \subseteq [\![f_1]\!]_P$). Although f_2 can be refined further toward $\mathcal{T}_{\mathsf{sim}}$, such refinement can be rejected as $\mathcal{T}_{f_2}$ may have poorer precision than $\mathcal{T}_{f_1}$ because tunneling abstraction is not monotone with respect to precision [Jeon et al. 2018]. Thus, the learning algorithm fails to make a further progress, ending up with a local minimum f_1. With our objective function, however, the learning algorithm can avoid such a local minimum because $\mathcal{T}_{f_1}(P) \cap \mathcal{T}_{\mathsf{sim}}(P) = \mathcal{T}_{f_2}(P) \cap \mathcal{T}_{\mathsf{sim}}(P) = \mathcal{T}_{\mathsf{sim}}(P)$. In Section 2 of the supplementary material, we provide a simple running example that illustrates how Algorithm 1 learns a formula f.

Feature Engineering. The success of learning depends also on the atomic features ($\mathbb{A}$). We used a total of 35 atomic features in Table 1, all of which describe syntactic properties of invocation sites. Here, we identify invocations with called methods on them. Features A1–A10 and B1–B6 came from [Jeon et al. 2018]. Features A1–A10 describe methods whose signatures contain the corresponding strings. For example, when a method's signature string is "java.lang.String: int length()", the method has features A1, A2, A4, A7, and A9. Features B1–B6 describe properties of

Table 1. Atomic features ($\mathbb{A}$) used in our method

A1	"java"	A2	"lang"	A3	"sun"	A4	"()"	A5	"void"
A6	"security"	A7	"int"	A8	"util"	A9	"String"	A10	"init"

B1	Method is contained in a nested class	B2	Method contains local assignments
B3	Method contains local variables	B4	Method is contained in a large class
B5	Method contains a heap allocation	B6	Method is a static method
C1	Method is called on `this` (i.e. `this.`m`(...)`)	C2	An argument is allocated in the same method
C3	Method is called on object of static field	C4	Method is called else where in the same method
C5	The base variable is passed to an initializer	C6	The containing method's modifier is "`protected`"
C7	The containing method has exception handling	C8	All parameters are passed from caller method
C9	All parameters are initialized just before the calls	C10	In "`java.util.regex`" class
C11	Invoke constructor (i.e. `new C(...)`)	C12	All the caller method's arguments' type is integer
C13	Caller method takes more than 2 arguments	C14	In "`java.io.*`" class
C15	In "`java.util.logging`" class	C16	Takes at least 2 arguments
C17	Virtual method calls in application class	C18	Callee method's name is "`clone`"
C19	C16 $\wedge\neg$ C6 $\wedge\neg$ C1		

method bodies. For example, feature B1 indicates whether a method is defined in a nested class or not

Using existing features only was insufficient and we newly designed features C1–C19 in Table 1. In our case, feature engineering was not very difficult as it can be guided by the simulated policy $\mathcal{T}_{\mathsf{sim}}$. To obtain those features, we initially ran our learning algorithm on training data with existing features (A1–A10 and B1–B6) only, which resulted in a parameter f with which the learned policy ($\mathcal{T}_f$) did not satisfy the learning objective. We investigated the reason why $\mathcal{T}_f$ fails to capture the behavior of $\mathcal{T}_{\mathsf{sim}}$ by analyzing the difference between $\mathcal{T}_{\mathsf{sim}}$ and $\mathcal{T}_f$ (i.e. $\mathcal{T}_{\mathsf{sim}}(P) \setminus \mathcal{T}_f(P)$ and $\mathcal{T}_f(P) \setminus \mathcal{T}_{\mathsf{sim}}(P)$). That is, our goal was to identify features a_1 and a_2 that minimize $\sum_{P\in\mathbf{P}}[\![a_1]\!]_P \oplus (\mathcal{T}_{\mathsf{sim}}(P) \setminus \mathcal{T}_f(P))$ and $\sum_{P\in\mathbf{P}}[\![a_2]\!]_P \oplus (\mathcal{T}_f(P) \setminus \mathcal{T}_{\mathsf{sim}}(P))$, respectively. We included the new features a_1 and a_2 in the feature set and ran the algorithm again. We repeated this process until the policy space was large enough to contain solutions and the learning algorithm could find one of them.

5 EXPERIMENTAL RESULTS

We experimentally prove our claim by evaluating Obj2Cfa on real-world programs. Main research questions are as follows:

- **Does our claim hold in the real-world?** Can call-site sensitivity be significantly superior to object sensitivity for real-world programs? How precise and scalable can the context-tunneled call-site-sensitive analysis be in practice?
- **Impact of simulation and learning**: Is simulation necessary? How accurately can our technique simulate object sensitivity? Is the simulation-guided learning necessary to capture the behavior of the simulated policy? How important are the features in learning?

Experimental Setting. We implemented Obj2Cfa on top of Doop [Bravenboer and Smaragdakis 2009], a popular pointer analysis framework for Java [Jeon et al. 2018; Jeong et al. 2017; Li et al. 2018b; Smaragdakis et al. 2014; Tan et al. 2016]. We used the publicly-available implementation of context tunneling given by Jeon et al. [2018] and newly implemented our simulation (Section 4.1) and learning (Section 4.2) techniques in Doop. We conducted all experiments on a machine with Intel i7 CPU and 64GB memory running the Ubuntu 16.04 64bit operating system.

We used 12 Java programs used by Jeon et al. [2018], of which 10 came from the DaCapo 2006 benchmarks [Blackburn et al. 2006] (luindex, lusearch, antlr, pmd, fop, eclipse, xalan, chart, bloat, and jython), and the remaining two (checkstyle and jpc) are real-world open-source programs.

Following prior work [Jeon et al. 2018; Smaragdakis et al. 2014], we classified those 12 programs into 4 small (luindex, lusearch, antlr, and pmd) and 8 large programs. We used the group of small programs as training data, from which our context-tunneling policy $\mathcal{T}_{\text{call}}$ is learned, and used large programs as test data to evaluate the policy for unseen programs.

