2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Module-Aware Context Sensitive Pointer Analysis

Haofeng Li[†], Chenghang Shi[†‡*], Jie Lu[†], Lian Li[†‡§*] and Zixuan Zhao[¶]
[†] SKLP, Institute of Computing Technology, CAS, China
[‡] University of Chinese Academy of Sciences, China
[§] Zhongguancun Laboratory, China
[¶] Huawei Technologies Co. Ltd, China
[†] {lihaofeng, shichenghang21s, lujie, lianli}@ict.ac.cn

***Abstract*—The Java Platform Module System (`JPMS`) has found widespread applications since introduced in Java 9. However, existing pointer analyses fail to leverage the semantics of `JPMS`. This paper presents a novel module-aware approach to improving the performance of pointer analysis. We model the semantics of keywords *provides* and *uses* in `JPMS` to recover missing points-to relations. We design a module-aware context-sensitive analysis, which can propagate and apply critical contexts (by exploiting modularity) to balance precision and efficiency better. We have implemented our module-aware pointer analysis named MPA in Tai-e and conducted extensive experiments to compare it with standard object-sensitivity. The evaluation results demonstrate that MPA finds more reachable methods and enhances existing context-sensitive approaches, striking a good balance between efficiency and precision. MPA can increase the number of reachable methods up to $90.9\times$ (`lombok`) under the same analysis. Performance-wise, MPA is nearly as fast as context-insensitivity for most benchmarks, while its precision is superior to that of 1-object-sensitivity on average.**

***Index Terms*—Pointer Analysis, Context Sensitivity, JPMS**

I. Introduction

Pointer analysis statically determines the points-to set of a pointer variable p, i.e., the set of abstract memory locations p may point to at runtime. Such pointer information serves as a foundation for an array of applications such as information flow analysis [1]–[3], program understanding [4], [5], bug detection [6]–[8], and program optimization [9]–[11], to name just a few. The ability of a pointer analysis to find more reachable methods precisely is crucial for the effectiveness of its client applications. For example, a taint analysis [1] may fail to capture information leaks if the underlying pointer analysis fails to find enough methods. In contrast, the imprecision of a pointer analysis can lead to over-tainting issues, causing poor performance and overwhelming false-positive reports, which makes the taint analysis less practical.

A prevalent approach to enhancing the precision of pointer analysis is to employ context-sensitivity [12]–[15]. Context-sensitive pointer analysis decorates a pointer with different calling contexts (representing distinct runtime paths), where a context is represented as a sequence of *context elements* $[e_k...e_1]$. According to context elements, there are three mainstream variants of context sensitivity, call-site-sensitivity (e_i is call site), object-sensitivity (e_i is receiver object), and type-sensitivity (e_i is the type of receiver object or type that contains the method which allocates receiver object). For object-oriented programs, object-sensitivity is believed to be a better choice than call-site-sensitivity [12], [13], and type-sensitivity is regarded as a more efficient, but less precise alternative to object-sensitivity [15]. Besides, choosing a proper set of context elements can also boost the performance and precision of a pointer analysis [16].

To ease software development and maintenance, modern software systems are organized into multiple modules, and commonly implemented based on other projects (such as standard libraries), which are also organized by modules. The concept of *module system* has gained popularity, such as the Java Platform Module System (`JPMS`) introduced in Java 9. The module system provides stronger encapsulation for code and imposes stricter constraints on access which are critical semantic information for pointer analysis.

Unfortunately, existing pointer analyses are unaware of the module system behind the program code. Specifically, if part of the program semantics is encoded in the `JPMS` specification, a pointer analysis would miss certain points-to relations if the specification is overlooked. For example, in one of the benchmarks we evaluated (`lombok`), a standard implementation of pointer analysis only discovered 16 application methods. Moreover, the internal implementation logic of a module can be quite intricate, which often confuses pointer analysis. Particularly, a standard k-limiting pointer analysis always chooses the most recent k context elements, making it ineffective in choosing appropriate context elements when analyzing a complex module. As a result, existing techniques often waste a lot of time for context sensitivity with little precision gain.

This paper aims to improve the preformance of pointer analysis for `JPMS`-based Java programs. Our observation is two-fold:

- Missing points-to relations. A pointer analysis can find the originally missed points-to relations when equipped with the semantics extracted from the `JPMS` specification.
- Precision. The module system can guide a pointer analysis to select critical context elements to exploit the tradeoff between precision and efficiency.

Based on the above insight, we propose the first *module-aware* pointer analysis to improve the precision of pointer analysis for `JPMS` and recover missing points-to relations. We have implemented our analysis on top of Tai-e [17], a recent

*Corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00227

static analysis framework for Java, and performed extensive experiments on the DACAPO benchmarks and seven popular real-world programs. The empirical results demonstrate a drastic improvement in precision and missing reachable methods by our approach.

To sum up, this paper makes the following contributions:

- We propose to model the semantics of keywords *provides* and *uses* in JPMS to recover missing points-to relations.
- We present a module-aware context-sensitive approach, which employs critical context elements to exploit a sweet spot between precision and efficiency. To this end, we introduce *module depth graph* (Definition III.1) – an extension of object allocation graph [18]– with an efficient on-the-fly construction algorithm.
- We realize our module-aware approach as MPA and perform extensive experiments on our benchmarks. Empirical results show that MPA finds more reachable methods and runs as fast as context-insensitivity for most benchmarks while obtaining better precision than 1-object-sensitivity.

The rest of the paper is organized as follows. Section II motivates our approach with an example. Section III formally describes how to model the semantics of keywords *provides* and *uses* in JPMS and apply module-aware context sensitivity. We evaluate the effectiveness and efficiency of module-aware pointer analysis in Section IV. Section V reviews related work and Section VI concludes this paper.

II. MOTIVATION

In this section, we first briefly introduce the module system. Then we use two examples to illustrate how to improve the performance of pointer analysis by utilizing the semantics of the module system.

A. Module System

The *module system* is gradually being introduced as a first-class citizen in programming languages. For instance, the module system was introduced in Java 9 named Java Platform Module System (JPMS), whose main purpose is to organize Java applications into modules for encapsulation. To define a module, JPMS introduces a specific file called "*module-info*", which uniquely defines the name and constraints of the module. The file begins with a module definition of the module name using the keyword *module*. Following the module definition, five principal directives, namely *requires*, *exports*, *provides*, *uses*, and *opens*, are employed to manage dependencies, encapsulate module internals, facilitate service providers and consumers, and reflective access and openness. The details can be found on the official website of ORACLE [19]. We will introduce these directives with code examples in the following sections. In addition, C++ 20 also introduced *module system*.

B. Service Provider Interface (SPI)

In this section, we will introduce two directives (*provides*, *uses*) and demonstrate their impact on points-to relations.

The SPI is an API that defines a set of plugin-style services or drivers, allowing applications to discover and load implementations of specific services at runtime. The idea behind the SPI is to separate the definitions (interfaces or abstract classes) of services from their implementations. Since Java 9, JPMS introduces the two keywords – *provides* and *uses* – to define the mappings from service definitions to implementations. The SPI scheme, with its features of loose coupling, extensibility, and dynamic loading, is friendly to software development but brings a lot of inconvenience to static program analysis. Due to the separation of interface and implementation, the implementation classes are not explicitly initialized in the code, making it impossible for existing pointer analyses to determine the specific points-to relationship of interface variables, leading to unsound results.

Let us study SPI since Java 9 with an example in Figure 1. There are two modules, M1 and M2, in the example. In M2 module, there are two classes, Dog (lines 16-18) and Cat (lines 19-21), both of which implement the Animal interface (lines 13-15). The *module-info* file of M2 (lines 22-24) defines the mappings from the Animal interface to its implementations, Dog and Cat, using the *provides* keyword. The M1 module depends on M2 module and uses the Animal interface according to the keywords, *requires* and *uses*, in lines 9-12. The main method demonstrates the use of the Java ServiceLoader mechanism to dynamically load and instantiate classes that implement the Animal interface. The ServiceLoader is a utility provided by the Java platform that enables the discovery and loading of service providers at runtime. At line 2, the ServiceLoader.load() method is invoked with the Animal interface as its argument, indicating that we are seeking implementations of this interface. At line 3, an iterator containing implementations corresponding to the Animal interface is obtained by invoking the iterator() method of the ServiceLoader. This iterator allows us to iterate and dynamically instantiate all available implementations of the Animal interface. According to the *module-info* files of M1 and M2, Dog and Cat are the implementations of the Animal interface. So, the target method at line 6 should be the run method of Dog and Cat. To construct a complete call graph, the SPI should be modeled precisely to maintain the mappings from interfaces to implementations. Firstly, we need to parse the *module-info* files of M1 and M2 to extract mappings from Animal to Dog and Cat. Then we need to bind the mappings to variable loader at service loading points (line 2) and maintain it across the iterator (lines 3-5). Finally, we retrieve implementations from the iterator, create Dog and Cat objects, assign them to the variable a, and restore missing points-to relations and call graphs.

