Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity

TIAN TAN, Nanjing University, China
YUE LI*, Nanjing University, China
XIAOXING MA, Nanjing University, China
CHANG XU, Nanjing University, China
YANNIS SMARAGDAKIS, University of Athens, Greece

Traditional context-sensitive pointer analysis is hard to scale for large and complex Java programs. To address this issue, a series of selective context-sensitivity approaches have been proposed and exhibit promising results. In this work, we move one step further towards producing highly-precise pointer analyses for hard-to-analyze Java programs by presenting the `Unity-Relay` framework, which takes selective context sensitivity to the next level. Briefly, `Unity-Relay` is a one-two punch: given a set of different selective context-sensitivity approaches, say $S = S_1, \ldots, S_n$, `Unity-Relay` first provides a mechanism (called `Unity`) to combine and maximize the precision of all components of S. When `Unity` fails to scale, `Unity-Relay` offers a scheme (called `Relay`) to pass and accumulate the precision from one approach S_i in S to the next, S_{i+1}, leading to an analysis that is more precise than all approaches in S.

As a proof-of-concept, we instantiate `Unity-Relay` into a tool called Baton and extensively evaluate it on a set of hard-to-analyze Java programs, using general precision metrics and popular clients. Compared with the state of the art, Baton achieves the best precision for *all* metrics and clients for *all* evaluated programs. The difference in precision is often dramatic—up to 71% of alias pairs reported by previously-best algorithms are found to be spurious and eliminated.

CCS Concepts: • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: Pointer Analysis, Alias Analysis, Context Sensitivity, Java

ACM Reference Format:
Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (October 2021), 27 pages. https://doi.org/10.1145/3485524

1 INTRODUCTION

Pointer analysis is important for an array of real-world applications such as bug detection [Chandra et al. 2009; Naik et al. 2006], security analysis [Arzt et al. 2014; Livshits and Lam 2005], program verification [Fink et al. 2008; Pradel et al. 2012] and program understanding [Li et al. 2016; Sridharan

*Corresponding author

Authors' addresses: Tian Tan, State Key Laboratory for Novel Software Technology, Nanjing University, China, tiantan@nju.edu.cn; Yue Li, State Key Laboratory for Novel Software Technology, Nanjing University, China, yueli@nju.edu.cn; Xiaoxing Ma, State Key Laboratory for Novel Software Technology, Nanjing University, China, xxm@nju.edu.cn; Chang Xu, State Key Laboratory for Novel Software Technology, Nanjing University, China, changxu@nju.edu.cn; Yannis Smaragdakis, Department of Informatics and Telecommunications, University of Athens, Greece, yannis@smaragd.org.

2475-1421/2021/10-ART147
https://doi.org/10.1145/3485524

et al. 2007]. A more precise pointer analysis is always favored by its clients as it implies, e.g., fewer false alarms of bugs and vulnerabilities, more accurate code navigation, or potential for optimization.

Context sensitivity has been demonstrated to be a major scheme for improving precision of Java pointer analysis [Fegade and Wimmer 2020; Feng et al. 2015; Kanvar and Khedker 2016; Lhoták and Hendren 2008; Lhoták and Hendren 2006; Milanova et al. 2002, 2005; Tan et al. 2017; Thakur and Nandivada 2019, 2020; Thiessen and Lhoták 2017; Whaley and Lam 2004; Xu and Rountev 2008]. With years of development, traditional context sensitivity performs well for small/medium programs, e.g., `2obj` (object-sensitive pointer analysis with context length being two) can analyze most DaCapo benchmarks rapidly and precisely [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011]; however, it is hard for it to scale to large and complex Java programs with good precision [Smaragdakis et al. 2014]. (An analysis not scaling means that it blows up in complexity and does not terminate even under very high time limits.)

To address this problem, a series of research work [Hassanshahi et al. 2017; Jeon et al. 2019; Jeong et al. 2017; Li et al. 2018a,b, 2020; Lu and Xue 2019; Minseok Jeon and Oh 2020; Oh et al. 2014, 2015; Smaragdakis et al. 2014; Wei and Ryder 2015] propose to apply context sensitivity *selectively* in the sense that contexts no longer apply to all methods, while context elements (e.g., objects, types, call-sites) and context lengths may vary for different selected methods. Other, non-selected, methods are treated context-insensitively, in order to enable good scalability.

Conversely, to gain better precision, selective context-sensitivity approaches usually have to sacrifice efficiency by allowing more methods to be analyzed context-sensitively, sometimes with longer context lengths or more precise but heavier context elements. In other words, these analyses need to make careful precision and efficiency trade-offs, where one more step towards precision may put the analysis at the risk of unscalability [Li et al. 2020]. As a result, seeking further precision improvement becomes hard, as limited efficiency margins are left to play with. Under such constraints, if the approach for more precise pointer analysis is not designed well, it may introduce significant overhead with minor or no precision improvement (e.g., by selecting for sensitive treatment many precision-useless methods). More commonly, the analysis will choose to scale, but will sacrifice precision (e.g., by selecting very few methods to treat context sensitively).

Problem. For many real-world applications where points-to or alias information is required, such as certain bug detectors, security analyzers, etc., good precision is much favored, even if the price is to run the analysis for a long time (e.g., several hours) [Christakis and Bird 2016]. Thus, the challenge we seek to address is to achieve, given a long but reasonable time allowance, *precise pointer analysis results* for large and complex Java programs, *for which traditional context-sensitive analyses fail to scale, and selective context-sensitive analyses scale but with limited precision.*

Method. We introduce a general (meta-)framework called `Unity-Relay` to produce highly precise pointer analyses for hard-to-analyze Java programs, by systematically exploiting and utilizing *any* selective context sensitivity technique. `Unity-Relay` is a one-two punch, with precision as its first priority:

- `Unity-Relay` takes as inputs a program `P` and a set of different selective context-sensitivity approaches, S;
- it first provides a method (called `Unity`) to unify all approaches in S (playing the role of a meta-heuristic), maximizing precision by applying the most stringent context-sensitivity selection made by any approach in S;
- if `Unity` fails to scale, the `Relay` method is employed. It divides the scalability burden of `Unity` into different passes (each pass is a run of context-sensitive pointer analysis guided by an approach in S), with the precision achieved by one pass being transmitted to and

accumulated in the next pass: in each pass, the points-to results retrieved from the previous pass are used to on-the-fly filter intermediate points-to results, keeping inferred points-to sets small and, thus, enhancing scalability.

Given *any* selective context-sensitive pointer analysis A (guided by an approach in S), if A is sound and scalable for `P`, no matter how precise A is, `Unity-Relay` can be expected to produce a sound and scalable analysis with a precision that is guaranteed to be at least as good as A's, in the worst case. (Experimental results show that the precision is always better.) This property also implies that even when new (possibly special-purpose) selective context-sensitivity insights are developed in the future, `Unity-Relay` will still be able to leverage them to produce a more precise analysis in practice.

Results. As a proof-of-concept, we instantiate the `Unity-Relay` framework in a tool called Baton. We extensively evaluate Baton on a set of hard-to-analyze Java programs (including the toughest programs evaluated in the past literature of selective context-sensitive pointer analysis for Java), using three general precision metrics and four popular clients (also the most complete set of precision metrics/clients used in recent literature). Compared to the state of the art, experimental results show that Baton is able to improve precision significantly, and achieves the best precision for *all* metrics and clients for *all* evaluated programs. To the best of our knowledge, this is the first time that this level of analysis precision has been attained for these hard-to-analyze programs. Moreover, because of `Unity-Relay`'s nature as a general meta-framework, we expect it to see more future instantiations and to unleash the power of more selective context-sensitivity approaches, to produce more precise pointer analyses.

In summary, this work makes the following contributions:

- We present `Unity-Relay`, a simple and practical framework to produce precise pointer analyses for hard-to-analyze Java programs, by unleashing the power of selective context sensitivity (Section 3).
- We formulate the `Unity-Relay` framework, prove its soundness and precision guarantees, and discuss its scalability (Section 4).
- We present Baton, a tool instantiated from the `Unity-Relay` framework, which will be released as an open-source tool (Section 5).
- We conduct extensive experiments by comparing Baton with the state of the art, to demonstrate its effectiveness in the real world (the experimental results can be obtained using the accompanying artifact [Tan et al. 2021]) (Section 6). Baton substantially improves on the precision of previous algorithms in the literature—e.g., eliminating over 33% of alias pairs (soundly determined to be spurious) on average, and up to 71%, compared to the best past contender evaluated.