Using Obj2Cfa, we transformed *1objH+T*, a context-tunneled 1-object-sensitive analysis developed by Jeon et al. [2018], into our 1-call-site-sensitive analysis, denoted *1callH+SL*. We chose *1objH+T* as baseline because it is one of the best object-sensitive analyses available today, which boosts the conventional 1-object-sensitive analysis using a well-tuned context-tunneling policy. For example, *1objH+T* is empirically more precise than conventional 2-object-sensitive analysis (*2objH*), which is considered to be highly precise [Jeon et al. 2020; Jeong et al. 2017; Li et al. 2018a,b; Smaragdakis et al. 2014], yet more scalable than 1-object sensitivity [Jeon et al. 2018]. We obtained the tunneling policy of *1objH+T* from the publicly available artifact of Jeon et al. [2018]. From *1objH+T*, we first applied our simulation technique (Section 4.1) to produce the corresponding simulated call-site sensitivity, denoted *1callH+S*. Then, we used our learning algorithm (Section 4.2) to obtain the final call-site-sensitive analysis, *1callH+SL*. Note that *1callH+S* runs *1objH+T* as a pre-analysis but *1callH+SL* does not (thanks to learning).

Our main objective is to compare *1objH+T* and *1callH+SL*, but we compare with some notable analyses as well to see the advance more clearly. In summary, we compare the following analyses:

- *1objH+T*: a state-of-the-art context-tunneled 1-object-sensitive analysis [Jeon et al. 2018]
- *1callH+S*: the simulated 1-call-site-sensitive analysis obtained from *1objH+T* via simulation
- *1callH+SL*: our final 1-call-site sensitive analysis (obtained from *1callH+S* via learning)
- *2objH*: 2-object-sensitive analysis without tunneling [Smaragdakis et al. 2011]
- *1callH+T*: the existing state-of-the-art 1-call-site sensitivity with tunneling [Jeon et al. 2018]

2objH is available in Doop. *1callH+T* is available in the artifact provided by Jeon et al. [2018]. All analyses use 1-context-sensitive heap. For precision metric, we mainly use may-fail casts.

5.1 Performance of *1callH+SL*

Table 2 shows that our analysis (*1callH+SL*) significantly outperforms other analyses in both precision and cost, confirming our claim that call-site sensitivity can be superior to object sensitivity for real-world programs. In particular, it beats by far the baseline object sensitivity (*1objH+T*) in precision for all programs. For example, *1objH+T* reports 1,253 may-fail casts for fop but *1callH+SL* reduces the number to 1,072. Also, *1callH+SL* is more scalable than *1objH+T*. For example, *1callH+SL* takes 2,731 seconds to analyze jython while *1objH+T* times out.

We note that our 1-call-site-sensitive analysis is even more precise than the traditional 3-object-sensitive analysis with 2-context-sensitive heap (*3obj2H*):

Program	*1callH+SL*		*3obj2H*	
	#fail-casts	Time(s)	#fail-cast	Time(s)
luindex	357	40	435	564
antlr	477	62	543	561
pmd	707	65	782	584

where we compare the results only for the three small programs because *3obj2H* does not scale for other programs. Note that *3obj2H* is the most precise object-sensitive analysis evaluated in the literature [Lu and Xue 2019; Tan et al. 2017] and *1callH+SL* substantially improves its precision with much smaller costs.

The performance of *1callH+SL* is completely beyond the reach of existing call-site-sensitive analyses. *1callH+T* is the state-of-the-art call-site sensitivity, which is more precise and faster than

Table 2. Precision and cost comparison of our analysis (*1callH+SL*) against various context-sensitive analyses: *1objH+T*, *2objH*, *1callH+T*, and *1callH+S*. #may-fail-casts: the number of potential may-fail casts (down-casting) in the program. #VarPtsTo: the size of the context-insensitive variable points-to sets. Time(s): the end-to-end, total amount of time needed to run each analysis. For all metrics, lower is better.

	program	Metric	*1callH+SL*	*1callH+S*	*1objH+T*	*2objH*	*1callH+T*
Training programs	luindex	VarPtsTo	250,012	245,470	256,531	255,545	800,715
		#may-fail-casts	357	360	462	496	784
		time elapsed(s)	40	86	37	40	82
	lusearch	VarPtsTo	264,728	260,204	271,765	270,710	890,529
		#may-fail-casts	371	374	469	508	843
		time elapsed(s)	45	94	39	82	85
	antlr	VarPtsTo	302,226	297,268	309,671	308,643	965,445
		#may-fail-casts	477	477	570	611	945
		time elapsed(s)	62	123	52	52	128
	pmd	VarPtsTo	306,462	300,391	329,415	327,295	1,116,506
		#may-fail-casts	707	711	812	846	1,200
		time elapsed(s)	65	128	56	138	138
Testing programs	eclipse	VarPtsTo	353,657	337,496	351,898	345,806	1,241,995
		#may-fail-casts	569	573	698	729	1,073
		time elapsed(s)	48	159	47	58	136
	xalan	VarPtsTo	410,440	394,522	401,556	400,872	1,660,901
		#may-fail-casts	576	586	680	720	1,137
		time elapsed(s)	71	590	377	2,288	208
	chart	VarPtsTo	501,615	496,676	502,913	500,357	4,694,330
		#may-fail-casts	883	942	1,011	1,055	2,376
		time elapsed(s)	96	575	84	382	805
	fop	VarPtsTo	650,218	637,213	726,777	720,031	3,467,105
		#may-fail-casts	1,072	1,072	1,253	1,270	1,977
		time elapsed(s)	206	407	137	493	500
	bloat	VarPtsTo	1,136,393	1,136,366	1,126,688	1,114,648	3,454,301
		#may-fail-casts	1,266	1,285	1,374	1,407	1,949
		time elapsed(s)	498	3,306	371	2,463	805
	jython	VarPtsTo	1,067,711	N/A	-	-	3,085,401
		#may-fail-casts	845	N/A	-	-	1,331
		time elapsed(s)	2,731	N/A	>10,800	>10,800	188
	jpc	VarPtsTo	1,304,810	1,118,622	1,142,496	1,114,946	6,667,910
		#may-fail-casts	1,639	1,642	1,795	1,814	2,620
		time elapsed(s)	493	699	262	1,737	1,511
	checkstyle	VarPtsTo	307,378	299,101	327,629	314,857	1,141,902
		#may-fail-casts	465	472	591	620	913
		time elapsed(s)	83	220	99	220	139