C. Precision

In Java 9, the JDK is divided into 99 modules, with the java.base module being the most fundamental and core module. It contains essential packages, such as java.lang, java.util, java.io, etc. Next, we will illustrate how to enhance the performance of context-sensitive pointer analysis

```
1  void main() {
2      ServiceLoader<Animal> loader =
           ServiceLoader.load(Animal.class);
3      Iterator<Animal> iter = loader.iterator();
4      while(iter.hasNext()) {
5          Animal a = iter.next();
6          a.run();
7      }
8  }
9  module M1 {
10     requires M2;
11     uses Animal;
12 }
```

```
13 interface Animal {
14     public void run();
15 }
16 class Dog implements Animal {
17     public void run() {}
18 }
19 class Cat implements Animal {
20     public void run() {}
21 }
22 module M2 {
23     provides Animal with Dog, Cat;
24 }
```

Fig. 1: An example of SPI with two keywords, *uses* and *provides* in JPMS.

by leveraging the semantics of the module system, using a simplified code example of `java.util.HashMap` from the `java.base` module.

As shown in Figure 2, the simplified code snippet of `HashMap` is given in lines 19 to 41. In this module, the `java.util` package is exported (line 40), allowing types in this package to be accessed by modules that require this module. The `HashMap` stores data in `table`, an array of `Node` objects (line 20). The `put` method creates a `Node` object and stores it in `table` (lines 21-24). The `get` method retrieves the corresponding `Node` object from `table`, then returns its value via the `getValue` method (lines 25-28).

There are two modules, `M3` and `M4`, where the methods `foo` and `bar` are defined, respectively. Both modules depend on the `java.base` module according to their *module-info* files (lines 7-9 and lines 16-18). It should be noted that the `java.base` module is always implicitly required by all other modules, meaning that developers do not need to explicitly declare a dependency on `java.base`. In the `foo` and `bar` methods, there are `HashMap` objects, O_1 (line 2) and O_2 (line 11), respectively. Object O_A is created and put into O_1 at line 3, then retrieved back via the `get` method at line 4. Similarly, object O_B is created and put into O_2 at line 12, then retrieved back at line 13. As a result, the two cast operations (lines 5 and 14) will never fail.

In a 1-object sensitive analysis (abbreviated as `1obj`), the receiver objects for the calls to the `put`/`get` methods at line 3/4 and line 12/13 are O_1 and O_2, respectively. Hence, the call to `put`/`get` methods at different call-sites can be distinguished using contexts $[O_1]$ and $[O_2]$. In `put` (lines 21-24), with `1obj` analysis, we get $pts([O_1],\texttt{n}) = \{O_4\}$ and $pts([O_2],\texttt{n}) = \{O_4\}$. Then, in the constructor of `Node` (lines 31-33), since O_4 is the only receiver object, we get

```
1  void foo() {
2      HashMap map1 = new HashMap();//O1
3      map1.put("A", new A());//OA
4      Object v1 = map1.get("A");
5      A a = (A) v1;//cast may fail?
6  }
7  module M3 {
8      requires java.base;
9  }
```

```
10 void bar() {
11     HashMap map2 = new HashMap();//O2
12     map2.put("B", new B());//OB
13     Object v2 = map2.get("B");
14     B b = (B) v2;//cast may fail?
15 }
16 module M4 {
17     requires java.base;
18 }
```

```
19 class java.util.HashMap ... {
20     Node[] table = new Node[16];//O3
21     public void put(K k, V v) {
22         Node n = new Node(k, v);//O4
23         table[hash(k)] = n;
24     }
25     public final V get(K k) {
26         Node n = table[hash(k)];
27         return n.getValue();
28     }
29     class Node ... {
30         K key; V value;
31         Node(K p, V q) {
32             key = p; value = q;
33         }
34         public final V getValue() {
35             return value;
36         }
37     }
38 }
39 module java.base {
40     exports java.util to M3, M4;
41 }
```

Fig. 2: Simplified code example of `java.util.HashMap` in `java.base` module.

$pts([O_4],\texttt{value}) = \{O_A, O_B\}$. As a result, call to O_1.`get` and O_2.`get` will return a value pointing to both O_A and O_B, leading to cast-may-fail false alarms at line 5 and line 14.

This example can only be precisely analyzed when the context depth is set to more than 1. In `put`, with `2obj` analysis, we get $pts([O_1],\texttt{n}) = \{\langle[O_1], O_4\rangle\}$ and $pts([O_2],\texttt{n}) = \{\langle[O_2], O_4\rangle\}$, where object O_4 is qualified with a heap context. Hence, the constructor of class `Node` (lines 31-33) is analyzed twice with 2 distinct contexts: $[O_1, O_4]$ and $[O_2, O_4]$. Thus, we can precisely compute the pointer values of `value` as $pts([O_1, O_4],\texttt{value}) = \{O_A\}$, and $pts([O_2, O_4],\texttt{value}) = \{O_B\}$. Finally, we can correctly analyze that $pts(\texttt{v1}) = \{O_A\}$ and $pts(\texttt{v2}) = \{O_B\}$, avoiding false cast-may-fail alarms.

It is noteworthy that O_1 and O_2 are critical context elements when analyzing the codes of `HashMap` and `Node`. We can also observe that O_1 and O_2 are the receiver objects for the `put` and `get` methods at lines 3, 4, 12, and 13, which are *module frontiers*, whose call-site and target method locate in distinct modules. For instance, the call-site at line 3 is in `M3` module, and the `put` method is in the `java.base` module.

Hence, we can use the receiver objects (module frontiers) as key context elements and propagate them when analyzing the target module. In that sense, O_1 and O_2 are critical context elements when analyzing `put`/`get` method at line 3/4 and line 12/13, and they are propagated when analyzing the code in the target module `java.base`. Hence, the constructor of class `Node` (lines 31-33) is analyzed twice with 2 distinct contexts: $[O_1]$ and $[O_2]$. Thus, we can precisely compute the pointer values of `value` as $pts([O_1],\texttt{value}) = \{O_A\}$, and $pts([O_2],\texttt{value}) = \{O_B\}$.

III. Methodology

This section formally illustrates module-aware pointer analysis, including the modeling of `SPI` and module-aware context sensitivity according to the semantics of `JPMS`.

A. Preliminaries

This section presents the notations and functions illustrated in Figure 3, which will be used in our formalism.

Before integrating the semantics of the module system into the pointer analysis, we need to extract the semantics from a given *module-info* file. Due to space constraints, the details are omitted. We use the following symbols to represent the semantic information of a module.

- $\mathbb{R}$: $\mathbb{M} \mapsto \wp(\mathbb{M})$ maintains the modules a given module depends on.
- $\mathbb{E}$: $(\mathbb{M}, \mathbb{M}') \mapsto \wp(\mathbb{N})$ keeps the packages that can be exported from $\mathbb{M}$ module to $\mathbb{M}'$ module.
- $\mathbb{P}$: $(\mathbb{M}, \mathbb{T}) \mapsto \wp(\mathbb{T})$ provides the mappings from interfaces to implementations.
- $\mathbb{U}$: $\mathbb{M} \mapsto \wp(\mathbb{T})$ declares the interfaces of `SPI` used by a module.

Let us revisit the examples in Figure 1 and Figure 2. For Figure 1, the results of parsing module constraints are: $\mathbb{R}(\texttt{M1}) = \{\texttt{M2}\}$, $\mathbb{U}(\texttt{M1}) = \{\texttt{Animal}\}$, $\mathbb{P}(\texttt{M2}, \texttt{Animal}) = \{\texttt{Dog}, \texttt{Cat}\}$. For Figure 2, the results are: $\mathbb{E}(\texttt{java.base}, \texttt{M3}) = \{\texttt{java.util}\}$, $\mathbb{E}(\texttt{java.base}, \texttt{M4}) = \{\texttt{java.util}\}$, $\mathbb{R}(\texttt{M3}) = \{\texttt{java.base}\}$, $\mathbb{R}(\texttt{M4}) = \{\texttt{java.base}\}$.

For forward compatibility, `JPMS` introduced the concept of an "*automatic module*" to migrate pre-existing non-modular libraries (regular Jar files). A regular Jar file in the classpath of an application is regarded as an automatic module, which implicitly requires all other modules, including JDK modules and all user-defined modules of the application. Moreover, an automatic module exports all its packages, making them accessible to other modules of the application. It is worth pointing out that our approach works identically for both normal modules and automatic modules.

