Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling

MINSEOK JEON, Korea University, Republic of Korea
SEHUN JEONG, Korea University, Republic of Korea
HAKJOO OH*, Korea University, Republic of Korea

We present context tunneling, a new approach for making k-limited context-sensitive points-to analysis precise and scalable. As context-sensitivity holds the key to the development of precise and scalable points-to analysis, a variety of techniques for context-sensitivity have been proposed. However, existing approaches such as k-call-site-sensitivity or k-object-sensitivity have a significant weakness that they unconditionally update the context of a method at every call-site, allowing important context elements to be overwritten by more recent, but not necessarily more important, context elements. In this paper, we show that this is a key limiting factor of existing context-sensitive analyses, and demonstrate that remarkable increase in both precision and scalability can be gained by maintaining important context elements only. Our approach, called context tunneling, updates contexts selectively and decides when to propagate the same context without modification.

We attain context tunneling via a data-driven approach. The effectiveness of context tunneling is very sensitive to the choice of important context elements. Even worse, precision is not monotonically increasing with respect to the ordering of the choices. As a result, manually coming up with a good heuristic rule for context tunneling is extremely challenging and likely fails to maximize its potential. We address this challenge by developing a specialized data-driven algorithm, which is able to automatically search for high-quality heuristics over the non-monotonic space of context tunneling.

We implemented our approach in the Doop framework and applied it to four major flavors of context-sensitivity: call-site-sensitivity, object-sensitivity, type-sensitivity, and hybrid context-sensitivity. In all cases, 1-context-sensitive analysis with context tunneling far outperformed deeper context-sensitivity with $k = 2$ in both precision and scalability.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computing methodologies → Machine learning approaches**;

Additional Key Words and Phrases: Points-to analysis, Context-sensitive analysis, Data-driven program analysis

ACM Reference Format:
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 (November 2018), 30 pages. https://doi.org/10.1145/3276510

*Corresponding author

Authors' addresses: Minseok Jeon, minseok_jeon@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; Sehun Jeong, gifaranga@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; 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.

2475-1421/2018/11-ART140
https://doi.org/10.1145/3276510

1 INTRODUCTION

Points-to analysis is a fundamental program analysis technique with diverse applications in software engineering tasks. The goal of points-to analysis is to safely yet accurately approximate the objects that pointer variables may point to at runtime, as precise pointer information is a key ingredient of modern software analysis techniques such as static bug-finding [Avots et al. 2005; Blackshear et al. 2015; Livshits and Lam 2003; Naik et al. 2006; Sui et al. 2014], program verification [Fink et al. 2008], security vulnerability detection [Arzt et al. 2014; Tripp et al. 2009; Yan et al. 2017], and automatic program repair [Gao et al. 2015; Lee et al. 2018]. The effectiveness and usefulness of these techniques are ultimately determined by the qualities of the underlying points-to analysis.

Context-sensitivity holds the key to the development of precise and scalable points-to analysis. In order to accurately track local variables and heap objects, a context-sensitive analysis treats multiple calls to the same method separately for its different calling contexts. For object-oriented and functional languages, context-sensitivity is the single most important factor that affects both precision and scalability. It has greater impact on the analysis precision than other techniques such as flow-sensitivity [Lhoták and Hendren 2006; Smaragdakis and Balatsouras 2015], and the improved precision by context-sensitivity is likely to increase scalability as well [Kashyap et al. 2014; Smaragdakis et al. 2011].

Consequently, developing effective approaches to context-sensitivity has been the major goal of research in points-to analysis. In particular, one practical approach to context-sensitivity is the so-called k-limited method [Sharir and Pnueli 1981] and over the last decades researchers have explored various flavors and policies with different tradeoffs, such as object-sensitivity [Milanova et al. 2005], type-sensitivity [Smaragdakis et al. 2011], hybrid context-sensitivity [Kastrinis and Smaragdakis 2013], and selective context-sensitivity [Jeong et al. 2017; Oh et al. 2014; Smaragdakis et al. 2014]. These approaches vary depending on the program entities that they use as context elements or policies to assign k values. Unlike the classical k-call-site-sensitivity [Shivers 1988], object- and type-sensitivity use a sequence of allocation-sites and a sequence of allocating-classes as context elements, respectively, providing new sweet spots for object-oriented languages. Hybrid context-sensitivity provides a way of combining call-site-sensitivity and object-sensitivity to enjoy benefits of both approaches. Selective context-sensitive analyses assign different k values to different methods.

In this paper, we present context tunneling, a new approach for performing precise and scalable k-limited context-sensitive analysis. The key observation behind our approach is that, although existing approaches for context-sensitivity differ in various characteristics such as flavors [Kastrinis and Smaragdakis 2013; Milanova et al. 2005; Shivers 1988; Smaragdakis et al. 2011] and policies to assign k values [Jeong et al. 2017; Oh et al. 2014; Smaragdakis et al. 2014], they all have a significant weakness in common: they update the context of a method unconditionally at *every* call-site and therefore important context elements are overwritten by more recent, but not necessarily more important, context elements. We found that this is a key factor that limits the effectiveness of the existing context-sensitive analyses, and show that dramatic increase in both precision and scalability can be gained by maintaining important context elements only. Our technique updates contexts selectively and decides when to preserve contexts without modification. We formalize our technique in a general setting, so that it is applicable to all major flavors of context-sensitivity.

We also present a new data-driven approach for realizing effective context tunneling. Although the potential of context tunneling is tremendous, maximizing its impact in practice is nontrivial. This is mainly because precision increase by context tunneling is very sensitive to the choice of important context elements and even not monotone with respect to the ordering of the choices. As a result, manually coming up with a good heuristic rule for context tunneling is extremely challenging and likely ends up with suboptimal results. To address this challenge, we leverage the recent trend of data-driven program analysis [Chae et al. 2017; Heo et al. 2016, 2017; Jeong et al. 2017; Oh et al. 2015; Singh et al. 2018; Wei and Ryder 2015] and aim at automatically learning a context-tunneling heuristic from data. Given a dataset of programs and a set of method features, our learning algorithm generates a heuristic that determines the context elements that contribute to improving the final analysis performance. Our algorithm is carefully designed for context tunneling, so it is able to produce effective heuristics over the non-monotonic space of context tunneling.

We implemented our approach on top of the Doop framework [Bravenboer and Smaragdakis 2009] and evaluated it with four major flavors of context-sensitivity: call-site-sensitivity [Shivers 1988], object-sensitivity [Milanova et al. 2002, 2005], type-sensitivity [Smaragdakis et al. 2011], and hybrid context-sensitivity [Kastrinis and Smaragdakis 2013]. In all analyses, context tunneling has improved the performance of the existing analyses remarkably. In particular, context tunneling enabled 1-context-sensitive analyses to far outperform 2-context-sensitive counterparts in both scalability and precision. We also demonstrate that the effectiveness of context tunneling comes from our learning algorithm; we evaluate our learning algorithm from various perspectives and justify the design decisions underlying the algorithm.

Contributions. In summary, our contributions are as follow:

- We present context tunneling, a new technique for improving k-limited context-sensitive points-to analysis by carefully maintaining important context elements.
- We present a new data-driven approach for learning context tunneling heuristics. Key technical contribution is the non-greedy learning algorithm that is able to effectively search for good heuristics over non-monotonic parameter space of context tunneling.
- We extensively evaluate the effectiveness of context tunneling and our learning approach with for four flavors of context sensitive analysis for Java.

2 MOTIVATING EXAMPLES

In this section, we illustrate our technique with examples. We describe our technique with k-call-site-sensitivity and k-object-sensitivity, but it is applicable to other flavors of context-sensitivity too. The general description is presented in Section 3.

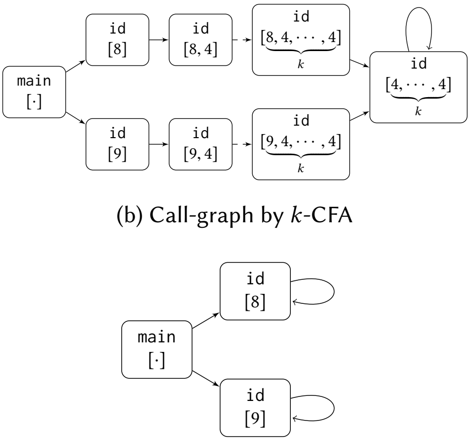
In this section, we restrict our discussion to the traditional k-limited context-sensitive analysis. Other approaches such as value-based context-sensitivity [Khedker and Karkare 2008; Padhye and Khedker 2013] may not suffer from the issue described in this section. However, their applicability to real-world points-to analysis remains to be seen, which is beyond the scope of this paper. On the other hand, the k-limited method is more widely used in practical settings [Bravenboer and Smaragdakis 2009; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011, 2014].

2.1 Call-Site-Sensitivity

Example Code. Suppose that we analyze the example Java code in Fig. 1a using a k-call-site-sensitive points-to analysis [Smaragdakis and Balatsouras 2015]. The example code has three class definitions, namely `A`, `B`, and `C`. Classes `A` and `B` are empty and class `C` has two methods: `id` and `main`. Method `id` is essentially the identity function on the first argument, which takes an object `v` and an integer `i`, and returns the same object after making recursive calls until `i` becomes 0. In `main`, `id` is invoked twice with different objects. Both method invocations share the same argument `i` that comes from the external environment (i.e. user input). The goal of the points-to analysis is to prove the safety of two type casts at lines 8 and 9.

```
 1 class A {} class B {}
 2 class C {
 3   static Object id (Object v, int i){
 4       return i >= 0 ? id(v, i-1) : v;
 5   }
 6   public static void main (){
 7     int i = input();
 8     A a = (A) id(new A(), i); //Query 1
 9     B b = (B) id(new B(), i); //Query 2
10   }
11 }
```

(a) Example code



(b) Call-graph by k-CFA

(c) Call-graph by 1-CFA with tunneling

Fig. 1. Example and call-graphs constructed by the conventional k-call-site-sensitivity (k-CFA) and our 1-call-site-sensitivity with context tunneling

Conventional Call-Site-Sensitivity. The traditional k-call-site-sensitive analysis fails to prove the queries no matter what k value is used. Because the value of i can be any integer, the following set K_∞ of infinite number of call-strings can be generated for method id at runtime:

$$K_\infty = \{[8], [9], [8, 4], [9, 4], [8, 4, 4], [9, 4, 4], [8, 4, 4, 4], [9, 4, 4, 4], \ldots\}$$

where, for example, $[8, 4]$ denotes the context that method id was called along the sequence of call-sites 8 and 4. The k-call-site-sensitive analysis approximates the call-strings in K_∞ by their suffixes of length up to k. For example, when $k = 2$, the analysis uses the following set K_2 of abstract contexts:

$$K_2 = \{[8], [9], [8, 4], [9, 4], [4, 4]\}$$

where an infinite number of contexts, $\{[8, 4, 4], [9, 4, 4], [8, 4, 4, 4], [9, 4, 4, 4], \ldots\}$, in K_∞ are approximated by their common suffix $[4, 4]$. The analysis analyzes the method id separately for each context in K_2. Although this approximation ensures termination, the analysis is now unable to differentiate the two separate calls to id at lines 8 and 9, causing the argument v of id to point to both objects A and B at the same time when the context becomes $[4, 4]$. Thus, both objects get returned to both queries simultaneously, making the analysis fail to prove their cast safety. Note that the analysis fails to prove the queries for any k values, because all the context strings longer than k are eventually merged into a single context $[\underbrace{4, \ldots, 4}_{k}]$ (see Fig. 1b).

Context Tunneling. With context tunneling, however, even 1-call-site sensitive analysis becomes to prove the queries. The main weakness of the conventional k-context-sensitive analysis is that it blindly updates the context of a method at every call-site, thereby allowing important context elements for the method to be easily overwritten by less important ones. In the example, distinguishing the two call-sites at lines 8 and 9 is essential to proving the queries. However, this information is eventually lost by repeatedly appending the less important call-site 4 to the context of id (Fig. 1b). Our technique aims to overcome this weakness by maintaining important context elements only during analysis. For example, we exclude the call-site 4 from the context strings generated for method id and thus approximate the set K_∞ to the set K_T of contexts:

$$K_T = \{[8], [9]\}.$$

Here abstract contexts [8] and [9] in K_T denote the sets of concrete contexts $\{[8], [8,4], [8,4,4], \ldots\}$ and $\{[9], [9,4], [9,4,4], \ldots\}$ in K_∞, respectively. The resulting context-sensitive analysis with K_T is able to prove the queries as it differentiates the two calls to id at lines 8 and 9 completely, producing the call-graph in Fig. 1c.