Table 3. Precision of call-graph related clients (#call-graph-edges, #reachable-methods, #polymorphic-calls) of the analyses. Again, lower is better for all metrics.

	program	Metric	*1callH+SL*	*1callH+S*	*1objH+T*	*2objH*	*1callH+T*
Training programs	luindex	#call-graph-edges	36,578	36,426	36,504	36,487	40,830
		#reachable-methods	7,710	7,699	7,702	7,702	7,879
		#polymorphic-calls	908	900	905	903	1,066
	lusearch	#call-graph-edges	39,456	39,304	39,381	39,362	44,007
		#reachable-methods	8,354	8,343	8,344	8,344	8,551
		#polymorphic-calls	1,086	1,078	1,078	1,075	1,243
	antlr	#call-graph-edges	55,467	55,396	55,474	55,455	59,818
		#reachable-methods	8,721	8,711	8,714	8,714	8,885
		#polymorphic-calls	1,722	1,709	1,709	1,716	1,876
	pmd	#call-graph-edges	42,980	42,909	43,015	42,998	47,889
		#reachable-methods	9,095	9,085	9,090	9,090	9,296
		#polymorphic-calls	951	943	947	946	1,117
Testing programs	eclipse	#call-graph-edges	44,947	44,842	44,926	44,824	51,724
		#reachable-methods	9,204	9,194	9,197	9,188	9,444
		#polymorphic-calls	1,184	1,175	1,181	1,179	1,399
	xalan	#call-graph-edges	50,061	49,985	50,065	50,051	55,644
		#reachable-methods	10,338	10,331	10,336	10,336	10,539
		#polymorphic-calls	1,637	1,630	1,633	1,628	1,858
	chart	#call-graph-edges	58,933	58,912	58,993	59,035	80,500
		#reachable-methods	12,500	12,495	12,510	12,510	16,020
		#polymorphic-calls	1,609	1,605	1,616	1,614	2,698
	fop	#call-graph-edges	59,663	59,440	61,975	61,923	71,741
		#reachable-methods	13,777	13,763	14,376	14,373	15,108
		#polymorphic-calls	1,962	2,063	2,063	2,047	2,522
	bloat	#call-graph-edges	61,249	60,990	60,638	60,601	68,674
		#reachable-methods	9,947	9,928	9,914	9,914	10,113
		#polymorphic-calls	1,679	1,667	1,652	1,650	1,925
	jython	#call-graph-edges	52,644	N/A	N/A	N/A	59,932
		#reachable-methods	10,625	N/A	N/A	N/A	10,987
		#polymorphic-calls	14,084	N/A	N/A	N/A	1,565
	jpc	#call-graph-edges	95,837	95,098	95,371	95,209	110,493
		#reachable-methods	18,634	18,581	18,655	18,631	19,854
		#polymorphic-calls	5,053	4,989	4,999	4,963	5,646
	checkstyle	#call-graph-edges	42,410	42,333	42,204	42,174	49,346
		#reachable-methods	8,435	8,424	8,428	8,428	8,672
		#polymorphic-calls	1,096	1,088	1,090	1,088	1,304

ordinary 2-call-site-sensitive analysis [Jeon et al. 2018]. However, *1callH+SL* reduced about 50% of may-fail casts of *1callH+T* for all programs.

Table 3 compares the precision of the analyses for three other call-graph construction related clients used in previous works [Li et al. 2018a,b; Tan et al. 2017]. #call-graph-edges presents the number of call-graph edges without contexts, #reachable-methods presents the number of reachable methods, and #polymorphic-calls presents the number of call-sites that cannot be determined as monomorphic calls. The results show that our simulated call-site sensitivity *1callH+S* overall shows better precision than the baseline object sensitivity *1callH+T*. This difference between *1callH+S* and *1callH+SL* comes from the learning objective. *1callH+SL* was not trained to optimize these metrics from *1callH+S* (the current implementation of our algorithm uses #may-fail-casts as the learning objective). The learning objective, however, can be easily adapted for other clients.

5.2 Impact of Simulation and Learning

Next, we discuss the impact of our technical contributions, simulation and learning.

Simulation Accuracy. We first note that our technique enabled call-site sensitivity to accurately simulate object sensitivity for real-world applications. For all benchmark programs except for jython, we ran the baseline object-sensitive analysis (*1objH+T*) and the corresponding call-site sensitive analysis (*1callH+SL*) and counted the number (A) of may-fail cast queries that both analyses can prove and the number (B) of queries that *1objH+T* can prove. The ratio A/B hints at how accurately *1callH+SL* covers the baseline object sensitivity. The average ratio over the 11 programs was 0.98, implying that Obj2Cfa can simulate object sensitivity almost completely.

Most of the remaining 2% of queries, which Obj2Cfa missed, were caused by imperfect simulation rather than learning; when we calculated the ratio using simulation only (i.e., *1callH+S*), the ratio A'/B was still 0.98, where A' denotes the number of may-fail casts that both *1callH+S* and *1objH+T* can prove. We found that this was because we used a coarse tunneling space, i.e., invocation sites, and that using a more fine-grained tunneling space (e.g., pairs of invocation-sites and receiver objects) would improve the simulation accuracy. We describe an example in Section 7.

Roles of Simulation and Learning. The results in Table 2 show that our simulation technique plays a key role in improving the precision of call-site sensitivity. The column *1callH+S* in Table 2 presents the performance of the call-site sensitive analysis obtained by simulating *1objH+T*. For all programs, *1callH+S* shows a far better precision than *1objH+T*.

