Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis

Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue

Abstract—Object-sensitive pointer analysis (denoted kOBJ under k-limiting) for an object-oriented program can be accelerated if context-sensitivity can be selectively applied to only some precision-critical variables/objects in a program. Existing pre-analyses for making such selections, which are performed as whole-program analyses to a program, are developed based on two broad approaches. One approach preserves the precision of object-sensitive pointer analysis but achieves limited speedups by reasoning about all the possible value flows in the program conservatively, while the other approach achieves greater speedups but sacrifices precision (often unduly) by examining only some but not all the value flows in the program heuristically. In this paper, we introduce a new pre-analysis approach, TURNER^m (where m stands for modularity), that represents a sweet spot between these two existing ones, as it is designed to enable kOBJ to run significantly faster than the former approach and achieve significantly better precision than the latter approach. TURNER^m is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects in the program can be approximated based on the object-containment relationship pre-established (from Andersen's analysis). In practice, this approximation introduces only a small degree of imprecision into kOBJ. Second, leveraging this initial approximation, we apply a novel object reachability analysis to the program by pre-analyzing its methods according to a reverse topological order of its call graph. When pre-analyzing each method, we make use of a simple DFA (Deterministic Finite Automaton) to reason about object reachability intra-procedurally from its entry to its exit along all the possible value flows established by its statements to identify its precision-critical variables/objects. In practice, this new modular object reachability analysis, which runs linearly in terms of the number of statements in the program, introduces again only a small loss of precision into kOBJ. We have validated TURNER^m with an open-source implementation in SOOT (already publicly available) against the state of the art by using a set of 12 widely used Java benchmarks and applications.

Index Terms—Object-Sensitive Pointer Analysis, CFL Reachability, Object Containment, Modular Static Analysis.

1 INTRODUCTION

Pointer analysis is a significant program analysis that approximates statically the runtime values (memory locations) for the pointer variables in a program. There are a wide range of real-world applications, including security analysis [1], [2], program verification [3], program slicing [4], [5], program understanding [5], [6], and bug detection [7], [8].

For object-oriented languages such as Java, *context sensitivity*, which distinguishes the variables declared and objects allocated locally in a method under different calling contexts, is adopted widely in developing highly precise pointer analyses. In general, a context is represented by a sequence of k context elements under k limiting. There are two common forms of context-sensitivity: (1) k-call-site-sensitivity [9] (which distinguishes the contexts of a method by its k-most-recent call sites) and (2) k-object-sensitivity [10] (which distinguishes the contexts of a method by its receiver object's k-most-recent allocation sites). The latter is widely regarded as a better abstraction in achieving precision and efficiency [11], [12], [13], [14], [15].

However, k-object-sensitive pointer analysis (with k-object-sensitivity as its context abstraction), denoted kOBJ, still does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [11], [12], [13], [14]. As k increases, blindly applying a k-limiting context abstraction uniformly to a program can cause the number of contexts handled to blow up exponentially (often without improving precision much).

In this paper, we address the problem of developing a pre-analysis for a Java program to enable kOBJ to apply context-sensitivity (i.e, a k-limited context abstraction) only to some of its variables/objects selected and context-insensitivity to all the other variables/objects in the program. Let us make it precise about what precision-critical variables/objects are (with respect to kOBJ) in a program.

Definition 1. *Let n be a variable/object in a program. Let $k\text{OBJ}^n$ be the version of kOBJ, where n is analyzed context-insensitively but all the remaining variables/objects in the program are analyzed context-sensitively in exactly the same way as in kOBJ. Then n is said to be* precision-critical *if kOBJ and $k\text{OBJ}^n$ fail to produce the identical points-to information for the program.*

A pre-analysis is said to be *precision-preserving* if it can identify the precision-critical variables/objects in a program precisely or over-approximately as being context-sensitive, and *non-precision-preserving* otherwise. Unfortunately, making such selections precisely is out of the question as solving kOBJ without k-limiting is undecidable [16]. When designing a practical pre-analysis, which aims to select the set of context-sensitive variables/objects, C_{ideal}, in the program, the main challenge are to ensure that (1) C_{ideal} includes as many precision-critical variables/objects as possible but

- *Dongjie He, Jingbo Lu, and Jingling Xue are from UNSW Sydney, Australia.*
  E-mail: {dongjieh, jlu, jingling}@cse.unsw.edu.au
- *Yaoqing Gao is from Huawei, Canada.*

Manuscript received April 19, 2005; revised August 26, 2015.

as few precision-uncritical variables/objects as possible, (2) C_{ideal} results in no or little precision loss, and (3) C_{ideal} is found in a lightweight manner to ensure that the pre-analysis overhead introduced is negligible (relative to kOBJ).

Recently, several pre-analyses have been proposed [15], [17], [18], [19], [20]. Broadly speaking, two approaches exist. EAGLE [15] represents a precision-preserving acceleration of kOBJ by reasoning about CFL (Context-Free-Language) reachability in the program. Designed to be precision-preserving, EAGLE analyzes conservatively and often efficiently the value flows reaching a variable/object and selects the set of context-sensitive variables/objects as a superset of the set of precision-critical variables/objects in the program over-approximately, thereby limiting the potential speedups thus achieved by kOBJ. On the other hand, ZIPPER [20], as a non-precision-preserving representative of the remaining pre-analyses [17], [18], [19], [20], examines the value flows reaching a variable/object heuristically and often efficiently by selecting the set of context-sensitive variables/objects to include some but not all the precision-critical variables/objects and also some precision-uncritical variables/objects in the program. As a result, ZIPPER can sometimes improve the efficiency of kOBJ more substantially than EAGLE in general, but at the expense of introducing a significant loss of precision for some programs.

In this paper, we introduce a new pre-analysis approach, TURNER^m (where m stands for modularity), that represents a sweet spot between EAGLE and ZIPPER: TURNER^m enables kOBJ to run significantly faster than EAGLE while achieving significantly better precision than ZIPPER. Despite a small loss of precision in the average points-to set size (#avg-pts), TURNER^m enables kOBJ to achieve usually the same or nearly the same precision for the other three commonly used precision metrics [11], [12], [13], [14], [15], call graph construction (#call-edges), may-fail casting (#may-fail-casts) and polymorphic call detection (#poly-calls), for a set of 12 widely used Java benchmarks and applications evaluated.

TURNER^m is simple, lightweight yet effective in accelerating kOBJ due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects in the program can be approximated initially based on the object-containment relationship that is inferred from the points-to information pre-computed by applying Andersen's analysis [21]. This approximation turns out to be practically accurate, as it introduces a small degree of imprecision into the final points-to information obtained. Second, leveraging this initial approximation, we apply a novel object reachability analysis to the program by pre-analyzing its methods according to a reverse topological order of its call graph. When pre-analyzing each method, we make use of a simple DFA (Deterministic Finite Automaton) to reason about object reachability intra-procedurally from its entry to its exit along all the possible value flows established by its statements to identify its precision-critical variables/objects. In practice, this new modular object reachability analysis, which runs linearly in terms of the number of statements in the program, introduces again only a small loss of precision into the final points-to information obtained. TURNER^m represents a significant extension of its earlier version, named TURNER and reported in our conference paper [22], which applies its object reachability analysis to all the methods in a program independently (albeit modularly), as discussed in Section 6. In our evaluation, kOBJ runs more efficiently (significantly for some large programs) under TURNER^m than under TURNER while only being negligibly less precise, as evaluated and analyzed in Section 5.

We have validated TURNER^m with an open-source implementation in SOOT against EAGLE and ZIPPER using a set of 12 Java benchmarks and applications. In general, TURNER^m enables kOBJ to run significantly faster than EAGLE due to fewer precision-uncritical variables/objects analyzed context-sensitively and achieve significantly better precision than ZIPPER due to more precision-critical variables/objects analyzed context-sensitively than ZIPPER.

In summary, we make the following contributions:

- We present a new pre-analysis approach, TURNER^m, that can accelerate k-object-sensitive pointer analysis (kOBJ) for Java significantly while introducing only some negligible loss of precision.
- We propose to first approximate the precision-criticality of the objects in a program based on object containment and then decide whether its variables/objects should be context-sensitive or not by conducting a modular object reachability analysis intra-procedurally with a DFA by processing the methods in the program according to a reverse topological order of its call graph, thereby obtaining a pre-analysis that is simple, lightweight and effective.
- TURNER^m enables kOBJ to run significantly faster than EAGLE and achieve significantly better precision than ZIPPER for a set of 12 widely used Java benchmarks and applications evaluated in terms of four common precision metrics, #avg-pts, #call-edges, #may-fail-casts, and #poly-calls (with TURNER^m losing no or little precision for the last three). In addition, the superiority of TURNER^m over TURNER is also demonstrated.
- TURNER^m has been open-sourced at https://www.cse.unsw.edu.au/~corg/turnerm. We hope it will provide a useful open-source framework for researchers and practitioners to develop pointer analysis algorithms and other static program analyses for Java and Android applications.

The rest of this paper is organized as follows. Section 2 motivates our TURNER^m approach. Section 3 gives a version of kOBJ that supports selective context-sensitivity. Section 4 formalizes our TURNER^m approach. In Section 5, we evaluate TURNER^m against the state of the art. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 MOTIVATION

We motivate the development of TURNER^m in the context of the two state-of-the-art pre-analyses, EAGLE [15] and ZIPPER [20]. EAGLE supports partial context-sensitivity as it enables kOBJ to analyze only a subset of variables/objects in a method context-sensitively. On the other hand, ZIPPER allows kOBJ to analyze a method either fully context-sensitively or fully context-insensitively. Like EAGLE, TURNER^m also supports partial context-sensitivity in order to maximize the potential speedups attainable.

As in both EAGLE and ZIPPER, TURNER^m (like TURNER [22]) also relies on the points-to information in a program pre-computed by Andersen's analysis [21] (context-insensitively) to make its context selections.

In Section 2.1, we give some background information. In Section 2.2, we examine several main challenges faced in developing a pre-analysis for accelerating kOBJ and discuss the methodological differences between our TURNER^m approach and two existing approaches, EAGLE and ZIPPER. We also highlight the major difference between TURNER^m and TURNER [22]. In Section 2.3, we introduce a motivating example abstracted from real code by highlighting the effects of these differences on the context-sensitivity selectively applied to kOBJ. In Section 2.4, we describe the basic idea behind TURNER^m (including our insights and trade-offs).

2.1 Background

In object-sensitive pointer analysis [10], the calling contexts of a method are distinguished by its receiver objects. Let each allocation site be abstracted by one unique object. In kOBJ, an object o_1 is modeled context-sensitively by a *heap context* of length $k-1$, $[o_2, ..., o_k]$, where o_i is the receiver object of a method in which o_{i-1} is allocated. As a result, a method with o_1 as its receiver object will be analyzed context-sensitively multiple times, once for each of o_1's heap contexts $[o_2, ..., o_k]$, i.e., once under every possible *method context* $[o_1, ..., o_k]$ of length k. Thus, kOBJ can be specified by either heap or method contexts alone.

Given a variable v analyzed under a method context c, its context-sensitive points-to set $\mathsf{pts}(v, c)$ is expressed as:

$$\mathsf{pts}(v, c) = \{(o_1, c_1), \cdots, (o_n, c_n)\} \quad (1)$$

where each pointed-to object o_i is identified by its heap context c_i. Let M_v be the set of method contexts under which v is analyzed. The context-insensitive points-to set $\overline{\mathsf{pts}}(v)$ for v can be deduced from $\mathsf{pts}(v, c)$ by dropping its contexts:

$$\overline{\mathsf{pts}}(v) = \bigcup_{c \in M_v} \{o \mid (o, c') \in \mathsf{pts}(v, c)\}. \quad (2)$$

When comparing different context-sensitive pointer analyses precision-wise, the context-insensitive points-to information thus obtained is used, as is often done in the literature [12], [14], [15], [17], [18], [20].

2.2 Challenges

A variable/object n in a program is precision-critical if kOBJ loses precision when it analyzes n context-insensitively instead of context-sensitively (Definition 1). In the case of a precision loss, there must exist some variable v such that its context-insensitive points-to information becomes less precise. In this case, $\overline{\mathsf{pts}}(v)$ will contain not only the pointed-to objects found before (when n is analyzed context-sensitively) but also some spurious pointed-to objects introduced now (when n is analyzed context-insensitively). As n and v can be further apart in the program, separated by a long sequence of method calls (with complex field accesses on n along the way), designing a practical pre-analysis P, which selects a set of variables/objects in a program for kOBJ to analyze context-sensitively, is challenging (since solving kOBJ without k-limiting is undecidable [16]). For a program, let C_{ideal} be the set of precision-critical variables/objects specified by Definition 1 and C_P be the set of context-sensitive variables/objects selected by P. The main challenges lie in how to ensure that (1) $|C_{\text{ideal}} - C_P|$ is minimized so that as many precision-critical variables/objects are selected and $|C_P - C_{\text{ideal}}|$ is minimized so that as few precision-uncritical variables/objects are selected, (2) C_P causes kOBJ to lose no or little precision, and (3) C_P is selected in a lightweight manner so that P introduces negligible overhead relative to kOBJ.

A pre-analysis for kOBJ exploits the following necessary condition stated as a fact (given and proved originally in [15]) to identify conservatively the precision-critical variables/objects in a program, with their accesses possibly triggered by some statements outside their containing methods. Without loss of generality, a method is assumed to contain only one return statement "**return** r", where r is a local variable in the method (referred to as its *return variable*).

Fact 1. *Consider a program being analyzed object-sensitively with the parameters and the return variable of each method being modeled as the (special) fields of its receiver objects as in [15]. A variable n in a method M is considered to be precision-critical* only if, *during program execution, there is a value flow entering and leaving M via a parameter or the return variable of M, by passing through n (i.e., by first writing into n via an access path $n.f_1.\cdots.f_r$, where $f_1.\cdots.f_r$ is a sequence of zero or more fields, and then reading it from the same access path).*

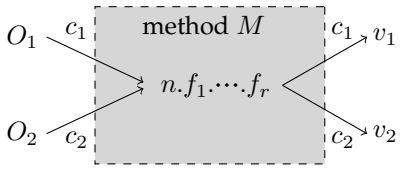

Fig. 1: A precision-critical variable/object n in M. An object O_1 (O_2) from outside M flows into $n.f_1.\cdots.f_r$ and then out of M into a variable v_1 (v_2) under context c_1 (c_2).

Figure 1 illustrates the necessary condition stated above for a variable/object n to be precision-critical. When this necessary condition holds for n, we can conservatively opt to analyze n context-sensitively (to maintain precision). This will allow several such value flows (as illustrated in Figure 1) to be tracked separately based on their calling contexts. Otherwise (i.e., if n is analyzed context-insensitively), the objects O_1 and O_2 that are pointed by $n.f_1.\cdots.f_r$ under the two different contexts, c_1 and c_2, will be conflated, causing v_1 (v_2) to point to the spurious target O_2 (O_1).

In principle, a pre-analysis must reason about the value flows in a program both forwards (by tracking the flow of objects to variables, i.e., def-use chains) and backwards (by discovering the objects pointed to by variables, i.e., use-def chains). This entails entering and exiting a method via its parameters and return variable to keep track of the value flows spanning across the method. As a result, a pre-analysis, as illustrated in Figure 2, should identify variable x as being precision-critical by considering (either directly or indirectly) a total of four possible value-flow patterns passing through x (which are essentially "entry-exit", "exit-entry", "entry-entry" and "exit-exit" classified according to

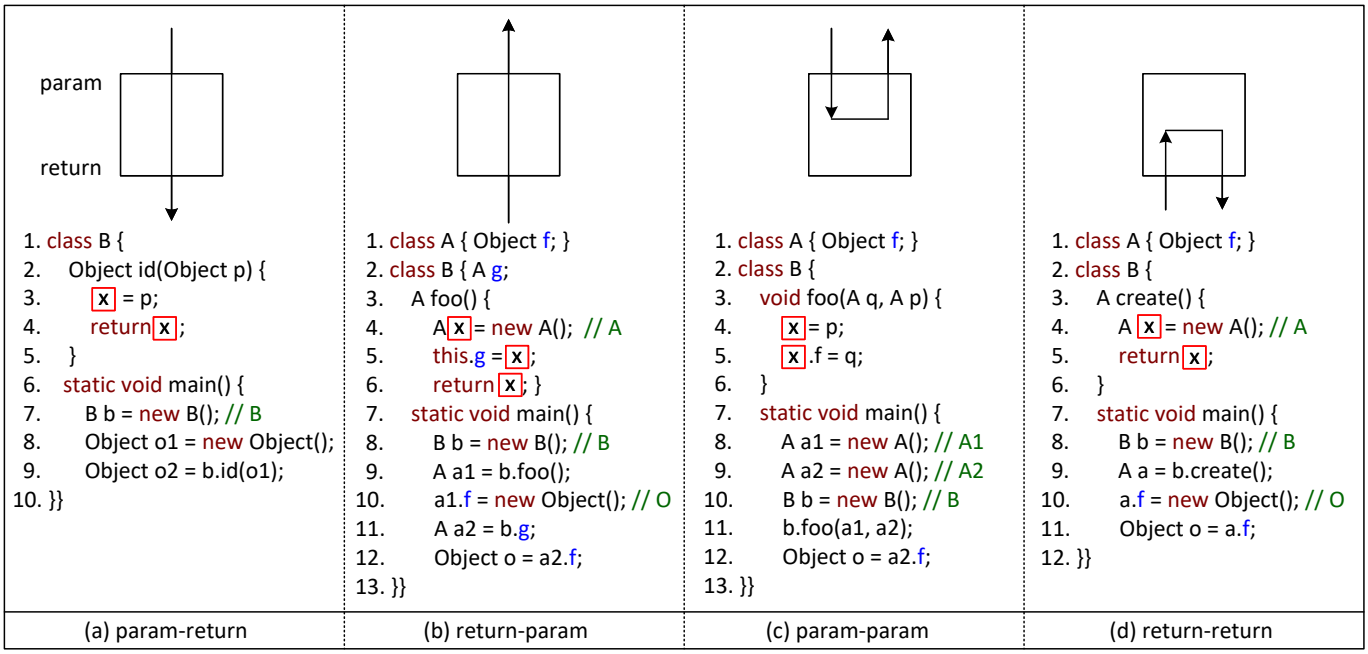

Fig. 2: Four value-flow patterns for determining whether variable x in each case should be precision-critical or not.

whether the two end points of a value-flow path are a parameter or the return variable of its containing method [15], [23]). The same four patterns are also applicable to a locally allocated object. In each case, x must be precision-critical, since if x points to distinct objects when its containing method is called from different call sites (with only one shown explicitly), these objects will be conflated, resulting in some potential precision loss (Figure 1).