Challenge. The example shows that the idea of context tunneling is potentially powerful; even 1-context-sensitivity with tunneling can be as precise as ∞-context-sensitivity. The goal of this paper is to maximize this tremendous, yet untapped, potential of the existing k-context-sensitive analysis. However, achieving effective context tunneling in practice is challenging. The effectiveness of the technique is very sensitive to the choice of important context elements. For example, choosing the call-site 4, rather than call-sites 8 and 9, does not work for the example program. In the worst case, the analysis precision degenerates into the context-insensitive case (e.g. when selecting no context elements at all), becoming even inferior to the ordinary k-context-sensitive analysis. Coming up with a good heuristic rule for choosing important context elements is nontrivial; hand-crafting such a rule not only requires a huge amount of engineering effort and domain knowledge but also likely fails to maximize the potential of the technique.

Data-Driven Context Tunneling. We address this challenge with a data-driven approach that automatically learns a good heuristic rule for context tunneling from a training set of programs. Our aim is to generate a heuristic $\mathcal{H}$, which takes a program and returns a relation $\mathcal{T}$ on methods. The relation contains a method pair (m_1, m_2) if the calling context of m_1 is more important than that of m_2. Thus, if $(m_1, m_2) \in \mathcal{T}$, the analysis applies context tunneling when m_2 is invoked; that is, m_2 inherits the context of m_1 ignoring the current context element. For the k-call-site-sensitivity example, $\mathcal{H}_\Pi$ produces the relation $\mathcal{T} = \{(\texttt{id}, \texttt{id})\}$, which means that tunneling is applied when id is called from id. As a result, the call-site 4, on which id is called from id, will not be used as important context elements during analysis. When main calls id, however, the analysis updates the contexts as the ordinary analysis since $(\texttt{main}, \texttt{id}) \notin \mathcal{T}$. In our approach, the heuristic is parameterized and the parameter determines the contents of the relation $\mathcal{T}$. The goal of our data-driven technique is to characterize the two classes of methods: 1) methods that increase the analysis precision by passing their contexts to others, and 2) methods that increase the analysis precision by inheriting contexts from others. Section 4 defines the learning problem and presents an efficient algorithm to solve the problem.

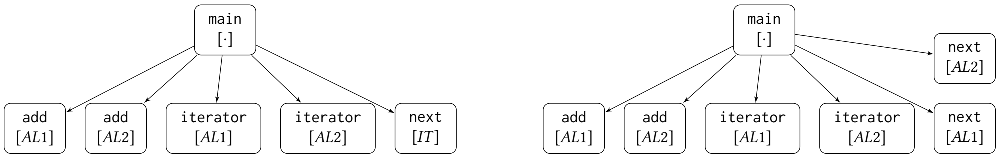
2.2 Object-Sensitivity

The use of context tunneling is not limited to call-site-sensitivity. Below, we describe a typical situation where object-sensitivity benefits from context tunneling.

Example Code. Fig. 2a shows a common code pattern that 1-object-sensitivity with a context-sensitive heap loses precision, which is found frequently in data structure implementations such as List and Map. In the example code, we have four top-level classes, A, B, C, and ArrayList, and an inner class ListIter inside of the ArrayList class. The ArrayList class implements a resizable array. For simplicity, it maintains data in the elementData array and has the add method to append new data at the end of elementData. It also has the iterator method that returns an object of type ListIter, providing access to elementData. The main method in class C performs usual allocate-add-retrieve tasks using ListIter. The goal of the points-to analysis is to prove the safety of two type casts at lines 13 and 14.

```
 1 class A {} class B {}
 2 class C {
 3   public static void main (){
 4     ArrayList al1 = new ArrayList();//AL1
 5     ArrayList al2 = new ArrayList();//AL2
 6
 7     al1.add(new A());
 8     al2.add(new B());
 9
10     ArrayList.ListItr it1 = al1.iterator();
11     ArrayList.ListItr it2 = al2.iterator();
12
13     A a = (A)it1.next(); //Query 1
14     B b = (B)it2.next(); //Query 2
15   }
16 }
17
18 class ArrayList{
19   Object[] elementData = new Object[10];
20   int size = 0;
21   void add(Object e){
22     elementData[size++] = e;
23   }
24   ListItr iterator(){
25     return new ListItr(); //IT
26   }
27   class ListItr{
28     int cursor = 0;
29     Object next(){
30       return elementData[cursor++];
31     }
32   }
33 }
```

(a) Example code



(b) Call-graph by 1-object-sensitive analysis

(c) Call-graph by 1-object-sensitive analysis with tunneling

Fig. 2. Example and call-graphs constructed by the conventional 1-object-sensitivity and our 1-object-sensitivity with context tunneling

Conventional Object-Sensitivity. Conventional object-sensitivity analyzes different method calls separately for their base objects (and their heap contexts, if needed) [Smaragdakis et al. 2011]. For example, when we analyze the `next` method at line 13, 2-object-sensitivity creates the context [`AL1`, `IT`], where `IT` is the allocation-site of the base object (`it1`) and `AL1` is its heap context. Likewise, the method invocation at line 14 creates the context [`AL2`, `IT`], enabling the analysis to distinguish the first and second calls to the same method. Using base objects as contexts makes object-sensitivity more favorable than call-site-sensitivity for object-oriented languages. However, it still updates contexts unconditionally at every method call and may lose important context elements when k is not sufficiently large. For example, when we analyze the code with 1-object-sensitivity, the two calls to `next` are analyzed under the same context [`IT`] (Fig. 2b). As a result, both objects `A` and `B` get returned to both queries simultaneously, making the analysis fail to prove the safety of type casting.

Object-Sensitivity with Context Tunneling. With context tunneling, however, even 1-object-sensitivity can prove the queries. Note that object-sensitivity has a different flavor from call-site-sensitivity; it updates the context with the heap context of the base object [Milanova et al. 2002, 2005]. Base variables `it1` and `it2` point to the same heap allocation with different heap contexts: `AL1` and `AL2`. Applying context tunneling for the two method calls to `next` corresponds to propagating the heap contexts `AL1` and `AL2` at lines 13 and 14, respectively, without modification. The resulting context-sensitive analysis proves the queries with a call-graph shown in Fig. 2c.

3 POINTS-TO ANALYSIS WITH CONTEXT TUNNELING

In this section, we formally describe the idea of context tunneling. In Section 3.1, we describe the conventional k-context-sensitive points-to analysis in a general setting, so that context tunneling in Section 3.2 is applicable to various flavors of context-sensitivity.

3.1 Conventional k-Context-Sensitive Analysis

Generic Analysis. We first formalize the well-known context-sensitive points-to analysis for Java [Smaragdakis and Balatsouras 2015], which is a flow-insensitive, field-sensitive, and subset-based analysis with on-the-fly call-graph construction. To reveal the core idea of context tunneling, we abstract the baseline analysis so that it is generic in terms of both flavors and depths of context-sensitivity. For example, the analysis is able to express various context-sensitivity flavors such as call-site-sensitivity [Shivers 1988], object-sensitivity [Milanova et al. 2005], hybrid context-sensitivity [Kastrinis and Smaragdakis 2013], and type-sensitivity [Smaragdakis et al. 2011] with arbitrary depths for calling contexts and heap contexts. In our formalism, such variations are characterized by the following data:

$$\langle \mathit{flavor}, \mathit{maxK}, \mathit{maxH} \rangle \quad (\mathit{maxK} \geq \mathit{maxH})$$

where *flavor* is a function that characterizes the flavor of context-sensitivity (e.g. call-site-sensitivity, object-sensitivity, etc). *maxK* and *maxH* represent the maximum depths of calling and heap contexts, respectively. For example, $\langle \mathit{flavor}_{call}, 1, 0\rangle$ defines the 1-call-site-sensitive analysis with context-insensitive heap and $\langle \mathit{flavor}_{obj}, 2, 1\rangle$ defines the 2-object-sensitive analysis with 1-context-sensitive heap, and so on. We will shortly define the *flavor* function for call-site-sensitivity (flavor_{call}), object-sensitivity (flavor_{obj}), and type-sensitivity (flavor_{type}).

Programs and Domains. We consider the five types of instructions in Java:

$$\mathit{linst} \rightarrow l : \mathit{inst}, \quad \mathit{inst} \rightarrow x = \mathit{new}\ C() \mid x = y \mid x = y.f \mid x.f = y \mid x = y.\mathit{sig}(\mathit{arg})$$

where each instruction is associated with a unique label. A labeled instruction (*linst*) consists of a label (l) and an instruction (*inst*). An instruction is object allocation ($x = \mathit{new}\ C()$), move ($x = y$), load ($x = y.f$), store ($x.f = y$), or method invocation ($x = y.\mathit{sig}(\mathit{arg})$). For simplicity, we assume every method has a single argument (except for the object itself).

The analysis uses the following domains: $\mathbb{V}$ is a set of program variables, $\mathbb{M}$ is a set of method identifiers, $\mathbb{S}$ is a set of method signatures (method name and type), $\mathbb{F}$ is a set of field names, $\mathbb{I}$ is a set of instruction labels, $\mathbb{H}$ is a set of abstract heaps, i.e., allocation-sites ($\mathbb{H} \subseteq \mathbb{I}$), $\mathbb{T}$ is a set of class types, $\mathbb{C}$ is a set of calling contexts, and $\mathbb{HC}$ is a set of heap contexts. $\mathbb{C}$ and $\mathbb{HC}$ will be defined depending on the flavor of context-sensitivity.

In addition, we assume four auxiliary functions: *methof*, *typeof*, *classof*, and *lookup*. $\mathit{methof} \in \mathbb{I} \rightarrow \mathbb{M}$ maps an instruction label to the method containing the instruction. $\mathit{typeof} \in \mathbb{H} \rightarrow \mathbb{T}$ maps an allocation-site to the type of the allocated object. $\mathit{classof} : \mathbb{I} \rightarrow \mathbb{T}$ maps an instruction to its containing class. $\mathit{lookup} \in \mathbb{T} \times \mathbb{S} \rightarrow \mathbb{M}$ maps a pair of a class type and a signature to the method that matches the signature inside the class. Given a method m, we write m_{this}, m_{param}, m_{return} for the 'this' variable, formal parameter, and return variable of the method, respectively.

Analysis Output. Given a program, the analysis computes the following information:

- $\mathit{ptsto} : \mathbb{V} \times \mathbb{C} \rightarrow \wp(\mathbb{H} \times \mathbb{HC})$
- $\mathit{fldptsto} : \mathbb{H} \times \mathbb{HC} \times \mathbb{F} \rightarrow \wp(\mathbb{H} \times \mathbb{HC})$
- $\mathit{reachable} : \mathbb{M} \rightarrow \wp(\mathbb{C})$

$$\frac{l : x = new\ C() \quad ctx \in reachable(methof(l)) \quad hctx = \lceil ctx \rceil_{maxH}}{(heap_l, hctx) \in ptsto(x, ctx)}$$

$$\frac{l : x = y \quad ctx \in reachable(methof(l))}{ptsto(y, ctx) \subseteq ptsto(x, ctx)}$$

$$\frac{l : x = y.f \quad ctx \in reachable(methof(l)) \quad (heap, hctx) \in ptsto(y, ctx)}{fldptsto(heap, hctx, f) \subseteq ptsto(x, ctx)}$$

$$\frac{l : x.f = y \quad ctx \in reachable(methof(l)) \quad (heap, hctx) \in ptsto(x, ctx)}{ptsto(y, ctx) \subseteq fldptsto(heap, hctx, f)}$$

$$\frac{\begin{array}{c} l : x = y.sig(arg) \quad ctx \in reachable(methof(l)) \\ (heap, hctx) \in ptsto(y, ctx) \quad t = typeof(heap) \quad m = lookup(t, sig) \\ (e, pctx, \_) = flavor(heap, hctx, l, ctx) \quad ctx' = \lceil pctx + e \rceil_{maxK} \end{array}}{\begin{array}{c} ctx' \in reachable(m) \quad (heap, hctx) \in ptsto(m_{this}, ctx') \\ ptsto(arg, ctx) \subseteq ptsto(m_{param}, ctx') \quad ptsto(m_{return}, ctx') \subseteq ptsto(x, ctx) \end{array}}$$

Fig. 3. Conventional context-sensitive points-to analysis

The function *ptsto* stores points-to sets for local variables: given a variable $x \in \mathbb{V}$ and a calling context $ctx \in \mathbb{C}$, $ptsto(x, ctx)$ denotes the set of heap objects and their heap contexts that x under ctx may point to. Other functions are used as intermediate information to compute *ptsto*: given $heap \in \mathbb{H}$, $hctx \in \mathbb{HC}$, and $f \in \mathbb{F}$, $fldptsto(heap, hctx, f)$ represents the set of heap objects possibly pointed to by the field of the heap object, and $reachable(m)$ records the set of calling contexts of m under which the method m is reachable from the entry of the program.