2 BACKGROUND

We assume that the reader is familiar with the high-level concepts of pointer analysis, as introduced in several surveys [Smaragdakis and Balatsouras 2015; Sridharan et al. 2013]. This section describes some necessary background knowledge about context sensitivity for whole-program pointer analysis for Java.

Traditional Context Sensitivity. A method is analyzed *context-sensitively* means that the static abstraction of different dynamic instantiations of variables and heap objects within the method will be treated separately during analysis under different abstract entities called *contexts*. Accordingly, spurious object flows will be reduced, thus making the analysis more precise. Given a method `m`, each of its contexts typically consists of a list of consecutive context elements in the form of `[c1,c2,...]` to model different run-time conditions. For example, for call-site sensitivity, `c1` is the call site of `m`, and `c2` is the call site of the method containing `c1`, etc. However, for efficiency, only the most recent l context elements are kept (l is also called the context length) and l is usually limited to a small number for whole-program analysis in practice [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2011]. There are three mainstream context-sensitivity variants for Java pointer analysis: call-site sensitivity, object sensitivity, and type sensitivity. Henceforth, we use `3obj` to denote object sensitivity with context length being 3. Longer context indicates more precision: more spurious data flows can be possibly eliminated. Thus, `3obj` is more precise than `2obj`. For different kinds of context elements with the same length, past work has demonstrated that object sensitivity typically outperforms call-site sensitivity, in terms of both precision and efficiency [Milanova et al. 2005; Smaragdakis et al. 2011], and is more precise but also more expensive than type sensitivity [Smaragdakis et al. 2011; Tan et al. 2017]. Traditionally, context-sensitive pointer analysis uniformly applies context sensitivity to every method in a program to maintain high precision. However, this approach often does not work for large and complicated programs as such treatment is too costly to scale [Li et al. 2018b; Smaragdakis et al. 2014].

Selective Context Sensitivity. Selective context sensitivity proposes to apply context sensitivity *selectively*, only for *precision-useful*/*non-scalability-threat* methods. The analysis precision of such methods will help improve the overall program analysis precision, while not incurring so much cost as to make the analysis hard to scale. The remaining methods are analyzed context-insensitively so that the space and time cost originally incurred for these methods (in traditional context sensitivity) is saved, leaving room to produce precise and scalable analyses even for hard-to-analyze programs. However, it is challenging to accurately determine which methods are precision-useful but not scalability-threat in general. To deal with this problem, in the past years, various selective context-sensitivity approaches have been presented, based on different insights and policies. For example, the target methods could be selected according to expert experience [WALA 2018], program patterns [Li et al. 2018a, 2020], abstracted memory capacity [Li et al. 2018b], parameterized heuristics [Hassanshahi et al. 2017; Smaragdakis et al. 2014], or machine learning approaches [Jeon et al. 2018; Jeong et al. 2017]. In addition, in selective context sensitivity, some approaches apply the same context sensitivity to the selected methods [Hassanshahi et al. 2017; Li et al. 2018a, 2020; Smaragdakis et al. 2014], while in other approaches context elements and lengths may vary for different subsets of the selected methods [Jeon et al. 2019; Jeong et al. 2017; Li et al. 2018b]. We refer the readers to Section 7 for more details.

3 THE UNITY-RELAY FRAMEWORK, INFORMALLY

We first give an overview of the `Unity-Relay` framework (Section 3.1), and then explain the key ideas of `Unity` (Section 3.2) and `Relay` (Section 3.3), respectively.

3.1 Overview

Figure 1 shows a pictorial overview of the `Unity-Relay` framework: given a program `P` and a set of selective context-sensitive (C.S.) approach/strategy components, *S*, the framework will yield highly precise pointer analysis results for `P`.

`Unity-Relay` is a one-two punch. It first calls its `Unity` component. Taking as input the results obtained by running each approach in *S* for `P`, the context-sensitivity selector of `Unity` (C.S. Selector in Figure 1) unifies them and makes new configuration about which portion of `P` should be analyzed by what context-sensitivity variants (i.e., what are the context elements and lengths) to maximize the precision, according to the unity principle, as introduced in Section 3.2. Intuitively, the `Unity`

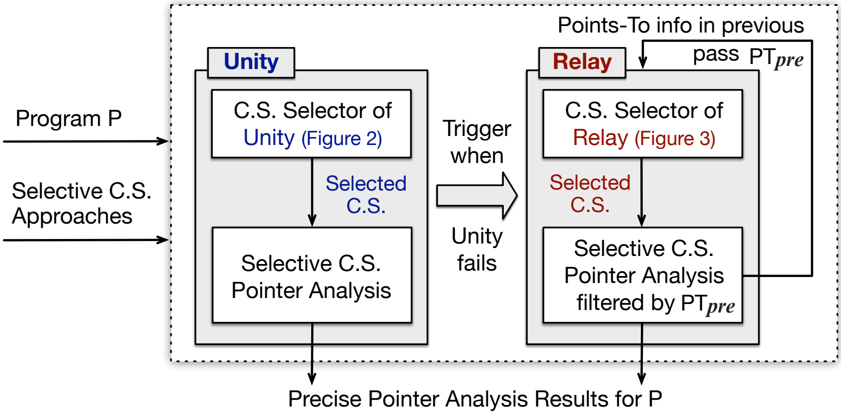
Program P
Selective C.S. Approaches
Unity
C.S. Selector of Unity (Figure 2)
Selected C.S.
Selective C.S. Pointer Analysis
Trigger when
Unity fails
Relay
C.S. Selector of Relay (Figure 3)
Selected C.S.
Selective C.S. Pointer Analysis filtered by PT$_{pre}$
Points-To info in previous pass PT$_{pre}$
Precise Pointer Analysis Results for P

Fig. 1. Overview of the Unity-Relay Framework.

principle maintains for each program element the most precise context that any component context-sensitivity approach would use. Then the new selected context-sensitivity information (Selected C.S. in Figure 1) will guide the selective context-sensitive pointer analysis to analyze P, and its results will be the output of Unity-Relay if the analysis is able to finish running within the time limit; otherwise (i.e., if Unity is too expensive to scale), Unity-Relay will trigger its Relay component to analyze P.

Generally, given n approaches in S, Relay will run the selective context-sensitive pointer analysis n times (i.e., we have n passes in Relay). In each pass, as shown in Figure 1, the selective context-sensitive pointer analysis is guided by the context-sensitivity information (Selected C.S.) output from the C.S. selector of Relay according to the relay principle as illustrated in Section 3.3. Between successive passes, the precision of the earlier pass will be transmitted to and accumulated in the next pass. This is achieved by soundly filtering the pointer analysis results in the current pass by using the points-to information (points-to sets of variables) from the previous pass (PT$_{pre}$ in Figure 1). As a result, in Relay, the scalability burden is shared in each analysis run (i.e., each pass): a component strategy is not burdened by others, but instead each strategy that completes helps both the precision and the scalability of the rest. Although overall precision is not guaranteed to reach that of Unity, precision improvements are reaped in a *stable* and *accumulative* way.

3.2 Unity

Recall that every selective context-sensitive analysis represents a precision and efficiency trade-off, where precision bumps up against the limits of analysis scalability. In other words, for past selective context-sensitive analyses, the degree of precision (typically: the proportion of the program to be analyzed context-sensitively) is chosen to be approximately at a point where further improving it would render the analysis non-scalable. According to previous work, treating context-sensitively even a small set of methods may significantly hinder scalability, for the "wrong" choice of methods [Li et al. 2020]. Now it seems that we are trapped: if a small step towards precision may hit the scalability wall (in the precision and efficiency balance made by each selective context-sensitivity approach), how can we reap a further noticeable precision improvement in a *general, policy-agnostic meta-framework*? To escape from the trap, in Unity, we propose to take a *big* step forward towards higher precision, by allowing many more methods to be analyzed under context sensitivity, with

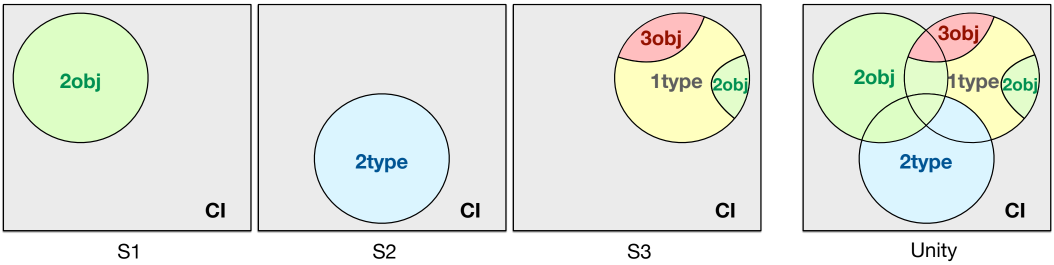

Fig. 2. An example for illustrating the idea of Unity. S1, S2 and S3 are three input selective context-sensitivity approaches. The four squares show the selected context-sensitivity information for the same program P (i.e., which methods of P should be analyzed by what context-sensitivity variants) yielded by the three input selective approaches (S1, S2 and S3) and Unity respectively. For example, the square of S1 means: the S1 approach determines that the methods enclosed in the green circle will be analyzed by 2obj while the remaining methods of P will be analyzed context-insensitively (i.e., CI).

heavier context elements and longer context lengths, if possible. This counterintuitive proposal is based on the following insight.