In "*param-return*" (Figure 2(a)), the pre-analysis should recognize that the object created in line 8 will flow into x in id() via its parameter p and then out of id() via a return variable, which happens to be x itself. In "*return-param*" (Figure 2(b)), the pre-analysis, when checking whether the object created in line 10 will flow into o in line 12 or not, will first need to establish whether a1 in line 10 and a2 in line 12 are aliases or not. This will entail reasoning about the value flow backwards from a1 to x, this.g, b.g, and finally a2, by entering foo() via its return statement (i.e., return variable) and leaving foo() from its this variable. In "*param-param*" (Figure 2(c)), the object A1 created in line 8 will flow into x.f in foo() via its parameter q and then out of foo() via its parameter p. In "*return-return*" (Figure 2(d)), the pre-analysis, when checking whether the object created in line 10 can flow into o in line 11 or not, will need to find what a points to, by entering and exiting create() from its return variable and visiting x in between (to discover that O flows to o via an intervening x.f).

We can now discuss how TURNER^m differs from EAGLE [15] and ZIPPER [20] methodologically. To start with, all the three are relatively lightweight with respect to kOBJ. Below we examine these pre-analyses in terms of their efficiency and precision tradeoffs made on approximating C_{ideal}. There are two caveats. First, C_{ideal} is conceptual but cannot be found exactly in a program. Second, some precision-critical variables/objects affect the precision and/or efficiency of kOBJ more profoundly than others, but they cannot be easily identified. How to do so approximately can be an interesting research topic in future work.

EAGLE is precision-preserving, since it accounts for all the four value-flow patterns in Figure 2 by reasoning about CFL reachability in the program inter-procedurally as a whole-program analysis to ensure that $C_{\text{ideal}} - C_{\text{EAGLE}} = \emptyset$. For some programs, EAGLE may conservatively misclassify many precision-uncritical variables/objects as being precision-critical, thereby causing $C_{\text{EAGLE}} - C_{\text{ideal}}$ to be unduly large and thus limiting the speedups attainable.

ZIPPER is not precision-preserving (implying that $C_{\text{ideal}} - C_{\text{ZIPPER}} \neq \emptyset$ in general), since it considers only "*param-return*" and "*return-param*" in Figure 2 heuristically by applying an inter-procedural pattern-matching algorithm and ignores "*param-param*" (according to its authors [20]) and "*return-return*" (according to its source code). For some programs, ZIPPER can achieve greater speedups than EAGLE (under certain configurations that pre-define how certain objects should be analyzed) but at some precision loss, since it has misclassified some precision- yet performance-critical variables/objects as context-insensitive.

In this paper, TURNER^m is designed to strike a good balance between EAGLE and ZIPPER. We aim to ensure that $|C_{\text{TURNER}^m} - C_{\text{ideal}}| < |C_{\text{EAGLE}} - C_{\text{ideal}}|$ so that TURNER^m can enable kOBJ to run significantly faster than EAGLE (due to fewer precision-uncritical variable/objects selected for kOBJ to analyze context-sensitively). At the same time, we aim to ensure that $|C_{\text{ideal}} - C_{\text{TURNER}^m}| < |C_{\text{ideal}} - C_{\text{ZIPPER}}|$ so that TURNER^m can also enable kOBJ to achieve significantly better precision than ZIPPER (due to more precision-critical variable/objects selected for kOBJ to analyze context-sensitively). We accomplish this by first exploiting object containment to approximate the precision-criticality of the objects in the program and then conducting a modular object reachability analysis for each method intra-procedurally by considering all the four value-flow patterns in Figure 2.

Unlike TURNER, an earlier version of our pre-analysis [22], which pre-analyzes the methods in the program independently (albeit modularly), TURNER^m pre-analyzes them in a reverse topological order of the call graph of the program, boosting the performance of kOBJ more significantly while introducing only some negligible loss of precision.

2.3 Example

Figure 3 gives a Java program abstracted from real code developed based on JDK. In lines 1-25, a simplified `HashMap` class is defined. In lines 26-31, a simplified `Entry` class is given. In lines 32-50, class `A` represents a use case of `HashMap`. In `foo()`, two instances of `HashMap`, `M1` and `M2`, and two instances of `java.lang.Object`, `O1` and `O2`, are created. Afterwards, `O1` (`O2`), pointed to by `v1` (`v2`), is deposited into `M1` (`M2`), pointed to by `map1` (`map2`), with `O` (received from its parameter `k`) as the corresponding key, and later retrieved and saved in `w1` (`w2`). In `main()`, n instances of `A`, $\mathtt{A}_1, ..., \mathtt{A}_n$, are created, where $n > 1$, and then used as the receivers for invoking `foo()`.

Table 1 lists the contexts used for analyzing this program by the five main pointer analyses, 2OBJ, E-2OBJ, Z-2OBJ, T-2OBJ, and T-2OBJ+M, which happen to produce the same points-to information but at different degrees of efficiency. P-2OBJ denotes the version of 2OBJ that adopts the selective context-sensitivity prescribed by $P \in \{\text{E (for EAGLE)}, \text{Z (for ZIPPER)}, \text{T (for TURNER)}\}$ and T-2OBJ+M is an extension of T-2OBJ by using a new modular object reachability proposed in this paper. EAGLE is always precision-preserving. For this program, ZIPPER happens to be also precision-preserving, but it is easy to modify it slightly so that Z-2OBJ will suffer from a loss of precision (as it does not consider the last two patterns in Figure 2). Both TURNER and TURNER^m also happen to be precision-preserving, but T-2OBJ and T-2OBJ+M differ from 2OBJ, Z-2OBJ and E-2OBJ substantially. Note that even for this small example, T-2OBJ+M is expected to run more efficiently than T-2OBJ due to a smaller number of contexts analyzed. Below we focus on examining how the context-insensitive points-to information for `w1` and `w2` in `foo()`, $\overline{\mathsf{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathsf{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$, is obtained by each of the five main analyses. For reasons of symmetry, Figure 4 illustrates only how $\overline{\mathsf{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ is obtained by these analyses.

First of all, 2OBJ analyzes `foo()` for a total of n times by identifying its variables/objects under the i-th invocation with its receiver object $\mathtt{A}_i$ (Figure 4(a)). Thus, we obtain $\forall\ 1 \leqslant i \leqslant n : \mathsf{pts}(\mathtt{w1}, [\mathtt{A}_i]) = \{\mathtt{O1}, [\mathtt{A}_i]\} \wedge \mathsf{pts}(\mathtt{w2}, [\mathtt{A}_i]) = \{\mathtt{O2}, [\mathtt{A}_i]\}$ context-sensitively. By projecting out all the contexts, we finally obtain $\overline{\mathsf{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathsf{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$ context-insensitively, as desired.

ZIPPER [20] makes its context-sensitivity selections for a program by conducting an inter-procedural pre-analysis to the program. For our example program, as shown in Table 1, Z-2OBJ behaves identically as 2OBJ except that it analyzes `containsKey()` context-insensitively. Note that both analyses compute the points-to information for `w1` and `w2` in `foo()` context-sensitively in the same way (Figure 4(a)).

EAGLE [15] operates also as an inter-procedural pre-analysis, but is designed to enable 2OBJ to support partial context-sensitivity without losing any precision. For our example program (Table 1), the variables/objects in $\{\mathtt{v1}, \mathtt{v2}, \mathtt{w1}, \mathtt{w2}, \mathtt{O1}, \mathtt{O2}\}$ from `foo()` and $\{\mathtt{e}, \mathtt{this}, \mathtt{t}\}$ from `containsKey()` will now be context-insensitive. In the case of `foo()`, however, `k`, `map1`, `map2`, `M1` and `M2` must still be analyzed context-sensitively due to a spurious "*param-param*" pattern established collectively due to (1) `k` is a parameter, (2) `put()` can write into `M1`/`M2`, and (3) `get()` can read from `M1`/`M2`. As a result, as illustrated in Figure 4(b), E-2OBJ will still need to analyze `foo()` for a total of n times, since it must distinguish the two `HashMap` objects `M1` and `M2` created in `foo()` context-sensitively as in 2OBJ, except that it can now analyze the two objects, `O1` and `O2`, created in `foo()` context-insensitively. Therefore, E-2OBJ yields $\mathsf{pts}(\mathtt{w1}, [\,]) = \{\mathtt{O1}, [\,]\}$ and $\mathsf{pts}(\mathtt{w2}, [\,]) = \{\mathtt{O2}, [\,]\}$, i.e., $\overline{\mathsf{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathsf{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$.

TURNER [22] conducts its pre-analysis intra-procedurally by processing all the methods in a program independently. For this particular program, T-2OBJ, as illustrated in Figure 4(c), goes beyond E-2OBJ by modeling `M1` and `M2` also context-insensitively. As a result, `foo()` is analyzed context-insensitively only once. Like E-2OBJ, T-2OBJ also concludes directly that $\mathsf{pts}(\mathtt{w1}, [\,]) = \{\mathtt{O1}, [\,]\}$ and $\mathsf{pts}(\mathtt{w2}, [\,]) = \{\mathtt{O2}, [\,]\}$, i.e., $\overline{\mathsf{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathsf{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$.

TURNER^m pre-analyzes the methods in a program in the reverse topological order of the call graph. Compared with T-2OBJ (Table 1), T-2OBJ+M enables `k` in both `get()` and `containsKey()` to be further analyzed context-insensitively. However, for this example, T-2OBJ+M will still compute the points-to information for `w1` and `w2` in `foo()` identically as T-2OBJ (Figure 4(c)).

2.4 Our Approach

TURNER^m is designed to accelerate kOBJ with partial context-sensitivity at a negligible loss of precision. Unlike EAGLE [15] and ZIPPER [20], TURNER^m works by exploiting first object containment and then object reachability to enable kOBJ to strike a better balance between efficiency and precision. In principle, TURNER^m may cause kOBJ to lose precision due to its first stage only. In practice, however, TURNER^m may also cause kOBJ to lose precision due to its second stage in some rare cases in the presence of type filtering applied during the pointer analysis (Section 4.3).

As discussed in Section 2.2, identifying both precision-critical variables and precision-critical objects in a program simultaneously can be either too conservative (as in EAGLE [15]) or too imprecise (as in ZIPPER [20]) in terms of the final precision-critical variables/objects identified. In TURNER^m, we first pre-select a set of precision-uncritical objects heuristically in a program based on the object containment relationship deduced from Andersen's analysis. We then determine the precision-critical variables/objects in the program by reasoning about CFL reachability intra-procedurally. We leverage CFL reachability in this second stage since a CFL formulation about the points-to information in a method can capture all its pointer-related value flows completely.

2.4.1 *Determining the Precision-Criticality of Objects in a Program based on Object Containment*

We exploit a key insight stated below to identify some precision-uncritical objects in a program approximately

```
1. class HashMap {
2.   Entry[] table;
3.   Object get(Object k) {
4.     int idx = k.hashCode();
5.     Entry[] t = this.table;
6.     Entry e = t[idx];
7.     Object r = e.value;
8.     return r;
9.   }
10.  void put(Object k, Object v) {
11.    int idx = k.hashCode();
12.    Entry e = new Entry(k, v);  // E
13.    Entry[] t = this.table;
14.    t[idx] = e;
15.  }
16.  boolean containsKey(Object k) {
17.    int idx = k.hashCode();
18.    Entry[] t = this.table;
19.    Entry e = t[idx]
20.    return e != null;
21.  }
22.  HashMap() {
23.    Entry[] t = new Entry[16]; // @
24.    this.table = t;
25. }}
26. class Entry {
27.   Object key, value;
28.   Entry(Object p, Object q) {
29.     this.key = p;
30.     this.value = q;
31. }}
32. class A {
33.   void foo(Object k) {
34.     HashMap map1 = new HashMap(); // M1
35.     HashMap map2 = new HashMap(); // M2
36.     Object v1 = new Object(); // O1
37.     Object v2 = new Object(); // O2
38.     if (!map1.containsKey(k)) {
39.       map1.put(k, v1);
40.       Object w1 = map1.get(k);
41.     }
42.     if (!map2.containsKey(k)) {
43.       map2.put(k, v2);
44.       Object w2 = map2.get(k);
45.   }}
46.   public static void main(String args[]) {
47.     Object k = new Object(); // O
48.     A ai = new A(); // Ai
49.     ai.foo(k);                    } 1 ≤ i ≤ n
50. }}
```

Fig. 3: A Java program abstracted from real code in JDK.

TABLE 1: The contexts used for analyzing the variables/objects in the program given in Figure 3 by 2OBJ, E-2OBJ, Z-2OBJ, T-2OBJ and T-2OBJ+M (where i in each context that contains $\mathtt{A}_i/\mathtt{a}_i$ ranges over $[1, n]$).

Method	Variables/Objects	2OBJ	Z-2OBJ	E-2OBJ	T-2OBJ	T-2OBJ+M
get	k	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]	[]
	e, r, this, t					[M1], [M2]
put	k, v, e, this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]	[M1], [M2]
	E	[M1], [M2]	[M1], [M2]	[M1], [M2]		
HashMap	this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]	[M1], [M2]
	@	[M1], [M2]	[M1], [M2]	[M1], [M2]		
containsKey	k	[M1, A_i], [M2, A_i]	[]	[M1, A_i], [M2, A_i]	[M1], [M2]	[]
	e, this, t			[]	[]	
Entry	p, q, this	[E, M1], [E, M2]	[E, M1], [E, M2]	[E, M1], [E, M2]	[E, M1], [E, M2]	[E, M1], [E, M2]
foo	v1, v2, w1, w2	[A_i]	[A_i]	[]	[]	[]
	O1, O2					
	k, map1, map2			[A_i]		
	M1, M2					
main	k, a_i	[]	[]	[]	[]	[]
	O, A_i					

based on the object containment relationship that is inferred from the points-to information pre-computed (context-insensitively) by Andersen's analysis [21]. We first explain how to pre-select a set of precision-uncritical objects in a program (Section 2.4.1.1) and then justify further this heuristic-based design choice (Section 2.4.1.2).

2.4.1.1 Precision-Uncritical Objects: We define what precision-uncritical objects are and also what we mean when we say such a design choice preserves the precision of kOBJ.

Definition 2. *Let the points-to information be pre-computed by applying Andersen's analysis [21]. A* top container *is an object O that is pointed to by neither (1) another object (which may be O itself) via a field of a declared type of C or $C[\,]$, where C is a class type nor (2) the return variable of the method in which O is allocated. A* bottom container *is an object O that does not point to another object (which may be O itself) via a field of a declared type of C or $C[\,]$, where C is a class type.*

Observation 1. *A top container is usually an object allocated and used locally in a method and thus independent of the calling contexts for the method. A bottom container is an object that typically encapsulates its primitive data (including arrays of primitive types), which is usually irrelevant to pointer analysis. Given a program, its top and bottom containers (selected according to Definition 2) are considered as being* precision-uncritical.

Definition 3. *Observation 1 is said to be precision-preserving for a program if kOBJ does not lose precision when it analyzes the precision-uncritical objects identified in the program according*

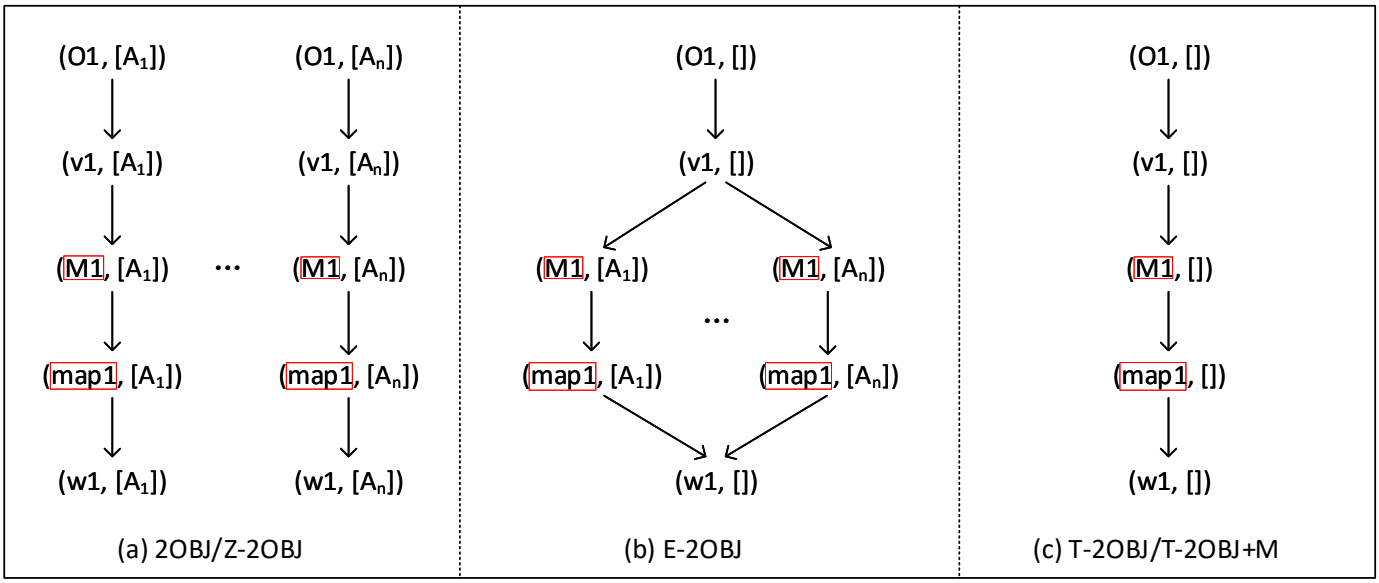

Fig. 4: Computing $\overline{\text{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ for the program in Figure 3 differently by 2OBJ, E-2OBJ, Z-2OBJ, T-2OBJ, and T-2OBJ+M.

to Observation 1 context-insensitively and the remaining variables/objects in the program exactly as before.

Therefore, an object created by a factory method (regarded here as a method that returns its own allocated objects via its return variable) is not a top container. Such an object is considered as being precision-uncritical iff it is a bottom container. For a program, its precision-uncritical objects will be analyzed by kOBJ context-insensitively (as justified below) and the remaining objects will be further classified as either precision-critical or precision-uncritical by an object reachability analysis (Section 2.4.2).

Consider `create()` in Figure 2(d). The object `A` created inside is not regarded as a top container, since it is pointed to by its return variable. In object-sensitive pointer analysis, when `create()` that is called on receiver object `B` in line 9 is analyzed, returning `A` to this caller is actually modeled as `this.ret = x` (line 5) and `a = b.ret` (line 9), where both `this` and `b` point to `B`, and `ret` can be understood as a special return variable introduced for `create()` (Section 4) [15]. Conceptually, `A` is not a top container. In this example, `A` is not a bottom container either, due to `A.f = O` in line 10, where `O` is an instance of `java.lang.Object`. As a result, `A` is considered as being precision-critical. However, if lines 10-11 were not present, then `A` would be deemed as being precision-uncritical as it is now a bottom container.