Analysis Rules. Figure 3 presents the analysis rules. The analysis aims to find the smallest functions of *ptsto*, *fldptsto*, and *reachable* that are closed under the analysis rules, where the order is defined pointwisely. Rules are defined for each instruction type. For example, the first rule is responsible for object allocation, where an abstract heap $heap_l \in \mathbb{H}$ is created from the allocation-site l. Its heap context $hctx$ is obtained from the calling context ctx of the method containing the instruction, possibly truncated so as to keep the last $maxH$ context elements, i.e., $hctx = \lceil ctx \rceil_{maxH}$. The operator $\lceil \cdot \rceil_k$ is used to truncate contexts: $\lceil \langle a_1, a_2, \ldots, a_n \rangle \rceil_k = \langle a_{n-k+1}, \ldots, a_n \rangle$.

The rules for the move, load, and store instructions are intuitive while the last rule for method invocation needs explanation. Given a method invocation $l : x = y.sig(arg)$, where l denotes the invocation-site, *sig* the method signature (name and type), and *arg* an actual argument, we identify the calling context *ctx* of the caller method, $(heap, hctx)$ the receiver object and its heap context that y points to, t the class type of the object *heap*, and m the callee method whose signature is *sig* and the enclosing class is t. Then we create a new calling context ctx' for the callee method by appending ($+$) the current context element e to the previous context *pctx*, i.e., $\lceil pctx + e \rceil_{maxK}$, possibly truncated to keep the last *maxK* context elements. We call the previous context *pctx* the "parent" context and the new one $\lceil pctx + e \rceil_{maxK}$ the "child" context.

Context-Sensitivity Flavors. The notions of context elements and parent contexts depend on the particular flavor of context-sensitivity and are defined by *flavor*. Given a heap $heap \in \mathbb{H}$, heap context $hctx \in \mathbb{HC}$, invocation-site $invo \in \mathbb{I}$, and calling context $ctx \in \mathbb{C}$ as input,

$$\frac{\begin{array}{c} l : x = y.sig(arg) \quad ctx \in reachable(methof(l)) \\ (heap, hctx) \in ptsto(y, ctx) \quad t = typeof(heap) \quad m = lookup(t, sig) \\ (e, pctx, p) = flavor(heap, hctx, l, ctx) \quad ctx' = \begin{cases} \lceil pctx +\!\!+ e \rceil_{maxK} & \text{if } (p, m) \notin \mathcal{T} \\ \lceil pctx \rceil_{maxK} & \text{if } (p, m) \in \mathcal{T} \end{cases} \end{array}}{\begin{array}{c} ctx' \in reachable(m) \quad (heap, hctx) \in ptsto(m_{this}, ctx') \\ ptsto(arg, ctx) \subseteq ptsto(m_{param}, ctx') \quad ptsto(m_{return}, ctx') \subseteq ptsto(x, ctx) \end{array}}$$

Fig. 4. Context-sensitive points-to analysis with tunneling

flavor(*heap*, *hctx*, *invo*, *ctx*) returns the current context element (*e*) to be used, the parent context (*pctx*), and the parent method (*p*) where *pctx* is used as a calling context. Various flavors of context-sensitivity can be obtained by defining *flavor* appropriately. For example, call-site-sensitivity [Shivers 1988], object-sensitivity [Milanova et al. 2005], and type-sensitivity [Smaragdakis et al. 2011] are characterized as follows:

$$flavor_{call}(heap, hctx, invo, ctx) = (invo, ctx, methof(invo)) \qquad \text{(Call-site-sensitivity)}$$
$$flavor_{obj}(heap, hctx, invo, ctx) = (heap, hctx, methof(heap)) \qquad \text{(Object-sensitivity)}$$
$$flavor_{type}(heap, hctx, invo, ctx) = (classof(heap), hctx, methof(heap)) \qquad \text{(Type-sensitivity)}$$

That is, the defining characteristic of call-site-sensitivity is that it uses invocation-sites (*invo*) as context elements, the parent context is the context (*ctx*) of the caller, and the parent method is the caller (*methof*(*invo*)). In other words, in call-site-sensitivity, the new context is created by appending the current context element to the calling context of the caller method. Object-sensitivity [Milanova et al. 2005] differs from call-site-sensitivity in both the context element and the parent context: it uses allocation-sites (*heap*) as context elements and the parent context comes from the heap context (*hctx*) of the receiver object (*heap*). The parent method is *methof*(*heap*) because the heap context (*hctx*) corresponds to the calling context of the method containing the allocation-site. Type-sensitivity [Smaragdakis et al. 2011] is similar to object-sensitivity except that it uses the containing classes (i.e. *classof*(*heap*)), instead of allocation-sites, as context elements. Hybrid context-sensitivity [Kastrinis and Smaragdakis 2013] combines call-site-sensitivity ($flavor_{call}$) and object-sensitivity ($flavor_{obj}$); it uses call-site-sensitivity at static calls and object-sensitivity at virtual calls.

3.2 Analysis with Context Tunneling

Now we describe context tunneling on top of the generic context-sensitive analysis. We assume that a binary relation $\mathcal{T}$ on methods, called tunneling relation, is given prior to the analysis:

$$\mathcal{T} \subseteq \mathbb{M} \times \mathbb{M}.$$

Intuitively, $\mathcal{T}$ relates the parent and child methods that require context tunneling. Suppose that a method m_2 is called during analysis where its parent method is m_1. If m_1 and m_2 are related by $\mathcal{T}$, i.e., $(m_1, m_2) \in \mathcal{T}$, we apply context tunneling by reusing the context of the parent method (m_1) in analyzing the child method (m_2) without creating a new context for m_2. Given a tunneling relation $\mathcal{T}$, the analysis rule for performing context tunneling is defined in Figure 4. All the existing rules except for method calls are carried over to the new analysis without any modifications. The rule for method call in Figure 3 is replaced by the rule in Figure 4, where the only difference is in the way of constructing the child context ctx'. If $(p, m) \notin \mathcal{T}$ (i.e. tunneling disabled), we update the calling context as the conventional k-context-sensitive analysis does. Otherwise, when $(p, m) \in \mathcal{T}$

(i.e. tunneling enabled), we just pass the parent context *pctx* to the child method (i.e. $ctx' = pctx$) without appending the current context element to *pctx*.

Although the idea of context tunneling is simple, realizing effective tunneling in practice is challenging. We found that the performance of context tunneling is very sensitive to the choice of the tunneling relation; with a good relation, context tunneling is able to improve the baseline analysis significantly but, with an inappropriate choice, the analysis becomes even inferior to the context-insensitive analysis. Manually finding a general heuristic rule to generate an optimal tunneling relation for a given program was difficult for real-world Java programs. The goal of Section 4 is to address this challenge with a data-driven technique that automatically learns a good tunneling heuristic from data.

4 LEARNING CONTEXT-TUNNELING HEURISTICS

In this section, we present a machine-learning algorithm specialized for generating good context tunneling heuristics from a dataset of programs.

4.1 Parametric Program Analysis

Let us first encapsulate our analysis in Section 3 as a parametric program analysis [Liang et al. 2011]. Let $P \in \mathbb{P}$ be a program to analyze. Let $\mathbb{M}_P$ be the set of methods in P. Then, we can define the set $\mathcal{R}_P$ of all tunneling relations for P as follows:

$$\mathcal{T} \in \mathcal{R}_P = \wp(\mathbb{M}_P \times \mathbb{M}_P).$$

The set $\mathcal{R}_P$ forms the parameter space of the analysis for P, where an element $\mathcal{T} \in \mathcal{R}_P$ — a set of method pairs — represents a tunneling relation. Abstractions are ordered by set inclusion. For each pair (m_1, m_2) of parent and child methods in $\mathcal{T}$, we apply context-tunneling by reusing the calling context of the parent method (m_1) when analyzing the child method (m_2) without creating a new context for m_2. The abstraction space covers the conventional context-insensitive and -sensitive analyses: with $\mathcal{T} = \emptyset$, the analysis becomes an ordinary k-context-sensitive analysis, and with $\mathcal{T} = \mathbb{M}_P \times \mathbb{M}_P$, the analysis equals to the context-insensitive analysis.

We assume that a set $\mathbb{Q}_P$ of assertions is given together with the input program P. For instance, $\mathbb{Q}_P$ may denote the set of all type casts in P and the analysis attempts to prove that they do not fail at runtime (i.e. no down-casting failures). Then, we can model points-to analysis for P by the function F_P:

$$F_P \in \mathcal{R}_P \rightarrow \wp(\mathbb{Q}_P) \times \mathbb{N}.$$

Given a program P and a tunneling relation $\mathcal{T} \in \mathcal{R}_P$, $F_P(\mathcal{T})$ returns the set $Q \subseteq \mathbb{Q}_P$ of proved assertions and the analysis time $n \in \mathbb{N}$ represented by a natural number (e.g. time took to analyze the program). We define two projection functions: $\mathrm{proved}(F_P(\mathcal{T}))$ and $\mathrm{cost}(F_P(\mathcal{T}))$ denote the set of proved assertions (Q) and the analysis cost (n) of the analysis $F_P(\mathcal{T})$, respectively.

Non-Monotonicity. One noticeable property of our analysis is non-monotonicity. Note that existing parametric program analyses are typically monotone with respect to their parameters; that is, the analysis precision is monotonically increasing (or decreasing) with respect to the parameters of the analysis (e.g. [Jeong et al. 2017; Liang and Naik 2011; Liang et al. 2011; Zhang et al. 2014]). That is, if $p \sqsubseteq p'$ then $\mathrm{proved}(F_P(p)) \subseteq \mathrm{proved}(F_P(p'))$ (or $\mathrm{proved}(F_P(p)) \supseteq \mathrm{proved}(F_P(p')))$. For example, in a selective context-sensitive analysis [Jeong et al. 2017], making more methods context-sensitive always increases precision. In our case, however, the analysis is not monotone with respect to the parameters (i.e. context-tunneling relations). That is, $\mathcal{T} \subseteq \mathcal{T}'$ does not imply $F_P(\mathcal{T}) \subseteq F_P(\mathcal{T}')$ or $F_P(\mathcal{T}) \supseteq F_P(\mathcal{T}')$. Consequently, neither $F_P(\emptyset)$ nor $F_P(\mathbb{M}_P \times \mathbb{M}_P)$ — the conventional k-context-sensitive analysis and context-insensitive analysis — is the most precise one in the parameter space of context tunneling. As we describe in Section 4.4, this unusual property makes learning challenging; in particular, we cannot use the existing learning algorithms (e.g. [Jeong et al. 2017; Liang et al. 2011]) that exploit the monotonicity of analysis.

4.2 Machine-Learning Model for Context Tunneling

Our goal is to learn a *tunneling heuristic*, denoted $\mathcal{H}$, from a dataset of programs, which takes a program P and returns a tunneling relation for P:

$$\mathcal{H}(P) \in \wp(\mathbb{M}_P \times \mathbb{M}_P).$$

To generate such a heuristic automatically, we need to define a space of possible tunneling heuristics, called model or inductive bias in the machine learning community. A standard method is to define the space by a generic heuristic with free parameters, reducing the problem of generating a good heuristic to the problem of finding appropriate parameter values. There is a number of different ways to define such a parameterized heuristic. For example, we can use a linear combination of input features [Oh et al. 2015] or a non-linear, disjunctive combination [Jeong et al. 2017]. We take the latter approach because the non-linear method is known to be more effective for points-to analysis than the simple linear approach [Jeong et al. 2017].