Insight. Analyzing more methods context-sensitively may not incur an efficiency decline (sometimes it may even accelerate an analysis as more false data flows are pruned away), as long as the right methods are chosen. Avoiding scalability-threat methods is essential [Li et al. 2020; Smaragdakis et al. 2014], but the net number of methods analyzed context-sensitively is not.

As described in Section 2, each of the existing selective context-sensitivity approaches (henceforth, just "selective approaches") has its own insight (e.g., based on expert experience, parameterized heuristics, principled program patterns, etc.) to identify which methods are more likely useful for improving precision *while not incurring much efficiency cost*, when analyzed context-sensitively. Limited by the effectiveness of each insight, every approach may *miss* a collection of methods which are truly precision-useful and efficiency-friendly; however, this problem can be alleviated when combining more selective approaches to identify more target methods. As a result, we propose to *simultaneously* and context-sensitively analyze *all* the methods identified by a set of selective approaches, S. Although now possibly many more methods need to be analyzed context-sensitively, we still have a chance to obtain an analysis that is scalable even if we analyze them together, as *each* of these extra methods is determined as *not a scalability threat* by at least one approach in S. Hence, although it is hard to give one principle to accurately decide which methods, say M, are precision-useful but not scalability-threatening, our proposal naturally takes advantages of the insights of different selective approaches for identifying M *from different perspectives*.

We illustrate the key idea of Unity in Figure 2. Assume we have three selective approaches S1, S2, and S3. Figure 2 depicts the selected context-sensitivity information determined by each approach for the same program P. For instance, after analyzing P, S1 reports that the set of methods in the green circle (in S1 of Figure 2) should be analyzed by 2obj and the remaining methods (the gray part in S1) need to be analyzed context-insensitively (CI). Similarly, we have the results for S2 and S3 in Figure 2, and in the case of S3, three context-sensitivity variants, 3obj, 2obj and 1type are selected for different sets of methods.

What does Unity produce given the above three approaches and their selected context-sensitivity information? Briefly, Unity finds out the *most precise configuration* by reassigning context-sensitivity variants to the methods which are reported to be analyzed context-sensitively by at least two input approaches. For example, among the overlapped methods reported by S1 and S3, from the point of view of S1, some methods that should have been analyzed under 2obj, are now assigned to 3obj (color turns from green to red) as 3obj is more precise than 2obj; for the rest of overlapped methods, they remain green (i.e., still be analyzed by 2obj) as 2obj is more precise than 1type (reported by S3). After applying similar treatment to other overlapped methods (with the disjoint ones unchanged), we get the output of Unity as in Figure 2. Note that the methods reported by all three input approaches (the middle part of the Unity figure) remain green as 2obj (reported by S1) is more precise than 2type (by S2) and 1type (by S3). Section 4 will detail the rules on how Unity deals with more input approaches with different context-sensitivity variants.

From the point of view of each selective approach, Unity modifies it by (1) adding many more methods (identified by other approaches) to be analyzed context-sensitively, and (2) upgrading the context-sensitivity variant to the most precise one for the methods which are also selected by other approaches. We have explained the insight for (1) at the beginning of Section 3.2, and the insight for (2) is similar: if a method m is reported to be analyzed under a certain context-sensitivity variant, cs, by an input approach *s*, this implies that according to the insight of *s*, m is considered as not a scalability-threat, regardless of how costly cs is. Then, if the input approaches are reliable, we still have a chance to produce a scalable analysis, even if selecting the most precise cs. Later, experimental results further demonstrate the validity of the Unity insight: even if many more methods are analyzed context-sensitively and are under the most precise configuration (w.r.t. the input approaches), Unity is still able to scale for 12 out of 13 hard-to-analyze programs (with default settings).

However, there is no guarantee regarding the scalability of Unity. What can we do if Unity fails to scale? As a second try, a straightforward approach is to not make the analysis too costly. For example, if some overlapped methods M are reported to be analyzed under 3obj and 1type by different input approaches, what about considering the less precise (also less costly) one, i.e., 1type? Nevertheless, this may introduce significant precision loss, and if M is critical form improving precision, such treatment may make the analysis even less precise than the original one(s) guided by certain input approach(es) individually. To reap as much precision as possible while achieving good scalability, we employ the Relay technique.

The core insight of Relay is to divide the scalability burden of Unity into different passes (each pass is a run of selective context-sensitive pointer analysis) and the precision achieved by the former pass can be transmitted to and accumulated in the next pass. Based on this insight, we design two options of Relay, called Relay-o1 and Relay-o2 that the former is more precise but less scalable than the latter. Figure 3 depicts the above idea with the same input selective approaches as in Figure 2. We introduce Relay-o1 first.

3.3 Relay

The first pass of Relay-o1 in Figure 3 differs from Unity in that it only analyzes the methods (say M1) selected by S1 (enclosed in the circle with solid line) context-sensitively, with the remaining ones being analyzed under CI. Similarly, only the methods, say M2 (M3) selected by S2 (S3) are analyzed context-sensitively in pass 2 (pass 3). Thus, for each *context-sensitive* method of M1 (M2 or M3), as shown in the figure, Relay selects the same context-sensitivity variant for it as in Unity, namely, *still the most precise configuration suggested by any of the combined approaches*. For example, for M1 in pass 1, its context-sensitivity variants distribution is the same as the one of Unity in Figure 2 (i.e., the same subset of methods are assigned to 3obj while the remaining methods are assigned to 2obj). For each pass of Relay, only a subset of methods of Unity are analyzed context-sensitively. Although this does not ensure better scalability than Unity, *no more methods than the base approach are analyzed context-sensitively*—only the same methods may be analyzed

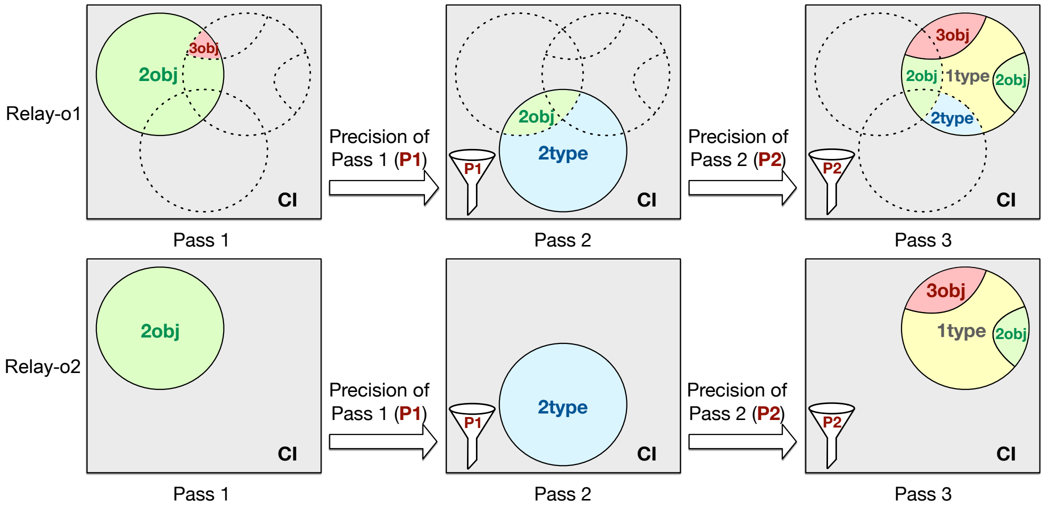

Fig. 3. Examples for illustrating the idea of Relay with the same input approaches setting as in Figure 2. Relay has two options, Relay-o1 and Relay-o2 and the former is more precise but less scalable than the latter. In each pass of both options, only the methods enclosed in the solid circles, say M, are analyzed context-sensitively (the dotted lines in Relay-o1 are to facilitate understanding). Both options adopt the same precision filtering mechanism, and their difference only lies in what context-sensitivity variants are assigned to M.

with a *more precise* context. Therefore, the probability to select scalability-threat methods is low, making Relay more likely to scale compared to Unity.