B. Module-Aware Context-Sensitivity

In this section, we formalize how to utilize the constraints of the module system to balance the efficiency and precision of context-sensitive pointer analysis. Section III-B1 presents the rules for the respective five kinds of statements. In New and Call, we use module frontier (Section III-B2) as the critical context element. To apply such the critical context element (module frontier) to the internal code of a module, Section III-B3 introduces several subroutines (in Algorithm 1 and Algorithm 2) used in the rules of Section III-B1 to construct MDG on the fly.

1) Rules of Module-Aware Context-Sensitivity:

Figure 4 illustrates our formalism to pointer analysis. Fundamentally, this approach aligns with the methodologies presented in previous works [12], [13], [20] as they all describe Andersen-style pointer analysis for Java. Without loss of generality, we use the five rules ([New], [Assign], [Load], [Store], and [Call]) to handle the respective five kinds of statements which can represent a simplified subset of Java. It should be noted that we omit the details found in previous works. To integrate with module-aware analysis, the traditional context c_1 (receiver objects, call-sites, etc.) is extended with module-aware elements c_2 (module frontiers), denoted as $\langle c_1, c_2 \rangle$. Given context c_1 with a sequence of elements $c_1 = [e_n, ..., e_1]$ and a new element e, we use the notation $c_1 +\!\!+ e$ for $[e_n, ..., e_1, e]$ and c_{1_k} for $[e_k, ..., e_1]$ where $k < n$. We introduce **moduleCtx** to select a particular module-aware context, with additional constraints to model the semantics of the module system (highlighted in blue).

For the rules [Assign], [Load], and [Store], we have not made any modifications; we merely extracted the points-to set of the right-hand side of the statement and incorporated it into the points-to set of the left-hand side.

In New, the declaring method of the statement $l \in \mathbb{L}$ is m. For m, we can utilize the **classOf** function to resolve its declaring class, which is then used to identify its containing module $\mathcal{M}$ using the **moduleOf** function. Meanwhile, an abstract heap object $O_l \in \mathbb{H}$ of type `C` is created. Similarly, the module $\mathcal{M}'$ containing type `C` is found using the **moduleOf** function. According to the semantics of `JPMS`, the module M needs to depend on the module $\mathcal{M}'$ ($\mathcal{M}' \in \mathbb{R}(\mathcal{M})$), or both $\mathcal{M}$ and $\mathcal{M}'$ have to be the same ($\mathcal{M} == \mathcal{M}'$).

In Call, similar to New, we sequentially invoke the functions **methodOf**, **classOf**, and **moduleOf**, resulting in the identification of the declaring module $\mathcal{M}$ of the statement l. We can resolve the target method m' using the **dispatch** function, with signature f and receiver object O_0 as arguments. Then we can determine the module $\mathcal{M}'$ in which the type $\mathcal{T}'$ (the containing class of m') is declared. Both modules $\mathcal{M}$ and $\mathcal{M}'$ need to satisfy the constraints of `JPMS` which have been discussed in the explanation of New. In addition, the package $\mathcal{P}$ of type $\mathcal{T}'$ should be accessible for module $\mathcal{M}$ ($\mathcal{P} \in \mathbb{E}(\mathcal{M}', \mathcal{M})$). The function **moduleCtx** is used to select the module-aware context. A detailed demonstration will be provided in the subsequent sections.

$$\textbf{moduleCtx}(O, c_2) = \begin{cases} [O] & \textbf{depthOf}(O, G_{mdg}) = 0 \\ c_2 & 0 < \textbf{depthOf}(O, G_{mdg}) < d \\ \emptyset & otherwise \end{cases}$$

2) Module Frontier:

As mentioned in Section II-C, the receiver object of a method invoked in another module is a critical context element. We define the term *module frontier* in Definition III.1 to represent this kind of receiver object. We select module

modules	$\mathcal{M} \in \mathbb{M}$	**moduleOf**: $\mathbb{T} \mapsto \mathbb{M}$	gives the containing module of a type
packages	$\mathcal{P} \in \mathbb{N}$	**methodOf**: $\mathbb{L} \mapsto \mathbb{W}$	gives the containing method of a statement
types	$\mathcal{T} \in \mathbb{T}$	**classOf**: $\mathbb{W} \mapsto \mathbb{T}$	gives the declaring class of a method
methods	$m \in \mathbb{W}$	**packageOf**: $\mathbb{T} \mapsto \mathbb{N}$	extracts the package of a type
fields	$f \in \mathbb{F}$	**constOf**: $\mathbb{V} \mapsto \mathbb{S}$	resolves the constant class literal of a variable
allocation sites	$O_i \in \mathbb{H}$	**typeOf**: $\mathbb{S} \mapsto \mathbb{T}$	returns the declared type of a constant class literal
local variables	$x, y \in \mathbb{V}$	**Gen**: $\mathbb{L} \times \mathbb{T} \mapsto \mathbb{H}$	generates object according to a type
statement labels	$l \in \mathbb{L}$	**service**: $\mathbb{H} \mapsto \mathbb{T}$	maps from object to service implementation
constant class literal	$s \in \mathbb{S}$	**methodCtx**:$\mathbb{W} \mapsto \wp(\mathbb{C})$	maintains the contexts used for analyzing a method
contexts	$\langle c_1, c_2 \rangle \in \mathbb{C}$	**dispatch**: $\mathbb{W} \times \mathbb{H} \mapsto \mathbb{W}$	resolves a call to a target method
actual parameters	a_i	**pts**: $((\mathbb{V} \bigcup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}) \mapsto \wp(\mathbb{H} \times \mathbb{C})$	records the points-to for a variable or field
formal parameters	p_i^m	ret^m	return variables

Fig. 3: Notations and auxiliary functions used in formalism.

$$\frac{\begin{array}{c} l : x = new\ C \quad m = \textbf{methodOf}(l) \\ ctx = \langle c_1, c_2 \rangle \in \textbf{methodCtx}(m) \quad hctx = \langle c_{1\,k-1}, c_2 \rangle \\ \mathcal{M} = \textbf{moduleOf}(\textbf{classOf}(m)) \quad \mathcal{M}' = \textbf{moduleOf}(C) \\ \mathcal{M} = \mathcal{M}' \mid \mathcal{M}' \in \mathbb{R}(\mathcal{M}) \end{array}}{(O_l, hctx) \in \textbf{pts}(x, ctx) \quad \textbf{SolveCaller}(m, O_l)} \text{ [NEW]}$$

$$\frac{l : x = y \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m)}{\textbf{pts}(y, ctx) \subseteq \textbf{pts}(x, ctx)} \text{ [ASSIGN]}$$

$$\frac{l : x = y.f \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m) \quad (O, hctx) \in \textbf{pts}(y, ctx)}{\textbf{pts}(O.f, hctx) \subseteq \textbf{pts}(x, ctx)} \text{ [LOAD]}$$

$$\frac{l : x.f = y \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m) \quad (O, hctx) \in \textbf{pts}(x, ctx)}{\textbf{pts}(y, ctx) \subseteq \textbf{pts}(O.f, hctx)} \text{ [STORE]}$$

$$\frac{\begin{array}{c} l : x = a_0.f(a_1) \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m) \\ (O_0, hctx) \in \textbf{pts}(a_0, ctx) \quad hctx = \langle c_1, c_2 \rangle \quad c_2' = \textbf{moduleCtx}(O_0, c_2) \quad ctx' = \langle c_1 +\!\!+ O_0, c_2' \rangle \\ m' = \textbf{dispatch}(f, O_0) \quad \mathcal{T} = \textbf{classOf}(m) \quad \mathcal{M} = \textbf{moduleOf}(\mathcal{T}) \quad \mathcal{T}' = \textbf{classOf}(m') \quad \mathcal{M}' = \textbf{moduleOf}(\mathcal{T}') \\ \mathcal{P} = \textbf{packageOf}(\mathcal{T}') \quad \mathcal{M} = \mathcal{M}' \mid \mathcal{M}' \in \mathbb{R}(\mathcal{M}) \quad \mathcal{P} \in \mathbb{E}(\mathcal{M}', \mathcal{M}) \end{array}}{\begin{array}{c} ctx' \in \textbf{methodCtx}(m') \quad (O_0, hctx) \in \textbf{pts}(this^{m'}, ctx') \\ \textbf{pts}(a_1, ctx) \subseteq \textbf{pts}(p_1^{m'}, ctx') \quad \textbf{pts}(ret^{m'}, ctx') \subseteq \textbf{pts}(x, ctx) \quad \textbf{SolveCallee}(O_0, m') \end{array}} \text{ [CALL]}$$

Fig. 4: Rules for module-aware context-sensitive pointer analysis.

frontiers as contexts and propagate them into the target modules, serving as contexts for the methods within those modules. However, the number of contexts will increase significantly if the propagation of module frontiers is not limited. In our insights, methods closer to the module frontiers are more significantly affected. Therefore, we designed an efficient approach to limit the propagation depth of module frontiers. The details are provided in the following section.