Comparing the performance of *1callH+S* and *1callH+SL* reveals that the use of learning reduced the overhead of *1callH+S* significantly. As the simulation needs to run the baseline object sensitivity (*1objH+T*), the simulated call-site-sensitive analysis (*1callH+S*) is inherently more expensive than the baseline object sensitivity (*1objH+T*). For example, *1callH+S* is unable to analyze the program jython because the baseline object sensitivity (*1objH+T*) failed to analyze it. Thanks to our learning technique, however, our call-site sensitivity (*1callH+SL*) removed the limitation. For example, *1callH+SL* successfully analyzed jython within the time budget.

We checked that the result of learning is not overfitted to the shared library. In eclipse, for example, *1callH+SL* proved 17% (resp., 15%) more may-fail casts compared to *1objH+T* in the application (resp., library) code, which implies that learning is equally effective in both application and library code. We also checked *1callH+SL* is not overfitted to DaCapo benchmarks. In a non-DaCapo benchmark checkstyle, for example, *1callH+SL* proved 41% (resp., 19%) more may-fail casts compared to *1objH+T* in the application (resp., library) code.

Impact of Simulation-Guided Learning. Our simulation-guided learning (Section 4.2) was essential for effectively capturing the precision of the simulated policy. To demonstrate this, we

Table 4. Impact of our simulation-guided learning (numbers indicate #may-fail-casts)

	luindex	lusearch	antlr	pmd	Total
1callH+SL	357	371	477	707	1,912
1callH+T_{new}	565	580	735	929	2,809
Decision tree	519	533	659	895	2,606

conducted two experiments. We first replaced our algorithm by the existing unguided algorithm for learning context tunneling [Jeon et al. 2018] and trained a policy using the same set of atomic features in Table 1 and training programs. Table 4 shows that the existing algorithm ended up with a much less precise policy. Over the four training programs, the context-tunneled call-site sensitivity obtained using the existing learning algorithm [Jeon et al. 2018], denoted *1callH+T_{new}* in Table 4, reported 2,809 may-fail casts while the number for *1callH+SL* is much smaller (1,912). This result shows that the use of simulation in our approach is critical.

Second, we replaced our algorithm by a simple supervised learning method. We generated labeled data, which consists of feature vectors of invocation sites and labels indicating whether selected by the simulated policy ($\mathcal{T}_{\mathsf{sim}}$) or not. We used the decision tree algorithm in Pedregosa et al. [2011] to learn a policy. Again, the resulting analysis (denoted *Decision tree* in Table 4) was unsatisfactory in precision; over the training programs, it reported 2,604 may-fail casts. This is mainly because the labeled data does not have enough information; although the simulated policy labels which invocations need context tunneling, it is unable to label which invocations are precision critical.

Impact of New Features. Another important factor for effective learning was the use of the features in Table 1 that are specifically designed for call-site sensitivity (recall that we crafted those features guided by the simulated policy $\mathcal{T}_{\mathsf{sim}}$). For example, the *1callH+T_{new}* analysis described above, which differs from *1callH+T* [Jeon et al. 2018] only in the use of the new set of features, is much more precise than *1callH+T*: *1callH+T_{new}* produces 970 fewer alarms than *1callH+T* that is learned using the same algorithm but with the different features designed by Jeon et al. [2018].

We note that our features in Table 1 are not appropriate for learning tunneling heuristics for object sensitivity. This is because our features are designed to reproduce the results of the simulated call-site sensitivity ($\mathcal{T}_{\mathsf{sim}}$) rather than object sensitivity. To clarify the impact of using a different set of features, we used our features and the existing algorithm [Jeon et al. 2018] to learn a tunneling heuristic for object sensitivity. The resulting analysis, denoted *1objH+T_{new}*, was overall less precise than *1objH+T*.

Learning Cost. The learning phase of Obj2Cfa spent 58 hours in total, which is slightly faster than the prior algorithm [Jeon et al. 2018]. Although our algorithm is expensive, we believe it is acceptable because learning is done off-line and saves otherwise more expensive human effort.

5.3 Comparison with Selective Object Sensitivity

We also checked how *1callH+SL* works in comparison with *Zipper* [Li et al. 2018a, 2020], a state-of-the-art technique that performs selective 2-object sensitivity. Unlike the analyses considered in Section 5.1, which perform uniform k-context sensitivity, *Zipper* performs a selective context-sensitive analysis and applies 2-object sensitivity only when it is necessary. In particular, compared to other selective approaches [Jeon et al. 2020; Jeong et al. 2017; Li et al. 2018b; Smaragdakis et al. 2014], *Zipper* is precision-focused and improves the scalability of uniform 2-object sensitivity while preserving most (98.8%) of its precision.

For direct comparison, we implemented *1callH+SL* and *1objH+T* on top of the artifact provided by Li et al. [2018a]. We used 14 programs (batik, fop, sunflow, bloat, xalan, chart, findbugs, eclipse,

Table 5. Performance comparison of *1callH+SL*, *2objH+Zip* (i.e., *Zipper* [Li et al. 2018a]), and *1callH+SL+Zip*.

Program	Analysis	#fail-casts	Time	Program	Analysis	#fail-casts	Time
eclipse	*1callH+SL*	460	62	fop	*1callH+SL*	1,359	368
	1callH+SL+Zip	482	42		*1callH+SL+Zip*	1,397	279
	2objH+Zip	586	45		*2objH+Zip*	1,471	318
bloat	*1callH+SL*	1,136	456	jpc	*1callH+SL*	1,279	201
	1callH+SL+Zip	1,168	397		*1callH+SL+Zip*	1,368	176
	2objH+Zip	1,224	1,942		*2objH+Zip*	1,415	147
chart	*1callH+SL*	810	80	xalan	*1callH+SL*	463	91
	1callH+SL+Zip	890	58		*1callH+SL+Zip*	482	72
	2objH+Zip	910	54		*2objH+Zip*	568	66
sunflow	*1callH+SL*	1,787	438	findbugs	*1callH+SL*	1,316	288
	1callH+SL+Zip	1,818	279		*1callH+SL+Zip*	1,360	164
	2objH+Zip	1,869	361		*2objH+Zip*	1,437	604
checkstyle	*1callH+SL*	522	106	batik	*1callH+SL*	1,602	1,422
	1callH+SL+Zip	535	88		*1callH+SL+Zip*	1,782	1,649
	2objH+Zip	607	248		*2objH+Zip*	1,614	667
sunflow09	*1callH+SL*	1,101	126	xalan09	*1callH+SL*	989	241
	1callH+SL+Zip	1,146	84		*1callH+SL+Zip*	1,004	166
	2objH+Zip	1,192	73		*2objH+Zip*	1,074	214
avrora09	*1callH+SL*	918	147	h2	*1callH+SL*	1,207	3,766
	1callH+SL+Zip	964	105		*1callH+SL+Zip*	1,225	2,318
	2objH+Zip	1,042	115		*2objH+Zip*	1,311	8,216