After running the first pass, its precision, which is reflected in the points-to set for every variable, will be transmitted to the next pass (pass 2). Then, in pass 2, we can use the precision of pass 1, i.e., the points-to information, to help soundly filtering the points-to sets for all variables when performing the analysis in pass 2. For example, assume a variable v points to three objects under CI with its points-to set, say pt(v), being {o1,o2,o3}. After pass 1, pt(v) becomes {o1,o3} where o2 is identified as a spurious object by analyzing M1 context-sensitively. However, in pass 2, purely analyzing M2 under the contexts shown in Figure 3 cannot recognize that v should not point to o2. However, this imprecision does not matter, as the pt(v) transmitted from pass 1 can be used to filter pt(v) *during the analysis* in pass 2, i.e., preventing o2 from being propagated to pt(v). Such filtering is safe as the pointer analysis in pass 1 is sound, so no true objects will be filtered away in Relay (this will be formally proven in Section 4).

Finally, although uncommon, what if Relay-o1 is not scalable? After all, given a program, we hope to confidently rely on Unity-Relay for yielding precise pointer analysis results all the time. Under this context, we design Relay-o2 as *the last line of defense* of Unity-Relay. Relay-o2 employs the same principle as Relay-o1 with the exception that context-sensitivity variants (selected by input approach $\mathtt{S}_i$) are *not* reassigned any more in each pass, say $\mathtt{Pass}_i$ (see Relay-o2 in Figure 3). In other words, $\mathtt{Pass}_i$ and the analysis guided by $\mathtt{S}_i$, say $\mathtt{A}_i$, adopt the same selective context sensitivity. The only difference between running the selective approaches independently is that precision accumulates, using the same filtering mechanism as in Relay-o1. Thus, in $\mathtt{Pass}_i$, the points-to set for each variable *is never larger* than the one in $\mathtt{A}_i$, resulting in less data propagation and faster analysis convergence. The only extra cost introduced in Relay-o2 is the process of filtering, which is small compared to the overhead of a context-sensitive pointer analysis, and easily offset by the aforementioned efficiency benefit. Thus, we can expect Relay-o2 to scale *as long*

```
1  x.f = a;
2  b = y.f;
3  ...
4  p.f = a;
5  b = q.f;
```

Variable	CI	S1	S2	S1&S2	Relay
a	o1	o1	o1	o1	o1
x	o2	o2	o2	o2	o2
y	o2,o3	o3	o2,o3	o3	o3
p	o4	o4	o4	o4	o4
q	o4,o5	o4,o5	o5	o5	o5
b	o1	o1	o1	o1	

Fig. 4. Example for illustrating precision intervention in Relay. Specifically, b is supposed to be a null pointer at run time, and only Relay can analyze it precisely in this case.

as A_i scales. We will further discuss the scalability of Relay-o2 in Section 4. Note that Relay-o2 is fundamentally different from merely intersecting the points-to results of all pointer analyses guided by the input approaches. In Relay-o2, there is *early precision intervention* of each analysis pass to the next: the pointer analysis in each pass is filtered by the points-to results produced by the previous passes *on the fly*, likely preventing the computation of *more* spurious value flows during the analysis. As a result, Relay-o2 is more precise than simply intersecting the final points-to sets after individually running each guided pointer analysis without such *precision intervention*.

We next use an example, in Figure 4, to illustrate the benefit of precision intervention. The left side of Figure 4 is an example code snippet, where the value of a is stored into x.f and p.f (through lines 1 and 4), and b loads values from y.f and q.f (through lines 2 and 5). The table on the right side contains the points-to sets of the variables in the code snippet under different pointer analyses. Column CI gives the points-to sets produced by a context-insensitive pointer analysis, and we consider two selective context-sensitive pointer analyses, S1 and S2, which identify o2 and o4 as spurious objects for variables y and q, respectively. If we simply intersect the points-to results of S1 and S2, then we can eliminate spurious objects for y and q, as shown in column S1&S2. However, b still points to the spurious object o1 under S1&S2, as o1 can flow to b through either line 2 or line 5: both S1 and S2 consider o1 to be valid for b, for different reasons each. Among the analyses in the table, only Relay can block the two flows (through lines 2 and 5) simultaneously. In Relay, with S1 as pass 1 and S2 as pass 2, S2 identifies o4 as spurious for q so that the flow through line 5 can be blocked (as p and q are not aliased). Further benefiting from the precision intervention, the flow through line 2 is also blocked as pass 2 also incorporates the result from pass 1, i.e., o2 is identified as spurious for y by S1 (so that x and y are not aliased). As a result, o1 is identified as spurious for b in pass 2. (The points-to set of b is empty, based on the fragment and example values shown, but other program statements could give it values, orthogonally to the example—e.g., by setting y.f or q.f.)

In the overall Relay mechanism, Relay-o1 and Relay-o2 apply independently to analysis passes and are mixed as necessary for scalability. The two Relay variants can be seen as the best and worst precision options, and there is design space between them for making precision and efficiency trade-offs. In our design, precision is the first priority, so for every $\mathtt{Pass}_i$, we first attempt Relay-o1. If Relay-o1 is not scalable, we select Relay-o2 (which scales if the base approach does), and repeat the same scheme (i.e., first trying Relay-o1, then Relay-o2 if it fails) to $\mathtt{Pass}_{i+1}$, etc. until all n passes complete (assuming there are n input selective approaches).

The running order for different passes of Relay does not affect the soundness and precision guarantees of Relay (including both Relay-o1 and Relay-o2) and also the scalability potential of Relay-o2 (as proved and discussed in Section 4.4). In other words, in practice, regardless of the pass ordering, users can rely on Relay to produce an analysis that is more precise than the one (say

A) guided by any input selective approach, and can expect `Relay-o2` to scale as long as A scales. However, the pass ordering may affect the final precision of `Relay` (although experimental results show that such effect is very minor). For example, assume two variables `v1` and `v2` point to `{o1,o2}` and `{o3,o4}` respectively under `CI`. Pass 1 identifies `o2` as a spurious object for `v1`, and pass 2 can identify `o4` as a false object if and only if it is aware that `v1` does not point to `o2` with the help of the result from pass 1. But when we swap the order of pass 1 and pass 2, `v2` may still point to `o4` (actually a spurious object) at the conclusion of all passes, resulting in different precision. We will discuss the pass ordering issue using the experimental results in Section 6.2.

4 FORMALISM AND PROPERTIES

We formalize the `Unity-Relay` framework, prove its key properties, and discuss its important features. Specifically, we detail how to unify various context-sensitivity approaches with different context elements and lengths, and how to guide selective context-sensitive pointer analysis in each pass to filter spurious objects using the results from the previous pass, while achieving soundness and precision guarantees.

4.1 Domain and Notations

Our notation is shown in Figure 5. For formalization purposes, two domains of context elements, $\mathbb{K}$ and $\mathbb{E}$ are introduced. The former denotes the *kinds* of context elements, i.e., `obj` for object sensitivity, `type` for type-sensitivity and `call` for call-site sensitivity (identified by their locations in the program) while the latter denotes the concrete context elements forming each context $c \in \mathbb{C}$. For instance, in object sensitivity, c consists of a list of heap objects (allocation sites) $o_i \in \mathbb{E}$. Each context-sensitivity variant $cs \in \mathbb{CS}$ can be expressed as a pair of context length l and a kind of context element k, denoted by lk, e.g., `2obj` or `1type`, hence we have $\mathbb{CS} = \mathbb{N} \times \mathbb{K}$, where $\mathbb{N}$ denotes the natural numbers.

pt and *fpt* denote the analysis results. $pt(c, x)$ represents the points-to set of variable x under context c. $fpt(c, o_i, f)$ represents the points-to set of instance field f of an abstract object o_i under context c. The objects in the points-to sets of *pt* and *fpt* are also qualified by contexts, i.e., heap contexts.

gen_{cs} denotes the function that generates contexts according to the length and kind of context elements of cs, which will be further explained in Section 4.2. *contextsOf* maps a method to a set of contexts under which the method is analyzed. *selectCS* is the core of selective context sensitivity, which selects a proper context-sensitivity variant cs for each method. We use S to denote a set of selective approaches, indexed by subscript: S_i. If a pointer analysis PA is guided by a selective approach S_i, we refer to it as PA-S_i.