Consider the program given in Figure 3 (which is free of factory methods), where a total of $n+7$ objects can be found: `E`, `@`, `M1`, `M2`, `O1`, `O2`, `O`, $\texttt{A}_1, ..., \texttt{A}_n$. Figure 5 depicts the object containment relationship deduced from Andersen's analysis for these objects. To ease understanding, we have also included the field names (inside the dashed boxes) along the points-to edges. According to the object containment relationship shown, `M1` and `M2` contain `@`, which contains `E`, which contains `O1`, `O2` and `O`. By Definition 2, the set of top containers is given by $\{\texttt{M1}, \texttt{M2}, \texttt{A}_1, ..., \texttt{A}_n\}$ and the set of bottom containers is given by $\{\texttt{O1}, \texttt{O2}, \texttt{O}, \texttt{A}_1, ..., \texttt{A}_n\}$ (which are not necessarily disjoint). By Observation 1, the $n+5$ objects in $\{\texttt{M1}, \texttt{M2}, \texttt{O1}, \texttt{O2}, \texttt{O}, \texttt{A}_1, ..., \texttt{A}_n\}$ are therefore

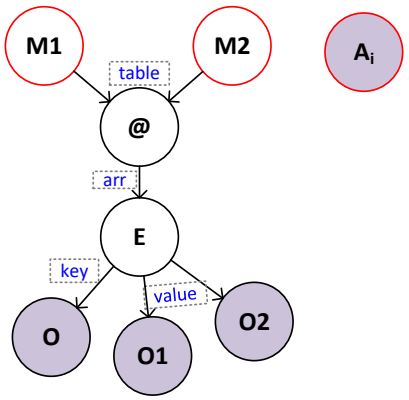

Fig. 5: The object containment relationship for the program given in Figure 3, inferred from the points-to information pre-computed by Andersen's analysis. The top containers are highlighted with their circles depicted in red and the bottom containers are highlighted with their circles filled with purple. $\texttt{A}_i, ..., \texttt{A}_n$, which are abbreviated by one single circle $\texttt{A}_i$ ($1 \leqslant i \leqslant n$), are both top and bottom containers.

considered as being precision-uncritical and will thus be analyzed by kOBJ context-insensitively.

2.4.1.2 Justifications: In our TURNER^m approach, our object containment analysis (based on Observation 1) may introduce some imprecision, which may propagate into its object reachability analysis. TURNER^m will suffer only a slight loss of precision in #avg-pts computed by T-2OBJ+M when some top or bottom containers that should be context-sensitive are misclassified as being precision-uncritical, and consequently, analyzed by T-kOBJ+M context-insensitively. However, this has no or little impact on the precision of #call-edges, #may-fail-casts, and #poly-calls for the set of 12 popular Java programs evaluated.

The set of top containers consists of the objects allocated and used locally in a method, such as `M1` and `M2` (two `HashMap` objects) in `foo()` in Figure 3. These objects do

not require context-sensitivity, since their encapsulated data does not usually flow out of its containing methods via their parameters or return variables. On the other hand, a bottom container also does not usually require context-sensitivity, as it represents an object that typically encapsulates its primitive data (if any), including arrays of primitive types if it ever contains pointers, such as `O`, `O1` and `O2` (three field-less `java.lang.Object` objects) in Figure 3. In Section 5.3, we will examine two example programs (Figures 18 and 19) to explain why TURNER^m causes kOBJ to lose some small precision in #avg-pts but still achieve the same or nearly the same precision in #call-edges, #may-fail-casts, and #poly-calls in real-world programs. Therefore, our heuristic for selecting precision-uncritical objects is practically effective with respect to these commonly used precision metrics.

2.4.2 Conducting Object Reachability for a Method

Given a method, TURNER^m relies on a simple DFA to reason about implicitly the four value-flow patterns in Figure 2 in the method in an intra-procedural manner to select its variables/objects to be analyzed by T-kOBJ+M context-sensitively. In Section 4, we will explain how to start with an object reachability analysis formulated as a CFL reachability analysis, which can only be solved in the same worst-case time complexity as kOBJ to be accelerated [24], and reduce it over-approximately to a DFA-based object reachability analysis, which can be solved linearly (in terms of the number of statements in the program). By design, all the precision-uncritical objects identified by Observation 1 in a method are deemed to be context-insensitive. The remaining objects and all the variables in the method will be classified by the DFA as either precision-critical (context-sensitive) or precision-uncritical (context-insensitive).

2.4.3 Conducting Object Reachability for a Program

We are required to classify the variables/objects in all the methods in a program as either context-sensitive or context-insensitive. In our preliminary investigation [22], our pre-analysis, TURNER, conducts its object reachability analysis by processing the methods in the program independently (albeit modularly). TURNER will be precision-preserving if Observation 1 is precision-preserving, as applying its object reachability analysis this way always over-approximates the precision-critical variables/objects in the program (Theorem 1). For our example in Figure 3, Table 1 gives the contexts selected by TURNER for kOBJ, i.e., T-2OBJ. We discuss only their differences with the contexts selected by EAGLE for kOBJ, i.e., E-2OBJ. By exploiting object containment as discussed in Section 2.4.1, `M1`, `M2`, `O1`, `O2`, and `O` have been identified as being precision-uncritical and will thus be analyzed context-insensitively. Given that `M1` and `M2` are now context-insensitive, `k`, `map1`, and `map2` in `foo()` will also be identified as being context-insensitive by our DFA, since the spurious "*param-param*" pattern that causes EAGLE to flag `M1`, `M2`, `k`, `map1`, and `map2` as being context-sensitive no longer exists (Section 2.3). As `M1` and `M2` are context-insensitive, the contexts $[\texttt{M1}, \texttt{A}_i]$ and $[\texttt{M2}, \texttt{A}_i]$ listed under E-2OBJ have been shortened to [`M1`] and [`M2`] under T-2OBJ.

In this paper, TURNER^m conducts its modular object reachability by adopting a new approach that is methodologically different from that adopted in TURNER. Our key insight is to take advantage of the precision-uncritical variables/objects discovered earlier in a method to increase the number of precision-uncritical variables/objects to be discovered later in another method. In our current design, we achieve this by processing the methods in a program in a reverse topological order of its call graph (with its strongly connected components being collapsed). By construction, the set of context-insensitive variables/objects found by TURNER^m is always a superset of the set of context-insensitive variables/objects found by TURNER (Theorem 3), which ensures the superiority of TURNER^m to TURNER in practice, as evaluated in this paper.

Consider a method M that contains a (possibly virtual) call site $l : b = a_0.m(a_1, \cdots, a_r)$. When performing our object reachability analysis for M, we can ignore all the value flows leaving a_i if its corresponding parameters in all target (callee) methods invoked at l are precision-uncritical. Similarly, we can ignore all the value flows leaving the return variables of the target (callee) methods invoked at l if all these return variables are precision-uncritical. This new modular object reachability analysis will boost the performance of kOBJ further while introducing only a negligible loss of precision in rare cases (as explained in Section 4.3).

For our example in Figure 3 (Table 1), T-2OBJ analyzes the parameters `k` in both `containsKey()` and `get()` context-sensitively. Under our new modular object reachability analysis, TURNER^m will analyze `hashCode()` before `containsKey()` and `get()`. As the *this* variable of `hashCode()` is precision-uncritical, which is a parameter corresponding to `k` in `k.hashCode()` in `containsKey()` (line 4) and `k.hashCode()` in `get()` (line 17), TURNER^m can now identify the parameters `k` in both `containsKey()` and `get()` as being precision-uncritical and thus instruct T-kOBJ+M to analyze both context-insensitively.

3 PRELIMINARIES

We take a standard formalization of kOBJ [10] from [25] and adapt it to support selective context-sensitivity. This gives a formal basis to understand our pre-analysis introduced.

3.1 A Simplified Object-Oriented Language

We consider a simple object-oriented language (a subset of Java), in which a program consists of a set of classes, where each class consists of static/instance fields and methods. Table 2 gives six kinds of statements, which are labeled by their line numbers, operated on by kOBJ. Note that "x = **new** T(...)" in Java is modeled as "x = **new** T; x.⟨`init`⟩(...)", where ⟨`init`⟩() is the corresponding constructor invoked. Section 5 discusses how to handle complex language features such as reflection and native code.

As kOBJ is context-sensitive but flow-insensitive, the control flow statements in a program are irrelevant. As is standard with several recent implementations of kOBJ [11], [12], [13], [14], static fields are analyzed context-insensitively as global variables, but static methods can be analyzed context-sensitively as instance methods as follows. For a static method $m()$ defined in class C, a call to $m()$ can be interpreted as $this.m()$ by proceeding as if $m()$ were an instance method defined in `java.lang.Object` and

TABLE 2: Six kinds of statements analyzed by kOBJ.

Kind	Statement	Description
new	$l : v =$ **new** T	v is a local variable and T is a class type
assign	$l : v = v'$	v and v' are local variables
assigngl	$l : v = v'$	v or v' is a global variable
load	$l : v = v'.f$	v and v' are local variables and f is a field
store	$l : v.f = v'$	v and v' are local variables and f is a field
call	$l : b = a_0.m(a_1, ..., a_r)$	b and a_i are local variables and m is an instance method

inherited by C. Given $this.m()$, $m()$ can then be analyzed context-sensitively under the receiver object pointed to by $this$, which is the receiver object of m's closest (instance) caller method, if any, on the call stack.

Finally, every method is assumed to have one single return statement of the form "**return** r", where r is a local variable (referred to as its return variable). Note that a return statement in a method is not listed explicitly in Table 2, as it will be handled implicitly at a call statement where the method is invoked (as shown in Figure 6).

3.2 Selective Object-Sensitive Pointer Analysis

Given a program, let $\mathbb{M}$, $\mathbb{F}$, $\mathbb{H}$, $\mathbb{V}$, $\mathbb{G}$ and $\mathbb{L}$ be its sets of methods, fields, allocation sites, local variables, global variables, and statements (identified by their labels, e.g., line numbers), respectively. Let $\mathbb{C} = \mathbb{H}^*$ be the universe of contexts. Given a context $ctx = [o_1, ..., o_n] \in \mathbb{C}$ and a context element o, we write $o +\!\!+ ctx$ for $[o, o_1, ..., o_n]$ and $\lceil ctx \rceil_k$ for $[o_1, ..., o_k]$. In the rules given for performing kOBJ, we will make use of the following functions:

- methodOf : $\mathbb{L} \mapsto \mathbb{M}$
- methodCtx : $\mathbb{M} \mapsto \wp(\mathbb{C})$
- dispatch : $\mathbb{M} \times \mathbb{H} \mapsto \mathbb{M}$
- len : $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H} \mapsto \mathbb{N}$
- pts : $(\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \mapsto \wp(\mathbb{H} \times \mathbb{C})$

where methodOf gives the containing method of a statement, methodCtx keeps track of the (method) contexts used for analyzing a method, dispatch resolves a virtual call to its target method, len defines the length of contexts used for analyzing a variable/object, and pts records the points-to information found for a variable or an object's field.

Figure 6 gives five rules used by kOBJ for analyzing six kinds of statements in Table 2 with two kinds of assignments processed together in one rule. In [NEW], v points to the object o_l uniquely identified by its allocation site l. Note that $\lceil ctx \rceil_{\mathsf{len}(o_l)}$ is the heap context of o_l (Section 2.1). In [ASSIGN/ASSIGNGL], two kinds of assignments, where v and v' are either local or global variables, are handled as copies. In [STORE] and [LOAD], field accesses are analyzed in the standard manner. In [CALL], a call to an instance method $b = a_0.m(a_1, ..., a_r)$ is analyzed. We write $this^{m'}$, $p_i^{m'}$ and $ret^{m'}$ for the "this" variable, i-th parameter and return variable of m', respectively, where m' is a target method resolved. Frequently, we also write $p_0^{m'}$ for $this^{m'}$. In the conclusion of this rule, $ctx' \in \mathsf{methodCtx}(m')$ reveals how the method contexts ctx' of a method m' are introduced. Initially, $\mathsf{methodCtx}(\text{"}\texttt{main}\text{"}) = \{[\,]\}$.

kOBJ represents a k-object-sensitive pointer analysis with a $(k-1)$-context-sensitive heap (by handling global variables context-insensitively as is standard) [11], [12], [13], [14]. Thus, kOBJ selects the context lengths for different entities e in $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ differently as follows:

$$\mathsf{len}_{k\text{OBJ}}(e) = \begin{cases} 0 & e \in \mathbb{G} \\ k & e \in \mathbb{V} \\ k-1 & e \in \mathbb{H} \end{cases} \tag{3}$$

TURNER^m will select a subset $CI_{\text{TURNER}^m} \subseteq \mathbb{V} \cup \mathbb{H}$ so that kOBJ will analyze CI_{TURNER^m} context-insensitively but $(\mathbb{V} \cup \mathbb{H}) \setminus CI_{\text{TURNER}^m}$ context-sensitively as follows:

$$\mathsf{len}_{\text{TURNER}^m}(e) = \begin{cases} 0 & e \in CI_{\text{TURNER}^m} \\ \mathsf{len}_{k\text{OBJ}}(e) & e \in (\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus CI_{\text{TURNER}^m} \end{cases} \tag{4}$$

As discussed earlier, EAGLE [15] will also enable kOBJ to analyze only a subset of variables/objects in a method context-sensitively but ZIPPER [20] will require a method (i.e., all its variables/objects) to be analyzed either fully context-sensitively or fully context-insensitively.

4 TURNER^m: OUR APPROACH

We first introduce TURNER^m, by describing its object containment analysis (Section 4.1), its object reachability analysis for one single method (Section 4.2), and its modular object reachability analysis for a program (Section 4.3). We then discuss its time complexity (Section 4.4).

4.1 Determining the Precision-Criticality of Objects in a Program based on Object Containment

During the object containment analysis, we identify some precision-uncritical objects in a program based on the points-to information pre-computed by Andersen's analysis [21] according to Observation 1. For an object o, we write ret_o to denote the return variable in the method where o is allocated. For two objects o_1 and o_2, we write $o_1 \xrightarrow{class\text{-}type(f)} o_2$ if $o_1.f = o_2$ for some field f whose declared type is either C or $C[]$, where C is some class type. As a result, the set of precision-uncritical objects in a program, denoted $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$, can be found as follows:

$$\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m} = \mathsf{TopCon} \cup \mathsf{BotCon} \tag{5}$$

where the sets of top and bottom containers are:

$$\mathsf{TopCon} = \left\{ o \;\middle|\; \left(\nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o' \xrightarrow{class\text{-}type(f)} o \right) \wedge ret_o \text{ does not point to } o \right\} \tag{6}$$

$$\mathsf{BotCon} = \left\{ o \;\middle|\; \nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o \xrightarrow{class\text{-}type(f)} o' \right\}$$

4.2 Conducting Object Reachability for a Method

During the object reachability analysis for a given method, we use a DFA to determine whether a variable/object in the method requires context-sensitivity or not. Let CI_{TURNER^m} be the set of context-insensitive variables/objects that are selected finally by TURNER^m to support selective context-sensitivity required in (4). By design, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m} \subseteq CI_{\text{TURNER}^m}$, i.e., the precision-uncritical objects selected during the object containment analysis will always be analyzed context-insensitively. Therefore, for a given method, its allocated objects that are not in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$, together with all the variables

$$\frac{l : v = \text{new } T \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M)}{(o_l, \lceil ctx \rceil_{\text{len}(o_l)}) \in \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad [\text{NEW}]$$

$$\frac{l : v = v' \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M)}{\text{pts}(v', \lceil ctx \rceil_{\text{len}(v')}) \subseteq \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad [\text{ASSIGN/ASSIGNGL}]$$

$$\frac{l : v.f = v' \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \quad (o, hctx) \in \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})}{\text{pts}(v', \lceil ctx \rceil_{\text{len}(v')}) \subseteq \text{pts}(o.f, hctx)} \quad [\text{STORE}]$$

$$\frac{l : v = v'.f \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \quad (o, hctx) \in \text{pts}(v', \lceil ctx \rceil_{\text{len}(v')})}{\text{pts}(o.f, hctx) \subseteq \text{pts}(v, \lceil ctx \rceil_{\text{len}(v)})} \quad [\text{LOAD}]$$

$$\frac{\begin{array}{c} l : b = a_0.m(a_1, ..., a_r) \quad M = \text{methodOf}(l) \quad ctx \in \text{methodCtx}(M) \\ (o, hctx) \in \text{pts}(a_0, \lceil ctx \rceil_{\text{len}(a_0)}) \quad m' = \text{dispatch}(m, o) \quad ctx' = o +\!\!+ hctx \end{array}}{\begin{array}{c} ctx' \in \text{methodCtx}(m') \quad \forall i \in [1, r] : \text{pts}(a_i, \lceil ctx \rceil_{\text{len}(a_i)}) \subseteq \text{pts}(p_i^{m'}, \lceil ctx' \rceil_{\text{len}(p_i^{m'})}) \\ (o, hctx) \in \text{pts}(this^{m'}, \lceil ctx' \rceil_{\text{len}(this^{m'})}) \text{pts}(ret^{m'}, \lceil ctx' \rceil_{\text{len}(ret^{m'})}) \subseteq \text{pts}(b, \lceil ctx \rceil_{\text{len}(b)}) \end{array}} \quad [\text{CALL}]$$

Fig. 6: Rules for kOBJ formalized to support selective context-sensitivity.

in the method, will be further classified as either context-sensitive or context-insensitive according to the DFA.

We first review a standard formulation for performing pointer analysis intra-procedurally based on CFL reachability, which can only be solved in the same worst-case time complexity as kOBJ [24] (Section 4.2.1). We then evolve it incrementally in stages into a DFA-based intra-procedural reachability analysis, which can be solved linearly in terms of the number of statements in a method (Section 4.2.2).

4.2.1 Standard CFL-Reachability-based Pointer Analysis

A parameterless method that contains no calls can be represented by a directed graph G, known as PAG (Pointer Assignment Graph), with its nodes drawn from $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ and its five types of edges added according to the rules given in Figure 7 [26], [27]. Loads and stores to the elements of an array are modeled by collapsing all the elements into a special field `arr` of the array. For each PAG edge $x \xrightarrow{\ell} y$ with its label ℓ, its inverse edge is denoted as $y \xrightarrow{\overline{\ell}} x$.

$$\frac{l : v = \textbf{new } T}{o_l \xrightarrow{\text{new}} v \quad v \xrightarrow{\overline{\text{new}}} o_l} \quad [\text{P-NEW}]$$

$$\frac{v = v'.f}{v' \xrightarrow{\text{load[f]}} v \quad v \xrightarrow{\overline{\text{load[f]}}} v'} \quad [\text{P-LOAD}] \qquad \frac{v.f = v'}{v' \xrightarrow{\text{store[f]}} v \quad v \xrightarrow{\overline{\text{store[f]}}} v'} \quad [\text{P-STORE}]$$

$$\frac{v = v'}{v' \xrightarrow{\text{assign}} v \quad v \xrightarrow{\overline{\text{assign}}} v'} \quad [\text{P-ASSIGN}] \qquad \frac{v = v'}{v' \xrightarrow{\text{assignl}} v \quad v \xrightarrow{\overline{\text{assignl}}} v'} \quad [\text{P-ASSIGNGL}]$$

Fig. 7: Creating the PAG edges for a method containing no calls inside.