Definition III.1. *The **Module Frontier** is a set of receiver objects, $\mathbb{MF}$, where $O_l \in \mathbb{MF}$ is a receiver object of a target method m' at a* CALL *statement, $l : x = a_0.f(...)$, whose declaring method is m. If the corresponding modules of methods m and m' are different, O_l is considered a module frontier. We can define the module frontier more formally using the following rule.*

$$\frac{\begin{array}{c} l : x = a_0.f(...) \quad m = \textbf{\textit{methodOf}}(l) \quad (O_0, \_) \in \textbf{\textit{pts}}(a_0, \_) \\ m' = \textbf{\textit{dispatch}}(f, O_0) \quad \mathcal{T} = \textbf{\textit{classOf}}(m) \quad \mathcal{M} = \textbf{\textit{moduleOf}}(\mathcal{T}) \\ \mathcal{T}' = \textbf{\textit{classOf}}(m') \quad \mathcal{M}' = \textbf{\textit{moduleOf}}(\mathcal{T}') \quad \mathcal{M} \neq \mathcal{M}' \end{array}}{O_0 \in \mathbb{MF}}$$

3) Module Depth Graph:

The *module frontier* can help us select critical context when solving context-sensitive pointer analysis. However, when processing the code within a module, the information of the module frontier will be missed. So we need to propagate the module frontier into the internal code of a module. To achieve this goal, we propose a module depth graph (MDG). We can propagate the module frontier along the MDG. In the meantime, we can limit the propagation depth based on MDG. Before presenting MDG, we first introduce the Object Allocation Graph which is the foundation of MDG.

As Definition III.2 shows, the OAG, proposed by Tan et al. [18], is used to describe the relationship of object allocation. A path with k nodes in an OAG, such as $O_k \rightarrow O_{k-1} \rightarrow ... \rightarrow O_2 \rightarrow O_1$, is exactly matched with a context, $[O_k, ..., O_2, O_1]$, of a method m for k-object-sensitivity where O_1 is the receiver object of m and O_2 is the receiver object of the method that created object O_1, etc.

Definition III.2. *The **Object Allocation Graph (OAG)** is a directed graph, $G = (N, E)$, where N is the set of nodes and E represents the set of edges. A node $O \in N$ represents an allocation site which is also the context element in object-sensitivity. An edge $O_2 \rightarrow O_1 \in E$ represents an object allocation relation where O_1 is allocated in a method with O_2 being the receiver object of this method.*

In order to construct OAG, it is necessary to aggregate all objects along with their respective receiver objects. Previous work [18] relies on a pre-analysis which is considerably slower to calculate such relationships. To improve the overall performance, we design an on-the-fly algorithm to build the OAG. It is worth noting that the on-the-fly algorithm leverages context-sensitive points-to information, making it more precise than the original algorithm. Algorithm 1 introduces two

subroutines, `SolveCallee` and `SolveCaller`, to build OAG on-the-fly. In procedure `SolveCallee`, each object O' allocated in method m (computed by the pre-analysis) is collected. The procedure `AddEdge` is then invoked to add the edge $O \rightarrow O'$ into OAG, signifying that O is the receiver object of the method allocating O'. The `SolveCaller` introduces an edge $O \rightarrow O'$ between O' and each receiver object O of the method m allocating O'. According to the definition of the OAG (Definition III.2), receiver objects of a method m and objects allocated in the method m are two kinds of primary elements to build an OAG. We focus on two kinds of statements: the `New` statement, which creates objects within a method, and the `Call` statement, which identifies the receiver objects of a method. As Figure 4 shows, we extends the rules of NEW and CALL. In the NEW rule, when an object O_l is created, the `SolveCaller` function is invoked with m and O_l as arguments. In the Call rule, when the receiver object O_0 of m' is resolved, the `SolveCallee` function is invoked with O_0 and m' as arguments. This way, all object allocation relations can be efficiently computed by extending rules NEW and CALL (highlighted in cyan), thereby avoiding a time-consuming pre-analysis.

Algorithm 1: Building object allocation graph

```
1  global Object allocation graph: G = (N, E);
2  Procedure SolveCallee(O, m):
3      for each object O' allocated in method m do
4          AddEdge(O, O');
5  Procedure SolveCaller(m, O'):
6      for each receiver object O of method m do
7          AddEdge(O, O');
8  Procedure AddEdge(O, O'):
9      add O into N;  add O' into N ;
10     add O → O' into E ;
11     BuildMDG(O);BuildMDG(O');
```

Theorem 1. *(Correctness). Both the original OAG algorithm and the on-the-fly OAG algorithm can construct the same OAG.*

Proof Sketch. The `WorkList` algorithm is usually used to solve Andersen-style pointer analysis. The algorithm iteratively processes statements in a program by applying the rules in Figure 4. When applying the NEW rule, there will be edges from all discovered receiver objects of the declaring method of the NEW statement to the object created at this statement. Similarly, when applying the CALL rule, there will be edges from all discovered receiver objects of the target method to the known objects allocated in the target method. Therefore, regardless of whether a receiver object O of a method m or an object O' allocated in m is discovered firstly, there will always be an edge, $O \rightarrow O'$, added to OAG. □

Let us apply the on-the-fly algorithm to build the OAG of the program in Figure 2. Figure 5 shows the resulting OAG where the object O_r represents a unique fake entry object.

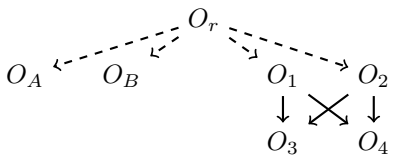

Fig. 5: The OAG of the program in Figure 2

Because the receiver objects of methods `foo` and `bar` are unknown in the code snippet, we use the dashed arrow (⇢) originating O_r to represent these unknown edges. Suppose the `foo` method is solved first, we can obtain the points-to set of variable `map1`, which is $\{O_1\}$ at line 2, and the receiver object of the constructor of `HashMap` and the `put` method is O_1. When solving these two methods, objects O_3 and O_4 are created, and the function `SolveCaller` is applied. In other words, edges $O_1 \rightarrow O_3$ and $O_1 \rightarrow O_4$ are added into the OAG. When solving the `bar` method, object O_2 is generated. When handling the CALL statements whose target methods are the constructor of `HashMap` and the `put` method respectively, the function `SolveCallee` is applied. Because objects O_3 and O_4 which are allocated in the two methods have been resolved according to previous steps, the edges $O_2 \rightarrow O_3$ and $O_2 \rightarrow O_4$ are added into the OAG.

Leveraging OAG, we can efficiently construct MDG in a timely manner. The formal definition of MDG is provided in Definition III.3.

Definition III.3. *The **Module Depth Graph (MDG)** is a directed graph, $G = (\langle N, D\rangle, E)$, where N and E are the same as that in the OAG, D represents the length of the shortest path from a module frontier node to the current node. For $\langle n, d\rangle \in \langle N, D\rangle$, if n is a module frontier, the d will be set to zero.*

The MDG is built by annotating the OAG with module depth. As defined in Definition III.3, a node in the MDG is a pair $\langle n, d\rangle$, where n is an object in the OAG, and d represents the propagation depth. Without sacrificing consistency, we incorporate D into N as a field in our implementation. By default, d is initialized to the unknown depth $\top$. Algorithm 2 augments the OAG algorithm (Algorithm 1) to build MDG on-the-fly: when an object O is added into the OAG (line 10 in Algorithm 1), Gather-Apply-Scatter (GAS model) [21] is performed in lines 2-10 to compute its propagation depth. If object O is a module frontier, we use the `Scatter` procedure to propagate it to its successor nodes whenever its depth changes. Otherwise, we gather its predecessors and calculate the minimum propagation depth (lines 13-14). Then, the minimum value will be increased by 1 and applied to object O using the `Apply` procedure (lines 16-21), during which the depth of O will be updated if the target value is less than the original depth. The updated depth will be scattered to the successor nodes (lines 9-10). In the `Scatter` procedure (lines 22-25), if the depth of an object changes, it will be recursively scattered to its successor nodes.