Following Jeong et al. [2017], we use a boolean formula over atomic features as a model parameter. Let us assume that a set of *atomic features* is given: $\mathbb{A} = \{a_1, a_2, \ldots, a_n\}$, where a feature a_i describes a property of methods. It is a function from programs to predicates on methods:

$$a_i(P) : \mathbb{M}_P \to \{\mathit{true}, \mathit{false}\}.$$

For example, a feature may express the set of methods that have heap allocation in their bodies. We shortly present our atomic features in detail. Given a set of atomic features, we can express more complex features of methods by combining the atomic features. We combine them with a boolean formula f in disjunctive normal form, i.e., a disjunction of conjunctions of literals:

$$f = \bigvee_i \bigwedge_j l_{i,j}$$

where $l_{i,j}$ is a literal: *true*, *false*, atomic feature $a \in \mathbb{A}$, or their negations. Note that the meaning of a boolean formula is a set of methods. Given a program P and a formula $f = \bigvee_i \bigwedge_j l_{i,j}$, let $[\![f]\!]_P$ be the set of methods on which the formula f evaluates to true: $[\![f]\!]_P = \bigcup_i \bigcap_j [\![l_{i,j}]\!]_P$ where $[\![\mathit{true}]\!]_P = \mathbb{M}_P$, $[\![\mathit{false}]\!]_P = \emptyset$, $[\![a_i]\!]_P = \{m \in \mathbb{M}_P \mid a_i(P)(m) = \mathit{true}\}$, and $[\![\neg a_i]\!]_P = \mathbb{M}_P \setminus [\![a_i]\!]_P$. In the rest of this paper, we often represent a formula in disjunctive normal form by a set of sets of literals. For example, the formula $f = (a_1 \wedge a_2) \vee (\neg a_3 \wedge a_4)$ can be represented by $\{\{a_1, a_2\}, \{\neg a_3, a_4\}\}$.

Our model uses two boolean formulas $\Pi = \langle f_1, f_2 \rangle$ and generates the tunneling relation for a given program P as follows:

$$\mathcal{H}_\Pi(P) = \{(m_1, m_2) \in \mathbb{M}_P \times \mathbb{M}_P \mid m_1 \in [\![f_1]\!]_P \vee m_2 \in [\![f_2]\!]_P\}. \tag{1}$$

The generated relation includes a pair (m_1, m_2) of methods if $m_1 \in [\![f_1]\!]_P$ or $m_2 \in [\![f_2]\!]_P$. This conditions says that we apply context tunneling when m_1 is implied by f_1 or m_2 by f_2. Intuitively, f_1 denotes the set of methods that improve the analysis performance by passing their contexts to child methods, and f_2 describes the methods that benefit by reusing the contexts of their parent methods. The goal of our learning algorithm is to discover the characteristics of such methods, represented by boolean combinations (f_1 and f_2) of features, that maximize the performance of the heuristic.

Table 1. Atomic features used in evaluation

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

Class B (Additional features)			
B1	Methods contained in nested class	B7	Methods containing static method invocation
B2	Methods taking multiple arguments	B8	Methods containing virtual method invocation
B3	Methods containing array load	B9	Static method
B4	Methods containing local assignments	B10	Methods containing a single heap allocation
B5	Methods containing local variables	B11	Methods taking an argument of Object type
B6	Methods containing field store	B12	Methods containing multiple heap allocations
		B13	Methods contained in a large class

Atomic Features. Table 1 shows 23 atomic features we have used in learning. Each feature in Table 1 describes a syntactic property of Java method definitions. The features are classified into two types: signature features (Class A) and additional features (Class B). Signature features (A1 – A10) came from the existing work [Jeong et al. 2017] and additional features (B1 – B13) have been newly designed in this work. Signature features consist of strings that most frequently appear in method signatures from the DaCapo suite [Blackburn et al. 2006]. For example, the feature A5 ("void") denotes the set of methods whose signature strings include "void" as a substring. On the other hand, features B1 – B13 describe slightly higher-level properties. For example, the feature B1 denotes the set of methods that belong to inner classes. When choosing atomic features, we focused on collecting as many simple features as possible and let the learning algorithm to discover meaningful combinations of them automatically. In Section 5.2, we discuss impact of using different atomic features.

4.3 Optimization Problem

Formally, the learning problem is expressed as an optimization problem. Given program analysis F, parameterized heuristic $\mathcal{H}_\Pi$ defined in (1), and training programs $\mathbf{P} = \{P_1, \dots, P_m\}$, our goal is to find the parameters f_1 and f_2 that maximize the precision of the analysis over the codebase:

$$\text{Find } \Pi = \langle f_1, f_2 \rangle \text{ that maximizes } \sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|$$

such that $\Pi = \langle f_1, f_2 \rangle$ satisfies the following constraint on the analysis cost:

$$\sum_{P \in \mathbf{P}} \mathsf{cost}(F_P(\mathcal{H}_\Pi(P))) \leq \sum_{P \in \mathbf{P}} \mathsf{cost}(F_P(\emptyset)).$$

The constraint says that the analysis with context tunneling ($F_P(\mathcal{H}_\Pi(P))$) is at least as scalable as the baseline analysis without tunneling ($F_P(\emptyset)$).

4.4 Learning Algorithm

In this paper, we present an algorithm that effectively solves the optimization problem. The key challenge, which makes our algorithm substantially differ from the existing learning algorithms [Jeong et al. 2017; Liang et al. 2011], is that the analysis F is not monotone with respect to the tunneling relations. In existing learning algorithms [Jeong et al. 2017; Liang et al. 2011], monotonicity plays a

Algorithm 1 Overall Algorithm

```
Input: Static analyzer $F$, codebase $\mathbf{P}$, atomic features $\mathbb{A}$
Output: Model parameters $f_1$ and $f_2$
1: procedure LEARN($F$, $\mathbf{P}$, $\mathbb{A}$)
2:     $f_2 \leftarrow$ LEARNPARAMETER($2, false, F, \mathbf{P}, \mathbb{A}$)      ▷ learn methods for child contexts
3:     $f_1 \leftarrow$ LEARNPARAMETER($1, f_2, F, \mathbf{P}, \mathbb{A}$)      ▷ learn methods for parent contexts
4:     return $\langle f_1, f_2 \rangle$
5: end procedure
```

central role in finding analysis parameters efficiently. They start from the most precise abstraction, and iteratively refine the abstraction until a smallest abstraction that satisfies a given constraint is found; here monotonicity allows the algorithms to safely rule out a large set of abstractions smaller than any previously failed (constraint-unsatisfying) abstractions. Consequently, these algorithms follow a (decreasing) chain of parameters and monotonicity ensures that this strategy is optimal. However, when the analysis is not monotone, simply following a chain of parameters no longer provides such a guarantee. Our algorithm is able to find a good parameter over the non-monotonic space of tunneling relations by exploring the search space in a non-greedy way that seeks to maximize the final benefit, instead of the immediate benefit.

Overall Algorithm. The overall algorithm consists of two phases. It first learns the formula f_2, aiming at characterizing the methods that increase the analysis precision by inheriting contexts from their parent methods instead of creating their own ones. With the learned f_2, the algorithm continues to learn f_1, the set of methods that improves the precision by passing their contexts to child methods. Those two formulas f_1 and f_2 become the parameter $\Pi = \langle f_1, f_2 \rangle$ of the heuristic $\mathcal{H}_\Pi$. Procedure LEARN in Algorithm 1 describes the overall procedure. It takes three inputs: a static analyzer F parameterized by tunneling relations, a set $\mathbf{P}$ of training programs, and a set $\mathbb{A}$ of atomic features. The algorithm calls the same subroutine, LEARNPARAMETER, twice with different parameters. Note that the formula f_2 learned in the first phase is used in the second phase at line 3. In the rest of this section, we write $\Pi(i)$ when $\Pi = \langle f_1, f_2 \rangle$ for f_1 (when $i = 1$) or f_2 (when $i = 2$), and write $\Pi[f_i \mapsto g]$ for $\langle g, f_2 \rangle$ (when $i = 1$) or $\langle f_1, g \rangle$ (when $i = 2$).

Learning a Single Parameter. Procedure LEARNPARAMETER in Algorithm 2 takes four inputs: index i indicating the formula to learn (i.e. $i = 1, 2$ when learning f_1, f_2, respectively), learned formula f_2 (if any), static analyzer F, training programs $\mathbf{P}$, and atomic features $\mathbb{A}$. At line 3, we initialize the parameter $\Pi = \langle f_1, f_2 \rangle$, where both f_1 and f_2 are *false* in the beginning (when we learn f_2). Note that, at this point, the heuristic $\mathcal{H}_\Pi$ indicates performing the conventional k-context-sensitive analysis without context tunneling. The goal of the algorithm is to discover a heuristic $\mathcal{H}_{\Pi'}$ that maximally increases the precision of the baseline analysis without sacrificing its scalability. Our strategy to do so is to identify what we call *seed features* and iteratively refine them to maximize their effectiveness. We say $a \in (\mathbb{A} \cup \neg\mathbb{A})$ is a seed feature if it describes methods for which applying context tunneling has potential to improve the precision of the baseline analysis. We define the predicate SeedFeature as follows:

$$\mathsf{SeedFeature}(a, i, \Pi, F, \mathbf{P}) = \Big(\bigcup_{P \in \mathbf{P}} \mathsf{proved}(F_P(\mathcal{H}_{\Pi[f_i \mapsto a]}(P))) \setminus \mathsf{proved}(F_P(\mathcal{H}_\Pi(P))) \Big) \neq \emptyset$$

In words, Π denotes the current baseline heuristic (e.g. conventional analysis without tunneling in the beginning). Let $A = \mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))$ be the set of queries proved by the baseline analysis. Let $B = \mathsf{proved}(F_P(\mathcal{H}_{\Pi[f_i \mapsto a]}(P)))$ be the set of queries proved by the analysis that applies context tunneling to the methods implied by the feature a. We call a seed if B includes at least one query

Algorithm 2 Learning a Single Parameter

```
Input: Index $i$ of parameter to learn, parameter $f_2$, static analyzer $F$, codebase $\mathbf{P}$, atomic features $\mathbb{A}$
Output: $i^{th}$ parameter $f_i$
 1: procedure LearnParameter($i, f_2, F, \mathbf{P}, \mathbb{A}$)
 2:     $f_1 \leftarrow false$
 3:     $\Pi \leftarrow \langle f_1, f_2 \rangle$
 4:     $W \leftarrow \{a \in (\mathbb{A} \cup \neg\mathbb{A}) \mid \mathsf{SeedFeature}(a, i, \Pi, F, \mathbf{P})\}$    ▷ collect seed features
 5:     while $W \neq \emptyset$ do
 6:         $s \leftarrow \mathsf{ChooseSeed}(i, \Pi, F, \mathbf{P}, W)$    ▷ pick a seed feature from $W$ with highest potential
 7:         $W \leftarrow W \setminus \{s\}$
 8:         $c \leftarrow$ RefineSeed($s, i, \Pi, \mathbb{A}, F, \mathbf{P}$)    ▷ refine seed feature $s$
 9:         $\Pi' \leftarrow \Pi[f_i \mapsto f_i \vee c]$    ▷ new parameter to be evaluated
10:         if $\mathsf{BetterHeuristicFound}(\Pi, \Pi', \mathbf{P})$ then    ▷ check whether new parameter improves
11:             $\Pi \leftarrow \Pi'$    ▷ update parameter
12:         end if
13:     end while
14:     return $\Pi(i)$    ▷ return $f_i$
15: end procedure
```

Algorithm 3 Refining a Seed Feature

```
Input: Seed feature $s$, parameter index $i$, parameters $\Pi$, atomic features $\mathbb{A}$, static analyzer $F$, codebase $\mathbf{P}$
Output: refined conjunction $c$
 1: procedure RefineSeed($s$, $i$, $\Pi$, $\mathbb{A}$, $F$, $\mathbf{P}$)
 2:     $c \leftarrow s$    ▷ initial conjunction
 3:     $Failed \leftarrow \emptyset$    ▷ $Failed$ will maintain features that fail to refine $c$
 4:     while ($a \leftarrow \mathsf{ChooseRefiner}(\mathbb{A}, f_i, c, \mathbf{P}, Failed)) \neq false$ do    ▷ iteratively refine conjunction $c$
 5:         $c' \leftarrow c \wedge a$    ▷ refine $c$ with $a$
 6:         $\Pi' \leftarrow \Pi[f_i \mapsto f_i \vee c]$    ▷ old parameter
 7:         $\Pi'' \leftarrow \Pi[f_i \mapsto f_i \vee c']$    ▷ new (refined) parameter
 8:         if $\mathsf{Prec}^{+}(\Pi', \Pi'', \mathbf{P}) \wedge \mathsf{HasPotential}(\Pi'', \Pi, \mathbf{P})$ then    ▷ precision improved
 9:             $c \leftarrow c'$
10:         else if $\mathsf{Prec}^{=}(\Pi', \Pi'', \mathbf{P}) \wedge \mathsf{Cost}^{-}(\Pi', \Pi'', \mathbf{P})$ then    ▷ cost reduced without precision loss
11:             $c \leftarrow c'$
12:         else
13:             $Failed \leftarrow Failed \cup \{a\}$    ▷ record failed attempt
14:         end if
15:     end while
16:     return $c$
17: end procedure
```

not in A, i.e., $B \setminus A \neq \emptyset$. At line 4 of Algorithm 2, we collect all such seed features and they constitute the initial workset W.