Let L be a CFL over Σ formed by the edge labels in G. A path p in G has a string $L(p)$ in Σ^* formed by concatenating in order the labels of edges in p. A node v in G is *L-reachable* from a node u in G if there exists a path p from u to v, known as *L-path* and denoted by $L(u, v)$, such that $L(p) \in L$. For a node n in G, we write $L(u, v)^n$ if n appears on $L(u, v)$. For a path p in G such that its label is $L(p) = \ell_1, \cdots, \ell_r$ in L, its inverse $\overline{p}$ has the label $L(\overline{p}) = \overline{\ell_r}, \cdots, \overline{\ell_1}$.

We start with a standard grammar that defines the following language L_0 [26], [27]:

$$L_0 : \begin{cases} \textit{flowsto} \longrightarrow \text{new } \textit{flows}^* \\ \textit{flows} \longrightarrow \text{assign} \mid \text{assignl} \mid \text{store[f]} \; \textit{alias} \; \text{load[f]} \\ \overline{\textit{flowsto}} \longrightarrow \overline{\textit{flows}}^* \; \overline{\text{new}} \\ \overline{\textit{flows}} \longrightarrow \overline{\text{assign}} \mid \overline{\text{assignl}} \mid \overline{\text{load[f]}} \; \textit{alias} \; \overline{\text{store[f]}} \\ \textit{alias} \longrightarrow \overline{\textit{flowsto}} \; \textit{flowsto} \end{cases} \quad (7)$$

If o *flowsto* v, then v is L_0-reachable from o, i.e., $L_0(o, v)$. To handle aliases, $\overline{\textit{flowsto}}$ is introduced as the inverse of the *flowsto* relation. A *flowsto* path p can be inverted to obtain its corresponding $\overline{\textit{flowsto}}$ path $\overline{p}$ using its inverse edges, and vice versa. Thus, o *flowsto* x iff x $\overline{\textit{flowsto}}$ o. This means that $\overline{\textit{flowsto}}$ actually represents the standard points-to relation. We can then conclude that x *alias* y iff x $\overline{\textit{flowsto}}$ o *flowsto* y for some object o, so that field accesses are handled precisely by solving a *balanced parentheses* problem.

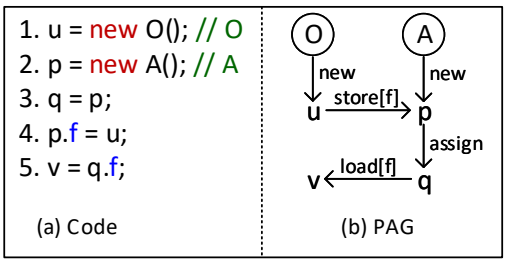

Fig. 8: The PAG for a code snippet.

For the code snippet (consisting of local variables only), together with its PAG, shown in Figure 8, we know that $L_0(O, v)$ holds, i.e., `O` *flowsto* `v`, implying that `v` points to `O`, which holds due to the existence of the *flowsto* path:

$$\texttt{O} \xrightarrow{\text{new}} \texttt{u} \xrightarrow{\text{store[f]}} \texttt{p} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{new}} \texttt{p} \xrightarrow{\text{assign}} \texttt{q} \xrightarrow{\text{load[f]}} \texttt{v} \quad (8)$$

By inverting all the edges in this *flowsto* path, a $\overline{flowsto}$ path showing v *flowsto* o can be obtained.

4.2.2 TURNER^m's Object Reachability Analysis

We now over-approximate L_0 incrementally to obtain a regular grammar, i.e., a DFA to decide intra-procedurally whether a variable/object needs context-sensitivity or not.

4.2.2.1 Ignoring Context-Insensitive Value Flows: Instead of computing points-to information in a program directly, TURNER^m analyzes the context-sensitive value flows across the parameters or return variables of its methods (Fact 1). Thus, we will ignore the assignGl statements and the precision-uncritical objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$, as the value-flows passing through them are context-insensitive.

$$\frac{l : v = \textbf{new}\ T \quad o_l \notin \mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}}{o_l \xrightarrow{\text{cs-likely}} o_l}\ \text{[P-OBJECT]}$$

Fig. 9: Treating the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$ as context-insensitive.

To handle the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$ context-insensitively as global variables, as shown in Figure 9, we have added a self-loop edge label, named cs-likely, for each object that is not in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$ to indicate that it is currently treated as being potentially context-sensitive but will be classified later as being either context-sensitive or context-insensitive by our reachability analysis. By deleting the two terminals assignGl and $\overline{\text{assignGl}}$ from and adding one new terminal cs-likely to the grammar for defining L_0, we obtain a revised grammar for defining L_1 as follows:

$$L_1 : \begin{cases} flowsto \longrightarrow \text{new}\ flows^* \\ flows \longrightarrow \text{assign} \mid \text{store[f]}\ alias\ \text{load[f]} \\ \overline{flowsto} \longrightarrow \overline{flows}^*\ \overline{\text{new}} \\ \overline{flows} \longrightarrow \overline{\text{assign}} \mid \overline{\text{load[f]}}\ alias\ \overline{\text{store[f]}} \\ alias \longrightarrow \overline{flowsto}\ \text{cs-likely}\ flowsto \end{cases} \quad (9)$$

Let us consider Figure 8 again by making two independent changes to the code snippet given. In one change, if q is assumed to be a global variable, then $\mathtt{p} \xrightarrow{\text{assign}} \mathtt{q}$ will become $\mathtt{p} \xrightarrow{\text{assignGl}} \mathtt{q}$. As a result, $L_1(\mathtt{o}, \mathtt{v})$ can no longer be established as in (8) earlier (due to the absence of assignGl in L_1). In a different change, if A is a cs-likely object, then $L_1(\mathtt{o}, \mathtt{v})$ can also be established as before, since we have:

$$\mathtt{o} \xrightarrow{\text{new}} \mathtt{u} \xrightarrow{\text{store[f]}} \mathtt{p} \xrightarrow{\overline{\text{new}}} \mathtt{A} \xrightarrow{\text{cs-likely}} \mathtt{A} \xrightarrow{\text{new}} \mathtt{p} \xrightarrow{\text{assign}} \mathtt{q} \xrightarrow{\text{load[f]}} \mathtt{v} \quad (10)$$

Otherwise (i.e., if A is precision-uncritical), $L_1(\mathtt{o}, \mathtt{v})$ will no longer be possible due to the absence of $\mathtt{A} \xrightarrow{\text{cs-likely}} \mathtt{A}$.

To simplify matters, returning values from a method can be treated identically as passing parameters for the method. In object-sensitive pointer analysis [11], [12], [13], [14], [15], a method M is analyzed context-sensitively under different receiver objects. Therefore, its return statement "**return** r" can be modeled as "$this.ret = r$", where ret is a fresh local variable (interpreted now as the *return variable* of M) and the return values in "*this.ret*" can be retrieved by its callers via its receiver objects. Given this simple transformation, the four value-flow patterns given in Figure 2 can be unified as just one "*param-param*" pattern.

Lemma 1. *A variable/object n in a method M requires context-sensitivity only if n lies on a sequence of statements, $s_1, ..., s_r$, so that (1) s_i and s_{i+1} form a def-use chain involving only local variables and cs-likely objects, (2) s_1 is a use of a cs-likely object or a parameter of M, and (3) s_r is a def of $P.f$, where P is a parameter of M (including this) and f is a field of the objects pointed by P (including M's return variable ret).*

Proof. Follows from the fact that the lemma is a restatement of Fact 1 based on the definition of cs-likely objects. □

In this case, n should be context-sensitive, since the modification effects of different definitions of n on $P.f$ under different calling contexts of M must be separated context-sensitively in order to avoid some potential precision loss.

4.2.2.2 Approximating the Value Flows Spanning across Method Calls: We now consider how to handle a method call made in a method being analyzed. TURNER^m will over-approximate the context-sensitive value flows spanning across a call site without analyzing its invoked methods. With L_1, we can only reason about CFL reachability starting from an object. With L_2 given below, we can also start from a variable (as needed in Lemma 1):

$$L_2 : \begin{cases} flows \longrightarrow (\text{new} \mid \text{assign} \mid \text{store[f]}\ alias\ \text{load[f]})^* \\ \overline{flows} \longrightarrow (\overline{\text{new}} \mid \overline{\text{assign}} \mid \overline{\text{load[f]}}\ alias\ \overline{\text{store[f]}})^* \\ alias \longrightarrow \overline{flows}\ \text{cs-likely}\ flows \end{cases} \quad (11)$$

Lemma 2. *Let G be the PAG built by the rules in Figures 7 and 9. Then $L_2 \supseteq L_1$.*

Proof. Follows simply from examining the structural differences in the productions of L_1 and L_2. □

In both languages, L_1 and L_2, the aliases between two variables are established in exactly the same way.

Next, we over-approximate L_2 to obtain L_3 by abstracting the field accesses with 1-limited access paths and handling aliases more conservatively (as explained below):

$$L_3 : \begin{cases} flows \longrightarrow (\text{new} \mid \text{assign} \mid \text{load} \mid \text{store}\ alias)^* \\ \overline{flows} \longrightarrow (\overline{\text{new}} \mid \overline{\text{assign}} \mid \overline{\text{load}} \mid alias\ \overline{\text{store}})^* \\ alias \longrightarrow \overline{flows}\ \text{cs-likely}\ flows \end{cases} \quad (12)$$

Thus, the fields in loads and stores are ignored, and loads and assignments become indistinguishable, but stores are treated differently (i.e., unsymmetrically as loads) in order to keep track of aliases as desired. Note that L_3 is still a CFL, since (1) a store is required to match a $\overline{\text{new}}$, $\overline{\text{assign}}$ or $\overline{\text{load}}$, and (2) a $\overline{\text{store}}$ is required to match a new, assign or load. However, this balanced-parentheses property is somehow hidden in the *alias*-production.

For the code snippet given in Figure 8, $L_3(\mathtt{o}, \mathtt{v})$ will still hold even if, say, `v = q.f` is replaced by `v = q.g` due to the existence of the following *flowsto* path:

$$\mathtt{o} \xrightarrow{\text{new}} \mathtt{u} \xrightarrow{\text{store}} \mathtt{p} \xrightarrow{\overline{\text{new}}} \mathtt{A} \xrightarrow{\text{cs-likely}} \mathtt{A} \xrightarrow{\text{new}} \mathtt{p} \xrightarrow{\text{assign}} \mathtt{q} \xrightarrow{\text{load}} \mathtt{v} \quad (13)$$

Lemma 3. *Let G be the PAG built by the rules in Figures 7 and 9. Then $L_3 \supseteq L_2$.*

$$\frac{b = a_0.m(a_1, ..., a_r)}{\forall\, i : a_i \xrightarrow{\text{store}[p_i^{m'}]} a_0 \quad \forall\, i : a_0 \xrightarrow{\overline{\text{store}[p_i^{m'}]}} a_i \quad a_0 \xrightarrow{\text{load}[ret^{m'}]} b \quad b \xrightarrow{\overline{\text{load}[ret^{m'}]}} a_0} \text{[P-CALL]}$$

Fig. 10: Analyzing a method call.

Proof. In L_3, the first two productions can be expressed equivalently as *flows* $\longrightarrow$ (new | assign | load | store *alias* load?)* and $\overline{\textit{flows}} \longrightarrow$ ($\overline{\text{new}}$ | $\overline{\text{assign}}$ | $\overline{\text{load}}$ | $\overline{\text{load}}$? *alias* $\overline{\text{store}}$)*. As is standard, (s)? indicates that s is optional, where '(' and ')' can be omitted if s represents one symbol. We can conclude that $L_3 \supseteq L_2$ by noting that the field access paths in L_3 are 1-limited. □

In L_3, a store can now also be matched (conservatively) with a $\overline{\text{store}}$ when looking for aliases:

$$\textit{flows} \Longrightarrow^+ \ldots \text{ store } \overline{\textit{flows}} \text{ cs-likely } \textit{flows } \overline{\text{store}} \ldots \tag{14}$$

For the code given in Figure 8, $L_3(\mathtt{o}, \mathtt{v})$ will thus still hold if we (1) replace `v = q.f` by `q.g = v` and (2) add `v = new V()`, where the allocated object, `V`, is assumed to be cs-likely:

$$\mathtt{o} \xrightarrow{\text{new}} \ldots \xrightarrow{\text{assign}} \mathtt{q} \xrightarrow{\overline{\text{store}}} \mathtt{v} \xrightarrow{\overline{\text{new}}} \mathtt{V} \xrightarrow{\text{cs-likely}} \mathtt{V} \xrightarrow{\text{new}} \mathtt{v} \tag{15}$$

where the subpath from `o` to `q` is the same as that in (13).

We exploit this property to avoid analyzing the methods invoked at a call site while still keeping track of all context-sensitive value flows spanning the call site (conservatively).

Consider how kOBJ analyzes a method call $b = a_0.m(a_1, ..., a_r)$, with a target method m' resolved when a_0 points to a receiver object O. Let its $r + 1$ parameters be $p_0^{m'}, ..., p_r^{m'}$, where $p_0^{m'}$ represents $this^{m'}$. Let its return variable $ret^{m'}$ be introduced as described in Section 4.2.2.1. Object-sensitively, $p_0^{m'}, ..., p_r^{m'}$ and $ret^{m'}$ are handled as if they were special fields of O [15]: $\forall\, i : a_0.p_i^{m'} = a_i$ for passing parameters and $b = a_0.ret^{m'}$ (for retrieving return values). As a result, Figure 10 gives a rule, [P-CALL], for adding the PAG edges required for a method call according to [P-LOAD] and [P-STORE]. When m' is analyzed by kOBJ, where its $this^{m'}$ variable points to O, its parameters will be initialized as $\forall\, i : p_i^{m'} = this^{m'}.p_i^{m'}$ and its return values will be made available in $this^{m'}.ret^{m'}$.

Given how $b = a_0.m(a_1, ..., a_r)$ is modeled, we can determine whether or not a context-sensitive value flow that enters one of its invoked methods via a parameter can also exit it via another parameter without actually analyzing the invoked method itself, by enforcing $L_3(a_i, a_j)$ conservatively, i.e., ensuring that whatever flows into a_i flows also into a_j, if necessary. As will be clear in Section 4.2.2.3, this call needs to be approximated this way if a_0 may point to some cs-likely object and can be ignored otherwise.

Lemma 4. *Let G be the PAG built by the rules in Figures 7, 9 and 10 for a method M (where how its parameters are modeled is irrelevant). When analyzing a call $b = a_0.m(a_1, ..., a_r)$ in M, $L_3(a_i, a_j)$ is established iff a_0 points to some cs-likely object.*

Proof. Let O be an object pointed by a_0. By [P-CALL], passing a_i and a_j to a target method m' at the call site is modeled by two stores $a_0.p_i^{m'} = a_i$ and $a_0.p_i^{m'} = a_j$. Thus,

$$\textit{flows} \Longrightarrow^+ \ldots a_i \xrightarrow{\text{store}} a_0\ \overline{\textit{flows}}\ O \cdots O\ \textit{flows}\ a_0 \xrightarrow{\overline{\text{store}}} a_j \ldots \tag{16}$$

$L_3(a_i, a_j)$ is then established (as far as this call site is concerned, regardless of its truthhood established elsewhere) iff O is a cs-likely object, in which case the "$\cdots$" between the two occurrences of O can be replaced by $\xrightarrow{\text{cs-likely}}$. □

4.2.2.3 Approximating the Incoming Value Flows from Parameters: We discuss how to handle the parameters of a method when it is analyzed. It is not computationally feasible to formulate our pre-analysis for a method in terms of L_3 directly (even after its parameters have been modeled in a certain way). As L_3 is a CFL (with balanced parentheses), the worst-time complexity for finding the points-to set of a variable is $O(N^3\Gamma_{L_3}^3)$, where N is the number of nodes in the PAG and Γ_{L_3} is the size of L_3 [24], [28].

To handle method parameters, we over-approximate L_3 by turning it into a regular language L_4 defined by:

$$L_4 : \begin{cases} \textit{flows} \longrightarrow (\text{new} \mid \text{assign} \mid \text{load})^*((\text{store} \mid \overline{\text{store}})\ \overline{\textit{flows}})? \\ \overline{\textit{flows}} \longrightarrow (\overline{\text{new}} \mid \overline{\text{assign}} \mid \overline{\text{load}})^*(\text{cs-likely}\ \textit{flows})? \end{cases} \tag{17}$$

Lemma 5. *Let G be the PAG built by the rules in Figures 7, 9 and 10. Then $L_4 \supseteq L_3$.*

Proof. Follows from the fact that L_4 is regularized from L_3 by no longer distinguishing store and $\overline{\text{store}}$. □

Thus, we are now even more conservative in abstracting aliases in L_4 than in L_3. If we replace `p.f = u` with `u.f = p` in Figure 8, $L_3(\mathtt{o}, \mathtt{v})$ will not hold but $L_4(\mathtt{o}, \mathtt{v})$ will, since

$$\mathtt{o} \xrightarrow{\text{new}} \mathtt{u} \xrightarrow{\overline{\text{store}}} \mathtt{p} \xrightarrow{\overline{\text{new}}} \mathtt{A} \xrightarrow{\text{cs-likely}} \mathtt{A} \xrightarrow{\text{new}} \mathtt{p} \xrightarrow{\text{assign}} \mathtt{q} \xrightarrow{\text{load}} \mathtt{v} \tag{18}$$

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\text{param}} p \quad p \xrightarrow{\overline{\text{param}}} p} \text{[P-PARAM]}$$

Fig. 11: Adding the PAG edges for method parameters.

We are ready to describe our final regular language L_5 used to decide if a variable/object in a method should be context-sensitive or not. By adding the two self-loop PAG edges for each parameter of a method according to [P-PARAM] given in Figure 11 and exploiting the fact that store and $\overline{\text{store}}$ are treated identically in L_4, we obtain:

$$L_5 : \begin{cases} s \longrightarrow \text{param}\ \textit{flows} \\ \textit{flows} \longrightarrow (\text{new} \mid \text{assign} \mid \text{load})^*((\text{store} \mid \overline{\text{store}})\ \overline{\textit{flows}})? \\ \overline{\textit{flows}} \longrightarrow (\overline{\text{new}} \mid \overline{\text{assign}} \mid \overline{\text{load}})^*(\text{cs-likely}\ \textit{flows})? \\ \overline{\textit{flows}} \longrightarrow \overline{\text{param}}\ e \\ e \longrightarrow \epsilon \end{cases} \tag{19}$$

This allows us to analyze a method without knowing what its parameters may point to, by treating it effectively as a parameterless method (discussed earlier).