Figure 6 illustrates the MDG of the program shown in Figure 2 constructed using Algorithm 2. As described in Section III-B3, object O_1 is discovered first identified as a module frontier according to Definition III.1. Therefore, its

Algorithm 2: Building module depth graph

```
1  global Module depth graph:
     G_mdg = (⟨N_mdg, D⟩, E_mdg);
2  Procedure BuildMDG(O):
3      if O ∈ 𝕄𝔽 then
4          if Apply(O, 0) then
5              Scatter(O);
6      else
7          predDepth = Gather(O);
8          if predDepth ≠ ⊤ then
9              if Apply(O, predDepth + 1) then
10                 Scatter(O);
11 Procedure Gather(O):
12     predDepth = ⊤;
13     for each predecessor node ⟨O', d'⟩ of ⟨O, d⟩ in
         G_mdg do
14         predDepth = min(d', predDepth);
15     return predDepth;
16 Procedure Apply(O, target):
17     Let ⟨O, d⟩ in G_mdg;
18     if target<d then
19         d = target;
20         return true;
21     return false;
22 Procedure Scatter(O):
23     for each successor node ⟨O', d'⟩ of ⟨O, d⟩ in G_mdg
         do
24         if Apply(O', d + 1) then
25             Scatter(O');
```

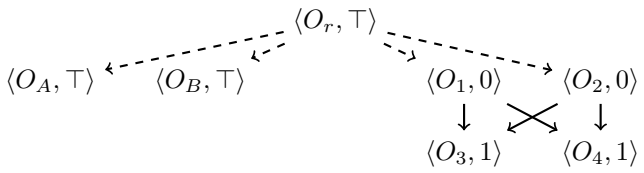

Fig. 6: The MDG of the program in Figure 2

depth is set to zero. At this stage, the successors of O_1 are unknown, therefore, the procedure `Scatter` does nothing. When objects O_3 and O_4 are generated, the depth of their unique predecessor, O_1, calculated by the `Gather` procedure, is determined to be zero. Consequently, the depths of O_3 and O_4 are each set to one. Another module frontier, O_2, also has its depth set to zero upon creation. When the `Scatter` procedure is applied, the depths of O_3 and O_4, which are successors of O_2, remain unchanged.

Let us revisit the [CALL] rule. Before introducing **moduleCtx**, we use the function **depthOf** to retrieve the module depth of object O in the MDG G_{mdg}. If the depth equals zero, which means the receiver object O is a module frontier ($O \in \mathbb{MF}$), $[O]$ is regarded as the context in analyzing a method call with O being a receiver object. When the module depth is greater than zero and less than d, the heap context $hctx$ is used as the context of the method. In other words, the module frontier is selected as critical context and propagated along the MDG until the depth reaches d. Finally, the module context and standard context are combined to form the context, ctx', of the target method m'.

4) Revisiting the Motivating Example:

Let us apply our module-aware context-sensitivity to perform pointer analysis on the example in Figure 2. In `foo`, object O_1 is created at line 2 and assigned to variable `map1` which is the receiver variable of method `put`. Since O_1 is a module frontier, the context of the method is $[O_1]$ at line 3. Similarly, O_2 is created at line 11 and used as the receiver object of the `put` method, with its context at line 12 being $[O_2]$. Consequently, we can differentiate the two invokes of `put` based on their distinct contexts. This also allows us to distinguish the points-to information of the variable `v` across different contexts. In `put`, O_4 is generated with two different heap contexts $[O_1]$ and $[O_2]$. Since the receiver object of the constructor of `Node` is O_4 (with a module depth of 1), the heap contexts $[O_1]$ and $[O_2]$ are used as the contexts of the constructor. Notably, O_1 and O_2 are module frontiers propagated to O_4 along the MDG. Therefore, the points-to information of the variable `q` can be distinguished under different contexts in line 31. Similarly, the methods `get` and `getValue` can be distinguished based on the contexts $[O_1]$ and $[O_2]$. Ultimately, this analysis allows us to correctly determine that $pts(v1) = \{O_A\}$ and $pts(v2) = \{O_B\}$, thereby avoiding false cast-may-fail alarms.

C. Modeling `SPI`

Java's SPI mechanism is now widely used in various frameworks and libraries to enhance modularity and flexibility, including `JDBC`, `Dubbo`, `JNDI`, `Spring`, `SLF4J`, etc. However, for pointer analysis, without handling `SPI`, we may miss objects that are created dynamically according to `SPI` configuration.

To model `SPI`, we separate it into three parts: loading a service according to an interface, iterating each implementation of the service, and applying each implementation. We use the following three sets to define these parts.

- LOADER, a set of pairs (m, i) where m represents the method that is used to load service according to a constant class literal which is the i-th parameter of method m.
- TRANSFER, a set of methods that propagate services from the receiver variable of the call site to its `LHS` variable, such as the `iterator` method of collection.
- RETRIEVE, a set of methods that are used to retrieve the implementations of service.

Figure 7 shows the formalization of modeling `SPI` by the following three rules.

- [DEF]. For an invoke statement, if its target method, m, is a LOADER method whose i-th parameter is the service interface, we can get the constant class literal according

$$\frac{\begin{array}{c} l: x = y.f(...) \quad s = \textbf{constOf}(a_k) \quad (O, \_) \in \textbf{pts}(x, \_) \\ (O_0, \_) \in \textbf{pts}(y, \_) \quad m' = \textbf{dispatch}(f, O_0) \quad (m', k) \in \textsc{Loader} \\ m = \textbf{methodOf}(l) \quad \mathcal{M} = \textbf{moduleOf}(\textbf{classOf}(m)) \\ \mathcal{M}' = \textbf{moduleOf}(\textbf{classOf}(m')) \quad \mathcal{T} = \textbf{typeOf}(s) \quad \mathcal{T} \in \mathbb{U}(\mathcal{M}) \\ \mathcal{T}' \in \mathbb{P}(\mathcal{M}', \mathcal{T}) \quad \mathcal{M} = \mathcal{M}' \mid \mathcal{M}' \in \mathbb{R}(\mathcal{M}) \end{array}}{\mathcal{T}' \in \textbf{service}(O)} \quad [\textsc{Def}]$$

$$\frac{l: x = y.f(...) \quad (O_0, \_) \in \textbf{pts}(y,) \quad (O, \_ \in \textbf{pts}(x, \_) \quad m' = \textbf{dispatch}(f, O_0) \quad m' \in \textsc{Transfer} \quad \mathcal{T} \in \textbf{service}(O_0)}{\mathcal{T} \in \textbf{service}(O)} \quad [\textsc{Prop}]$$

$$\frac{\begin{array}{c} l: x = y.f(...) \quad ctx \in \textbf{methodCtx}(\textbf{methodOf}(l)) \\ (O_0, \_) \in \textbf{pts}(y, ctx) \quad m' = \textbf{dispatch}(f, O_0) \\ m' \in \textsc{Retrieve} \quad \mathcal{T} \in \textbf{service}(O_0) \end{array}}{O = \textbf{Gen}(l, \mathcal{T}) \quad (O, ctx) \in \textbf{pts}(x, ctx)} \quad [\textsc{Use}]$$

Fig. 7: Rules for modeling `SPI`.

to the actual parameter a_i and resolve its type $\mathcal{T}$ by the function **typeOf**. It should be noted that when the service interface type is not directly given as a class literal, we query its points-to sets to obtain the set of class literals propagated to it. The only case that cannot be handled is when the propagated value is from user input. However, this is an extremely rare case, and we did not encounter such a case in our evaluated applications. Then we will check whether current module $\mathcal{M}$ *uses* $\mathcal{T}$ by $\mathbb{U}$ and get the implementation $\mathcal{T}'$ of the interface $\mathcal{T}$ according to $\mathbb{P}$. Then, these implementations will be bound to the object pointed to by the `LHS` variable of this invoke statement by the **service** function. It should be noted that the points-to set of the `LHS` variable may be unknown when processing this invoke statement, we will bind the implementations to the `LHS` variable and rebind them to the objects pointed to by `LHS` variable once the points-to set has been resolved.

- [Prop]. For an invoke statement whose target is a Transfer method, the implementations are propagated from the object pointed to by the receiver variable of this call site to the object pointed to by the `LHS` variable.
- [Use]. The Retrieve methods apply an `SPI` service. For an invoke statement whose target is a Retrieve method, we use the **service** function to retrieve implementations corresponding to the receiver object of this invoke and then create objects according to these implementations by the **Gen** function, and then add these objects into the points-to set of the `LHS` variable to maintain complete points-to relations.