In the loop at lines 5–13, we iterate to refine each seed feature. At line 6, the ChooseSeed function chooses the seed feature that has the highest potential:

$$\mathsf{ChooseSeed}(i, \Pi, F, \mathbf{P}, W) = \underset{a \in W}{\operatorname{argmax}} \left| \bigcup_{P \in \mathbf{P}} \mathsf{proved}(F_P(\mathcal{H}_{\Pi[f_i \mapsto f_i \vee a]}(P))) \setminus \mathsf{proved}(F_P(\mathcal{H}_{\Pi}(P))) \right|$$

Among the seed features in W, we pick a feature $a \in W$ that maximizes the number of queries provable with the feature (i.e. $\Pi[f_i \mapsto f_i \vee a]$) but unprovable withtout it (Π). Note that we evaluate the potential of a feature by the number of exclusively provable queries, not by the total number of provable queries. That is, we do not choose the feature

$$\underset{a \in W'}{\text{argmax}} \Big| \bigcup_{P \in \mathbf{P}} \text{proved}(F_P(\mathcal{H}_{\Pi[f_i \mapsto f_i \vee a]}(P))) \Big|$$
$$\text{where } W' = \{a \in W \mid \Big| \bigcup_{P \in \mathbf{P}} \text{proved}(F_P(\mathcal{H}_{\Pi[f_i \mapsto f_i \vee a]}(P))) \Big| > \Big| \bigcup_{P \in \mathbf{P}} \text{proved}(F_P(\mathcal{H}_{\Pi}(P))) \Big| \} \quad (2)$$

which maximizes the immediate precision benefit. Instead, we deliberately choose a feature that has a small (or even negative) immediate benefit but may lead to greater benefit after refinement. This is a key decision point that our algorithm makes in order to maximize the performance in the long run, when the analysis is not monotone with respect to its parameter space. Existing learning algorithms designed with monotonicity in mind [Jeong et al. 2017; Liang et al. 2011] do not explore the search space this way—they simply seek immediate benefit— and not appropriate for learning context tunneling heuristics. We discuss the effect of our search strategy in Section 5.2 in detail.

Once we choose seed s, we refine it to maximize its potential benefit by conjoining other features (line 8). Procedure RefineSeed is responsible for refining s and returns a conjunctive clause $s \wedge a_1 \wedge \cdots \wedge a_k$. When the refinement procedure finishes, we update the parameter with the refined clause and check whether the refined heuristic indeed produces improved performance (lines 9 and 11). Because we pick an atomic feature at the beginning of a clause refinement phase solely based on its potential, overall precision of intermediate clauses can be lower than the baseline. We accept the refined clause only if adding them to the formula improves the precision while satisfying the cost constraint, which is checked by the BetterHeuristicFound predicate:

$$\text{BetterHeuristicFound}(\Pi_1, \Pi_2, \mathbf{P}) =$$
$$\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi_1}(P)))| < \sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathcal{H}_{\Pi_2}(P)))| \wedge \sum_{P \in \mathbf{P}} \text{cost}(F_P(\mathcal{H}_{\Pi_2}(P))) \leq \sum_{P \in \mathbf{P}} \text{cost}(F_P(\emptyset))$$

If a better heuristic is found, we update the parameter and go back to line 5 where the next seed feature is selected and refined. Note that the loop always terminates as the workset W never grows after line 4.

Refining a Seed Feature. Algorithm 3 presents the procedure for refining a seed feature (s). The conjunction is initially s (line 2) and iteratively refined in the loop at lines 4–15. At line 4, we choose a refiner (i.e. an atomic feature) a from $\mathbb{A}$ and conjunct c with a, resulting in $c \wedge a$ (line 5).When refining the current clause c, the algorithm behaves conservatively by choosing the feature a that strengthens c as little as possible. We define the ChooseRefiner function as follows:

$$\text{ChooseRefiner}(\mathbb{A}, f, c, \mathbf{P}, \mathit{Failed}) = \underset{a \in (\mathbb{A} \cup \neg\mathbb{A}) \setminus (c \cup \mathit{Failed})}{\text{argmax}} \sum_{P \in \mathbf{P}} |[\![f \vee (c \wedge a)]\!]_P|$$

Note that we exclude the features in the current clause c and those failed in the previous refinement steps (*Failed*), which ensures that the refinement loop always terminates. At lines 5–7, c and c' denote the current and refined clauses, respectively, and Π', Π'' are the corresponding, current and refined parameters. At lines 8–12, the performance of the refined parameter is evaluated on the training programs. The refined parameter is accepted if it improves the analysis precision (lines 8–9) or it improves the scalability while retaining the precision (lines 10–11). Otherwise (lines 12–13), we do not refine the current clause c and store the feature a in *Failed*. The predicates Prec$^+$,

$\mathsf{Prec}^{=}$, and Cost^{-} are defined as follows:

$$\begin{array}{rcl}
\mathsf{Prec}^{+}(\Pi_1, \Pi_2, F, \mathbf{P}) & = & \sum_{P\in\mathbf{P}}|\mathsf{proved}(F_P(\mathcal{H}_{\Pi_1}(P)))| < \sum_{P\in\mathbf{P}}|\mathsf{proved}(F_P(\mathcal{H}_{\Pi_2}(P)))| \\
\mathsf{Prec}^{=}(\Pi_1, \Pi_2, F, \mathbf{P}) & = & \sum_{P\in\mathbf{P}}|\mathsf{proved}(F_P(\mathcal{H}_{\Pi_1}(P)))| = \sum_{P\in\mathbf{P}}|\mathsf{proved}(F_P(\mathcal{H}_{\Pi_2}(P)))| \\
\mathsf{Cost}^{-}(\Pi_1, \Pi_2, F, \mathbf{P}) & = & \sum_{P\in\mathbf{P}}\mathsf{cost}(F_P(\mathcal{H}_{\Pi_1}(P)) > \sum_{P\in\mathbf{P}}\mathsf{cost}(F_P(\mathcal{H}_{\Pi_2}(P)))
\end{array}$$

Because our goal is to find a conjunctive clause c that helps increase precision of the baseline analysis (i.e. analysis with Π), we additionally check whether the refined heuristic has potential to improve the precision at line 15:

$$\mathsf{HasPotential}(\Pi_1, \Pi_2, F, \mathbf{P}) = \Big(\bigcup_{P\in\mathbf{P}}\mathsf{proved}(F_P(\mathcal{H}_{\Pi_1}(P))) \setminus \mathsf{proved}(F_P(\mathcal{H}_{\Pi_2}(P)))\Big) \neq \emptyset$$

HasPotential returns *true* iff the heuristic with Π_1 proves queries that cannot be proved by the heuristic with Π_2.

5 EVALUATION

In this section, we experimentally evaluate our techniques. We aim to answer the following research questions:

- **Effectiveness of context tunneling**: How effective is context tunneling in practice? How much does context tunneling improve the performance of the state-of-the-art pointer analysis?
- **Necessity and efficacy of our learning approach**: Is learning necessary to find good context-tunneling heuristics? How effectively does the non-greedy algorithm help to find good tunneling heuristics? How sensitive is the algorithm to the choice of atomic features?
- **Learned heuristics**: What insight do the learned heuristics provide about context tunneling?

Setting. We implemented our approach in Doop [Bravenboer and Smaragdakis 2009], a widely used framework for points-to analysis for Java [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011; Tan et al. 2016, 2017]. The Doop framework provides four kinds of context-sensitivity: hybrid context-sensitivity [Kastrinis and Smaragdakis 2013], object-sensitivity [Milanova et al. 2002, 2005], type-sensitivity [Smaragdakis et al. 2011], and call-site-sensitivity [Smaragdakis and Balatsouras 2015]. Hybrid context-sensitivity is currently considered the state-of-the-art, which selectively combines object-sensitivity and call-site-sensitivity to enjoy the benefits of both approaches [Kastrinis and Smaragdakis 2013]. Thus, our primary objective is to apply context tunneling to hybrid context-sensitivity, aiming at advancing the state-of-the-art, but we also show that context tunneling is useful for other types of context-sensitivity as well. In short, we consider the following 12 context-sensitive analyses for Java (all analyses use 1-context-sensitive heap):

- Hybrid context-sensitivity[1]:
  - *S1objH*, *S2objH*: 1 and 2-hybrid-context-sensitivity with 1 context-sensitive heap
  - *S1objH+T*: *S1objH* with context tunneling
- Object-sensitivity:
  - *1objH*, *2objH*: 1 and 2-object-sensitivity with 1 context-sensitive heap
  - *1objH+T*: *1objH* with context tunneling
- Type-sensitivity:
  - *1typeH*, *2typeH*: 1 and 2-type-sensitivity with 1 context-sensitive heap
  - *1typeH+T*: *1typeH* with context tunneling
- Call-site-sensitivity
  - *1callH*, *2callH*: 1 and 2-call-site-sensitivity with 1 context-sensitive heap
  - *1callH+T*: *1callH* with context tunneling

[1] We write *S1objH* to mean *Selective 1-object-sensitive hybrid A* in original paper [Kastrinis and Smaragdakis 2013].

All experiments were done on a machine with Intel i7 CPU and 16 GB RAM running on Ubuntu 14.04 64bit operating system and JDK 1.6.0_30.

Benchmarks. We used the DaCapo 2006-10-MR2 benchmark suite [Blackburn et al. 2006], a standard benchmark for evaluating Java points-to analysis [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011, 2014; Tan et al. 2016, 2017; Thiessen and Lhoták 2017]. Following Jeong et al. [2017] and Smaragdakis et al. [2014], we split ten programs in DaCapo into 4 small (`luindex`, `lusearch`, `antlr`, `pmd`) and 6 large (`eclipse`, `xalan`, `fop`, `chart`, `bloat`, `jython`) programs. We did not use `hsqldb` because 1-context-sensitive points-to analysis did not scale (with 5,400s timeout) when a conservative reflection analysis is used. On the 4 small programs, we ran our learning algorithm to obtain a tunneling heuristic for each analysis and then the learned heuristic was evaluated on the 6 large programs. For a precision metric, we report the number of type casts proven to be safe.

5.1 Effectiveness of Context Tunneling

Column ***S1objH+T*** of Table 2 presents the highlight of our results. It shows that context tunneling significantly improves the precision and scalability of the state-of-the-art hybrid context-sensitivity. We generated *S1objH+T* by applying context tunneling to *S1objH*, where a tunneling heuristic was learned by running Algorithm 1 on the training set of programs (`luindex`, `lusearch`, `antlr`, `pmd`). Table 2 compares the performance of *S1objH+T* and *S1objH* on both training and test programs. On the training set, *S1objH+T* was much more precise and scalable than the baseline (*S1objH*). For example, while *S1objH* analyzes `lusearch` in 79s and reports 850 may-fail casts, our analysis with context tunneling (*S1objH+T*) reduces the analysis time and the number of alarms to 37s and 380, respectively. The improvement becomes more dramatic in the (larger) test programs. For example, *S1objH* reports 2,290 alarms and takes 1,299s in analyzing `chart`. Our analysis (*S1objH+T*) reduces the number of alarms to 876 while only taking 73s.

Remarkably, *S1objH+T* is even better than *S2objH* (2-hybrid-context-sensitivity) in both precision and scalability on all benchmark programs. *S2objH* is currently considered the state-of-the-art points-to analysis for Java as it provides most precise results while scaling well to large programs [Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Tan et al. 2016]. Our result shows that *S1objH+T* can supplant the state-of-the-art. For example, when analyzing `chart`, *S2objH* takes 488s and reports 915 may-fail casts while *S1objH+T* reduces the numbers into 73s and 876, respectively.