Lemma 6. *Let G be the PAG built for a method by the rules in Figures 7 and 9–11. Let P_1 and P_2 be its two (not necessarily different) parameters. Then $L_4(P_1, P_2) \iff L_5(P_1, P_2)$.*

Proof. Follows straightforwardly reasoning about the minor differences in the productions of L_4 and L_5. □

As discussed in Section 4.2.1, if L is a CFL, $L(u,v)^n$ holds if $L(u,v)$ holds due to an L-path that contains a node n. Thus, CI_{TURNER^m} that appears in (4) can be defined as:

$$CI_{\text{TURNER}^m} = \bigcup_{M \in \mathbb{M}} CI_{\text{TURNER}^m}(M) \tag{20}$$

where

$$CI_{\text{TURNER}^m}(M) = \{n \mid n \in \mathbb{V} \cup \mathbb{H} \wedge n \text{ is a node in } G_M \wedge \nexists\, P_1, P_2 \in \mathit{param}(M) : L_5^{G_M}(P_1, P_2)^n\} \tag{21}$$

Here, $\mathit{param}(M)$ is the set of parameters of a method M and L_5 is superscripted with the PAG, G_M, built for M. By construction, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m} \subseteq CI_{\text{TURNER}^m}$ holds due to the absence of a self-loop edge, labeled cs-likely, around each object in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$. According to (3), all the global variables in $\mathbb{G}$ are handled context-insensitively. Therefore, we do not have to include $\mathbb{G}$ in CI_{TURNER^m} explicitly even though this inclusion can be recognized automatically by our approach.

Let us apply TURNER^m to the four programs in Figure 2 to see how it has successfully selected `x` to be context-sensitive (where "return `x`" in each program has been replaced by "`this.ret = x`" and the objects `A` created in Figure 2(b) and 2(d) are assumed to be cs-likely objects):

- **Figures 2(a):** $L_5(\texttt{p},\texttt{this})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\text{assign}} \texttt{x} \xrightarrow{\text{store}} \texttt{this}$.
- **Figures 2(b):** $L_5(\texttt{this},\texttt{this})^{\texttt{x}}$: $\texttt{this} \xrightarrow{\overline{\text{store}}} \texttt{x} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{cs-likely}} \texttt{A} \xrightarrow{\text{new}} \texttt{x} \xrightarrow{\text{store}} \texttt{this}$.
- **Figure 2(c).** $L_5(\texttt{p},\texttt{q})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\text{assign}} \texttt{x} \xrightarrow{\overline{\text{store}}} \texttt{q}$.
- **Figure 2(d):** $L_5(\texttt{this},\texttt{this})^{\texttt{x}}$: $\texttt{this} \xrightarrow{\overline{\text{store}}} \texttt{x} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{cs-likely}} \texttt{A} \xrightarrow{\text{new}} \texttt{x} \xrightarrow{\text{store}} \texttt{this}$.

Finally, we show that our intra-procedural object reachability analysis is precision-preserving if Observation 1 is precision-preserving. In this case, TURNER [22], which pre-analyzes the methods in a program independently, will not cause kOBJ to lose precision. The basic idea is to show that if a variable/object in a given method is identified as being context-sensitive according to Lemma 1, i.e., Fact 1 (Figure 2), then it must always reside on an L_5-path.

Theorem 1. *Suppose Observation 1 is precision-preserving. Let G be the PAG built for a method M (according to the rules given in Figures 7 and 9–11). If a variable/object n in M is context-sensitive by Lemma 1, then $L_5(P_1, P_2)^n$ holds, where P_1 and P_2 are two (not necessarily different) parameters of M.*

Proof. Our proof proceeds in the following three steps:

1) We assume that M is analyzed equivalently under one cs-likely receiver object, O_M. Let M' be obtained from M by augmenting it with (1) "$this^M$ = **new** T // O_M" and (2) "P = $this^M.P$" for every parameter P of M. Let G' be the resulting PAG augmented from G. For every parameter P of M, we now have $P \xrightarrow{\overline{\text{assign}}} this^M \xrightarrow{\overline{\text{new}}} O_M \xrightarrow{\text{cs-likely}} O_M \xrightarrow{\text{new}} this^M \xrightarrow{\text{assign}} P$. Thus, $L_5(P_1, P_2)^n$ holds over G, where P_1 and P_2 are two parameters of M iff $L_5(P', P')^n$ holds over G', where P' is a parameter of M. In L_5, every variable is now guaranteed to point to at least one object, which can be O_M.
2) We show now that all the context-sensitive value flows that enter M under its different calling contexts are tracked in L_5 if they pass through a method call $b = a_0.m_0(a_1, ..., a_r)$ (via $a_0, ..., a_r$). Thus, it suffices to consider each call site in M in isolation. Note that the loads and stores in a program can always be modeled as getters and setters.
   By Lemmas 5 and 6, Lemma 4 applies also to L_5: $L_5(a_i, a_j)$ is established in analyzing $b = a_0.m_0(a_1, ..., a_r)$ iff a_0 points to at least one cs-likely object. Thus, we only need to argue that if a_0 points to only context-insensitive objects, recorded in F_{a_0}, then each invoked method at this call site can be ignored in this sense. In this case (where $O_M \notin F_{a_0}$ as O_M is context-sensitive by construction), if some pointed-to objects of a_0 are missing in F_{a_0} (as our pre-analysis is intra-procedural), then there must exist a call chain, $a_0 = x_1.m_1(...), x_1 = x_2.m_2(...), ..., x_{t-1} = x_t.m_t(...)$ (modeled effectively as $a_0 = x_1 = ... = x_t$ in L_5), where all the pointed-to objects of x_t in the program are found intra-procedurally (under the assumption that all the receiver objects of M are abstracted by one single context-sensitive object, O_M, as explained in Step 1). Since Observation 1 is assumed to be precision-preserving, the value flows that enter M under its different calling contexts (i.e., receiver objects) need not be tracked, i.e., separated context-sensitively at each call site $m_i()$. To prove this claim inductively, we write $x_{-1} = x_0.m_0(...)$ to represent $b = a_0.m_0(...)$. Let R_{m_i} be the set of objects returned by $m_i()$ but missed by L_5, as $m_i()$ is not analyzed. Our claim is true for $x_{t-1} = x_t.m_t(...)$, since all the objects pointed to by x_t in the program are context-insensitive. This also implies that the objects in R_{m_t} are all conflated under different calling contexts of M. Suppose that our claim holds for $m_i()$, in which case, the objects in R_{m_i} are conflated. Let us consider $x_{i-2} = x_{i-1}.m_{i-1}(...)$. As x_{i-1} can only point to either some context-insensitive objects in F_{a_0} found intra-procedurally by L_5 or the conflated objects in R_{m_i}, our claim must also be true for $m_{i-1}()$.
3) If a variable n is context-sensitive by Lemma 1, there must exist a cs-likely O due to Step 1 such that $L_1(O, P)^n$: O *flows* $n' \xrightarrow{\text{store}} P$, which contains n, where n' is a variable (which may be n) and P is a parameter of M. By applying Lemmas 2 – 6 and the result established in Step 2, we must have $L_5(O, P)^n$: O *flows* $n' \xrightarrow{\text{store}} P$ (passing through n). As a result, $L_5(P, P)^n$: $P \xrightarrow{\overline{\text{store}}} n'$ *flows* $O \xrightarrow{\text{cs-likely}} O$ *flows* $n' \xrightarrow{\text{store}} P$ holds. If an object n is context-sensitive by Lemma 1, $L_5(P, P)^n$ can be established similarly.

□

4.2.2.4 Computing CI_{TURNER^m} with a DFA: For a method M, we give an efficient algorithm for computing $CI_{\text{TURNER}^m}(M)$ defined in (21) with a DFA (shown in Figure 12), which is obtained equivalently from the regular grammar for L_5. Our algorithm proceeds in linear time of

the number of nodes in the PAG of M by exploiting an antisymmetric property inherent in our DFA.

The DFA is a quintuple $\mathscr{A} = (Q, \Sigma, \delta, s, e)$, where $Q = \{s, \mathit{flows}, \overline{\mathit{flows}}, e\}$ is the set of states, $\Sigma = \{\mathsf{param}, \overline{\mathsf{param}}, \mathsf{new}, \overline{\mathsf{new}}, \mathsf{assign}, \overline{\mathsf{assign}}, \mathsf{load}, \overline{\mathsf{load}}, \mathsf{store}, \overline{\mathsf{store}}, \mathsf{cs\text{-}likely}\}$ is the alphabet, $\delta : Q \times \Sigma \mapsto Q$ is the state transition function, s is the start state, and e is the accepting, i.e., final state.

Definition 4. *Given a PAG edge $n_1 \xrightarrow{\sigma} n_2$ with a corresponding state transition $\delta(q_1, \sigma) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition.*

We describe an antisymmetric property of our DFA in Lemmas 7 and 8 below.

Lemma 7. *Let n_1 and n_2 be two PAG nodes. We have (1) $(n_1, s) \rightarrowtail^+ (n_2, \overline{\mathit{flows}}) \implies (n_2, \mathit{flows}) \rightarrowtail^+ (n_1, e)$ and (2) $(n_1, s) \rightarrowtail^+ (n_2, \mathit{flows}) \implies (n_2, \overline{\mathit{flows}}) \rightarrowtail^+ (n_1, e)$.*

Proof. To prove (1), we note that n_1 *flows* $n_2 \implies n_2 \; \overline{\mathit{flows}} \; n_1$ in L_5. To prove (2), we note that n_1 *flows* $n \xrightarrow{\overline{\mathsf{store}} \mid \mathsf{store}} n_2 \implies n_2 \xrightarrow{\mathsf{store} \mid \overline{\mathsf{store}}} n \; \overline{\mathit{flows}} \; n_1$ in L_5, where n is a PAG node. □

Lemma 8. *Let n_1 and n_2 be two PAG nodes. We have $(n_2, \overline{\mathit{flows}}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \mathit{flows})$ and $(n_2, \mathit{flows}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \overline{\mathit{flows}})$.*

Proof. Proceeds similarly as in the proof of Lemma 7 by noting [P-PARAM] given in Figure 11. □

In (21), we include a variable/object n in a method M (with its PAG denoted by G_M) into $CI_{\text{TURNER}^m}(M)$ if $L_5^{G_M}(P_1, P_2)^n$ does not hold for any two parameters P_1 and P_2 of M. In terms of our DFA, $L_5^{G_M}(P_1, P_2)^n$ holds iff $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e)$, where $q \in \{\mathit{flows}, \overline{\mathit{flows}}\}$.

The antisymmetric property of our DFA is exploited below.

Theorem 2. *Let n be a variable/object in a method with P_1 and P_2 as its two parameters. $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e) \iff (P_2, s) \rightarrowtail^+ (n, \overline{q}) \rightarrowtail^+ (P_1, e)$, where $q \in \{\mathit{flows}, \overline{\mathit{flows}}\}$.*

Proof. Lemmas 7 and 8. □

As a result, we have designed an efficient algorithm for verifying $L_5^{G_M}(P_1, P_2)^n$ by verifying $n \in R_M(\mathit{flows}) \cap R_M(\overline{\mathit{flows}})$ for a method M (with G_M as its PAG), in which, $R : Q \mapsto \wp(\mathbb{V} \cup \mathbb{H})$ returns a set of nodes in G_M reached at a given state $q \in Q$ and $R^{-1} : \mathbb{V} \cup \mathbb{H} \mapsto \wp(Q)$ is the inverse of R. These two functions are computed according to the two rules given in Figure 13. The two rules are simple: [A-I] performs the initializations needed while [A-II] computes a fixed point for each function iteratively.

Given R_M computed above, we can now obtain $CI_{\text{TURNER}^m}(M)$ given in (21) efficiently as follows:

$$CI_{\text{TURNER}^m}(M) = \{n \mid n \text{ is a node in } G_M \wedge n \notin R_M(\mathit{flows}) \cap R_M(\overline{\mathit{flows}})\} \quad (22)$$

4.3 Conducting Object Reachability for a Program

TURNER^m applies a modular object reachability analysis to the methods in a program according to a reverse topological order of its call graph. This can increase the number of precision-uncritical variables/objects found in a caller method based on the precision-uncritical variables/objects that are already found earlier in its callee methods.

Our final pre-analysis, TURNER^m, given in Algorithm 1, takes a program $\mathscr{P}$ as input and returns CI_{TURNER^m} constructed for $\mathscr{P}$ according to (20) as output. To start with, we obtain a call graph G_{cg} for $\mathscr{P}$ by applying Andersen's analysis [21] (line 1). As Andersen's analysis is context-insensitive, G_{cg} always over-approximates the call graph constructed for $\mathscr{P}$ by kOBJ (with or without selective context-sensitivity being enforced) as desired. Let G_{scc} be obtained from G_{cg} with its strongly connected component (SCCs) being merged (line 2). As G_{scc} is now a DAG, $sccList$ is obtained simply as a list containing all the SCC nodes sorted according to some reverse topological order in G_{scc} (line 3). In lines 4-7, we apply our object reachability analysis to the methods in $\mathscr{P}$ according to the order in which their containing SCCs appear in $sccList$, with all the methods in the same SCC being ordered randomly. For a given method M contained in a SCC scc (lines 4-5), we call `buildPAG(M, scc)` to build its PAG G_M (line 6) and then find the set of precision-uncritical variables/objects in M, i.e., $CI_{\text{TURNER}^m}(M)$, according to (22) (line 7). When building G_M (lines 9-21), we perform one significant optimization enabled by our modular object reachability analysis at a call site $l : b = a_0.m(a_1, ..., a_r)$ in M when all its callee methods in C_M^l are outside scc. In this case, we ignore the value flows leaving an argument a_i if its corresponding parameters $p_i^{m'}$ in all the callee methods contained in C_M^l are precision-uncritical (lines 14-17), since such value-flows are usually context-insensitive (Fact 1). Similarly, we ignore the value flows originating from the return variables in all the callee methods in C_M^l (into b) if all these return variables are precision-uncritical (lines 18-20). Finally, we obtain CI_{TURNER^m} by (20) in line 8 as desired.

TURNER [22] introduced in our preliminary investigation can be regarded as the version of TURNER^m simplified with lines 11-20 in Algorithm 1 ignored, so that TURNER ends up applying our object reachability analysis to the methods in a program independently (in any order). Thus, TURNER^m is always no less effective than TURNER in identifying the context-insensitive variables/objects in a method.

Theorem 3. *For any given method M analyzed by* TURNER^m *and* TURNER, *$CI_{\text{TURNER}^m}(M) \supseteq CI_{\text{TURNER}}(M)$ always holds.*

Proof. TURNER [22] pre-analyzes the methods in a program independently. Thus, G_M built for M by TURNER, denoted G_M^{TURNER}, is simply the one given in line 10 of Algorithm 1. On the other hand, G_M built by TURNER^m must be a subgraph of G_M^{TURNER} due to the existence of lines 11-20 of Algorithm 1. Thus, $CI_{\text{TURNER}^m}(M) \supseteq CI_{\text{TURNER}}(M)$. □

Below we first state the principle behind the development of our modular object reachability analysis in a theorem and then explain why it may cause kOBJ to lose precision only in some rare cases due to type filtering being applied during the pointer analysis.

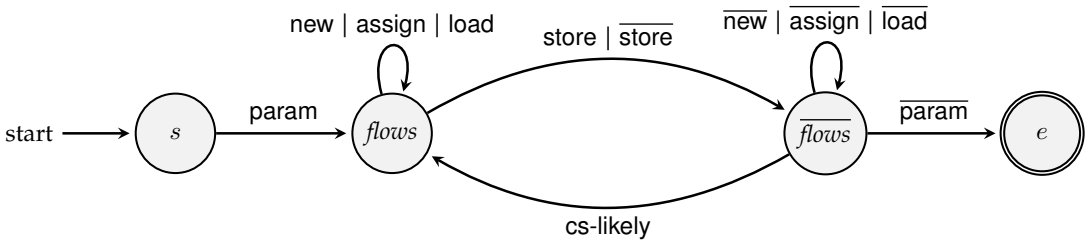

Fig. 12: The DFA representing the regular grammar for defining L_5.

$$\frac{n \in N_M}{n \in R_M(s) \quad s \in R_M^{-1}(n)} \quad \text{[A-I]}$$

$$\frac{n_1 \xrightarrow{\sigma} n_2 \in E_M \quad q_1 \in R_M^{-1}(n_1) \quad \delta(q_1, \sigma) = q_2 \quad q_2 \notin R_M^{-1}(n_2)}{n_2 \in R_M(q_2) \quad q_2 \in R_M^{-1}(n_2)} \quad \text{[A-II]}$$

Fig. 13: Computing R_M and R_M^{-1} for a method M with $G_M = (N_M, E_M)$.

Algorithm 1: The TURNER$^{\mathscr{m}}$ pre-analysis.

Input : A program $\mathscr{P}$
Output: $CI_{\text{TURNER}^{\mathscr{m}}}$
1 $G_{cg} \leftarrow$ call graph of $\mathscr{P}$ built by Andersen's analysis;
2 $G_{scc} \leftarrow$ `CollapsingSCCs` (G_{cg});
3 $sccList \leftarrow$ `ReverseTopologicalSort` (G_{scc});
4 **for** $scc \in sccList$ **do**
5 **for** $M \in scc$ **do**
6 $G_M \leftarrow$ `buildPAG` (M, scc);
7 Build $CI_{\text{TURNER}^{\mathscr{m}}}(M)$ for M by (22);
8 $CI_{\text{TURNER}^{\mathscr{m}}} \leftarrow \bigcup_{M \in \mathbb{M}} CI_{\text{TURNER}^{\mathscr{m}}}(M)$ by (20);
9 **Function** `buildPAG` (M, scc)
10 $G_M \leftarrow$ PAG built for $\mathscr{P}$ as in Figures 7, 9 and 10;
11 **for** $l : b = a_0.m(a_1, ..., a_r) \in M$ **do**
12 $C_M^l \leftarrow$ set of callee methods invoked at l;
13 **if** $C_M^l \cap scc = \varnothing$ **then**
14 **for** $i \in [0, r]$ **do**
15 **if** $\forall\, m' \in C_M^l : p_i^{m'} \in CI_{\text{TURNER}^{\mathscr{m}}}(m')$ **then**
16 Remove $a_i \xrightarrow{\text{store}[p_i^{m'}]} a_0$ from G_M;
17 Remove $a_0 \xrightarrow{\overline{\text{store}[p_i^{m'}]}} a_i$ from G_M;
18 **if** $\forall\, m' \in C_M^l : ret^{m'} \in CI_{\text{TURNER}^{\mathscr{m}}}(m')$ **then**
19 Remove $a_0 \xrightarrow{\text{load}[ret^{m'}]} b$ from G_M;
20 Remove $b \xrightarrow{\overline{\text{load}[ret^{m'}]}} a_0$ from G_M;
21 **return** G_M

Theorem 4. *Let* T-kOBJ (T-kOBJ+M) *be* kOBJ *performed with selective context-sensitivity prescribed by* TURNER (TURNER$^{\mathscr{m}}$). *Suppose that if* kOBJ *is applied to analyze a program,* $\textsf{pts}(a_0, c) = \varnothing \implies \forall\, 1 \leqslant i \leqslant r : \textsf{pts}(a_i, c) = \varnothing$ *always holds for each of its call sites,* $a_0.m(a_1, ..., a_r)$, *analyzed under every possible context* c. *Then* T-kOBJ+M *and* T-kOBJ *yield exactly the same precision (in terms of* $\overline{\textsf{pts}}$*) for the program.*

Proof. By Theorem 3, $CI_{\text{TURNER}^{\mathscr{m}}}(M) \supseteq CI_{\text{TURNER}}(M)$ holds for every method M analyzed by both TURNER$^{\mathscr{m}}$ and TURNER. By noting further lines 11-20 in Algorithm 1, we can conclude that for each call site $a_0.m(a_1, ..., a_r)$ in a program analyzed by kOBJ under every possible context c, $\overline{\textsf{pts}}(a_i)$ obtained under T-kOBJ+M is a strict superset of $\overline{\textsf{pts}}(a_i)$ obtained under T-kOBJ only if $\textsf{pts}(a_0, c) = \varnothing \implies \textsf{pts}(a_i, c) \neq \varnothing$, where $1 \leqslant i \leqslant r$. Thus, under the stated hypothesis, T-kOBJ+M and T-kOBJ will yield the same precision (expressed in terms of $\overline{\textsf{pts}}$) for the given program. □

We use an example (abstracted from JDK) in Figure 14 to help understand this theorem by explaining why T-kOBJ+M may be slightly less precise than T-kOBJ in some rare cases due to type filtering being applied during the pointer analysis. In particular, T-1OBJ has the same precision as 1OBJ but T-1OBJ+M is slightly less precise. We focus on $\overline{\textsf{pts}}$(`v`) obtained in line 27, as it is affected by whether `v1` defined in line 18 is analyzed context-sensitively or not.