Let us apply the three rules above to perform pointer analysis on the example in Figure 1. At line 2, there is a Loader method, `ServiceLoader.load`, with a service interface `Animal`. According to the *module-info* files of modules, `M1` and `M2`, the `M1` module *uses* `Animal` interface, and the `M2` module *provides* two implementations, `Dog` and `Cat`. So, we bind `Dog` and `Cat` to the objects pointed to by variable `loader` which can be represented to $\textbf{service}(O_1) = \{\texttt{Dog}, \texttt{Cat}\}$. At line 3, the `ServiceLoader.iterator` method is a Transfer method, so we just propagate implementations from objects pointed to by variable `loader` to objects pointed to by variable `iter` which means that $\textbf{service}(O_2) = \{\texttt{Dog}, \texttt{Cat}\}$. At line 5, the `Iterator.next` method is a Retrieve method, so we retrieve implementations, `Dog` and `Cat`, from $\textbf{service}(O_2)$ and create two objects O_{dog} and O_{cat} whose types are `Dog` and `Cat` respectively. Then the points-to set of variable a is $\{O_{dog}, O_{cat}\}$. Finally, we can resolve the target methods to the `run` methods of `Dog` and `Cat` at line 6. Without modeling `SPI`, the points-to set of variable a is empty, and the `run` methods of `Dog` and `Cat` will be unreachable. Therefore, more methods can be analyzed by modeling the `SPI`.

IV. Evaluation

To evaluate the impact of our research, we focused on the following research questions in our assessment of context-insensitivity, k-object-sensitivity, and MPA which applies our module-aware approach to pointer analysis:

- **RQ1.** Does MPA find more reachable methods?
- **RQ2.** Can MPA improve performance?
- **RQ3.** Is on-the-fly algorithm of the MDG efficient?
- **RQ4.** How is the precision and efficiency of MPA under different depths of MDG?

Experimental Settings: When analyzing a program, we set the timeout budget to 1.5 hours, consistent with previous works [16], [22]–[27]. All experiments were conducted on an Intel Xeon 2.0GHz machine with 1 TB of RAM, running Clear Linux OS. Since `JPMS` was introduced in JDK9, differing from previous works that relied on experiments with JDK1.6, we analyze programs on JDK17. This Long Term Support version is also recommended by Tai-e [28]. The maximum heap size of the JVM is set to 500GB (with `-Xmx`).

Implementation: We implemented our module-aware approach on top of Tai-e, a recent popular static analysis framework for Java. We implemented the on-the-fly algorithms (Algorithm 1 and Algorithm 2) to build the OAG and the MDG based on the original algorithm in Tai-e. The default depth of MDG is set to 4 ($d = 4$). We use MPA as module-aware context-insensitive Andersen's analysis [29]. The notation MPA+`kobj` represents the MPA scheme applied to k-object-sensitive analysis.

Benchmarks: Similar to previous works [15], [16], [23]–[27], we evaluated MPA using the Dacapo suite, where each JAR file is treated as "automatic modules" by `JPMS` to ensure forward compatibility, as these benchmarks were developed before `JPMS`. The "automatic modules" require all other modules implicitly. In Dacapo, `jython` is excluded because context-insensitivity fails to scale for it. In addition, we select three widely used programs (`ant`, `antlr4`, `checkstyle`) that were migrated from earlier versions of JDK before JDK9 to JDK9 or later versions, and the three programs are also treated as "automatic modules" in our module-aware analysis. At last, we also evaluate against four popular programs (`jboss`, `logback`, `questdb`, and `lombok`, with 270, 2.9k, 13.5k, and 12.6k stars on GitHub respectively) that were developed based on JDK9 or later and fully utilize the features of `JPMS`. The diversity selection of both traditional applications (Dacapo, `ant`, `antlr4`, and `checkstyle`) and modularized applications (`jboss`, `logback`, `questdb`

TABLE I: Comparing the metrics of three programs with or without modeling SPI (SPI or non-SPI) for context-insensitive analysis. For all numbers, larger is better.

Programs	Analyses	Metrics			
		#fail-cast	#reach-mtd	#call-edge	#poly-call
logback	non-SPI	269	1,711	3,556	162
	SPI	277	1,785	3,782	165
lombok	non-SPI	6	16	15	11
	SPI	310	1,455	4,146	477
questdb	non-SPI	1,692	14,694	148,158	12,581
	SPI	2,590	20,069	315,891	16,015

and lombok) demonstrate that our approach is applicable to all kinds of Java programs.

Precision Metrics: Following previous works [15], [16], [23]–[27], we assess the precision of pointer analyses utilizing four essential metrics: the number of casts that may fail (#fail-cast), the number of reachable methods (#reach-mtd), the number of call graph edges (#call-edge), and the number of polymorphic calls discovered (#poly-call). Same as previous work [15], we collect the four metrics in the application code, which is the part that developers are most concerned about.

A. RQ1: Discovering More Reachable Methods

Excluding the DACAPO suite and three programs migrated from early versions of the JDK that did not utilize the SPI scheme, we use the remaining four programs to evaluate the ability to find more reachable methods. For jboss, MPA cannot find more reachable methods because it does not use *provides*. Table I gives the number of four metrics with or without modeling SPI when analyzing the other benchmarks under context-insensitivity. For lombok, only 16 application methods are analyzed without modeling SPI, while 1,455 methods are discovered after modeling SPI. For questdb, after modeling SPI, 5,375 methods were newly discovered compared to the original approach. For logback, only 74 methods are newly handled. A similar conclusion can be drawn for the other three precision metrics.

The following code snippet illustrates the reason for the significant improvement in reachable methods for lombok. In the entry method, lombok loads the service of LombokApp and then invokes the runApp method (lines 2-4). Without modeling SPI, objects of LombokApp implementations will be excluded from the points-to set of the variable app, resulting in the inability to analyze runApp methods of these implementations.

```
1 void main() {
2       Iterable apps = SpiLoadUtil.findServices(
            LombokApp.class);
3       for (LombokApp app : apps) {
4          app.runApp();
5       }
6 }
```

The three benchmarks, logback, lombok, and questdb invoke SPI interfaces once, 10 times, and once, respectively. As described in Section III-C, traditional pointer analyses (here, we use context-insensitive analysis) cannot determine the target methods of those SPI interface invocations, often resulting in a large number of unreachable methods. Nevertheless, our approach precisely resolves the target methods of SPI invocations by faithfully modeling the semantics of SPI. We manually examined each resolved SPI invocation target and confirmed that there were no false positives. Those newly discovered SPI invocation targets are then analyzed using the same context-insensitive analysis, making more methods invoked by those targets reachable. As a result, we observe a significant increase across all four metrics in Table I.

B. RQ2: Performance

In this section, we investigate the efficiency and precision of the module-aware approach by applying MPA to standard context-insensitive (CI), 1-object-sensitive (1obj), 2-object-sensitive (2obj), and selective-sensitive (ZIPPER^{E}) pointer analyses. The SPI modeling is integrated into all tools to ensure a fair comparison.

Efficiency: Table II gives the time and precision metrics of all tools for our benchmarks. The times represent the total time for each analysis. Compared to CI, MPA runs as fast as CI for most benchmarks such as antlr, eclipse, hsqldb, luindex, lusearch, etc. On average, MPA is 1.6× slower than CI for all benchmarks.

Compared to 1obj and 2obj, MPA+kobj can achieve slight speedup (0.3× and 0.1× respectively) on average, with precision improvements for all benchmarks.

Precision: As Table II shows, MPA is noticeably more precise than CI for all precision metrics across all benchmarks. For #fail-cast, #reach-mtd, #call-edge, and #poly-call, the ratio of the number reported by MPA against that reported by CI is 54.9%, 97.5%, 96.1%, and 85.2% respectively. Especially for #fail-cast, MPA can reduce almost half of the false positives reported by CI on average.

Compared to 1obj, MPA is significantly more precise on #fail-cast metric and slightly more precise on the other three metrics on average. For #fail-cast, #reach-mtd, #call-edge, and #poly-call, the ratio of the number reported by MPA against that reported by 1obj is 58.7%, 99.0%, 97.7%, and 98.6% respectively. In addition, MPA+kobj is always more precise than kobj for all benchmarks.

Compared to ZIPPER^{E}*:* The last two columns in Table II compare precision and efficiency of 1obj and MPA+1obj with or without applying ZIPPER^{E} (the state-of-the-art selective context-sensitivity approach) on our benchmarks. Consistent with the conclusion in [30], ZIPPER^{E} can achieve substantial speedup with a slight loss of precision than standard k-object-sensitivity. Compared to 1obj +ZIPPER^{E}, MDG +1obj +ZIPPER^{E} offers higher precision, reduces reports on the metric #fail-cast by 8%, and is only 0.55× slower. Compared to 1obj, MDG +1obj +ZIPPER^{E} achieves 8.2× speedup with similar precision.