Note that recent approaches [Jeong et al. 2017; Tan et al. 2016] aiming at enhancing *S2objH* do so by compromising either precision or scalability. Tan et al. [2016] presented Bean, a technique for improving the precision of object-sensitivity. Jeong et al. [2017] presented a technique for performing selective context-sensitivity, which applies context-sensitivity only to a selected set of methods. Both approaches have been used primarily for improving *S2objH*. However, Bean increases the precision at the expense of scalability [Tan et al. 2016] and the technique by Jeong et al. [2017] improves the scalability at the expense of precision. Our analysis, *S1objH+T*, increases both the precision and scalability of *S1objH* at the same time.

Table 2 and 3 show that context tunneling is useful for other types of context-sensitivity as well; overall, context tunneling enabled 1-object-sensitivity, 1-type-sensitivity, and 1-call-site-sensitivity to perform better than their counterparts with $k = 2$. However, hybrid context-sensitivity benefitted the most from our technique. This is because hybrid context-sensitivity provides the best precision when we use deeper context-sensitivity ($k = 2$) in baseline analyses. Such result implies that we have more chance to find tunneling relations for higher precision than others.

The results show that context tunneling also improves memory consumption. Context tunneling can reduce memory consumption because increased precision reduces spurious facts (e.g., spurious

Table 2. Effectiveness of context tunneling for hybrid context-sensitivity and object-sensitivity. In all metrics, lower is better. Entries with "-" mean that the analysis did not terminate within the given time budget (5400 sec.). For precision metrics, we count may-fail casts, virtual calls that have multiple targets, and reachable methods. For cost metrics, we measure number of call-graph edges and analysis time.

			Hybrid Context Sensitivity			Object Sensitivity		
			S1objH+T	*S1objH*	*S2objH*	*1objH+T*	*1objH*	*2objH*
Training programs	luindex	**may-fail casts**	**371**	783	415	462	796	496
		analysis time(s)	**34**	66	36	37	51	37
		reachable mthds	7,700	7,907	7,702	7,702	7,876	7,702
		call-graph-edges	0.2M	0.5M	0.9M	0.3M	0.5M	1M
	lusearch	**may-fail casts**	**380**	850	420	469	812	508
		analysis time(s)	**37**	79	63	41	56	64
		reachable mthds	8,342	8,580	8,344	8,344	8,526	8,344
		call-graph-edges	0.2M	0.5M	2M	0.3M	0.5M	2.1M
	antlr	**may-fail casts**	**483**	956	530	570	985	611
		analysis time(s)	**47**	85	50	50	67	45
		reachable mthds	8,712	8,917	8 ,714	8,714	8,886	8,714
		call-graph-edges	0.2M	0.6M	0.9M	0.3M	0.6M	1M
	pmd	**may-fail casts**	**713**	1,217	761	812	1,210	846
		analysis time(s)	**53**	129	56	57	77	57
		reachable mthds	9,086	9,322	9,090	9,090	9,277	9,090
		call-graph-edges	0.3M	0.7M	1.3M	0.3M	0.6M	1.3M
Testing programs	eclipse	**may-fail casts**	**586**	1,061	625	698	1,092	729
		analysis time(s)	**41**	129	49	47	94	51
		poly v-calls	1,180	1,404	1,179	1,181	1,395	1,179
		reachable mthds	9,195	9,461	9,188	9,197	9,408	9,188
		call-graph-edges	0.3M	0.8M	1.4M	0.4M	0.8M	1.5M
	xalan	**may-fail casts**	**572**	1,129	623	680	1,055	720
		analysis time(s)	**64**	187	465	400	179	2,047
		poly v-calls	1,628	1,916	1,624	1,633	1,861	1,628
		reachable mthds	10,325	10,560	10,327	10,336	10,511	10,336
		call-graph-edges	0.4M	1M	9M	1.9M	1M	35M
	fop	**may-fail casts**	**1,080**	1,975	1,107	1,253	1,968	1,270
		analysis time(s)	**121**	916	513	176	1,797	428
		poly v-calls	2,081	2,733	2,041	2,063	2,650	2,047
		reachable mthds	14,374	15,741	14,373	14,376	15,733	14,373
		call-graph-edges	1M	3.2M	12.1M	1.6M	3.9M	11.4M
	chart	**may-fail casts**	**876**	2,290	915	1,011	2,226	1,055
		analysis time(s)	**73**	1,299	488	107	2248	316
		poly v-calls	1,614	2,792	1,614	1,616	2,670	1,614
		reachable mthds	12,503	16,037	12,510	12,510	15,977	12,510
		call-graph-edges	0.5M	2.6M	11M	0.7M	3.2M	11.3M
	bloat	**may-fail casts**	**1,251**	1,931	1,326	1,374	1,911	1,407
		analysis time(s)	**464**	707	2,211	463	557	2,314
		poly v-calls	1,668	2,075	1,650	1,652	2,071	1,650
		reachable mthds	9,928	10,159	9,914	9,914	10,116	9,914
		call-graph-edges	1.4M	2.1M	35M	1.4M	1.9M	35.3M
	jython	**may-fail casts**	**837**	1,308	-	-	-	-
		analysis time(s)	**425**	730	>5,400	>5,400	>5,400	>5,400
		poly v-calls	1,394	1,619	-	-	-	-
		reachable mthds	10,626	11,012	-	-	-	-
		call-graph-edges	1.1M	2.1M	-	-	-	-

Table 3. Effectiveness of context tunneling for type-sensitivity and call-site-sensitivity. All notations and measures are the same with those in Table 2.

			Type Sensitivity			Call-Site Sensitivity		
			1typeH+T	*1typeH*	*2typeH*	*1callH+T*	*1callH*	*2callH*
Training programs	luindex	**may-fail casts**	575	888	624	784	837	796
		analysis time(s)	37	43	34	60	50	348
		reachable mthds	7,706	7,884	7,704	7,879	7,953	7,902
		call-graph-edges	0.1M	0.2M	0.3M	0.2M	0.3M	9.2M
	lusearch	**may-fail casts**	616	926	664	843	938	875
		analysis time(s)	40	45	36	62	52	364
		reachable mthds	8,348	8,536	8,346	8,551	8,626	8,575
		call-graph-edges	0.1M	0.2M	0.3M	0.2M	0.3M	9.3M
	antlr	**may-fail casts**	707	1,061	753	945	1,037	995
		analysis time(s)	59	56	50	98	75	430
		reachable mthds	8,718	8,894	8,716	8,885	8,961	8,910
		call-graph-edges	0.1M	0.2M	0.3M	0.3M	0.4M	9.5M
	pmd	**may-fail casts**	949	1,302	995	1,200	1,273	1,216
		analysis time(s)	63	65	58	103	78	454
		reachable mthds	9,096	9,290	9,094	9,296	9,371	9,319
		call-graph-edges	0.1M	0.2M	0.3M	0.2M	0.4M	9.4M
Testing programs	eclipse	**may-fail casts**	839	1,191	876	1,073	1,154	1,098
		analysis time(s)	43	60	45	111	94	574
		poly v-calls	1,246	1,440	1,247	1,399	1,507	1,402
		reachable mthds	9,234	9,451	9,220	9,444	9,511	9,443
		call-graph-edges	0.2M	0.3M	0.4M	0.3M	0.4M	10.5M
	xalan	**may-fail casts**	907	1,229	950	1,137	1,203	1,143
		analysis time(s)	75	77	80	142	121	672
		poly v-calls	1,673	1,890	1,685	1,858	1,967	1,903
		reachable mthds	10,383	10,560	10,375	10,539	10,613	10,560
		call-graph-edges	0.2M	0.3M	0.9M	0.3M	0.4M	9.8M
	fop	**may-fail casts**	1,690	2,133	1,728	1,977	2,070	1,998
		analysis time(s)	143	496	212	508	420	3,480
		poly v-calls	2,349	2,712	2,370	2,522	2,665	2,582
		reachable mthds	15,386	15,794	15,363	15,108	15,217	15,129
		call-graph-edges	0.6M	1M	4M	0.4M	0.5M	10.6M
	chart	**may-fail casts**	1,451	2,449	1,502	2,376	2,485	2,410
		analysis time(s)	92	422	95	618	400	4,339
		poly v-calls	1,752	2,731	1,775	2,698	2,892	2,783
		reachable mthds	13,294	16,018	13,280	16,020	16,134	16,022
		call-graph-edges	0.2M	0.6M	0.8M	0.4M	0.6M	10.7M
	bloat	**may-fail casts**	1,692	2,037	1,723	1,949	2,007	-
		analysis time(s)	71	87	73	785	634	>5400
		poly v-calls	1,772	2,139	1,896	1,925	2,129	-
		reachable mthds	9,952	10,128	9,956	10,113	10,200	-
		call-graph-edges	0.3M	0.3M	0.8M	0.4M	0.6M	-
	jython	**may-fail casts**	1,118	1,440	-	1,331	1,388	1,344
		analysis time(s)	603	431	>5400	188	163	841
		poly v-calls	1,459	1,665	-	1,565	1,705	1,616
		reachable mthds	10,768	11,117	-	10,987	11,066	11,006
		call-graph-edges	0.6M	0.6M	-	0.3M	0.5M	10M

call-graph edges). For example, while *1objH* generated 0.5 million call-graph edges for `lusearch`, *S1objH+T* generated 0.2 million edges, confirming that more than a half of the call-graph edges in the *1objH* analysis are spurious.

5.2 Efficacy of Learning Algorithm

The key enabler for effective context tunneling is our learning algorithm. In this subsection, we discuss our learning approach from various perspectives.

Necessity of learning. One important lesson we learned from this work is that using an automated learning algorithm is essential for discovering effective tunneling heuristics. Any simple heuristic we could come up with manually did not achieve good-enough performance. For example, we tried all "single-feature" heuristics with the parameters in $\{\langle f_1, f_2\rangle \mid (f_1 = \mathit{false} \wedge f_2 = a) \vee (f_1 = a \wedge f_2 = \mathit{false}), a \in \mathbb{A} \cup \neg\mathbb{A}\}$, where $\mathbb{A}$ is the set of atomic features in Table 1. Among the 92 ($4 \times |\mathbb{A}|$) such heuristics, the B1 feature with $\langle f_1 = \mathit{false}, f_2 = B1\rangle$ was most effective for the family of object-based context-sensitivities (i.e. *1objH*, *S1objH*, *1typeH*). By using the B1 feature alone, however, we managed to reduce the total of 9,694 alarms of *S1objH* to 7,218 and the analysis time from 3,968s to 1,367s for the test programs. We failed to further improve its precision substantially with manual tuning. On the other hand, our algorithm effectively refined the feature to the following:

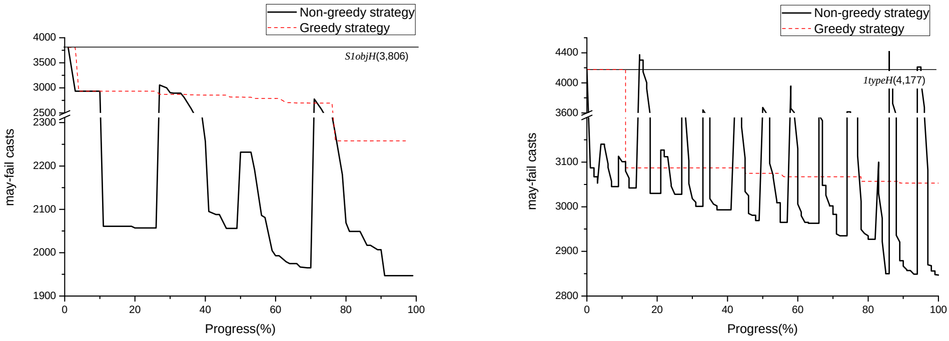
$$\begin{aligned}
&f_2\text{: } (\underline{B1} \wedge B5 \wedge \neg A6 \wedge \neg A9 \wedge \neg B13 \wedge A1 \wedge \neg A3) \vee (B9 \wedge \neg A6 \wedge A1 \wedge B5 \wedge B4 \wedge \\
&\neg B6 \wedge \neg A9 \wedge \neg A3 \wedge \neg A10 \wedge \neg A4 \wedge A8 \wedge \underline{\neg B1}) \vee (B13 \wedge B5 \wedge \neg B3 \wedge \neg A6 \wedge \neg B9 \wedge \\
&B4 \wedge \neg A9 \wedge \neg B10 \wedge \neg B6 \wedge \neg A3 \wedge \neg A5 \wedge \neg B8 \wedge A4 \wedge A8 \wedge A2 \wedge B7 \wedge \underline{\neg B1})
\end{aligned}$$

Note that our algorithm not only was able to refine the feature (i.e. the first conjunct containing $B1$), but also found room for further improvement with $\neg B1$ (i.e. the second and third conjuncts). With this refinement, the precision of the heuristic improved significantly, reducing the number of alarms from 7,218 to 5,202, finally outperforming *S2objH*. Discovering such a complex heuristic manually is totally nontrivial for real-world programs.