- **1OBJ.** In lines 18-19, `v1` and `v2` will be analyzed under two contexts, [`P3`] and [`P4`], due to the two calls in lines 39-40. Under [`P3`], we obtain $\textsf{pts}(\texttt{v1}, [\texttt{P3}]) = \{(\texttt{S1}, [\,])\}$ and $\textsf{pts}(\texttt{v2}, [\texttt{P3}]) = \{(\texttt{S3}, [\,])\}$. Under [`P4`], we obtain $\textsf{pts}(\texttt{v1}, [\texttt{P4}]) = \varnothing$ (due to type-filtering that happens in line 14) and $\textsf{pts}(\texttt{v2}, [\texttt{P4}]) = \{(\texttt{S4}, [\,])\}$. Thus, `endWith()` in line 27 will be analyzed under [`S1`] only, yielding $\textsf{pts}(\texttt{v}, [\texttt{S1}]) = \{(\texttt{S3}, [\,])\}$. Context-insensitively, $\overline{\textsf{pts}}(\texttt{v}) = \{\texttt{S3}\}$.
- **T-1OBJ.** T-1OBJ also analyzes `v1` and `v2` context-sensitively, achieving the same precision as 1OBJ.
- **T-1OBJ+M.** As the two (formal) parameters, `this` and `v`, in `endWith()` are precision-uncritical, TURNER$^{\mathscr{m}}$ will now make both `v1` and `v2` context-insensitive. For T-1OBJ+M, we then have $\textsf{pts}(\texttt{v1}, [\,]) = (\texttt{S1}, [\,])$ and $\textsf{pts}(\texttt{v2}, [\,]) = \{(\texttt{S3}, [\,]), (\texttt{S4}, [\,])\}$. When `endWith()` is analyzed, we obtain $\textsf{pts}(\texttt{v}, [\,]) = \{(\texttt{S3}, [\,]), (\texttt{S4}, [\,])\}$. Context-insensitively, $\overline{\textsf{pts}}(\texttt{v}) = \{\texttt{S3}, \texttt{S4}\}$, where `S4` is spurious. This happens, since, for the call `v1.endsWith(v2)` in line 20, $\textsf{pts}(\texttt{v1}, [\texttt{P4}]) = \varnothing \implies \textsf{pts}(\texttt{v2}, [\texttt{P4}]) = \{(\texttt{S4}, [\,])\} \neq \varnothing$ under 1OBJ (violating the hypothesis stated in Theorem 4 for guaranteeing the precision equivalence between T-1OBJ and T-1OBJ+M). As discussed in Section 5, T-kOBJ+M can be significantly faster than T-kOBJ for some programs despite such a slight precision loss.

In the presence of type filtering, how to improve TURNER$^{\mathscr{m}}$ so that it can preserve the precision of kOBJ without losing much efficiency (under the assumption that

```
1. abstract class Permission {
2.   String name;
3.   Permission(String n1) {
4.     this.name = n1;
5.   }
6.   abstract boolean implies(Permission t);
7. }}
8. class SocketPermission extends Permission {
9.   SocketPermission(String n3) { super(n3); }
10.  boolean implies(Permission p) {
11.    if (!(p instanceof SocketPermission)) {
12.      return false;
13.    }
14.    SocketPermission s = (SocketPermission) p;
15.    return impliesIgnoreMask(s);
16.  }
17.  boolean impliesIgnoreMask(SocketPermission q) {
18.    String v1 = q.name;
19.    String v2 = this.name;
20.    return v1.endsWith(v2);}}
21. class AllPermission extends Permission {
22.   AllPermission(String n2) { super(n2); }
23.   boolean implies(Permission p) {
24.     return true;
25. }}
26. class String {
27.   boolean endsWith(String v) {
28.     return false;
29. }}
30. static void main(String args[]) {
31.   String s1 = new String(); // S1
32.   String s2 = new String(); // S2
33.   String s3 = new String(); // S3
34.   String s4 = new String(); // S4
35.   Permission p1 = new SocketPermission(s1); // P1
36.   Permission p2 = new AllPermission(s2); // P2
37.   Permission p3 = new SocketPermission(s3); // P3
38.   Permission p4 = new SocketPermission(s4); // P4
39.   p3.implies(p1);
40.   p4.implies(p2); }
```

Fig. 14: Precision loss incurred in a program abstracted from real code.

Observation 1 is precision-preserving) can be an interesting research topic. This can be non-trivial, since type filtering done on a receiver object of a method may also filter out the objects flowing into its other parameters context-sensitively but not context-insensitively, as illustrated by this example. One possible solution is to develop a type-filtering-aware constraint solver for pointer analysis so that the effects of type filtering on an receiver object of a method can also be reflected on the other objects passed into the method.

4.4 Time Complexity

The worst-case time complexity of TURNER^m in analyzing a program is linear in terms of its number of statements, for two reasons. First, $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$ given in (5) and (6) can be found in $O(|\mathbb{H}|)$ based on the points-to information already computed by Andersen's analysis [21]. Second, R_M used in (22) for a method M, with its PAG denoted $G_M = (N_M, E_M)$, can be computed by the rules in Figure 13 in $O(|E_M|\times|Q|)$, where $|E_M|$ is the number of edges in G_M (constructed linearly based on the number of statements in M according to the rules in Figures 7 and 9–11) and $|Q|$, i.e., the number of states in the DFA (Figure 12), is 4.

5 EVALUATION

We demonstrate that TURNER^m can accelerate kOBJ significantly with only negligible precision loss, by being both substantially faster than EAGLE [15] (the currently best precision-preserving pre-analysis) and substantially more precise than ZIPPER [20] (the currently best non-precision-preserving pre-analysis). In addition, we also demonstrate that TURNER^m is substantially more effective than TURNER (an earlier version of our pre-analysis [22] under which all the methods in a program are processed independently).

We address the following three research questions:

- RQ1. Is TURNER^m precise?
- RQ2. Is TURNER^m efficient?
- RQ3. Is TURNER^m effective (by exploiting object containment and object reachability)?

We have implemented TURNER^m in SOOT [29], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, SPARK [30], and an object-sensitive version of SPARK (i.e., kOBJ) developed by ourselves. Our pre-analysis is implemented in about 1300 lines of Java code, which has been open-sourced at https://www.cse.unsw.edu.au/ corg/turnerm/. To compare TURNER^m with EAGLE [15] and ZIPPER [20], we have implemented EAGLE based on its three rules (in 600 lines of Java code) and used ZIPPER's latest version (b83b038).

As ZIPPER is evaluated in DOOP [31], we have used a setting as close as possible to its original one in several aspects. First, objects that are instantiated from `StringBuilder` and `StringBuffer` as well as `Throwable` (including its subtypes) are merged per dynamic type and then analyzed context-insensitively as is often done in DOOP [32] and WALA [33]. Second, we perform an exception analysis together with kOBJ as in DOOP by handling exception objects caught in terms of so-called exception-catch links [34]. Third, for type-filtering purposes performed on the elements of an array, we use the declared type of its elements instead of `java.lang.Object`. Finally, we use the summaries provided in SOOT to handle native code.

We have done our experiments on an Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM. We have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo2006 [35], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`), which are commonly used in evaluating kOBJ [12], [14], [17], [18], [36]. The Java library used is `JRE1.6.0_45` (as the DaCapo2006 benchmarks rely only on an older version of JRE). We use TAMI-

FLEX [37], a dynamic reflection analysis tool, to resolve Java reflection as is often done in the pointer analysis literature [11], [12], [15], [17], [20]. We have excluded `jython` and `hsqldb` since their context sensitive analyses do not scale due to overly conservative handling of Java reflection [13].

The time budget used for running each object-sensitive pointer analysis on a program is set as 24 hours. The analysis time of a program is an average of three runs.

Table 3 gives our main results. We compare TURNER^m with EAGLE, ZIPPER and TURNER in terms of their efficiency and precision tradeoffs made on improving kOBJ. For each $k \in \{2, 3\}$ considered, kOBJ is the baseline, Z-kOBJ, E-kOBJ and T-kOBJ are the versions of kOBJ for performing selective context-sensitivity under ZIPPER, EAGLE and TURNER, respectively. T-kOBJ+M is the version of kOBJ for performing selective context-sensitivity under TURNER^m, representing a significant extension of T-kOBJ proposed in this paper for supporting our new modular object reachability analysis.

5.1 RQ1: Precision

Table 3 lists four common metrics used for measuring the precision of a context-sensitive pointer analysis [11], [13], [15], [20] in terms of its context-insensitive points-to information obtained (as described in Section 2.1): (1) *#may-fail-casts*: the number of type casts that may fail, (2) *#call-edges*: the number of call graph edges discovered, (3) *#poly-calls*: the number of polymorphic calls discovered, and (4) *#avg-pts*: the average number of objects pointed by a variable, i.e., the average points-to set size.

EAGLE [15] is designed to be precision-preserving by ensuring that E-kOBJ produces exactly the same context-insensitive points-to information as kOBJ. Thus, E-2OBJ and E-3OBJ achieve trivially the same precision in all the four metrics. ZIPPER [20] is designed to accelerate kOBJ heuristically as much as possible (by also ignoring the last two value-flow patterns in Figure 2) while allowing sometimes a significant loss of precision. For 2OBJ, Z-2OBJ has caused its #avg-pts to increase by 18.1% on average, resulting in the average percentage precision losses of 7.8%, 0.7%, and 1.7% for #may-fail-casts, #call-edges, and #poly-calls, respectively. For 3OBJ, Z-3OBJ has caused its #avg-pts to increase by 16.2% on average, resulting in the average percentage precision losses of 10.8%, 0.7%, and 2.0% for #may-fail-casts, #call-edges, and #poly-calls, respectively.

In this research, we are motivated to develop a pre-analysis approach that can enable kOBJ to run significantly faster while suffering from only a small loss of precision. In our preliminary investigation, [22], TURNER pre-analyzes the methods in a program independently and reasons about all the four value-flow patterns in Figure 2 in each method implicitly using a DFA based on object containment and reachability. Despite some slightly imprecise points-to information produced (with #avg-pts increasing by 0.6% and 0.5% under T-2OBJ and T-3OBJ, respectively, on average), both T-2OBJ and T-3OBJ preserve the precision for #may-fail-casts, #call-edges, and #poly-calls across all the 12 programs. In this paper, TURNER^m is designed to further improve TURNER's efficiency while introducing no or negligible loss of precision, by applying a new modular object reachability analysis. Compared with TURNER, TURNER^m has caused a similar degree of precision loss across the 12 programs for *#avg-pts* (with the same average percentage increases of 0.6% under T-2OBJ+M and 0.5% T-3OBJ+M). For the other three precision metrics, while TURNER^m no longer preserves precision for *#may-fail-casts* (with the average percentage precision losses of 0.3% under T-2OBJ+M and 0.4% under T-3OBJ+M), TURNER^m still preserves precision for *#poly-calls* for all the 12 benchmarks and for *#call-edges* for 10 out of the 12 programs except for `chart` and `findbugs`.

5.2 RQ2: Efficiency

On average, as shown in Table 3, T-kOBJ+M is always faster than E-kOBJ but can sometimes be slower than Z-kOBJ. By adopting the context selections prescribed by each of the four pre-analyses, kOBJ runs faster under all the configurations than before. We compare TURNER^m with EAGLE, ZIPPER, and TURNER below.

- **T-kOBJ+M vs. E-kOBJ.** Both achieve nearly the same precision for #may-fail-casts, #call-edges, and #poly-calls across the 12 programs for $k \in \{2, 3\}$ (as discussed above), but T-kOBJ+M is always faster in each case. For k = 2, the speedups of T-2OBJ+M over 2OBJ range from 2.5x (for `JPC`) to 34.6x (for `findbugs`) with an average of 6.0x. In contrast, the speedups of E-2OBJ over 2OBJ range from 1.4x (for `bloat` and `lusearch`) to 2.7x (for `findbugs`) with an average of 1.8x only. For k = 3, the speedups of T-3OBJ+M over 3OBJ range from 2.9x (for `lusearch`) to 30.5x (for `xalan`) with an average of 8.1x, while the speedups of E-3OBJ over 3OBJ range from 1.1x (for `antlr`, `fop`, `luindex`, `lusearch`, and `pmd`) to 3.8x (for `xalan`) with an average of 1.6x only. Thus, the speedups of T-kOBJ+M over E-kOBJ are 2.9x when k = 2 and 4.0x (with `chart` included even though it cannot be analyzed by 3OBJ scalably) when k = 3.
  In addition, T-kOBJ+M exhibits better scalability than E-kOBJ. For the four programs, `chart`, `eclipse`, `checkstyle` and `findbugs`, that are unscalable under 3OBJ, T-3OBJ+M can now analyze `chart` and `findbugs` but E-3OBJ can analyze `chart` only.
- **T-kOBJ+M vs. Z-kOBJ.** Despite its substantially better precision, T-kOBJ+M is faster in nine programs when k = 2 and three when k = 3. Compared with the kOBJ baseline, the average speedups achieved by T-kOBJ+M and Z-kOBJ are 6.0x and 3.9x, respectively, when k = 2, and 8.1x and 9.3x, respectively, when k = 3. As a result, Z-kOBJ is actually slightly slower than T-kOBJ+M by 0.9x when k = 2 but faster than T-kOBJ+M by 2.1x (with `chart` and `findbugs` included) when k = 3, on average. In terms of scalability, T-kOBJ+M is on par with Z-kOBJ for $k \in \{2, 3\}$.
- **T-kOBJ+M vs. T-kOBJ.** Despite some negligible loss of precision, T-kOBJ+M is not only faster than T-kOBJ across all the 12 programs for $k \in \{2, 3\}$, but also substantially faster for some programs (especially some large ones, such as `chart`, `eclipse`, `xalan`, and `findbugs`). By using kOBJ as the baseline, we can see that T-kOBJ+M is faster than T-kOBJ by boosting the average speedup achieved from 3.6x to 6.0x when