4.2 Selective Context-Sensitive Pointer Analysis

We present a formulation of general selective context-sensitive pointer analysis in Figure 6, which covers five basic statements: object allocation ([New]), local assignment ([Assign]), field load ([Load]) and store ([Store]), and method call ([Call]). Similar to [Sridharan et al. 2013; Tan et al. 2016], we elide the rules for static members and arrays, as the former is straightforward and the latter can be modeled as load and store to an artificial field of each array. In Figure 6, the premises within boxes are only related to `Unity-Relay`, and thus can be ignored for now.

For each rule in Figure 6, m denotes the method containing the corresponding statement (in blue color). Here we only explain [New] and [Call] as the other three rules are standard.

In context-sensitive pointer analysis, an abstract object is typically represented by a heap context and the label of the allocation site, as in our [New] rule. For simplicity, we directly use the context of the method containing the allocation site (i.e., c) as the heap context for the created abstract

variables	$x, y \in \mathbb{V}$
heap objects	$o_i, o_j \in \mathbb{O}$ (obj)
methods	$m \in \mathbb{M}$
fields	$f \in \mathbb{F}$
types	$T \in \mathbb{T}$ (type)
program locations	$i, j \in \mathbb{L}$ (call)
context elements	$o_i, T, j \in \mathbb{E} = \mathbb{O} \cup \mathbb{T} \cup \mathbb{L} \ldots$
kinds of context elements	$k \in \mathbb{K} = \{$obj, type, call, ...$\}$
contexts	$c \in \mathbb{C} = \mathbb{E}^0 \cup \mathbb{E}^1 \cup \mathbb{E}^2 \ldots$
context-sensitivity variants	$cs = lk \in \mathbb{CS} = \mathbb{N} \times \mathbb{K}$
pt :	$\mathbb{C} \times \mathbb{V} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{O})$
fpt :	$\mathbb{C} \times \mathbb{O} \times \mathbb{F} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{O})$
gen_{cs} :	$\mathbb{C} \times \mathbb{O} \times \mathbb{C} \times \mathbb{L} \rightarrow \mathbb{C}$
$contextsOf$:	$\mathbb{M} \rightarrow \mathcal{P}(\mathbb{C})$
$selectCS$:	$\mathbb{M} \rightarrow \mathbb{CS}$

Fig. 5. Domain and notations.

object. This is a bit different from some existing literature [Jeon et al. 2018; Milanova et al. 2005; Smaragdakis and Balatsouras 2015; Sridharan et al. 2013], which derives the heap context from the method context—e.g., allowing to cut out part of the method context, as the heap context may be shorter. We omit this general treatment in the formal model, for simplicity of illustration purposes. Our implementation supports the more general, full choice in selecting heap contexts.

In [Call], function $dispatch(o_i, g)$ is applied to resolve the virtual dispatch of g on receiver object o_i to the callee m'.

Without loss of generality, we model the results of each selective context-sensitivity approach as a function *selectCS*, which is invoked to select a proper context-sensitivity variant cs for m'. Actually, traditional context sensitivity can be seen as a special case of *selectCS*, which returns the same cs for all methods.

Each context-sensitivity variant cs corresponds to a function gen_{cs} (see Figure 5), which generates contexts for target methods based on the information available at the call site. For generality, as shown in Figure 5, gen_{cs} accepts four arguments: the heap context (c') of the receiver object (o_i), the context of caller m' (c), and the call site (j), so that it can generate contexts for different context-sensitivity variants, including object, type, and call-site sensitivity. The generated context for the target method is denoted c^t.

We use m'_{this} to represent the this pseudo-variable of method m', m'_{pk} to represent the k-th parameter of m', and define for every m' a variable m'_{ret} holding all its return values. The rest of [Call] propagates arguments and return values between caller in context c and callee in context c^t.

4.3 Unity

We formalize Unity by defining its C.S. selector (Figure 1), i.e., *selectCS* in Figure 6. Based on a set of input selective approaches S, Unity selects the most precise context-sensitivity variant for each method. We define $\preceq$, a binary relation of precision between two context-sensitivity variants. If cs is less precise than (or as precise as) cs', we write $cs \preceq cs'$.

The precision of different context-sensitivity variants can be compared based on their lengths and kinds of context elements. The impact of length on precision is well understood, i.e., longer length has better precision (e.g., 2obj $\preceq$ 3obj).

Are context-sensitivity variants with different kinds of elements comparable in terms of precision? The answer is yes, but not always. Here, we say two variants are *comparable* means that one of them

$$\frac{i:\ x = new\ T() \quad c \in contextsOf(m)}{\langle c, o_i \rangle \in pt(c, x)}\ [\textsc{New}]$$

$$\frac{\begin{array}{c} x = y \quad c \in contextsOf(m) \\ \langle c, o_i \rangle \in pt(c, y) \quad \boxed{o_i \in ppt(x)} \end{array}}{\langle c, o_i \rangle \in pt(c, x)}\ [\textsc{Assign}]$$

$$\frac{\begin{array}{c} x = y.f \quad c \in contextsOf(m) \quad \langle c', o_i \rangle \in pt(c, y) \\ \langle c'', o_j \rangle \in fpt(c', o_i, f) \quad \boxed{o_j \in ppt(x)} \end{array}}{\langle c'', o_j \rangle \in pt(c, x)}\ [\textsc{Load}]$$

$$\frac{\begin{array}{c} x.f = y \quad c \in contextsOf(m) \\ \langle c', o_i \rangle \in pt(c, x) \quad \langle c'', o_j \rangle \in pt(c, y) \end{array}}{\langle c'', o_j \rangle \in fpt(c', o_i, f)}\ [\textsc{Store}]$$

$$\frac{\begin{array}{c} j:\ x = y.g(a_1, ..., a_n) \quad c \in contextsOf(m) \\ \langle c', o_i \rangle \in pt(c, y) \quad \boxed{o_i \in ppt(m'_{this})} \\ m' = dispatch(o_i, g) \\ cs = selectCS(m') \quad c^t = gen_{cs}(c', o_i, c, j) \\ \langle c^a, o_a \rangle \in pt(c, a_k), 1 \le k \le n \quad \boxed{o_a \in ppt(m'_{pk})} \\ \langle c^r, o_r \rangle \in pt(c^t, m'_{ret}) \quad \boxed{o_r \in ppt(x)} \end{array}}{\begin{array}{c} c^t \in contextsOf(m') \quad \langle c', o_i \rangle \in pt(c^t, m'_{this}) \\ \langle c^a, o_a \rangle \in pt(c^t, m'_{pk}) \quad \langle c^r, o_r \rangle \in pt(c, x) \end{array}}\ [\textsc{Call}]$$

Fig. 6. Rules for selective context-sensitive pointer analysis. The premises enclosed in boxes are specific to Unity-Relay-based pointer analysis.

is always more precise than (or as precise as) the other one. For such comparison, we introduce notation $k \sqsubseteq k'$ to represent that using context element k is less precise than (or as precise as) k'.

The variants of some kinds of context elements are comparable, e.g., obj and type. As type is a coarser context abstraction over (and always less precise than) object [Smaragdakis et al. 2011], we have type $\sqsubseteq$ obj, and accordingly, $l$type $\preceq$ $l$obj. In other cases, such as $l$obj and $l$call, their precision are theoretically incomparable. We will discuss how to handle such cases at the end of this section. For now, we assume that all concerned context-sensitivity variants are comparable.

Now we define $cs \preceq cs'$:

$$\begin{aligned} cs \preceq cs' \text{ iff } cs = \texttt{CI} \ \vee (l \leqslant l' \wedge k \sqsubseteq k') \\ \text{where } cs = lk \text{ and } cs' = l'k' \end{aligned} \tag{1}$$

Formula (1) states that any context sensitivity variant is not less precise than CI, and $cs \preceq cs'$ only when both length and kind of context elements of cs are at most as precise as those of cs'.