C. RQ3: Efficiency of MDG Builder

Unlike the original algorithm for OAG construction depending on a pre-analysis, MPA constructs OAG and MDG on the fly. To illustrate the efficiency of our on-the-fly algorithm, we run the standard context-insensitive analysis (CI) equipped with or without the MDG builder and use the time difference

TABLE II: Efficiency and precision results for tools on our benchmarks. For all numbers, smaller is better.

Index	Program	Metrics	CI	MPA	1obj	MPA+1obj	2obj	MPA+2obj	1obj +ZIPPER^E	MDG +1obj +ZIPPER^E
1	antlr	Time(s)	61.83	60.66	315.39	312.67	1,882.18	1,916.12	34.14	34.47
		#fail-cast	64	38	58	38	30	30	61	38
		#reach-mtd	755	755	754	754	754	754	754	754
		#call-edge	4,546	4,546	4,545	4,545	4,545	4,545	4,545	4,545
		#poly-call	487	483	483	483	483	483	483	483
2	bloat	Time(s)	71.43	113.88	413.37	404.43	2,874.76	2,271.67	49.35	89.21
		#fail-cast	1,219	896	1,183	883	958	867	1,189	1,189
		#reach-mtd	2,500	2,479	2,486	2,478	2,463	2,459	2,499	2,499
		#call-edge	18,866	18,201	18,395	18,002	17,627	17,582	18,552	18,552
		#poly-call	1,015	800	822	779	750	746	993	993
3	chart	Time(s)	208.89	1,729.80	4,991.38	2,198.48	-	-	136.45	638.91
		#fail-cast	607	283	574	275	-	-	599	544
		#reach-mtd	2,396	2,314	2,391	2,313	-	-	2,394	2,389
		#call-edge	8,550	8,317	8,509	8,288	-	-	8,540	8,513
		#poly-call	335	277	314	275	-	-	330	311
4	eclipse	Time(s)	82.68	75.88	528.80	344.85	2,114.33	2,213.71	47.03	47.46
		#fail-cast	214	151	197	149	141	140	207	185
		#reach-mtd	984	939	956	933	927	927	973	972
		#call-edge	2,377	2,116	2,315	2,112	2,109	2,109	2360	2290
		#poly-call	155	133	144	133	130	130	154	151
5	fop	Time(s)	112.66	204.66	944.38	592.96	4,121.28	3,169.51	74.91	132.41
		#fail-cast	136	43	93	43	49	43	103	97
		#reach-mtd	1,197	1,023	1,021	1,021	1,021	1,021	1,021	1,021
		#call-edge	3,638	3,157	3,156	3,155	3,155	3,155	3,156	3,156
		#poly-call	125	71	87	71	71	71	96	95
6	hsqldb	Time(s)	59.17	61.45	308.57	330.11	2,625.69	2,044.25	31.84	31.41
		#fail-cast	7	0	4	0	0	0	7	7
		#reach-mtd	53	53	53	53	53	53	53	53
		#call-edge	79	79	79	79	79	79	79	79
		#poly-call	6	6	6	6	6	6	6	6
7	luindex	Time(s)	60.53	59.89	333.28	323.82	2,262.29	2,078.96	30.54	33.76
		#fail-cast	44	10	37	6	6	6	40	40
		#reach-mtd	353	350	350	347	347	347	350	350
		#call-edge	892	891	883	882	882	882	883	883
		#poly-call	39	33	29	29	29	29	35	35
8	lusearch	Time(s)	64.65	63.53	326.78	321.43	2,373.99	1,938.67	33.3	35.09
		#fail-cast	120	26	112	24	8	8	118	87
		#reach-mtd	732	708	727	705	701	701	728	728
		#call-edge	2,196	2,094	2,180	2,083	2,082	2,082	2,182	2,182
		#poly-call	189	173	169	167	167	167	179	179
9	pmd	Time(s)	86.59	129.55	649.92	477.00	2,714.70	2,434.85	56.86	77.15
		#fail-cast	491	363	480	358	380	357	485	476
		#reach-mtd	1,752	1,723	1,748	1,723	1,726	1,723	1,748	1,746
		#call-edge	5,702	5,647	5,695	5,647	5,654	5,647	5,695	5,693
		#poly-call	120	94	112	94	100	92	120	120
10	xalan	Time(s)	130.00	548.14	1,658.19	1,331.16	-	-	91.31	157.8
		#fail-cast	1,097	707	1,019	698	-	-	1,064	964
		#reach-mtd	5,245	5,163	5,203	5,118	-	-	5,210	5,180
		#call-edge	22,819	20,993	22,592	20,942	-	-	22,778	22,626
		#poly-call	1,952	1,667	1,882	1,663	-	-	1,950	1,937
11	ant	Time(s)	71.02	101.59	364.74	382.45	2,449.22	2,247.85	40.52	51.02
		#fail-cast	68	38	67	37	31	30	68	68
		#reach-mtd	391	373	383	367	362	362	383	383
		#call-edge	742	675	726	665	665	665	730	730
		#poly-call	87	73	76	66	65	65	86	86
12	antlr4	Time(s)	68.65	71.60	395.64	341.02	2,140.78	2,066.97	39.29	54.52
		#fail-cast	692	382	678	376	455	370	686	661
		#reach-mtd	2,688	2,656	2,685	2,646	2,650	2,646	2,685	2,685
		#call-edge	5,598	5,568	5,593	5,548	5,549	5,548	5,593	5,593
		#poly-call	187	146	175	142	151	142	187	187
13	checkstyle	Time(s)	85.27	129.97	514.03	497.06	2,579.87	3,030.36	50.95	74.22
		#fail-cast	236	208	228	207	207	207	230	220
		#reach-mtd	946	946	945	944	943	942	945	945
		#call-edge	2,505	2,504	2,504	2,503	2,503	2,502	2,504	2,504
		#poly-call	37	33	19	19	19	19	35	35
14	jboss	Time(s)	87.59	115.03	734.25	521.31	3,009.79	2,244.53	54.19	77.95
		#fail-cast	106	56	92	51	50	37	97	97
		#reach-mtd	823	814	814	805	769	767	818	818
		#call-edge	2,031	1,999	1,995	1,966	1,879	1,878	2,023	2,023
		#poly-call	168	132	150	130	128	117	168	168
15	logback	Time(s)	134.10	154.83	973.19	500.75	2,746.72	2,873.99	72.26	100.39
		#fail-cast	277	161	255	155	147	135	263	247
		#reach-mtd	1,785	1,736	1,758	1,724	1,720	1,720	1,772	1,767
		#call-edge	3,782	3,571	3,697	3,563	3,563	3,563	3,779	3,661
		#poly-call	165	143	150	134	129	129	162	160
16	lombok	Time(s)	236.19	2,811.02	4,996.46	2,946.13	-	-	152.82	294.43
		#fail-cast	310	215	304	199	-	-	307	291
		#reach-mtd	1,455	1,437	1,454	1,436	-	-	1,454	1,454
		#call-edge	4,146	4,127	4,146	4,126	-	-	4,146	4,132
		#poly-call	477	433	472	429	-	-	476	469
17	questdb	Time(s)	101.85	321.87	908.70	891.50	3,562.37	3,008.23	80.74	85.08
		#fail-cast	2,590	2,300	2,553	2,281	623	613	2,553	2,187
		#reach-mtd	20,069	19,918	20,037	19,901	19,828	19,828	20,053	19,851
		#call-edge	315,891	298,164	311,695	297,972	287,230	287,226	311,837	289,338
		#poly-call	16,015	15,630	15,779	15,586	15,544	15,544	15,814	15,601

between the two analyses as the time for building the MDG. Figure 8 shows the percentage of time spent by MDG against standard context-insensitive analysis. The percentage of the time for building MDG against the time for `CI` ranges from 2.4% (`antlr4`) to 14.3% (`xalan`), with an average of 7.8%. Compared to building an MDG using a context-insensitive pre-analysis, our on-the-fly approach is more efficient.

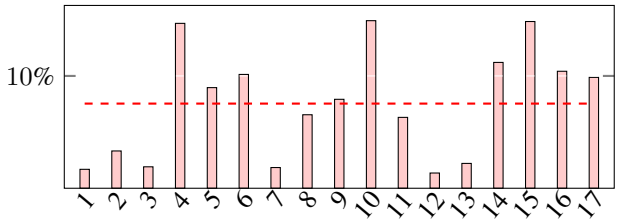

Fig. 8: The percentage of time spent by MDG against context-insensitive analysis. The dashed line represents the average.