jpc, checkstyle, h2, xalan09, avrora09, sunflow09) used in Li et al. [2018a]. Five programs (fop, xalan, bloat, chart, eclipse) are DaCapo 2006 benchmarks [Blackburn et al. 2006], four programs (xalan09, sunflow09, avrora09, and h2) are from DaCapo 2009 benchmarks, and the remaining five programs (sunflow, findbugs, jpc, checkstle, batik) are real-world Java applications. Note that the artifact of Li et al. [2018a] uses a quite different reflection analysis from the one we used in Table 2. The number of alarms and analysis time varies significantly depending on how reflection is supported. The baseline analysis on top of *Zipper* is implemented supports reflection less conservatively than our implementation in Section 5.1 and therefore Table 5 reports fewer alarms than Table 2. In Table 5, our analysis uses the same reflection analysis as *Zipper*.

Table 5 compares the performance of *1callH+SL* and *Zipper* (denoted *2objH+Zip* in Table 5). For all programs, *1callH+SL* shows a better precision than *Zipper*. Especially for bloat, *1callH+SL* is more precise and scalable than than *2objH+Zip*; *2objH+Zip* took 1,942 seconds and reports 1,224 alarms but *1callH+SL* analyzed it within 456 seconds with 1,136 may-fail cast alarms. However, for batik, *1callH+SL* was slower than *2objH+Zip*.

Because context tunneling and selective context sensitivity are orthogonal techniques, we can combine our analysis (*1callH+SL*) and *Zipper*, resulting in a selective 1-call-site-sensitive analysis with context tunneling (*1callH+SL+Zip*). This is possible because the idea of *Zipper* is general and applicable to various context-sensitivity flavors including call-site sensitivity [Li et al. 2020]. Table 5 shows the performance of *1callH+SL+Zip*. Thanks to *Zipper*, *1callH+SL+Zip* becomes faster than *1callH+SL* at the small cost of the precision. In h2, for example, *1callH+SL+Zip* took 2,318

Table 6. Precision and scalability comparison between 1-hybrid-object sensitivity with tunneling (*S1objH+T*) against its simulated call-site sensitivity (*1callH+S'*) and the call-site sensitivity obtained from learning (*1callH+SL'*). The columns Time(s) of *1callH+S'* present sum of its baseline analysis cost(s) (e.g., *S1objH+T*) and the simulated call-site sensitive analysis cost(s).

Program	Analysis	#fail-casts	Times(s)	Program	Analysis	#fail-casts	Time(s)
luindex	*1callH+SL'*	366	36	lusearch	*1callH+SL'*	380	35
	1callH+S'	359	67		*1callH+S'*	373	69
	S1objH+T	371	32		*S1objH+T*	380	33
antlr	*1callH+SL'*	486	54	eclipse	*1callH+SL'*	586	40
	1callH+S'	479	95		*1callH+S'*	573	77
	S1objH+T	483	48		*S1objH+T*	586	36
xalan	*1callH+SL'*	576	68	chart	*1callH+SL'*	942	67
	1callH+S'	562	123		*1callH+S'*	861	139
	S1objH+T	572	59		*S1objH+T*	876	62
bloat	*1callH+SL'*	1,267	456	jython	*1callH+SL'*	856	194
	1callH+S'	1,248	870		*1callH+S'*	836	8,985
	S1objH+T	1,251	375		*S1objH+T*	837	342
jpc	*1callH+SL'*	1,676	131	pmd	*1callH+SL'*	721	59
	1callH+S'	1,644	3,002		*1callH+S'*	710	50
	S1objH+T	1,593	186		*S1objH+T*	713	52
fop	*1callH+SL'*	1,084	123	checkstyle	*1callH+SL'*	466	63
	1callH+S'	1,055	959		*1callH+S'*	468	135
	S1objH+T	1,080	94		*S1objH+T*	474	69

seconds while *1callH+SL* took 3,766 seconds. One exception is batik, where *1callH+SL+Zip* is slower than *1callH+SL*. This is because *1callH+SL+Zip* in this case loses precision significantly (reporting 180 more alarms than *1callH+SL*), producing spurious points-to sets that make the analysis slow.

5.4 Applicability to Variations of Object Sensitivity

In this paper, we focused on the original object sensitivity [Milanova et al. 2002] as it is the most widely known and used (e.g., [Feng et al. 2014; Gordon et al. 2015; Li et al. 2018a,a,b; Lu and Xue 2019; Smaragdakis et al. 2014; Tan et al. 2016]). Therefore, though we believe our claim holds for variations of object sensitivity as well (e.g., hybrid context sensitivity [Kastrinis and Smaragdakis 2013], type sensitivity [Smaragdakis et al. 2011]), we do not claim that Obj2Cfa in its present form is readily applicable to them. Note that we have designed Obj2Cfa by exploiting the properties of original object sensitivity (e.g., call-graph patterns and atomic features). To apply Obj2Cfa to its variations, we may need domain-specific tuning of the simulation and learning techniques (e.g., the inference rules in Section 4.1 and feature engineering in Section 4.2). It would be interesting future work to generalize our results for other variants of object sensitivity.