TABLE 3: Main results. For a given $k \in \{2, 3\}$, the speedups of E-kOBJ, Z-kOBJ, T-kOBJ, and T-kOBJ+M are normalized with kOBJ as the baseline. For all the metrics except "Speedup", smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	T-2o+M	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ	T-3o+M
antlr	Time (s)	24.5	12.4	12.7	6.8	5.2	628.9	570.8	141.4	196.5	173.1
	Speedup	-	2.0x	1.9x	3.6x	4.7x	-	1.1x	4.4x	3.2x	3.6x
	#fail-casts	516	516	565	516	518	456	456	513	456	458
	#call-edges	50975	50975	51203	50975	50975	50948	50948	51176	50948	50948
	#poly-calls	1607	1607	1629	1607	1607	1600	1600	1622	1600	1600
	#avg-pts	6.110	6.110	6.585	6.125	6.127	4.927	4.927	5.427	4.945	4.947
bloat	Time (s)	412.6	290.9	324.2	138.9	129.0	10648.2	6994.7	6878.9	1902.8	1734.2
	Speedup	-	1.4x	1.3x	3.0x	3.2x	-	1.5x	1.5x	5.6x	6.1x
	#fail-casts	1295	1295	1349	1295	1297	1198	1198	1256	1198	1200
	#call-edges	56488	56488	56988	56488	56488	56258	56258	56837	56258	56258
	#poly-calls	1549	1549	1587	1549	1549	1535	1535	1577	1535	1535
	#avg-pts	14.796	14.796	15.672	14.816	14.816	13.995	13.995	14.802	14.019	14.019
chart	Time (s)	206.2	107.5	28.3	75.1	63.3	*OoM*	12346.4	522.7	7886.1	5599.5
	Speedup	-	1.9x	7.3x	2.7x	3.3x	-	-	-	-	-
	#fail-casts	1339	1339	1410	1339	1343	-	1239	1316	1239	1243
	#call-edges	72426	72426	73009	72426	72432	-	71987	72640	71987	71993
	#poly-calls	1988	1988	2011	1988	1988	-	1962	1989	1962	1962
	#avg-pts	4.905	4.905	5.363	4.971	4.974	-	4.149	4.799	4.168	4.171
eclipse	Time (s)	10680.5	5885.3	4122.8	4686.0	2649.1	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.8x	2.6x	2.3x	4.0x	-	-	-	-	-
	#fail-casts	3551	3551	3718	3551	3571	-	-	-	-	-
	#call-edges	162208	162208	163186	162208	162208	-	-	-	-	-
	#poly-calls	9525	9525	9572	9525	9525	-	-	-	-	-
	#avg-pts	17.334	17.334	19.691	17.519	17.521	-	-	-	-	-
fop	Time (s)	18.7	10.2	6.9	5.2	5.0	728.1	651.6	123.8	187.3	184.7
	Speedup	-	1.8x	2.7x	3.6x	3.8x	-	1.1x	5.9x	3.9x	3.9x
	#fail-casts	414	414	460	414	416	362	362	416	362	364
	#call-edges	34173	34173	34406	34173	34173	34146	34146	34379	34146	34146
	#poly-calls	816	816	841	816	816	809	809	834	809	809
	#avg-pts	3.577	3.577	4.132	3.597	3.597	3.359	3.359	3.942	3.383	3.384
luindex	Time (s)	15.7	9.4	6.3	4.6	4.3	596.3	532.6	131.7	185.0	172.2
	Speedup	-	1.7x	2.5x	3.4x	3.6x	-	1.1x	4.5x	3.2x	3.5x
	#fail-casts	402	402	455	402	404	348	348	405	348	350
	#call-edges	33449	33449	33689	33449	33449	33422	33422	33662	33422	33422
	#poly-calls	905	905	932	905	905	898	898	925	898	898
	#avg-pts	3.595	3.595	4.285	3.612	3.612	3.352	3.352	4.072	3.374	3.374
lusearch	Time (s)	22.3	15.8	11.1	10.4	8.3	1968.0	1736.8	523.5	881.1	686.7
	Speedup	-	1.4x	2.0x	2.1x	2.7x	-	1.1x	3.8x	2.2x	2.9x
	#fail-casts	417	417	473	417	419	366	366	425	366	368
	#call-edges	36247	36247	36485	36247	36247	36220	36220	36458	36220	36220
	#poly-calls	1103	1103	1131	1103	1103	1096	1096	1124	1096	1096
	#avg-pts	3.611	3.611	4.229	3.627	3.628	3.358	3.358	3.959	3.381	3.381
pmd	Time (s)	42.1	23.9	23.8	18.3	14.1	1504.0	1380.1	358.6	266.2	243.0
	Speedup	-	1.8x	1.8x	2.3x	3.0x	-	1.1x	4.2x	5.7x	6.2x
	#fail-casts	1174	1174	1252	1174	1176	1116	1116	1199	1116	1118
	#call-edges	59664	59664	59832	59664	59664	59599	59599	59767	59599	59599
	#poly-calls	2329	2329	2354	2329	2329	2322	2322	2347	2322	2322
	#avg-pts	4.943	4.943	6.378	4.954	4.954	4.684	4.684	5.973	4.698	4.698
xalan	Time (s)	243.2	121.8	54.2	90.9	82.8	25424.4	6771.9	694.2	1386.4	834.2
	Speedup	-	2.0x	4.5x	2.7x	2.9x	-	3.8x	36.6x	18.3x	30.5x
	#fail-casts	569	569	629	569	571	516	516	582	516	518
	#call-edges	45916	45916	46113	45916	45916	45884	45884	46086	45884	45884
	#poly-calls	1589	1589	1611	1589	1589	1582	1582	1604	1582	1582
	#avg-pts	4.253	4.253	5.258	4.272	4.272	4.096	4.096	5.014	4.119	4.119
checkstyle	Time (s)	1240.6	710.2	484.3	339.3	322.3	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.7x	2.6x	3.7x	3.8x	-	-	-	-	-
	#fail-casts	1129	1129	1203	1129	1131	-	-	-	-	-
	#call-edges	66702	66702	67528	66702	66702	-	-	-	-	-
	#poly-calls	2188	2188	2246	2188	2188	-	-	-	-	-
	#avg-pts	6.380	6.380	10.070	6.491	6.491	-	-	-	-	-
JPC	Time (s)	101.9	59.2	31.0	44.0	41.1	2371.1	1172.9	175.9	316.8	303.1
	Speedup	-	1.7x	3.3x	2.3x	2.5x	-	2.0x	13.5x	7.5x	7.8x
	#fail-casts	1364	1364	1438	1364	1364	1209	1209	1281	1209	1209
	#call-edges	81003	81003	81590	81003	81003	79315	79315	79893	79315	79315
	#poly-calls	4255	4255	4301	4255	4255	4115	4115	4159	4115	4115
	#avg-pts	5.050	5.050	5.486	5.065	5.067	4.434	4.434	4.752	4.458	4.460
findbugs	Time (s)	1820.6	681.1	128.7	150.9	52.6	*OoM*	*OoM*	2133.8	1947.0	1333.9
	Speedup	-	2.7x	14.1x	12.1x	34.6x	-	-	-	-	-
	#fail-casts	2037	2037	2100	2037	2040	-	-	1884	1650	1699
	#call-edges	87532	87532	88134	87532	87532	-	-	87289	86599	86600
	#poly-calls	3472	3472	3487	3472	3472	-	-	3463	3441	3441
	#avg-pts	8.011	8.011	8.804	8.058	8.059	-	-	7.203	6.632	6.636

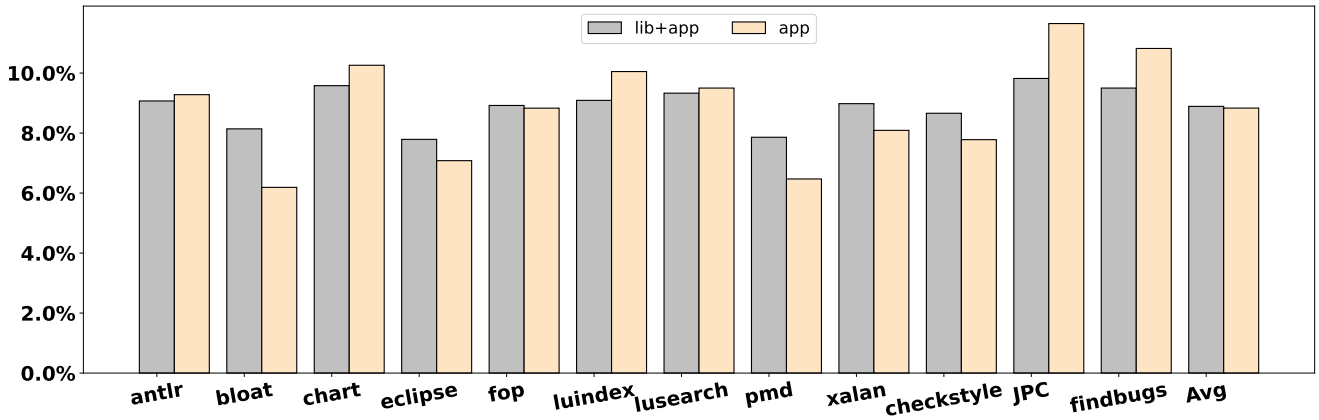

Fig. 15: Percentage increases of context-insensitive variables/objects expressed as $|CI_{\text{TURNER}^m} - CI_{\text{TURNER}}|/|\mathbb{V} \cup \mathbb{H}|$ when TURNER is replaced by TURNER^m for a program (including the JDK and its application code) and its application code only.

k = 2 and from 6.2x to 8.1x when k = 3. Thus, T-kOBJ+M outperforms T-kOBJ by 1.3x when k = 2 and 1.2x when k = 3, on average. However, it should be emphasized that T-kOBJ+M can be substantially faster than T-kOBJ for some large programs (Table 3), achieving, e.g., the speedups of 1.4x for `chart` (when k = 3 by reducing its analysis time from 2.2 hours to 1.6 hours, 1.8x for `eclipse` (when k = 2 by reducing its analysis time from 1.3 hours to 44.2 minutes), 1.7x for `xalan` (when k = 3 by reducing its analysis time from 23.1 minutes to 13.9 minutes), and 2.9x for `findbugs` (when k = 2 by reducing its analysis time from 150.9 seconds to 52.6 seconds).

Figure 15 gives the percentage increases of context-insensitive variables/objects calculated according to $|CI_{\text{TURNER}^m} - CI_{\text{TURNER}}|/|\mathbb{V} \cup \mathbb{H}|$ when we switch from TURNER to TURNER^m (Theorem 3) for a program when it consists of (1) both the JDK and its application code) and (2) its application code only, respectively. For half of the 12 benchmarks evaluated, such percentage increases are slightly higher in their application codes than in their entire programs.

In general, the speedups of T-kOBJ+M over T-kOBJ are not expected to be linearly proportional to such percentage increases, as some context-insensitive variables/objects affect the analysis time of a pointer analysis algorithm more significantly than others (Section 2.2). However, TURNER^m is more effective than TURNER in accelerating kOBJ across the 12 programs while introducing no or little loss of precision.

Table 4 gives the numbers of context-sensitive facts established by kOBJ, E-kOBJ, Z-kOBJ, T-kOBJ, and T-kOBJ+M with #cs-gpts, #cs-pts and #cs-fpts representing the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls representing the number of context-sensitive call edges. In general, the speedups of a pointer analysis over a baseline come from a significant reduction in the number of context-sensitive facts computed by the baseline. For example, Z-3OBJ is significantly faster than E-3OBJ, T-3OBJ and T-3OBJ+M for `chart` as its number of context-sensitive facts is significantly less than the other three. Similarly, T-3OBJ+M and T-3OBJ are much faster than E-3OBJ and Z-3OBJ for `bloat`. However, as is well-known, the analysis time of a pointer analysis is correlated with but not linearly proportional to the number of context-sensitive facts computed [13]. For example, T-3OBJ+M (T-3OBJ) is faster than 3OBJ by 3.6x (3.2x) for `antlr` but achieves a percentage time reduction of 50.3% (49.7%) only.

Table 5 gives the times spent by SPARK [30] (an implementation of context-insensitive Andersen's analysis [21]) and the four pre-analyses, EAGLE, ZIPPER, TURNER and TURNER^m. As discussed earlier, each pre-analysis relies on the points-to information computed by SPARK to make its context selection decisions. TURNER^m, which is as lightweight as TURNER, is significantly faster than EAGLE and ZIPPER across all the 12 programs evaluated. On average, the pre-analysis times of the five tools are 1.1 seconds (TURNER), 1.2 seconds (TURNER^m), 8.9 seconds (EAGLE), 12.2 seconds (ZIPPER), and 14.8 seconds (SPARK), respectively. Note that ZIPPER is multi-threaded (with 16 threads used in our experiments), but SPARK, EAGLE, TURNER and TURNER^m are all currently single-threaded. Without any parallelization, TURNER^m, like TURNER, exhibits already negligible analysis times as it runs linearly in terms of the number of statements in a program (Section 4.4).

5.3 RQ3: Effectiveness

We evaluate the effectiveness of TURNER^m by examining how its two stages contribute in terms of the performance speedups achieved by kOBJ and the number of context-insensitive objects selected, and why Observation 1 causes some small precision loss for #avg-pts but no or little precision loss for #call-edges, #may-fail-casts, and #poly-calls.

TURNER^m relies on object containment and object reachability to make its context selections. In order to understand roughly their percentage contributions to the speedups achieved by T-kOBJ+M over kOBJ, let us consider two versions of T-kOBJ+M: (1) T-kOBJ+M^C, where only object containment is exploited, i.e., the objects in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ are context-insensitive and all the rest (the variables/objects in $(\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$) are

TABLE 4: Context-sensitive facts (in millions). For all the metrics, smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	T-2o+M	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ	T-3o+M
antlr	#cs-gpts	4.0K	3.8K	4.8K	2.2K	2.2K	6.6K	6.0K	12.2K	2.8K	2.8K
	#cs-pts	8.7M	4.9M	8.8M	1.5M	1.3M	83.4M	63.4M	72.4M	33.3M	32.9M
	#cs-fpts	0.4M	0.3M	0.4M	0.2M	0.2M	10.2M	9.9M	10.3M	8.0M	8.0M
	#cs-calls	2.4M	1.8M	1.0M	0.7M	0.6M	38.5M	33.5M	6.8M	25.1M	24.7M
	Total	11.5M	7.1M	10.2M	2.4M	2.1M	132.1M	106.7M	89.6M	66.4M	65.6M
bloat	#cs-gpts	3.2K	3.0K	4.0K	2.2K	2.2K	5.1K	4.3K	11.3K	3.1K	3.1K
	#cs-pts	120.4M	82.4M	111.1M	36.9M	35.6M	1196.0M	856.5M	1137.5M	230.8M	225.7M
	#cs-fpts	4.0M	4.0M	5.1M	3.7M	3.6M	35.8M	35.4M	51.3M	30.6M	30.3M
	#cs-calls	35.5M	32.1M	29.5M	15.0M	14.7M	371.7M	340.5M	298.2M	109.9M	107.9M
	Total	159.9M	118.4M	145.7M	55.6M	53.9M	1603.6M	1232.5M	1487.0M	371.3M	363.8M
chart	#cs-gpts	14.3K	13.0K	10.8K	8.2K	8.2K	-	34.5K	26.3K	22.0K	21.9K
	#cs-pts	64.3M	36.7M	17.0M	19.9M	18.0M	-	1378.0M	171.2M	1005.7M	628.2M
	#cs-fpts	1.5M	1.1M	0.8M	1.0M	1.0M	-	55.4M	24.8M	48.8M	48.1M
	#cs-calls	20.5M	12.2M	2.5M	8.7M	7.6M	-	356.0M	23.9M	260.8M	240.9M
	Total	86.4M	49.9M	20.4M	29.7M	26.6M	-	1789.4M	220.0M	1315.3M	917.2M
eclipse	#cs-gpts	40.6K	39.9K	28.8K	10.0K	10.0K	-	-	-	-	-
	#cs-pts	991.9M	742.7M	744.5M	565.5M	518.9M	-	-	-	-	-
	#cs-fpts	21.8M	21.4M	20.4M	16.2M	16.2M	-	-	-	-	-
	#cs-calls	609.1M	342.7M	188.6M	296.5M	292.2M	-	-	-	-	-
	Total	1622.8M	1106.8M	953.6M	878.2M	827.3M	-	-	-	-	-
fop	#cs-gpts	3.1K	2.9K	3.7K	2.1K	2.1K	4.5K	3.8K	9.8K	2.7K	2.7K
	#cs-pts	3.7M	2.1M	3.6M	1.0M	0.9M	70.3M	56.1M	48.8M	33.5M	33.2M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	9.4M	7.9M	7.9M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	0.4M	33.7M	29.8M	4.2M	25.0M	24.6M
	Total	5.0M	3.2M	4.2M	1.6M	1.4M	113.7M	95.3M	62.5M	66.4M	65.7M
luindex	#cs-gpts	2.8K	2.6K	3.8K	1.9K	1.9K	4.5K	3.9K	11.0K	2.7K	2.7K
	#cs-pts	3.8M	2.2M	4.2M	1.1M	0.9M	67.6M	54.2M	56.5M	33.2M	32.9M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	10.8M	8.0M	8.0M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	0.4M	33.1M	29.6M	4.7M	25.1M	24.7M
	Total	5.2M	3.3M	4.9M	1.7M	1.5M	110.4M	93.2M	72.0M	66.3M	65.6M
lusearch	#cs-gpts	3.0K	2.7K	3.8K	1.9K	1.9K	4.2K	3.5K	10.3K	2.5K	2.5K
	#cs-pts	5.8M	3.9M	5.1M	2.2M	2.0M	167.7M	151.6M	115.3M	92.2M	91.4M
	#cs-fpts	0.3M	0.2M	0.2M	0.2M	0.2M	11.2M	11.0M	11.0M	9.4M	9.4M
	#cs-calls	2.3M	1.9M	1.0M	1.4M	1.3M	108.1M	94.9M	40.5M	80.8M	80.1M
	Total	8.4M	6.0M	6.4M	3.8M	3.5M	287.1M	257.5M	166.9M	182.4M	180.9M
pmd	#cs-gpts	3.9K	3.6K	5.9K	2.5K	2.5K	5.6K	4.9K	23.8K	3.4K	3.4K
	#cs-pts	12.2M	7.6M	15.1M	4.1M	3.9M	144.6M	108.8M	184.5M	45.5M	44.9M
	#cs-fpts	1.1M	1.0M	1.1M	0.9M	0.9M	15.9M	15.3M	19.0M	11.7M	11.7M
	#cs-calls	3.6M	2.6M	2.1M	1.7M	1.6M	58.5M	49.0M	17.0M	33.3M	32.9M
	Total	16.9M	11.1M	18.4M	6.7M	6.3M	219.0M	173.1M	220.5M	90.6M	89.5M
xalan	#cs-gpts	3.9K	3.6K	3.6K	2.4K	2.4K	15.5K	13.5K	10.3K	6.1K	6.1K
	#cs-pts	99.1M	45.9M	20.1M	14.3M	12.7M	1795.3M	987.3M	253.0M	104.5M	92.9M
	#cs-fpts	2.5M	2.4M	1.8M	1.9M	1.9M	70.9M	63.6M	18.8M	27.0M	27.0M
	#cs-calls	26.0M	19.3M	4.7M	17.2M	17.0M	432.4M	300.8M	35.3M	168.1M	167.5M
	Total	127.6M	67.6M	26.6M	33.3M	31.6M	2298.6M	1351.7M	307.1M	299.6M	287.4M
checkstyle	#cs-gpts	7.8K	7.5K	11.5K	3.9K	3.9K	-	-	-	-	-
	#cs-pts	145.0M	107.2M	118.2M	38.0M	30.1M	-	-	-	-	-
	#cs-fpts	2.5M	2.3M	3.0M	1.6M	1.6M	-	-	-	-	-
	#cs-calls	78.6M	34.5M	23.2M	21.1M	19.6M	-	-	-	-	-
	Total	226.1M	144.0M	144.4M	60.7M	51.4M	-	-	-	-	-
JPC	#cs-gpts	7.9K	7.1K	7.7K	5.7K	5.7K	22.1K	19.5K	17.5K	10.2K	10.2K
	#cs-pts	28.7M	18.8M	13.9M	12.1M	11.1M	618.1M	319.8M	68.6M	69.1M	66.1M
	#cs-fpts	1.2M	0.9M	1.0M	0.9M	0.9M	22.8M	20.0M	13.0M	13.0M	13.0M
	#cs-calls	9.6M	7.1M	2.7M	5.8M	5.0M	95.2M	61.4M	7.2M	38.4M	36.9M
	Total	39.6M	26.9M	17.6M	18.8M	17.0M	736.1M	401.3M	88.8M	120.5M	116.0M
findbugs	#cs-gpts	33.5K	32.9K	10.7K	4.0K	4.0K	-	-	45.6K	6.0K	6.0K
	#cs-pts	326.4M	245.0M	57.2M	37.8M	19.3M	-	-	545.9M	183.3M	170.3M
	#cs-fpts	15.7M	15.5M	4.7M	1.1M	1.1M	-	-	59.4M	26.6M	26.6M
	#cs-calls	120.0M	58.3M	11.9M	9.6M	7.4M	-	-	96.4M	138.5M	134.1M
	Total	462.0M	318.9M	73.8M	48.5M	27.8M	-	-	701.7M	348.5M	331.0M

handled as in kOBJ, and (2) T-kOBJ+M^R, where only object reachability is exploited by assuming $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m} = \varnothing$. Let T-$k$OBJ+M$^S_{\text{Speedup}}$ be the speedup obtained by T-kOBJ+M^S over kOBJ, where $S \in \{C, R, \epsilon\}$, for a program. Certainly, T-kOBJ+M$^C_{\text{Speedup}}$ + T-kOBJ+M$^R_{\text{Speedup}}$ = T-kOBJ+M$_{\text{Speedup}}$ is not expected for a program, as the common contribution made by T-kOBJ+M^C and T-kOBJ+M^R towards T-kOBJ+M$_{\text{Speedup}}$ cannot be meaningfully isolated. Instead, we consider T-kOBJ+M$^S_{\text{Speedup}}$/(T-kOBJ+M$^C_{\text{Speedup}}$ + T-kOBJ+M$^R_{\text{Speedup}}$), where $S \in \{C, R\}$, as the relative percentage contribution made by T-kOBJ+M^S towards T-kOBJ+M$_{\text{Speedup}}$ in order to gain a rough understanding about whether both stages are indispensable. Figure 16 illustrates the case for accelerating 2OBJ by T-2OBJ+M, demonstrating that both object containment and object reachability are indeed exploited beneficially for real-world programs.