Next we can formalize the C.S. selector of Unity as a function $selectCS\text{-}unity^S(m)$:

$$selectCS\text{-}unity^S(m) = \max \preceq_{S_i \in S} selectCS_i(m) \tag{2}$$

Based on $\preceq$, max $\preceq_{S_i \in S}$ chooses the most precise cs from a set of selective approaches S for method m. $selectCS_i$ corresponds to the selector function for S_i. By specifying $selectCS$ in [Call] to $selectCS\text{-}unity^S$, we get a pointer analysis guided by Unity, i.e., PA-Unity^S.

THEOREM 4.1 (SOUNDNESS OF Unity). PA-Unity^S *is sound.*

PROOF. By rules in Figure 6, PA-Unity^S is a standard Andersen-style pointer analysis, and the only difference from other selective context-sensitive pointer analyses is *selectCS*. Selecting different contexts for methods affects precision but not soundness [Jeon et al. 2018; Jeong et al. 2017; Li et al. 2018b, 2020; Smaragdakis et al. 2014]. Hence PA-Unity^S is sound. □

Following existing literature [Kastrinis and Smaragdakis 2013; Li et al. 2018b; Lu and Xue 2019; Minseok Jeon and Oh 2020], contexts in points-to results are ignored when measuring the precision of context-sensitive pointer analyses. To this end, we use $\overline{pt}_{\mathrm{PA}}(x)$ to represent the points-to set of variable x in pointer analysis PA, without contexts, e.g., if $pt(c, x) = \{\langle c^i, o_i\rangle\}$ and $pt(c', x) = \{\langle c^j, o_j\rangle\}$ in PA, then $\overline{pt}_{\mathrm{PA}}(x) = \{o_i, o_j\}$.

Definition 4.2 (Precision of Pointer Analysis). Given two sound pointer analyses PA and PA′, PA is at most as precise than PA′, denoted PA ⪯ PA′, if for each variable x: $\overline{pt}_{\mathrm{PA}}(x) \supseteq \overline{pt}_{\mathrm{PA}'}(x)$.

(We informally also use the convenient forms “less precise”, as a synonym of “at most as precise”, and “more precise” as a synonym of the converse. “Less than or equally” and “more than or equally” would have been more accurate, but inconvenient.)

We define the precision of a pointer analysis based on the points-to sets of each variable, as they are the most important results of pointer analysis and used by virtually all its clients.

THEOREM 4.3 (PRECISION OF Unity). *Given a set of selective approaches S,* $\forall S_i \in S$: PA-S_i ⪯ PA-Unity^S.

PROOF. By Equation (2), the *cs* selected by *selectCS-unity*^{S} for each method is always more precise than the one selected by any $S_i \in S$. This ensures that for each variable x in any PA-S_i, $\overline{pt}_{\mathrm{PA}\text{-}S_i}(x) \supseteq \overline{pt}_{\mathrm{PA}\text{-}\mathtt{Unity}^S}(x)$. Hence, $\forall S_i \in S$: PA-S_i ⪯ PA-Unity^S. □

4.4 Relay

Given a set of selective approaches S, in Relay, we run $|S|$ pointer analysis passes to divide the scalability burden into different passes (Section 3.3). Now we formalize the C.S. selector (Figure 1) of the two options of Relay (Relay-o1 and Relay-o2) for any pass$_i$.

We define the C.S. selector of Relay-o1 for pass$_i$ (denoted by *selectCS-relay*$_i^S$), as follows:

$$\textit{selectCS-relay}_i^S(m) = \begin{cases} \textit{selectCS-unity}^S(m) & \text{if } \textit{selectCS}_i(m) \neq \mathtt{CI} \\ \mathtt{CI} & \text{otherwise} \end{cases} \tag{3}$$

For a method m, if *selectCS*$_i$ (result of the i-th selective approach) selects a non-CI variant for m, then *selectCS-relay*$_i^S$ leverages *selectCS-unity*^{S} to select the most precise *cs*, otherwise, CI is selected, as illustrated in Figure 3.

In Relay-o2, to ensure scalability in pass$_i$, as explained in Section 3, we use the same context selector function as S_i:

$$\textit{selectCS-relay}_i^S(m) = \textit{selectCS}_i(m) \tag{4}$$

In each pass of Relay (except pass$_1$), we apply filtering on the points-to sets of variables based on the points-to results from the previous pass, to improve precision by reducing spurious data flows. Such filtering is formalized as the premises in boxes of [ASSIGN], [LOAD] and [CALL] (Figure 6), where $ppt(x)$ denotes the points-to set of variable x (without contexts) in the previous analysis pass (*ppt* reads as “previous points-to set”). We ignore contexts in *ppt* as the contexts of different passes may not be possible to match. These boxed premises ensure that if the previous analysis concludes that variable x does not point to object o_i, then all $\langle \_, o_i\rangle$ will be filtered out from $pt(\_, x)$ in the current (and future) passes, as they are identified as spurious points-to information by previous analyses.

Filtering prevents spurious objects from being propagated to variables. [New] does not propagate spurious objects (for statement $i : x = new\, T()$, x must point to o_i). [Store] propagates objects to instance fields (not variables), and the stored objects can only be accessed by a load operation. Then with the filtering by *ppt* in [Load], spurious objects from instance fields will not be propagated to other variables. Thus we do not need to apply filtering for [New] and [Store].

With filtering, the precision achieved in each pass can be transmitted to and accumulated in later passes.

Lemma 4.4 (Soundness of Filtering). *In Figure 6, if ppt(_) are the results of a sound pointer analysis* PA*, then the resulting filtered pointer analysis* PA′ *is also sound.*

Proof. By [Assign], [Load], and [Call], the filtering only removes the points-to results which are absent in PA. Since PA is sound, it will not filter out true points-to results. Hence the filtering based on PA does not affect soundness in PA′. □

Theorem 4.5 (Soundness of `Relay`). *For any pass$_i$ in* `Relay`*,* PA-`Relay`$_i^S$ *is sound.*

Proof. For any pass$_i$ in `Relay`, we assume that PA-`Relay`$_i^S$ may use any approach in S, i.e., the approaches in S can be arranged in any order. By the rules in Figure 6, PA-`Relay`$_i^S$ without pointer analysis filtering is sound (refer to the proof of Theorem 4.1). As pass$_1$ of `Relay` does not have filtering, PA-`Relay`$_1^S$ is sound (regardless of the approach it uses). The filtering of PA-`Relay`$_i^S$ is based on the results of PA-`Relay`$_{i-1}^S$. Thus, by Lemma 4.4, no matter what approaches PA-`Relay`$_{i-1}^S$ and PA-`Relay`$_i^S$ use (i.e., the order of approaches is irrelevant), if PA-`Relay`$_{i-1}^S$ is sound, PA-`Relay`$_i^S$ is sound. By induction, for any pass$_i$, PA-`Relay`$_i^S$ is sound. □

Theorem 4.6 (Precision of `Relay`). *Given a set of selective approaches S, $\forall S_i \in S$:* PA-$S_i \preceq$ PA-`Relay`$_i^S \preceq$ PA-`Relay`$_{|S|}^S$ *(the last pass of* `Relay`*) for both* `Relay-o1` *and* `Relay-o2`.

Proof. Similarly to the proof of Theorem 4.5, we assume that PA-`Relay`$_i^S$ may use any approach in S. By Equations (3) and (4), for each method, the *cs* selected by *selectCS-relay*$_i^S$ (of both `Relay-o1` and `Relay-o2`) is more precise than that selected by *selectCS*$_i$, thus, $\forall S_i \in S$: PA-$S_i \preceq$ PA-`Relay`$_i^S$ regardless of what approach S_i is. By pointer analysis filtering, each variable x in pass$_i$ must point to fewer (or the same) objects than x in pass$_{i-1}$, i.e., PA-`Relay`$_{i-1}^S \preceq$ PA-`Relay`$_i^S$; in other words, each pass is more precise than all its previous passes, no matter what approaches these passes use (i.e., the order of approaches is irrelevant). As PA-`Relay`$_{|S|}^S$ is the last pass of `Relay`, we have $\forall S_i \in S$: PA-`Relay`$_i^S \preceq$ PA-`Relay`$_{|S|}^S$. Hence, $\forall S_i \in S$: PA-$S_i \preceq$ PA-`Relay`$_i^S \preceq$ PA-`Relay`$_{|S|}^S$. □

Interestingly, Theorem 4.6 holds for `Relay-o2` regardless of whether the context-sensitivity variants selected by *selectCS-relay*$_i^S$ are comparable or not. Since by Equation (4), `Relay-o2` selects the same context-sensitivity variants as the selective approach S_i in pass$_i$, and further by the precision accumulation (via filtering), PA-`Relay`$_i^S$ is at least as precise as (practically more precise than) PA-S_i. Accordingly, the last pass of `Relay-o2` is at least as precise as any PA-S_i.