Impact of non-greedy strategy. Our algorithm is able to find a good heuristic because it explores the search space in a non-greedy manner. That is, it seeks to maximize the final performance of heuristics rather than blindly pursuing the immediate improvement. To see the impact of this strategy, we compared our algorithm against a greedy algorithm that maximizes the immediate benefit. We made this greedy version of our algorithm by modifying the ChooseSeed and HasPotential functions, so that they measure the potential of a heuristic based on the total number of proved queries, not on the number of *exclusively* proved queries. Precisely, the ChooseSeed is redefined as in Eq. 2 and the HasPotential is changed to return *true* always. With this strategy, the learning algorithm becomes to always favor a parameter that maximally improves the current heuristic; it never explores parameters that are worse than the current one.

Figure 5 shows that our non-greedy strategy has significant impact on the final quality of learned heuristics. It depicts how the precision of the current heuristic changes over time during the learning algorithm. The x-axis reports progress of the learning procedure[2] and the y-axis reports the precision in terms of the number of unproven queries on the training set of programs (lower is better). Note that our algorithm (black solid line) repeatedly explores parameters that are much worse than the current one but doing so enables to find a good parameter in the end. With the greedy strategy (red dotted line), the algorithm converges to suboptimal outcomes. Crucially, the heuristics learned by this greedy algorithm could not beat the conventional 2-context-sensitive approaches on test programs.

[2] We collected refinements that passed the condition at line 10 in Algorithm 2 for better understanding.



(a) Hybrid context-sensitivity

(b) Type-sensitivity

Fig. 5. Impact of our non-greedy learning strategy.

Table 4. Generalization to other benchmarks

Benchmarks	*S1objH+T*		*S1objH*		*S2objH*	
	may-fail casts	time(s)	may-fail casts	time(s)	may-fail casts	time(s)
JPC	1,593	230	2,595	2,578	1,627	1,597
checkstyle	474	80	902	111	508	158

Generality of learned heuristics. The algorithm is able to learn heuristics that do not overfit to training data and generalize well to unseen programs. Tables 2 and 3 show that the context-tunneling heuristics learned with the small programs (luindex, lusearch, antlr, pmd) in the DaCapo suite perform well on the large programs (eclipse, xalan, fop, chart, bloat, jython). We also checked the generality of the heuristics beyond the DaCapo benchmarks. We evaluated the learned heuristic for *S1objH+T* on two large applications, JPC[3] and checkstyle[4]. Table 4 shows that our heuristic substantially outperforms both *S1objH* and *S2objH* for these programs.

Impact of using more features. Our algorithm is likely to produce a better heuristic as more diverse features are used. We evaluated our algorithm with 1) the A features only, 2) the B features only, and 3) the A and B features. We learned a heuristic for each set of features from the training programs, and then evaluated its performance on the test programs. Table 5 presents the results. Overall, using all of the A and B features was most effective. The high-level features (B) primarily helped to increase the precision of the learned heuristic. However, using the B features alone was unable to find scalable heuristics. Additionally using the low-level features (A) enabled the algorithm to refine the B features delicately, generating a heuristic outstanding in both precision and scalability.

Learning cost. Our learning algorithm took 53 − 137 hours to generate the heuristics used in Table 2 and 3. For hybrid context-sensitivity, it took 57 hours in total (21 hours for generating the parameter f_1 and 36 hours for f_2). For object-sensitivity, the algorithm required 26 hours for f_1 and 28 hours for f_2. For type-sensitivity, it took 76 hours for f_1 and 61 hours for f_2. For

[3] http://jpc.sourceforge.net/home_home.html

[4] http://checkstyle.sourceforge.net

Table 5. Impact of using more features

Benchmarks	Baseline(*S1objH*)		With A only		With B only		With A and B	
	alarms	time(s)	alarms	time(s)	alarms	time(s)	alarms	time(s)
eclipse	1,061	129	807	69	583	47	586	41
xalan	1,129	187	866	179	585	137	572	64
fop	1,975	916	1,250	179	1,102	163	1,080	121
chart	2,290	1,299	1,200	120	887	98	876	73
bloat	1,931	707	1,634	1,156	1,250	548	1,251	464
jython	1,308	730	1,039	379	844	3,747	837	425
TOTAL	9,694	3,968	6,796	2,082	5,251	4,740	5,202	1,188

Table 6. Tradeoff between learning cost and performance.

	Learning Cost		eclipse	xalan	fop	chart	bloat	jython
Full learning ($\langle f_1, f_2 \rangle$)	54 hours	alarms	586	572	1,080	876	1,251	837
		time(s)	41	64	121	73	464	425
Approximate ($\langle \mathit{false}, f_2 \rangle$)	29 hours	alarms	605	588	1,099	897	1,317	855
		time(s)	33	54	90	57	335	367

call-site-sensitivity, the algorithm took 53 hours in total (24 hours for f_1 and 29 hours for f_2). Our algorithm was most expensive for type-sensitivity because it found relatively a large set of seed features and required more iterations to refine all of them (see Figure 5b).

Our algorithm is expensive, but it is useful because learning occurs off-line and only consumes machine-time. In particular, generating such a heuristic manually would require much more expensive human costs. Nonetheless, we could reduce the learning cost by approximation. One possible way is to only learn the formula f_2 and then merely set f_1 to *false*. The resulting parameter $\langle \mathit{false}, f_2 \rangle$ makes the heuristic less discerning, giving sub-optimal results, but the learning cost can be halved. Table 6 shows this trade-off between learning cost and optimality. Overall, *S1objH+T* learned with the approximate learning is less precise and slightly faster than the analysis with full learning.

5.3 Learned Heuristics

The learned features in Appendix A hint at when and where context tunneling is useful in practice. For example, our learning algorithm automatically discovered the characteristics of methods that benefit from deeper ($k \geq 2$) object-sensitivity, which was originally conjectured by Milanova et al. [2005]. Milanova et al. [2005] stated that deeper object-sensitivity may be useful for methods that belong to sub-objects. Consider the code snippet that defines a composite class using a sub-object:

```
class A {} class B {}
class SubObject {
  Object id(Object v) { return v; }
}
class CompositeClass {
  public SubObject att;
  public CompositeClass(){
```

```
 8     att = new SubObject();                    //SO
 9   }
10 }
11 class Main{
12   public static void main(String[] args){
13     CompositeClass cc1 = new CompositeClass(); //CC1
14     CompositeClass cc2 = new CompositeClass(); //CC2
15     A a = (A)cc1.att.id(new A());             //Query 1
16     B b = (B)cc2.att.id(new B());             //Query 2
17   }
18 }
```

To prove the safety of two down-casting queries at lines 15 and 16, two method calls to id should be analyzed separately. However, object-sensitivity with $k = 1$ fails to prove the queries because the analysis merges the two contexts into the same context [SO]. On the other hand, deeper object-sensitivity (e.g., $k = 2$) can accommodate the composite class's allocation sites along with the one of sub-objects ([CC1, SO] and [CC2, SO]), concluding that each invocation of id returns one of A or B, not both. Here, we can apply context tunneling instead of using deeper context; the sub-objects' method should inherit the context from the parent method, i.e., the constructor of CompositeClass, instead of updating the context. In other words, contexts of the constructor CompositeClass are important and they should be propagated without modification. By doing so, object-sensitivity with $k = 1$ and context tunneling can prove the queries. Our learning algorithm captured this situation automatically. It produced the f_1 formula for object-sensitivity with the following conjunction:

$$\cdots \vee (A8 \wedge B5 \wedge \neg B3 \wedge A1 \wedge \neg A6 \wedge \neg A3 \wedge \neg B9 \wedge B4 \wedge \neg A9 \wedge \neg B11 \wedge \neg A2 \wedge \neg B7 \wedge \\ \neg B1 \wedge \neg B8 \wedge \underline{B12} \wedge \underline{B10} \wedge \neg B13 \wedge \underline{B6} \wedge A5 \wedge \underline{A10} \wedge A7 \wedge \neg A4)$$

Note the four underlined atomic features: $A10$ for constructor methods, $B12$ and $B10$ for methods with heap allocations, and $B6$ for methods containing field store. Combining these features denotes a set of constructor methods that allocate heaps inside and store something into their member attributes, which describes methods such as one given above.

5.4 Threats to Validity

- The DaCapo suite we used for training may not be representative. We presented the results for other benchmarks beyond DaCapo as well, but it still may not be inclusive.
- Using different sets of features may produce different results. Although we have evaluated our approach with a number of combinations of atomic features, the results might be different if a totally different set of atomic features is used. In particular, our learning algorithm would be unable to work if atomic features do not have any potentials as described in Section 4.4.
- Using a different type of queries, may produce different results as we have learned heuristics with may-fail-casts in mind.

6 RELATED WORK

Points-to analysis has a large body of past literature[Chatterjee et al. 1999; Hind 2001; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006, 2008; Liang and Harrold 1999; Liang et al. 2005; Might et al. 2010; Milanova et al. 2002, 2005; Sharir and Pnueli 1981; Shivers 1988; Smaragdakis and Balatsouras 2015; Smaragdakis et al. 2011; Whaley and Lam 2004; Wilson and Lam 1995]. Below, we discuss prior works that are closely related to ours.

6.1 Context-Sensitive Analysis

To our knowledge, all existing techniques for context-sensitive analyses (e.g. [Jeong et al. 2017; Karkare and Khedker 2007; Kastrinis and Smaragdakis 2013; Khedker and Karkare 2008; Milanova et al. 2002, 2005; Oh et al. 2014; Padhye and Khedker 2013; Sharir and Pnueli 1981; Shivers 1988; Smaragdakis et al. 2011, 2014; Tan et al. 2016; Wei and Ryder 2015; Whaley and Lam 2004]) suffer from the problem we tackle in this paper. All of those techniques update the calling contexts at every call-site and therefore may lose important context elements when k-limiting is used. For example, techniques for varying context depths adaptively [Jeong et al. 2017; Oh et al. 2014; Smaragdakis et al. 2014; Wei and Ryder 2015] have proven their effectiveness for tuning the performance of context-sensitivity. However, the primary goal of these techniques is to selectively analyze methods with appropriate k values, rather than to be selective in context construction.

The technique, called Bean, by Tan et al. [2016] is similar to ours in that it aims to improve precision of k-object-sensitive points-to analysis without increasing k. The idea is to identify "redundant" context elements by running a pre-analysis and building a so-called object allocation graph. An object allocation graph is analogous to a call-graph in call-site-sensitivity and captures how context strings are generated during an object-sensitive analysis. The technique identifies redundant nodes in the graph, which can be removed without reducing the number of distinct contexts; in fact, this technique guarantees to result same or more distinct contexts than conventional analysis with same k. The redundant nodes are excluded when selecting heap contexts for an object. The main focus of Bean is in choosing good heap contexts and therefore it still suffers from the core problem we tackle in this paper; that is, Bean always append the last context element to the parent context on method calls (i.e. it uses the last rule in Figure 3, not Figure 4). As the result, when $k = 1$ for example, Bean analyzes every method under the same calling context as the ordinary 1-object-sensitive analysis. The goal of context tunneling is to mitigate this problem.

Merging equivalent contexts [Xu and Rountev 2008] or abstract heaps [Tan et al. 2017] also have been studied to increase scalability of context-sensitive points-to analysis. Xu and Rountev [2008] first defines a fine-grained notion of context equivalence that guarantees to increase scalability without losing original analysis's precision, which is ∞-context-sensitivity. Based on the equivalence, the paper introduces its abstracted version to extend scalability even further at the cost of precision. Tan et al. [2017] merges two abstract heaps if their fields-points-to-graphs indicate that any field access sequence (i.e., $o.f.g.\cdots.h$) ends up with same typed objects for two graphs. This approach demonstrated that it brings little or no negative impact on precision but scales deep context sensitive analysis.