In this paper, our research is driven by three insights, stated in Observation 1, Theorem 1, Theorem 4, respectively. Therefore, TURNER^m is designed to exploit both object containment and object reachability modularly to classify

TABLE 5: Times spent by SPARK and the four pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs	Avg
SPARK	9.0	10.7	17.2	38.6	8.1	7.4	7.9	13.5	9.5	16.8	19.3	19.8	14.8
EAGLE	3.5	3.8	9.9	34.6	2.8	2.7	3.0	9.3	6.1	9.2	9.6	12.1	8.9
ZIPPER	5.4	6.5	17.1	38.9	4.4	4.2	4.6	9.5	9.0	17.9	11.5	17.4	12.2
TURNER	0.8	0.9	1.4	2.4	0.5	0.5	0.5	1.1	0.8	1.2	1.2	1.3	1.1
TURNER^m	0.8	1.1	1.3	3.2	0.7	0.6	0.6	1.3	0.8	1.5	1.4	1.6	1.2

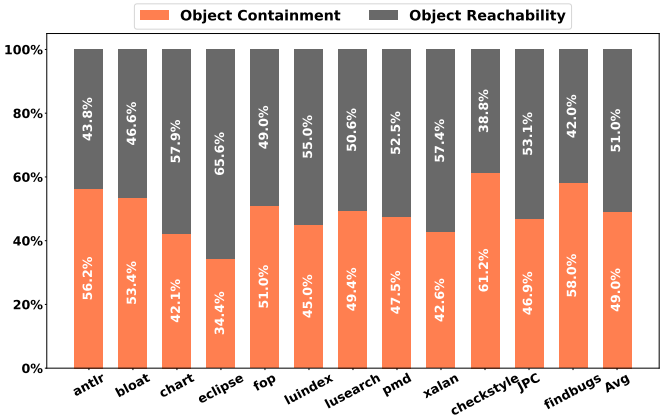

Fig. 16: Percentage contributions made by TURNER^m's two analysis stages for the speedups of T-2OBJ+M over 2OBJ.

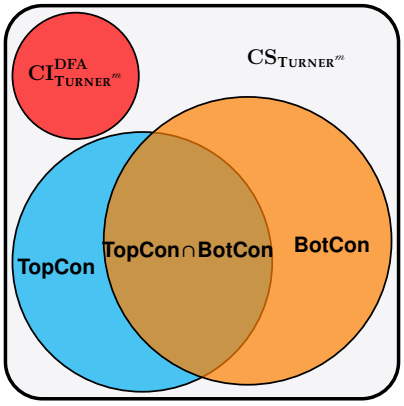

Fig. 17: The Venn diagram of the objects in a program, where $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ = TopCon ∪ BotCon according to (5).

the objects, and consequently, the variables in a program as context-sensitive or context-insensitive.

Figure 17 gives a Venn diagram showing how TURNER^m classifies the containers, i.e., objects in a program. Based on object containment (Observation 1), $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ = TopCon ∪ BotCon gives the set of precision-uncritical, i.e., context-insensitive objects identified. Based on object reachability (performed by our DFA), $CI^{DFA}_{\text{TURNER}^m} \subseteq \mathbb{H} \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ gives an additional set of context-insensitive sets found. Thus, $CS_{\text{TURNER}^m} = \mathbb{H} \setminus (\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m} \cup CI^{DFA}_{\text{TURNER}^m})$ represents the set of context-sensitive objects identified. On average, across the 12 programs evaluated, the ratios of $|\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}|$, $|CI^{DFA}_{\text{TURNER}^m}|$ and $|CS_{\text{TURNER}^m}|$ over $|\mathbb{H}|$ are 61.3%, 5.5%, and 33.2%, respectively. As the performance benefits of making different objects context-insensitive can be drastically different (which are hard to measure individually), these ratios, together with Figure 16, demonstrate again the effectiveness of TURNER^m's two analysis stages.

Finally, we give two examples abstracted from the JDK library to explain why TURNER^m causes kOBJ to suffer from a small loss of precision in #avg-pts but no or a negligible loss of precision in #call-edges, #may-fail-casts, and #poly-calls across the 12 programs evaluated. TURNER^m can render some points-to sets imprecise when some top/bottom containers that are classified as precision-uncritical in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ should have been analyzed context-sensitively.

Figure 18 illustrates a case in which whether the object `P` created in line 4 (a top container according to Observation 1) is analyzed context-sensitively or not affects $\overline{\mathsf{pts}}$(`str`) obtained in line 23. Consider 2OBJ, which will analyze `P` context-sensitively. When analyzing lines 19–22, we find that $\mathsf{pts}(ui,[\,]) = \{(Ui,[\,])\} \wedge \mathsf{pts}(Ui.\texttt{file},[\,]) = \mathsf{pts}(\texttt{P.path},[Ui]) = \{(\texttt{Si},[\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 23, we find that $\mathsf{pts}(\texttt{str},[\,]) = \{(\texttt{S1},[\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\mathsf{pts}}(\texttt{str}) = \{\texttt{S1}\}$. In the case of T-2OBJ+M, however, `P` ∈ $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ will be analyzed context-insensitively instead. When analyzing lines 19–22, we have $\mathsf{pts}(ui,[\,]) = \{(Ui,[\,])\} \wedge \mathsf{pts}(Ui.\texttt{file},[\,]) = \mathsf{pts}(\texttt{P.path},[\,]) = \{(\texttt{S1},[\,]),(\texttt{S2},[\,])\}$, where $1 \leqslant i \leqslant 2$. As `P` is context-insensitive, analyzing line 23 this time will give rise to $\mathsf{pts}(\texttt{str},[\,]) = \{(\texttt{S1},[\,]),(\texttt{S2},[\,])\}$. Thus, context-insensitively, T-2OBJ+M yields $\overline{\mathsf{pts}}(\texttt{str}) = \{\texttt{S1},\texttt{S2}\}$, which contains a spurious target `S2` introduced for `str`. Despite this loss of precision in #avg-pts, however, T-2OBJ+M (just like T-2OBJ) does not lose any precision in #may-fail-casts, #call-edges, and #poly-calls, as both `S1` and `S2` have exactly the same type, `java.lang.String`.

Figure 19 illustrates another case in which whether the object `D` created in line 14 (a bottom container according to Observation 1) is analyzed context-sensitively or not affects $\overline{\mathsf{pts}}$(`t`) obtained in line 7. Consider 2OBJ, which will analyze `D` context-sensitively. When analyzing lines 17–20, we find that $\mathsf{pts}(vi,[\,]) = \{(Vi,[\,])\} \wedge \mathsf{pts}(Vi.\texttt{buffer},[\,]) = \{(\texttt{D},[Vi])\} \wedge \mathsf{pts}(\texttt{D.buf},[Vi]) = \{(Bi,[\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 7, we find that $\mathsf{pts}(\texttt{t},[\texttt{D},\texttt{V1}]) = \{(\texttt{B1},[\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\mathsf{pts}}(\texttt{t}) = \{\texttt{B1}\}$. In the case of T-2OBJ+M, however, `D` ∈ $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}^m}$ will be analyzed context-insensitively instead. When analyzing lines 17–20, we have $\mathsf{pts}(vi,[\,]) = \{(Vi,[\,])\} \wedge \mathsf{pts}(Vi.\texttt{buffer},[\,]) = \{(\texttt{D},[\,])\} \wedge \mathsf{pts}(\texttt{D.buf},[\,]) = \{(Bi,[\,])\}$, where $1 \leqslant i \leqslant 2$. As `t` is context-insensitive, analyzing line 7 will give rise to $\mathsf{pts}(\texttt{t},[\,]) = \{(\texttt{B1},[\,]),(\texttt{B2},[\,])\}$. Thus, context-insensitively, T-2OBJ+M yields $\overline{\mathsf{pts}}(\texttt{t}) = \{\texttt{B1},\texttt{B2}\}$, which contains a spurious target `B2` introduced for `t`. Despite this loss of precision in #avg-pts, T-2OBJ+M (just like T-2OBJ) loses no precision in #may-fail-casts, #call-edges, and #poly-calls, as both `B1` and `B2` have exactly the same type, `java.lang.byte[]`, and in addition, each array object pointed by `t` is used in

```
1. class URL {
2.    String file;
3.    URL(String s) {
4.       Parts parts = new Parts(s); // P
5.       this.file = parts.getPath();
6. }
7.    String getFile() {
8.       return this.file;
9. }}
10. class Parts {
11.    String path;
12.    Parts(String p) {
13.       this.path = p;
14. }
15. String getPath() {
16.       return this.path;
17. }}

18. void main() {
19.    String s1 = new String(); // S1
20.    String s2 = new String(); // S2
21.    URL u1 = new URL(s1); // U1
22.    URL u2 = new URL(s2); // U2
23.    String str = u1.getFile();
24.    InputStream in = new FileInputStream(str);
25.    // parse content of the Stream.
26.    in.close();
27. }
```

Fig. 18: An example with imprecise points-to information computed by T-2OBJ+M (and T-2OBJ) for a top container P.

```
1. class DerInputBuffer {
2.    byte[] buf;
3.    DerInputBuffer (byte[] p) {
4.       this.buf = p;
5.    }

6.    Date getTime() {
7.       byte[] t = this.buf;
8.       long l = t[0];
9.       return new Date(l);
10. }}
11. class DerValue {
12.    DerInputBuffer buffer;
13.    DerValue(byte[] buf) {
14.       this.buffer = new DerInputBuffer(buf); // D
15. }}
16. void main() {
17.    byte[] b1 = new byte[10]; // B1
18.    byte[] b2 = new byte[10]; // B2
19.    DerValue v1 = new DerValue(b1); // V1
20.    DerValue v2 = new DerValue(b2); // V2
21.    Date d1 = v1.buffer.getTime();
22. }
```

Fig. 19: An example with imprecise points-to information computed by T-2OBJ+M (and T-2OBJ) for a bottom container D.

line 8 for obtaining a long integer only.

6 Related Work

We review only existing pre-analysis techniques developed for accelerating whole-program context-sensitive pointer analysis algorithms that represent calling contexts by context strings such as object-sensitivity and callsite-sensitivity. There are other types of approaches for conducting pointer analysis. Thiessen and Lhoták [13] propose to use context transformations rather than context strings as a new context abstraction for kOBJ, making it theoretically possible for kOBJ to run more efficiently with better precision. Instead of solving kOBJ as a whole-program analysis [10], [30], [32], [38], [39], demand-driven pointer analyses [23], [26], [27], [40], [41], [42] typically compute the points-to information for particular variables of interest, with call-site-sensitivity instead of object-sensitivity being often used.

There are two approaches for developing pre-analyses for improving the efficiency and scalability of object-sensitive pointer analysis (kOBJ) for Java: the precision-preserving approach [15] and non-precision-preserving approach [17], [18], [19], [20]. EAGLE [15] aims to improve the efficiency of kOBJ while preserving its precision by reasoning about all the four value-flow patterns in Figure 2 implicitly via CFL reachability to make its context selections conservatively, thereby limiting the speedups achieved. In this paper, TURNER^m addresses its limitation by trading a slight loss of precision for greater performance speedups. On the other hand, ZIPPER [20], as a representative non-precision-preserving pre-analysis [17], [18], [19], [20], aims to trade precision for efficiency by examining the first two value-flow patterns in Figure 2 heuristically to make its context selections, achieving sometimes greater speedups than EAGLE but at a substantial loss of precision for some programs. In this paper, TURNER^m addresses its limitation by trading possibly a slight loss of efficiency for greater precision. TURNER^m achieves this by exploiting object containment (Observation 1) and reasoning about all the four value-flow patterns in Figure 2 implicitly via an a new modular object reachability analysis (Theorems 1, 3 and 4).

In comparison with our earlier conference paper [22], where TURNER is introduced, we have made a number of significant contributions in introducing TURNER^m in this journal paper. First, TURNER^m differs from TURNER by performing a novel modular object reachability analysis in a program according to a reverse topological order of its call graph, boosting the performance of kOBJ more substantially (especially for large programs) while introducing no or little precision loss. Their key difference is also illustrated by a motivating example given in Section 2. Second, we have now formalized our new pre-analysis precisely in terms of an algorithm presented in Algorithm 1. Third, we provide a theoretical justification for the superiority of TURNER^m over TURNER (Theorems 3 and 4), providing insights for

developing better pre-analyses in future work. Fourth, we recognize that TURNER^m may be slightly less precise than TURNER due to type-filtering that happens during the pointer analysis (Figure 14), suggesting that some new type-aware pointer analysis algorithms may be developed to eliminate such type-filtering-induced imprecision (although this can be non-trivial (Section 4.3)). Finally, we have open-sourced our TURNER^m analysis framework to enable other researchers to leverage it to develop new pointer analyses and other down-stream client analysis tools.

There are other types of pre-analyses for kOBJ. MAHJONG [12] sacrifices the precision of alias analysis (by merging objects of the same dynamic type) in order to improve the efficiency of kOBJ at a small loss of precision for a class of so-called type-dependent clients, such as call graph construction, may-fail casting, and polymorphic call detection. In contrast, TURNER^m is designed to be a general-purpose pointer analysis to support all possible applications that rely on points-to information, including not only type-dependent clients but also alias analysis. Jeong et al. [18] apply machine learning to select the lengths of calling contexts for different methods analyzed by kOBJ for a particular client (e.g., may-fail-casting). In contrast, TURNER^m makes its context selections by exploiting object containment and modularity-enabled object reachability.

There are also research efforts for developing pre-analyses for other programming languages. For example, Wei and Ryder [43] present an adaptive context-sensitive analysis for JavaScript. They extract user-specific function characteristics from an inexpensive pre-analysis and then apply a decision-tree-based machine learning technique to correlate these features with different types of context-sensitivity, e.g., 1-callsite, 1-object and i-th-parameter, achieving better precision and efficiency than any single context-sensitive analysis evaluated.

Elsewhere [14], [36], [44], pre-analyses are applied to improve the precision of kOBJ at the cost of its efficiency. This line of research is orthogonal to ours considered here.

In some recent CFL-reachability-guided pre-analyses [15], [22], CFLs are approximated by regular languages in order to make such pre-analyses lightweight. Mohri and Nederhof [45] introduce an approach for over-approximating a context-free grammar (CFG) by a non-deterministic finite automaton (NFA). Prasanna *et al.* [46] adopt this approach to compute the liveness information required by a garbage collector for functional programs. For object-oriented pointer analysis, however, our work is the first for introducing an intra-procedural pre-analysis for determining selective context-sensitivity in a program, based on a DFA over-approximated from a CFG that defines pointer analysis inter-procedurally.

7 CONCLUSION

We have introduced TURNER^m, a simple, lightweight yet effective pre-analysis that can accelerate object-sensitive pointer analysis for Java programs with negligible precision loss. We exploit a key insight that many precision-uncritical objects in the program can be identified based on a pre-computed object containment relationship. Leveraging this approximation, we rely on a modular object reachability analysis to determine whether the remaining objects, together with all the variables, in the program are precision-critical or not. As a result, we obtain a new pre-analysis (already open-sourced) that can improve the efficiency of object-sensitive pointer analysis significantly while introducing only some small precision loss into the points-to information produced. In addition, there is no or little precision loss observed for three important clients, call graph construction, may-fail casting, and polymorphic call detection, over a set of 12 popular Java programs evaluated.

This research can be extended in several directions. First, we can incorporate the object allocation relationship (exploited earlier [36]) into our framework to mitigate some precision loss incurred in the scenarios illustrated in Figures 18 and 19. Second, we can explore with sharpening the precision of $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}^m}$ with a more precise yet faster algorithm than Anderson's analysis [21]. Finally, we can try to generalize TURNER^m to work for other types of context-sensitivity, such as call-site-based context-sensitivity.

ACKNOWLEDGMENTS

Thanks to all the reviewers for their constructive comments. This research is supported by ARC Grants DP180104069 and DP210102409.

REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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*. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269.

[2] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, 2019, pp. 267–279.

[3] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 2, pp. 1–34, 2008.

[4] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[5] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 112–122.

[6] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program tailoring: Slicing by sequential criteria," in *30th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 15:1–15:27.

[7] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 308–319.

[8] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in Android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 167–177.

[9] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

[10] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.

[11] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 17–30.

[12] T. Tan, Y. Li and J. Xue, "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*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 278–291.

[13] R. Thiessen and O. Lhoták, "Context transformations for pointer analysis," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017, p. 263–277.

[14] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[15] J. Lu, D. He, and J. Xue, "Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity," *ACM Transactions on Software Engineering and Methodology*, 2021, to appear.

[16] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 162–186, 2000.

[17] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 485–495.

[18] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 100, 2017.

[19] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu, "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*. New York, NY, USA: Association for Computing Machinery, 2017, p. 13–18.

[20] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[21] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[22] D. He, J. Lu, Y. Gao, and J. Xue, "Accelerating object-sensitive pointer analysis by exploiting object containment and reachability," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, ser. LIPIcs, A. Møller and M. Sridharan, Eds., vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 16:1–16:31.

[23] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2006, p. 387–400.

[24] J. Kodumal and A. Aiken, "The set constraint/cfl reachability connection in practice," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2004, pp. 207–218.

[25] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Berlin, Heidelberg: Springer, 2013, pp. 196–232.

[26] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for Java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: Association for Computing Machinery, 2005, p. 59–76.

[27] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 264–274.

[28] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. USA: IBM Corp., 2010, pp. 214–224.

[30] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using spark," in *International Conference on Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–169.

[31] Y. Smaragdakis, "Doop-framework for Java pointer and taint analysis (using p/taint)," 2021.

[32] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 243–262.

[33] I. T. W. R. Center, "WALA: T.J. Watson Libraries for Analysis," 2020. [Online]. Available: http://wala.sourceforge.net/

[34] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: Better together," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–12.

[35] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapobenchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 169–190.

[36] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *International Static Analysis Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 489–510.

[37] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*. Honolulu, HI, USA: IEEE, 2011, pp. 241–250.

[38] J. Whaley and M. S. Lam, "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*. New York, NY, USA: Association for Computing Machinery, 2004, pp. 131–144.

[39] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 343–353.

[40] D. Yan, G. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for Java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 155–165.

[41] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 460–473.

[42] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java," in *30th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26.

[43] S. Wei and B. G. Ryder, "Adaptive context-sensitive analysis for JavaScript," in *29th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 712–734.

[44] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–434.

[45] M. Mohri and M.-J. Nederhof, "Regular approximation of context-free grammars through transformation," in *Robustness in Language and Speech Technology*, J.-C. Junqua and G. van Noord, Eds. Dordrecht: Springer Netherlands, 2001, pp. 153–163.

[46] P. Kumar K., A. Sanyal, and A. Karkare, "Liveness-based garbage collection for lazy languages," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 122–133.