As for `Unity` and `Relay-o1`, in the case that some selected variants are theoretically incomparable, e.g., `2obj` and `2call`, their precision guarantee does not hold in theory. But in practice, users can choose an appropriate one empirically. E.g., `2obj` is typically much more precise than `2call` [Kastrinis and Smaragdakis 2013; Tan et al. 2016]. In this way, we can still obtain from `Unity-Relay` a pointer analysis that is likely more precise than any S_i in practice.

Scalability. `Relay-o2` is the last line of defense of `Unity-Relay`: given a set of selective approaches S, we can expect `Relay-o2` to scale as long as $\forall S_i \in S$: PA-S_i scales. With conventional approaches, when context-sensitively analyzing a set of scalability-threat methods, (1) the size of context-sensitive points-to relations may become large (thus needing much more time to handle in every analysis iteration), and (2) the analysis may need many more iterations to reach a fixed point. The simultaneous effects of (1) and (2) make the analysis complexity blow up, becoming unscalable. However, for `Relay-o2`, in each pass the context-sensitivity variant selected is the same as that of PA-S_i for every method, and due to the filtering mechanism, the points-to set for every variable is never larger than the one in PA-S_i (note that this holds regardless of the pass ordering in `Relay-o2`); therefore, in `Relay-o2`, there will be no more data propagated in each iteration and no more iterations until convergence. Thus `Relay-o2` scalability is ensured, as long as each PA-S_i scales. The only extra cost introduced in `Relay-o2` is its filtering process and its essence is querying whether an element (object) belongs to a set (points-to set), which can be implemented efficiently, and unlike (1) and (2) described above, it is not a key factor in making a pointer analysis unscalable. The experimental results in Section 6.2 further validate the scalability of `Relay`.

Other Context Abstractions for Selective Context Sensitivity. Our current formalism only focuses on the widely-used method-level selective approaches, i.e., it assumes that the input selective approaches choose context based on the target method (m in $selectCS(m)$). Still, the idea of `Unity-Relay` also applies to other context abstractions for selective context sensitivity. For example, introspective analysis [Smaragdakis et al. 2014] selects context based on call sites (i.e., different context-sensitivity variants for different call sites). To support such a selective approach, we only need to change the input domain of *selectCS* to further include call sites, and the context selection policy of `Unity-Relay` remains the same. As another alternative, Lu and Xue [2019] select context on a per-variable basis. The `Unity-Relay` idea applies seamlessly, but at the granularity of variables, not methods, i.e., one should see the circles in Figures 2 and 3 as selected variables.

5 BATON

As a proof-of-concept, we introduce Baton, an instantiation of the `Unity-Relay` framework. Given plenty of existing selective context sensitivity approaches [Hassanshahi et al. 2017; Jeon et al. 2019; Jeong et al. 2017; Li et al. 2018a,b, 2020; Lu and Xue 2019; Minseok Jeon and Oh 2020; Oh et al. 2014, 2015; Smaragdakis et al. 2014; Wei and Ryder 2015], the design space of `Unity-Relay` is large. For more precision improvement, the chosen approaches should be *diverse* so that they could cover precision-useful methods from *different perspectives*; and they should exhibit good scalability, as explained in Section 3.3. Accordingly, Baton considers three such approaches as inputs: one ad-hoc approach based on expert experience, called Collection [WALA 2018], and two state-of-the-art approaches, called Zipper^e [Li et al. 2020] and Scaler [Li et al. 2018b].

Collection (or container) methods are important to pointer analysis, as a huge amount of objects may flow to and are merged in them, if the collection is not being analyzed context-sensitively. Thus, in pointer analysis frameworks such as Wala [WALA 2018], the experts provide an option to analyze only collection methods (in JDK) context-sensitively for good precision (with also good scalability). Inspired by [WALA 2018], in Collection, we apply `3obj` to the collection methods whose declaring classes implement interfaces `java.util.Collection` and `java.util.Map` in both application and library code.

Zipper^e (short for Zipper express) [Li et al. 2020] is a variant of a selective approach called Zipper [Li et al. 2018a] which identifies precision-useful methods based on principled precision-loss patterns [Li et al. 2020]. By exploiting the patterns, Zipper^e adopts simple but effective heuristics to further exclude the scalability-threat methods. As a result, by applying `2obj` only to the finally selected methods (with the others being analyzed context-insensitively), compared to Zipper, Zipper^e substantially improves efficiency with similar precision.

Scaler [Li et al. 2018b] leverages the object allocation graph in [Tan et al. 2016] to estimate the context-sensitive points-to sizes that would be needed for each method. Then it selects an appropriate context-sensitivity variant (2obj, 2type, 1type or CI) for each method, while keeping the overall points-to size bounded (to a quantity that represents the memory capacity available for running the analysis), resulting in good scalability.

For a given program P, Baton applies the above three approaches to analyze P to obtain their context-sensitivity variants selected for each method of P. Based on these results, Baton produces new context-sensitivity configurations according to the Unity and Relay principles. In particular, in Relay, we run Collection, Zipper^e and Scaler in this order. We switch between Relay-o1 and Relay-o2, choosing the more precise option (Relay-o1) if it scales, per the discussion of Section 3.

As all context-sensitivity variants selected by Collection, Zipper^e and Scaler are comparable in precision, by Theorems 4.3 and 4.6, we can expect Baton to yield more precise results than all of them in practice. This will be further validated in Section 6.

6 EVALUATION

This section examines how Baton performs when addressing the challenging research problem raised in Section 1: "Given reasonably long time, can we achieve precise pointer analysis results for hard-to-analyze programs, for which traditional context-sensitive analyses fail to scale, and selective context-sensitive approaches scale but with limited precision?". We investigate the following research questions:

RQ1. Given that Baton (Unity) picks the "most precise configuration" based on the input selective approaches, how does this design scale for hard-to-analyze programs in practice? How does it fare against state-of-the-art analyses in terms of the precision gain that we aim for?

RQ2. In the cases for which Baton (Unity) fails to scale, how does Baton (Relay), as the second punch of Unity-Relay, perform in terms of scalability and precision?

Experimental Settings. We conduct all experiments on a machine with an Intel Xeon 2.2GHz CPU and 128GB of memory. All pointer analyses are performed on Doop [Bravenboer and Smaragdakis 2009], the state-of-the-art pointer analysis framework for Java (with the version published as the artifact of [Smaragdakis et al. 2014]). All pointer analyses adopt the same reflection handling configuration for the same program. Specifically, for each program, we first run the dynamic reflection analysis tool TamiFlex [Bodden et al. 2011] and its results are used if TamiFlex analyzes the program successfully; otherwise (if TamiFlex throws exceptions), we use Doop's default reflection analysis setting. Time budget is set to 2 hours for each analysis. In evaluation, all benchmarks are analyzed with a large Java library: OpenJDK 1.6.0_24, which is widely used in recent work [Jeon et al. 2019, 2018; Li et al. 2018a, 2020; Minseok Jeon and Oh 2020].

Hard-to-analyze Programs. We consider 13 large and complex Java programs as our benchmarks, including all the hard-to-analyze programs in the standard DaCapo benchmarks [Blackburn et al. 2006] and recent literature for Java pointer analysis [Jeon et al. 2019, 2018; Jeong et al. 2017; Kastrinis and Smaragdakis 2013; Li et al. 2018b, 2020; Minseok Jeon and Oh 2020; Smaragdakis et al. 2014], for which traditional 2obj fails to scale within time budget (2 hours). To our knowledge, this is the largest set of hard-to-analyze programs evaluated in related literature.