6.2 Parametric and Data-driven Program Analysis

Our work presents a new instance of parametric program analysis. Previously, parametric program analyses have been used for context-sensitivity [Jeong et al. 2017; Liang and Naik 2011; Liang et al. 2011; Oh et al. 2014; Tan et al. 2017; Zhang et al. 2014], flow-sensitivity [Oh et al. 2015], variable clustering for relational analysis [Heo et al. 2016], and widening thresholds [Cha et al. 2016]. Typically, the goal of these analyses is to find a parameter which is as scalable as possible while sacrificing as little precision as possible. For example, Jeong et al. [2017]; Liang et al. [2011]; Oh et al. [2014, 2015]; Smaragdakis et al. [2014] sacrifice the precision of full flow- or context-sensitivity to obtain tractable scalability. Heo et al. [2016] compromises the full precision of relational analysis and cluster variables whose relationships should be kept during analysis. Hassanshahi et al. [2017] and Tan et al. [2017] aim to find appropriate heap abstraction that scales well without losing precision too much. In this paper, we propose a new knob, called context tunneling, which is able to improve both precision and scalability of the conventional context-sensitive analysis.

From the perspective of data-driven program analysis, our work presents a new learning algorithm for disjunctive model that is applicable even when the underlying analysis is not monotone with respect to the parameter space. Recently, data-driven approaches to program analysis have been popular to address the challenge of generating program-analysis heuristics [Chae et al. 2017; Heo et al. 2016, 2017; Jeong et al. 2017; Oh et al. 2015; Wei and Ryder 2015]. In particular, Jeong et al. [2017] recently proposed an algorithm with disjunctive model based on boolean formulas and used the model to learn a heuristic to determine appropriate context depths for each method. This algorithm drastically reduces the search space by exploiting the monotonicity of the analysis (i.e. applying deeper context-sensitivity will never decrease precision). Our approach is based on the disjunctive model proposed by [Jeong et al. 2017] but uses a different, non-greedy algorithm for learning context tunneling heuristics. The learning algorithm by Oh et al. [2015] is applicable to non-monotone analyses, but its applicability is limited to analysis heuristics based on the linear combination of input features. Chae et al. [2017] solves an orthogonal problem, automatic generation features for learning heuristics, where feature programs are generated by running a program reducer and get abstracted to features represented by abstract data-flow graphs. However, it remains to be seen whether the technique is applicable beyond intraprocedural settings.

7 CONCLUSION AND FUTURE WORK

Developing a precise and scalable context-sensitive analysis is a major challenge in program analysis research. This paper demonstrates that we can effectively address this challenge by applying context tunneling, which carefully maintains the k *most important* context elements, as opposed to the traditional approaches that simply maintain the k *most recent* ones. Experimental results with four flavors of context-sensitivity show that the new approach improves the state-of-the-art points-to analyses remarkably in both precision and scalability. To achieve this, we developed a new machine-learning algorithm specialized for generating context-tunneling heuristics automatically from a dataset of programs.

We believe that the use of context tunneling is not limited to points-to analysis for Java. As future work, we plan to apply context tunneling to static analysis for dynamic languages or control-flow analysis for functional languages, where deeper context-sensitivity (e.g., $k > 5$) is known to be greatly beneficial [Kashyap et al. 2014; Park and Ryu 2015]. Our conjecture is that the same (or even higher) precision can be obtained with smaller k values if they use context tunneling.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. DOI: http://dx.doi.org/10.1145/2594291.2594299

Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. 2005. Improving Software Security with a C Pointer Analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 332–341. DOI: http://dx.doi.org/10.1145/1062455.1062520

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 (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. DOI: http://dx.doi.org/10.1145/1167473.1167488

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2015. Selective Control-flow Abstraction via Jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 163–182. DOI: http://dx.doi.org/10.1145/2814270.2814293

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 (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. DOI: http://dx.doi.org/10.1145/1640089.1640108

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. *Learning a Strategy for Choosing Widening Thresholds from a Large Codebase*. Springer International Publishing, Cham, 25–41. DOI: http://dx.doi.org/10.1007/978-3-319-47958-3_2

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant Context Inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 133–146. DOI: http://dx.doi.org/10.1145/292540.292554

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. DOI: http://dx.doi.org/10.1145/1348250.1348255

Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 459–470. http://dl.acm.org/citation.cfm?id=2818754.2818812

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. ACM, New York, NY, USA, 13–18. DOI: http://dx.doi.org/10.1145/3088515.3088519

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. *Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 237–256. DOI: http://dx.doi.org/10.1007/978-3-662-53413-7_12

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*. ACM.

Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61. DOI: http://dx.doi.org/10.1145/379605.379665

Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

Bageshri Karkare and Uday P. Khedker. 2007. An Improved Bound for Call Strings Based Interprocedural Analysis of Bit Vector Frameworks. *ACM Trans. Program. Lang. Syst.* 29, 6, Article 38 (Oct. 2007). DOI: http://dx.doi.org/10.1145/1286821.1286829

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 121–132. DOI: http://dx.doi.org/10.1145/2635868.2635904

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 (PLDI '13)*. ACM, New York, NY, USA, 423–434. DOI: http://dx.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 *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 213–228. http://dl.acm.org/citation.cfm?id=1788374.1788394

Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. DOI: http://dx.doi.org/10.1007/11688839_5

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. DOI: http://dx.doi.org/10.1145/1391984.1391987

Donglin Liang and Mary Jean Harrold. 1999. Efficient Points-to Analysis for Whole-program Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering (ESEC/FSE-7). Springer-Verlag, London, UK, UK, 199–215. http://dl.acm.org/citation.cfm?id=318773.318943

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 (PASTE '05)*. ACM, New York, NY, USA, 6–12. DOI: http://dx.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 (PLDI '11)*. ACM, New York, NY, USA, 590–601. DOI: http://dx.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 (POPL '11)*. ACM, New York, NY, USA, 31–42. DOI: http://dx.doi.org/10.1145/1926385.1926391

V. Benjamin Livshits and Monica S. Lam. 2003. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. ACM, New York, NY, USA, 317–326. DOI: http://dx.doi.org/10.1145/940071.940114

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 (PLDI '10)*. ACM, New York, NY, USA, 305–315. DOI: http://dx.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 (ISSTA '02)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.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. DOI: http://dx.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 (PLDI '06)*. ACM, New York, NY, USA, 308–319. DOI: http://dx.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 (PLDI '14)*. ACM, New York, NY, USA, 475–484. DOI: http://dx.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 (OOPSLA 2015)*. ACM, New York, NY, USA, 572–588. DOI: http://dx.doi.org/10.1145/2814270.2814309

Rohan Padhye and Uday P. Khedker. 2013. Interprocedural data flow analysis in Soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis, SOAP 2013, Seattle, WA, USA, June 20, 2013*. 31–36. DOI: http://dx.doi.org/10.1145/2487568.2487569

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 735–756. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.735

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 (PLDI '88)*. ACM, New York, NY, USA, 164–174. DOI: http://dx.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. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. DOI: http://dx.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 (POPL '11)*. ACM, New York, NY, USA, 17–30. DOI: http://dx.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 (PLDI '14)*. ACM, New York, NY, USA, 485–495. DOI: http://dx.doi.org/10.1145/2594291.2594320

Y. Sui, D. Ye, and J. Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (Feb 2014), 107–122. DOI: http://dx.doi.org/10.1109/TSE.2014.2302311

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 (PLDI 2017)*. ACM, New York, NY, USA, 278–291. DOI: http://dx.doi.org/10.1145/3062341.3062360

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 (PLDI 2017)*. ACM, New York, NY, USA, 263–277. DOI: http://dx.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 (PLDI '09)*. ACM, New York, NY, USA, 87–97. DOI: http://dx.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, July 5-10, 2015, Prague, Czech Republic*. 712–734. DOI: http://dx.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 (PLDI '04)*. ACM, New York, NY, USA, 131–144. DOI: http://dx.doi.org/10.1145/996841.996859

Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/207110.207111

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 (ISSTA '08)*. ACM, New York, NY, USA, 225–236. DOI: http://dx.doi.org/10.1145/1390630.1390658

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, New York, NY, USA, 42–54. DOI: http://dx.doi.org/10.1145/3134600.3134620

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 (PLDI '14)*. ACM, New York, NY, USA, 239–248. DOI: http://dx.doi.org/10.1145/2594291.2594327

A LEARNED FORMULAS

Following tables show the learned tunneling heuristics. Each row means a conjunctive formula, and each formula has one seed feature, which is underlined. We denote positive and negative features using "T" and "F", respectively, and an empty space means "don't care".

A.1 Hybrid context-sensitivity

	Signature Features (A)										Other Features (B)												
	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11	12	13
f_1	T	F	F	T		F		F	F		F		F	T	T		F		F	$\underline{T}$	F	T	
	T	F	F	F	T	F	T	$\underline{T}$	F	T	F		F	F	T	T	F	F	F	T	F	T	F
f_2	T		F			F			F		$\underline{T}$				T								F
	T		F	F		F		T	F	F	F			T	T	F			$\underline{T}$				
		T	F	T	F	F		T	F		F		F	T	T	F	T	F	F	F			$\underline{T}$

A.2 Object-sensitivity

	Signature Features (A)										Other Features (B)												
	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11	12	13
f_1		F	F	F	T	F	F	F	F	T	F	F	F	T	T	$\underline{T}$	F	F	F	T	F	T	F
	T	F	F	$\underline{T}$	F	F	F		F	F	F	F	F	T	T	F	F	F	F	T	F		F
	T	F	F	F	T	F	T	T	F	T	F	$\underline{T}$	F	T	T	T	F	F	F	T	F	T	F
	T	T	F	F	F	F	F	T		F	F	F	F	T	T	F	F	F			$\underline{T}$		T
f_2	T		F			F			F		$\underline{T}$			T	T				F				
	T	T	T	F	T	T	F	F	T	T	F	$\underline{T}$	F	T	T	T	T	T	F	F	F	T	T

A.3 Call-site-sensitivity

	Signature Features (A)										Other Features (B)												
	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11	12	13
f_1	T	F	F	$\underline{T}$	T	F	F	T		T	F	F	F	T	T	F	F	F	F	F	F	F	T
						F		T		F	F			T	T	F	F	$\underline{F}$	F	F		F	
	T	T	F	F		F		T	F	F	F	T		T	T	F	T	T	$\underline{T}$	F	T	F	T
f_2			F			F			F					T	T			$\underline{F}$	F				F
	T		F	F			T		F			$\underline{T}$		T	T	F			F		F		
	T	F	F			F			F		$\underline{T}$		F	T	T	F	F		F	F	F	F	F

A.4 Type-sensitivity

	Signature Features (A)										Other Features (B)												
	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11	12	13
f_1	T	F	F			F	F	T	F	F	F	F		T	T	F	F		F	$\underline{T}$		T	
	T	F	F	T	T	F	F	T	F	T	F	F	F	F	T	F	F	$\underline{F}$	F	F	F	T	T
	T	T	F	F	T	F	F	F	F	T	F	$\underline{T}$	F	T	T	T	F	F	F	F	F	T	T
	T	F	F	F	T	F	T	T	F	T	F	F	F	T	T	$\underline{T}$	F	F		T	F	T	F
	F	F	F	F	F	F	T	F	F	F	F	T	F	T	T	F	$\underline{F}$	T	F	F	F	T	T
	T	F	F	F	T	F	$\underline{T}$	T	F	T	F	T	F	T	T	T	F	F	F	T	F	T	F
	F	F	T	$\underline{T}$	T	F	F	F	F	T	F	F	F	F	T	F	F	F	F	F	F	T	F
				F					F	F	F		$\underline{T}$	T	T	F		T				T	
	T	F	F	T	F	F	F	T	F	F	F	F	F	T	T	F	F	T	$\underline{F}$	F	F	T	T
	F	$\underline{F}$	F	F	T	F	F		F		F	F	F	T	T	F	F		F		F	T	
f_2	T		F			F			F		$\underline{T}$			T	T				F				F
	T	T			F		F	F	$\underline{T}$	F	F			T	F	F	T		F	F		T	F
			$\underline{T}$		T	F	F	F	F		F		F	T	T			T	F	F		F	T
	T	F	F	F	$\underline{T}$	F	F	F	F	T	F			T	T	T	F		F	F	F		
	$\underline{F}$	F	F	F		F	T		F		F	T	F	T	T		F		F	F	F		F