When we simply used Obj2Cfa to hybrid object sensitivity (in the setting of Section 5.1), for example, we found that the results are encouraging yet suboptimal. Hybrid context sensitivity [Kastrinis and Smaragdakis 2013] is a variant of object sensitivity that selectively combines call-site and object sensitivity, and the state-of-the-art is the context-tunneled version, *S1objH+T* [Jeon et al. 2018]. We applied Obj2Cfa to *S1objH+T* and transformed it into a call-site-sensitive analysis. We compared the performance of *S1objH+T* with the simulated call-site sensitivity without learning

(denoted *1callH+S'*), and the final analysis with learning (denoted *1callH+SL'*). Table 6 shows that, though suboptimal, our simulation technique overall makes hybrid context sensitivity more precise (*1callH+S'* vs. *S1objH+T*), indicating that hybrid context sensitivity can benefit from our approach. However, the learned analysis (*1callH+SL'*) shows overall worse precision than hybrid context sensitivity. The failure of learning comes from the lack of atomic features appropriate for hybrid object sensitivity. We leave adaptation to hybrid object sensitivity as future work.

6 RELATED WORK

Context Tunneling. Our work builds on the prior work by Jeon et al. [2018], where the ideas of context tunneling and learning tunneling policies were first proposed. However, our main message (call-site sensitivity can be superior to object sensitivity) as well as the simulation and simulation-guided learning techniques are entirely original in this work.

Object Sensitivity. Our work deviates significantly from past works on improving context-sensitive analyses for object-oriented programs. While almost all past works took the "safe" direction of building upon object sensitivity [Jeong et al. 2017; Li et al. 2018a,b; Liang and Naik 2011; Liang et al. 2011; Lu and Xue 2019; Smaragdakis et al. 2011, 2014; Sridharan et al. 2012; Tan et al. 2016], our work calls for attention to call-site sensitivity.

Over the past decades, a large amount of research has been devoted to improving object sensitivity. For example, Smaragdakis et al. [2011] proposed a variant of object sensitivity, called type sensitivity, which is more scalable than object sensitivity with little compromise on precision. Sridharan et al. [2012] proposed parameter sensitivity that uses parameter values as contexts rather than the values of receiver objects. Tan et al. [2016] presented a technique to make k-object-sensitive analyses more precise without increasing k. Thiessen and Lhoták [2017] proposed a new variation of object sensitivity that combines CFL-reachability and k-limited approach. Xu and Rountev [2008] improved scalability of object sensitivity by identifying and merging equivalent contexts.

Selective object sensitivity has been particularly popular for boosting object sensitivity [Jeong et al. 2017; Li et al. 2018a,b; Liang and Naik 2011; Liang et al. 2011; Lu and Xue 2019; Smaragdakis et al. 2014; Wei and Ryder 2015]. As applying deep object sensitivity to all methods does not scale to large programs, several techniques have been proposed to apply deeper contexts only to a set of methods that are likely to benefit. For example, Smaragdakis et al. [2014] proposed a technique that runs a pre-analysis and identifies the methods that should not receive context sensitivity. Jeong et al. [2017] developed a machine learning algorithm that can produce method-selection heuristics automatically and showed that the data-driven approach outperforms existing hand-crafted approaches. As we showed in Section 5.3, selective context sensitivity is effective at improving the scalability of call-site sensitivity as well [Li et al. 2020; Oh et al. 2014; Smaragdakis et al. 2014], which would make our technique more practical.

Call-Site Sensitivity. Call-site sensitivity has received little attention in static analysis of object-oriented languages. However, call-site sensitivity has been studied extensively in other contexts [Dan et al. 2017; Khedker and Karkare 2008; Khedker et al. 2012; Oh et al. 2014; Sridharan and Bodík 2006; Sridharan et al. 2005; Thakur and Nandivada 2019; Tripp et al. 2009; Zhang et al. 2014]. For example, several works developed techniques to run the most precise, ∞-call-site-sensitive analysis. Khedker and Karkare [2008] used value contexts to identify redundant contexts that are worthless to maintain and remove them to scale up the full call strings analysis without precision loss. In a similar context, Thakur and Nandivada [2019] proposed a way to handle the scalability issue by reducing comparison costs involved in value contexts. Another line of works aimed to improve scalability of call-site sensitivity [Dan et al. 2017; Sridharan and Bodík 2006; Sridharan

et al. 2005; Whaley and Lam 2004; Xu et al. 2009]. For example, Whaley and Lam [2004] used binary decision diagrams to represent context strings efficiently, and Sridharan and Bodík [2006] proposed a refinement-based demand-driven analysis based on CFL-reachability, where the idea is to remove client-irrelevant part of the program. It is challenging to compare ours with the demand-driven approaches [Späth et al. 2016; Sridharan and Bodík 2006] because our technique is currently formulated on top of a whole-program analysis and does not consider the CFL-reachability formulation.

Data-Driven Static Analysis. Learning analysis policies has been popular in recent works [Bielik et al. 2017; Grigore and Yang 2016; He et al. 2020; Heo et al. 2019, 2017; Jeon et al. 2019, 2020; Jeong et al. 2017; Oh et al. 2015; Peleg et al. 2016; Singh et al. 2018; Wei and Ryder 2015]. At a high-level, our work is in this line of research, but we apply learning to a new application, i.e., transforming object sensitivity to call-site sensitivity. To do so, we adapt the existing ideas [Jeon et al. 2018; Jeong et al. 2017] to learn the behavior of the simulated policy. Technically, the main difference is that ours is a supervised learning algorithm (in that the search process is guided by the simulated policy) while the existing algorithms [Jeon et al. 2018; Jeong et al. 2017] are unsupervised.