D. RQ4: Performance under Different Depths

As described in Section III-B1, as d increases, the number of contexts increases, and the precision of analyses also improves. To select suitable d to balance the precision and efficiency, we evaluate MPA under different d (d = 1,2,3,4,5). Hereafter, we use MPA_i to denote MPA under depth i. In this experiment, performance-wise, MPA_4 outperforms all other variants. In detail, the ratio of the time spent by MPA_1, MPA_2, MPA_3, MPA_5 against that of MPA_4 is 119.48%, 106.17%, 100.25%, and 100.31% respectively. Importantly, MPA_4 achieves the same precision as MPA_5 in all evaluated metrics. Specifically, the ratio of the number of #fail-cast reported by MPA_1, MPA_2, MPA_3, MPA_5 against that of MPA_4 is 155.77%, 108.60%, 100.27%, and 100% respectively. For #reach-mtd, the ratio is 100.73%, 100.14%, 100%, and 100%. For #call-edge, the ratio is 101.16%, 100.16%, 100%, and 100%. For #poly-call the ratio is 108.18%, 102.15%, 100%, and 100%. As d increases, the improvement in the precision of MPA slows down. Compared to MPA_3, MPA_4 only achieves slight improvement on #fail-cast, and maintains the same precision on the other three precision metrics. So, we use 4 as the default value of d.

V. Related Work

In this section, we review the previous works that are closely related to this research.

A. Context Representation

According to the representation of context, there are three mainstream variants of context sensitivity: *k*-object sensitivity, *k*-type sensitivity, and *k*-call-site sensitivity. In addition to the above three variants, the work [27] proposed a hybrid approach that applies object sensitivity to instance method invocations and call-site sensitivity to static method invocations. The value-contexts approach [31] used the data flow values of parameters to represent contexts. Generic-sensitivity [23] utilized the generic instantiation sites as contexts. Our module-aware approach can be adapted to the above context-sensitive variants by selecting the corresponding kind of context at module frontiers.

B. Selective Context-Sensitivity

Selective context-sensitivity allows applying context to partial methods or variables to provide a better trade-off between precision and efficiency. SCALER [26] assesses whether a method should be analyzed with context sensitivity based on an estimate of its potential memory usage. ZIPPER [25] introduces three types of value-flow patterns for pinpointing methods that are critical for precision, these patterns are determined by solving a graph reachability problem on a precision flow graph. Consequently, this allows for substantial speedup while maintaining most of the precision. The extended version of ZIPPER, ZIPPER-E [16], significantly enhances ZIPPER's speed while maintaining similar precision. EAGLE [24] conducts a pre-analysis based on CFL-reachability to enable selective context sensitivity in k-obj, ensuring precision is maintained. TURNER [32] strikes an optimal balance between ZIPPER and EAGLE, facilitating k-obj analysis to operate much faster than EAGLE while achieving higher precision than ZIPPER. CONCH [32] identifies objects that depend on context, preventing the excessive growth of contexts. BATON [33] introduces a Unit-Relay framework by integrating various context selectors collectively. Rather than choosing which methods should undergo context-sensitive analysis, BEAN [18] enhances the precision of k-obj sensitive analysis by omitting irrelevant context elements, [34] propose a context-tunneling approach that offers an elegant and general data-driven method for selecting critical context elements, and [22], [23] introduce generic-sensitivity which always preserves generic instantiation sites as key context elements. In this paper, we provide an alternative that specifically targets module systems—we leverage the semantics of module systems to select module frontiers, which commonly suggest encapsulation boundaries, as critical context elements. Moreover, our approach also discovers more reachable methods by modeling SPI. In addition, some manually-selected metrics and heuristics [35]–[37], or learning-based approaches [38]–[41] have been imposed to selectively analyze a subset of methods with context sensitivity.

VI. Conclusion

We proposed a module-aware pointer analysis including modeling `SPI` and module-aware context-sensitivity based on an on-the-fly MDG. We implemented a pointer analysis tool named MPA based on our module-aware approach. MPA can increase reachable methods up to $90.9\times$. Meanwhile, the MPA can run as fast as context-insensitivity for most evaluated programs while obtaining better precision than 1-object-sensitivity.

VII. Acknowledgment

We thank all reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (2022YFB3103900), the National Natural Science Foundation of China (62402474, 62132020, and 62202452), and the China Postdoctoral Science Foundation (2024M753295).

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*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[2] J. Späth, K. Ali, and E. Bodden, "Ideal: Efficient and precise alias-aware dataflow analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133923

[3] 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 *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 267–279. [Online]. Available: https://doi.org/10.1109/ASE.2019.00034

[4] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program Tailoring: Slicing by Sequential Criteria," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, pp. 15:1–15:27. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.15

[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*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 112–122. [Online]. Available: https://doi.org/10.1145/1250734.1250748

[6] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 254–264. [Online]. Available: https://doi.org/10.1145/2338965.2336784

[7] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 317–326. [Online]. Available: https://doi.org/10.1145/1882291.1882338

[8] C. Liu, J. Lu, G. Li, T. Yuan, L. Li, F. Tan, J. Yang, L. You, and J. Xue, "Detecting tensorflow program bugs in real-world industrial environment," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 55–66. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678891

[9] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," *SIGPLAN Not.*, vol. 41, no. 6, p. 387–400, jun 2006. [Online]. Available: https://doi.org/10.1145/1133255.1134027

[10] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast algorithms for dyck-cfl-reachability with applications to alias analysis," *SIGPLAN Not.*, vol. 48, no. 6, p. 435–446, jun 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462159

[11] M. Das, B. Liblit, M. Fähndrich, and J. Rehof, "Estimating the impact of scalable pointer analysis on optimization," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 260–278. [Online]. Available: https://doi.org/10.1007/3-540-47764-0_15

[12] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for java," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 1–11. [Online]. Available: https://doi.org/10.1145/566172.566174

[13] Milanova, Ana and Rountev, Atanas and Ryder, Barbara G., "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, p. 1–41, jan 2005. [Online]. Available: https://doi.org/10.1145/1044834.1044835

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

[15] 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*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 17–30. [Online]. Available: https://doi.org/10.1145/1926385.1926390

[16] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381915

[17] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1093–1105. [Online]. Available: https://doi.org/10.1145/3597926.3598120

[18] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *International Static Analysis Symposium*. Springer, 2016, pp. 489–510. [Online]. Available: https://doi.org/10.1007/978-3-662-53413-7_24

[19] P. Deitel. (2017) Understanding java 9 modules. [Online]. Available: https://www.oracle.com/corporate/features/understanding-java-9-modules.html

[20] 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*. Springer, 2013, pp. 196–232. [Online]. Available: https://doi.org/10.1007/978-3-642-36946-9_8

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[22] H. Li, T. Tan, Y. Li, J. Lu, H. Meng, L. Cao, Y. Huang, L. Li, L. Gao, P. Di, L. Lin, and C. Cui, "Generic sensitivity: Generics-guided context sensitivity for pointer analysis," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1144–1162, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3377645

[23] H. Li, J. Lu, H. Meng, L. Cao, Y. Huang, L. Li, and L. Gao, "Generic sensitivity: customizing context-sensitive pointer analysis for generics," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1110–1121. [Online]. Available: https://doi.org/10.1145/3540250.3549122

[24] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360574

[25] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276511

[26] Li, Yue and Tan, Tian and Møller, Anders and Smaragdakis, Yannis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3236024.3236041

[27] 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*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–434. [Online]. Available: https://doi.org/10.1145/2491956.2462191

[28] T. Tan and Y. Li. (2023) Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. [Online]. Available: https://tai-e.pascal-lab.net/docs/current/reference/en/setup-in-intellij-idea.html

[29] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[30] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381915

[31] R. Padhye and U. P. Khedker, "Interprocedural data flow analysis in soot using value contexts," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, ser. SOAP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 31–36. [Online]. Available: https://doi.org/10.1145/2487568.2487569

[32] D. He, J. Lu, and J. Xue, "Context debloating for object-sensitive pointer analysis," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 79–91. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678880

[33] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485524

[34] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276510

[35] 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*, ser. SOAP 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 13–18. [Online]. Available: https://doi.org/10.1145/3088515.3088519

[36] S. Wei and B. G. Ryder, "Adaptive Context-sensitive Analysis for JavaScript," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. T. Boyland, Ed., vol. 37. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 712–734. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2015.712

[37] 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*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 485–495. [Online]. Available: https://doi.org/10.1145/2594291.2594320

[38] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133924

[39] M. Jeon, S. Jeong, S. Cha, and H. Oh, "A machine-learning algorithm with disjunctive model for data-driven program analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 2, jun 2019. [Online]. Available: https://doi.org/10.1145/3293607

[40] M. Jeon, M. Lee, and H. Oh, "Learning graph-based heuristics for pointer analysis without handcrafting application-specific features," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428247

[41] M. Jeon and H. Oh, "Return of cfa: call-site sensitivity can be superior to object sensitivity even for object-oriented programs," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498720