7 OPEN QUESTION: IS COMPLETE SIMULATION POSSIBLE?

In this paper, we showed that it is practically possible to outperform object sensitivity via context-tunneled call-site sensitivity. However, it remains to be seen whether or not call-site sensitivity can be fundamentally superior to object sensitivity. Let us define the notion of 'superiority' as follows:

Definition 7.1 (Superiority of Call-Site Sensitivity). Let $\mathbb{P}$ be a set of target programs. Let $\mathbb{S}$ be a context-tunneling space for the target programs. We say call-site sensitivity is superior to object sensitivity with respect to $\mathbb{S}$ if is always possible to simulate object sensitivity via call-site sensitivity:

$$\forall P \in \mathbb{P}. \forall T_{obj} \in \mathbb{S}. \exists T_{call} \in \mathbb{S}. \forall k \in [0, \infty].\ \mathit{fix} F_{P,k}^{T_{call}, U_{call}} \succeq \text{ (more precise than) } \mathit{fix} F_{P,k}^{T_{obj}, U_{obj}} \tag{5}$$

where U_{call} and U_{obj} are context-update functions (Eq. (1) and (2)) that are naturally given together with the tunneling space $\mathbb{S}$, and the precision order $\succeq$ is defined in terms of the context-insensitive points-to sets, as follows:

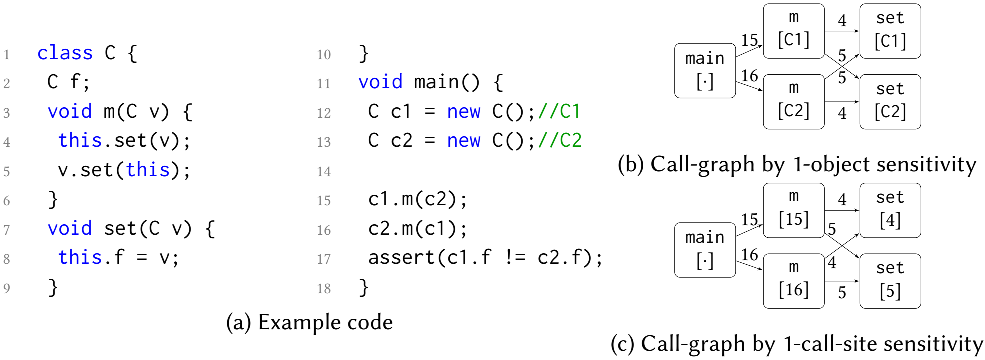
$$\mathit{fix} F_{P,k}^{T,U} \succeq \mathit{fix} F_{P,k}^{T',U'} \iff \forall x \in \mathit{Var}_P.\ \Pi(\mathit{fix} F_{P,k}^{T,U})(x) \subseteq \Pi(\mathit{fix} F_{P,k}^{T',U'})(x)$$

where $\Pi(X, Y, R, G) = \lambda x.\ \bigsqcup_{ctx \in Ctx} \{h \mid (h, hctx) \in X(x, ctx)\}$.

Then, the question is restated as follows: Is there a context-tunneling space $\mathbb{S}$ that makes the condition (5) true?

Object Sensitivity is Not Fundamentally Superior. We first show that object sensitivity is not fundamentally superior to call-site sensitivity. That is, it is not possible to find a tunneling abstraction space $\mathbb{S}$ that makes k-object sensitivity with tunneling always equal or more precise than k-call-site sensitivity with tunneling. The example code in Figure 1a is a universal counter-example for all tunneling spaces $\mathbb{S}$. By definition of context tunneling (i.e., selective update of contexts), the method `id` in the example (Figure 1a) can have `[D]` (updating context) or `[·]` (inheriting context) as a context no matter what tunneling abstraction space is used. Thus, k-object sensitivity with tunneling is unable to separate the three method calls (invoked at the three different invocation sites) with the two available context choices. In summary, we conclude that object sensitivity cannot simulate call-site sensitivity even in the generalized setting with context tunneling.

The Case for Call-Site Sensitivity. On the other hand, the situation for call-site sensitivity is nontrivial and it remains to be seen whether or not call-site sensitivity can simulate object sensitivity completely. Let us explain in more detail with examples.



(a) Example code

(b) Call-graph by 1-object sensitivity

(c) Call-graph by 1-call-site sensitivity

Fig. 7. Example such that call-site sensitivity cannot simulate object sensitivity w.r.t. tunneling space $\mathbb{S} = \mathit{Invo}$.

First, we point out that that the condition (5) does not hold w.r.t. the tunneling space considered in this paper. In Section 3, we defined the tunneling space $\mathbb{S}$ to be the set of invocation-sites, i.e., $\mathbb{S} = \mathit{Invo}$. With this space, there exists a tricky counter-example program that context-tunneled call-site sensitivity is unable to simulate object sensitivity. Consider the program in Figure 7. In the example code, class `C` contains two methods, `set` and `m`. Method `set` is a setter that stores the parameter value in the the field of the base object, and method `m` calls the `set` method twice with the receiver object and parameter being swapped. The `main` method creates two objects, `C1` and `C2`, and stores them in `c1` and `c2`. Method `m` is called on `c1` with parameter `c2` at line 15, and it is called again on `c2` with parameter `c1` at line 16. At line 17, an assertion asks if the fields of `c1` and `c2` are not aliased. In the real execution, the query holds because `c1.f` points to `C2` and `c2.f` points to `C1`.

The conventional 1-object-sensitive analysis can prove the query but 1-call-site sensitivity cannot do so no matter what tunneling abstraction from the space $\mathbb{S} = \mathit{Invo}$ is chosen. Figure 7b presents the call-graph of 1-object sensitivity. The object-sensitive analysis is effectively producing the call-graph above as each context of `set` determines the value of each parameter. When the context is `[C1]`, the receiver object is `C1` and the value of the parameter is `C2`. Otherwise, when the context is `[C2]`, the receiver object and the parameter value are `C2` and `C1`, respectively. On the other hand, conventional 1-call-site sensitivity constructs a similar but different call-graph in Figure 7c. Note that the two edges labeled 4 go to the same method but they are heading to different methods in the call-graph of object sensitivity. Thus, it is not possible to simulate object sensitivity with the invocation-site-based context tunneling.

However, this counter-example does not mean that it is fundamentally impossible to simulate object sensitivity via call-site sensitivity, as the counter-example becomes no longer valid if we use a more fine-grained tunneling abstraction. For example, suppose we define the tunneling space to be pairs of receiver objects and invocation-sites, i.e., $\mathbb{S} = \mathit{Heap} \times \mathit{Invo}$ (recall that choosing a tunneling abstraction does not affect the analysis soundness). With this tunneling space, a context-tunneled 1-call-site-sensitive analysis can now prove the query. Suppose we use a tunneling abstraction $T = \{(C1, 4), (C1, 5)\}$, which means that we apply context tunneling only when the receiver object is $C1$ and the invocation-site is either 4 or 5. 1-call-site sensitivity with T produces the call-graph in Figure 8. In the call-graph, `m[15]` $\overset{(C1,4)}{\rightarrow}$ `set[15]` indicates that the caller (`m`) and callee (`set`) have the same context 15 where the callee method is called at invocation-site 4 and its receiver object is $C1$. With this call-graph, we can prove the query in Figure 7a as it is strictly more precise than the call-graph produced by object sensitivity (Figure 7b).

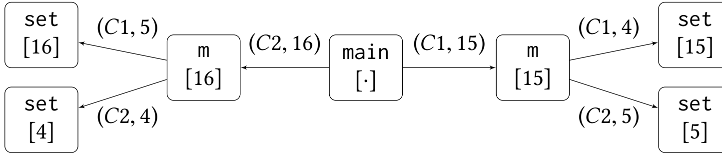


Fig. 8. Call-graph with a fine-grained tunneling space

This way, we conjecture that it would be always possible to find a suitable context-tunneling space $\mathbb{S}$ that satisfies the condition (5) w.r.t. the given set of programs ($\mathbb{P}$). We leave an in-depth theoretical analysis as future work.

8 CONCLUSION AND FUTURE WORK

Unfortunately, the program analysis community for object-oriented programs has dismissed call-site sensitivity for a long time. In this paper, showed that call-site sensitivity has vast untapped potential, even more than object sensitivity, when the notion of k-limiting is generalized. We provided an insight that call-site sensitivity with context tunneling can simulate object sensitivity and experimentally proved that the observation holds in practice by developing a technique to transform a baseline object-sensitive analysis into more precise, context-tunneled call-site sensitivity. Based on our results, we hope that the community reconsiders call-site sensitivity from now on.

Many problems remain as future work. We already discussed a theoretical issue in Section 7. Other problems include the following.

- *Can we learn better tunneling strategies than just simulating object sensitivity?* Our goal in this paper was to show that call-site sensitivity can be superior to object sensitivity, and simulating object sensitivity was an effective means of achieving this goal. However, simulating object sensitivity would be a suboptimal strategy for call-site sensitivity; we believe an optimal tunneling strategy would enable call-site sensitivity to show far better precision than ours (*1callH+SL*). Thus, an interesting direction for future work is to develop a powerful learning algorithm to find such strategies, where the main challenge is how to efficiently explore the huge and non-monotone tunneling space [Jeon et al. 2018]. Using reinforcement learning, for example, could be a promising approach to address this challenge.
- *Can our approach be adapted for other flavors of context-sensitive analyses?* Our current simulation technique relies on properties specific to k-CFA (e.g., I_2 in Section 4.1 leverages a unique property of context-tunneled k-CFA). However, the high-level idea (i.e., simulating object sensitivity) would be applicable to other analyses. For example, it would be interesting if our idea could be adapted for m-CFA [Might et al. 2010].

ACKNOWLEDGMENTS

We thank the anonymous POPL reviewers for their constructive feedback. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair) and by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2021-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation), and by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944).

REFERENCES

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 233–253.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Andrei Marian Dan, Manu Sridharan, Satish Chandra, Jean-Baptiste Jeannin, and Martin Vechev. 2017. Finding Fix Locations for CFL-Reachability Analyses via Minimum Cuts. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 521–541.

Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

Radu Grigore and Hongseok Yang. 2016. Abstraction Refinement Guided by a Learnt Probabilistic Model. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 485–498. https://doi.org/10.1145/2837614.2837663

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning Fast and Precise Numerical Analysis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. https://doi.org/10.1145/3385412.3386016

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware Program Analysis via Online Abstraction Coarsening. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 94–104. https://doi.org/10.1109/ICSE.2019.00027

Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 519–529. https://doi.org/10.1109/ICSE.2017.54

Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (June 2019), 41 pages. https://doi.org/10.1145/3293607

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428247

Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven Context-sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133924

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228.

Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. 2012. Liveness-Based Pointer Analysis. In *Static Analysis*, Antoine Miné and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–282.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction*, Alan Mycroft and Andreas Zeller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 47–64.

Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018a. Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276511

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (May 2020), 40 pages. https://doi.org/10.1145/3381915

Donglin Liang, Maikel Pennings, and Mary Jean Harrold. 2005. Evaluating the Impact of Context-sensitivity on Andersen's Algorithm for Java Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal) *(PASTE '05)*. ACM, New York, NY, USA, 6–12. https://doi.org/10.1145/1108792.1108797

Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 590–601. https://doi.org/10.1145/1993498.1993567

Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning Minimal Abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/1926385.1926391

Jingbo Lu and Jingling Xue. 2019. Precision-preserving Yet Fast Object-sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 572–588. https://doi.org/10.1145/2814270.2814309

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

Hila Peleg, Sharon Shoham, and Eran Yahav. 2016. D^3: Data-Driven Disjunctive Abstraction. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 185–205.

Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis.* Prentice-Hall, Englewood Cliffs, NJ, Chapter 7, 189–234.

O. Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/53990.54007

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham,

211–229.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-to Analysis of Javascript. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 435–458. https://doi.org/10.1007/978-3-642-31057-7_20

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 263–277. https://doi.org/10.1145/3062341.3062359

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

Guoqing Xu and Atanas Rountev. 2008. Merging Equivalent Contexts for Scalable Heap-cloning-based Context-sensitive Points-to Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/1390630.1390658

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–122.

X. Xu, Y. Sui, H. Yan, and J. Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 512–523.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327