Precision Metrics. To thoroughly measure precision, we consider the most complete set of precision metrics that were used in recent literature [Jeon et al. 2019, 2018; Jeong et al. 2017; Kastrinis and

Smaragdakis 2013; Li et al. 2018a,b, 2020; Lu and Xue 2019; Minseok Jeon and Oh 2020; Smaragdakis et al. 2014]. It consists of three general precision metrics, i.e., the total size of points-to sets for all variables (VarPts), the average size of points-to set per variable (AvgPts), and the number of all may-alias variable pairs (Aliases); and four independently useful client analyses that are often adopted in the literature to measure precision. The clients include a cast-resolution analysis (with the metric being the number of cast operations that may fail, denoted FCasts), a devirtualization analysis (with the metric being the number of virtual call sites that may have multiple call targets, denoted PCalls), a method reachability analysis (with the metric being the number of reachable methods, denoted RMtds), and a call-graph construction analysis (with the metric being the number of call graph edges, denoted CEdges).

State of the art. The nature of Unity-Relay enables Baton to take advantage of the precision achieved by any selective approach (as input), thereby whatever the state-of-the-art is, Baton can achieve practically better precision (Theorems 4.3 and 4.6). In this context, we compare against two state-of-the-art selective approaches, Zipper^e [Li et al. 2020] and Scaler [Li et al. 2018b] (also adopted by Baton as inputs in Section 5), to evaluate how much more precision Baton can achieve. Zipper^e and Scaler are open-source tools that try to maximize the precision while not threatening scalability, and have been demonstrated to be significantly more precise than other selective approaches [Smaragdakis et al. 2014]. We do not consider Zipper [Li et al. 2018a] as it fails to scale for all our benchmarks.

6.1 RQ1. Precision and Scalability of Baton (Unity).

For Zipper^e and Scaler, we use their default configurations as in Li et al. [2018b, 2020], which are well-tuned and achieve good precision and efficiency trade-offs. As part of Baton, we also include Collection for comparison. To provide a sense of how precise the pointer analyses (guided by these selective approaches) are, we consider CI as a baseline.

Figure 7 shows graphically the precision improvement for a single metric: Aliases (i.e., may-alias variable pairs). Table 1 presents the results for all analyses, in full detail. For each program, there are five rows of data, corresponding to the five analyses evaluated. Columns 3–9 list the seven precision metrics explained above, and the rightmost column shows the analysis time in seconds. Note that Zipper^e, Scaler and Baton (Unity) require a pre-analysis to select contexts, however, these computations are very fast (from seconds to minutes) and are negligible compared to pointer analysis time, thus the pre-analysis time is elided in our evaluation. Next, we examine the detailed results in Table 1. For all numbers in the table, lower is better.

Precision. From Table 1, we can see that Zipper^e, Scaler and Collection are much more precise than CI. For Baton (Unity), as expected, it achieves better precision than *all* these three analyses for *all* precision metrics for *all* programs in Table 1. This result also validates Theorem 4.3. The precision improvements are significant in general, and are striking for some programs such as hsqldb and heritrix. Considering that the compared analyses are already highly precise, such precision improvements made by Baton (Unity) are impressive. Baton (Unity) performs well especially for VarPts, AvgPts and Aliases, which are the most important outputs of pointer analysis, necessary for numerous clients. The precision improvements for PCalls and RMtds are not as remarkable as other metrics. Actually, existing work has shown that the precision improvement of a pointer analysis has less impact on these two clients compared to others [Kastrinis and Smaragdakis 2013]. This can also be observed in the experimental results from recent selective context-sensitive pointer analysis work [Jeon et al. 2019; Jeong et al. 2017; Li et al. 2020]. Finally, due to high precision, Baton (Unity) can avoid spending time on propagating many spurious data flows. Thus, sometimes, it can run even faster than individual analyses (Collection, Zipper^e or Scaler) as discussed below.

Table 1. Precision and performance metrics for context-insensitive (CI) and selective context-sensitive pointer analyses guided by Collection, Zipper^e, Scaler, and Baton (Unity) respectively. For all numbers, lower is better.

Program	Analysis	VarPts	AvgPts	Aliases	FCasts	PCalls	RMtds	CEdges	Time (s)
hsqldb	CI	5,120,447	53.4	54,502,418	1,662	1,592	11,486	63,790	59
	Collection	2,239,445	24.2	25,676,463	1,342	1,343	10,995	56,426	69
	Zipper^e	1,110,244	12.5	8,650,411	1,032	1,261	10,440	51,261	119
	Scaler	1,062,721	11.8	12,096,236	1,120	1,197	10,639	52,063	357
	Baton (Unity)	738,878	8.4	6,905,176	842	1,145	10,304	49,864	450
galleon	CI	61,536,682	224.6	774,840,941	5,063	7,046	31,137	195,026	571
	Collection	21,484,100	80.9	338,974,351	3,647	6,071	30,038	164,764	1,012
	Zipper^e	25,881,909	97.8	330,526,330	3,923	5,903	29,786	165,085	583
	Scaler	28,084,055	105.2	383,443,213	3,739	6,175	30,194	167,957	4,697
	Baton (Unity)	7,831,380	31.5	110,490,509	3,023	5,458	28,138	139,979	533
jedit	CI	16,874,067	98.0	224,287,546	3,382	4,749	21,006	118,426	104
	Collection	3,895,227	23.1	66,401,696	2,516	4,152	20,562	99,574	79
	Zipper^e	3,588,763	21.4	59,605,295	2,304	4,065	20,418	98,290	100
	Scaler	3,583,465	21.3	61,720,584	2,377	3,990	20,499	97,999	1,580
	Baton (Unity)	3,140,272	18.8	52,291,827	2,098	3,895	20,340	96,872	2,028
soot	CI	110,901,529	365.2	2,006,757,243	16,570	16,532	32,459	415,476	1,013
	Collection	31,431,433	105.1	744,789,961	10,296	14,765	31,952	367,708	1,221
	Zipper^e	33,832,881	113.1	859,999,654	10,673	14,666	31,965	326,092	797
	Scaler	34,952,536	116.8	817,587,505	10,549	14,822	31,982	374,877	1,257
	Baton (Unity)	24,080,734	80.7	579,686,802	9,483	14,499	31,878	308,306	574
gruntspud	CI	23,261,792	112.6	301,389,233	3,583	5,703	24,887	148,874	171
	Collection	8,099,569	40.1	103,258,215	2,636	4,820	24,210	120,835	137
	Zipper^e	7,670,883	38.2	104,660,677	2,549	4,739	24,117	120,316	288
	Scaler	7,577,272	37.6	100,355,028	2,479	4,699	24,207	120,177	2,761
	Baton (Unity)	6,275,349	31.4	81,988,438	2,096	4,572	23,992	116,842	3,254
heritrix	CI	19,576,304	120.3	208,124,549	2,514	3,264	18,820	117,551	198
	Collection	6,268,412	41.0	71,227,109	1,628	2,553	17,549	94,501	217
	Zipper^e	7,179,587	47.2	78,727,889	1,734	2,501	17,419	96,633	348
	Scaler	7,106,210	46.7	80,018,315	1,637	2,461	17,461	94,785	2,601
	Baton (Unity)	2,546,396	17.7	20,679,819	1,200	2,182	16,428	82,301	847
pmd	CI	7,713,272	57.8	86,502,490	2,948	4,183	15,254	104,457	67
	Collection	2,765,814	21.1	28,677,101	2,237	3,684	14,967	94,750	74
	Zipper^e	2,760,540	21.1	29,809,145	2,153	3,634	14,908	93,516	386
	Scaler	2,410,883	18.4	28,460,705	2,176	3,536	14,895	92,775	697
	Baton (Unity)	2,141,253	16.5	24,976,066	1,956	3,463	14,803	91,762	3,602
jython	CI	16,581,997	157.7	183,658,141	2,234	2,778	12,718	114,856	108
	Collection	13,281,462	131.5	148,452,466	1,901	2,572	12,205	107,903	152
	Zipper^e	12,948,335	129.8	142,634,911	1,781	2,486	12,026	107,113	134
	Scaler	13,016,241	129.1	134,214,889	1,852	2,500	12,167	107,410	459
	Baton (Unity)	12,687,718	127.3	129,984,727	1,719	2,438	11,997	106,645	773
jasperreports	CI	73,783,904	276.9	586,687,036	5,305	7,060	33,019	187,173	577
	Collection	14,935,335	59.9	151,406,154	3,501	5,544	30,774	152,250	627
	Zipper^e	32,834,048	126.5	258,484,882	4,202	6,083	32,044	167,340	1,043
	Scaler	19,025,792	75.9	176,489,911	3,615	5,577	30,904	154,745	1,551
	Baton (Unity)	6,453,254	26.5	74,908,878	2,994	5,267	30,008	142,256	834
eclipse	CI	24,841,670	123.4	261,199,033	4,190	9,197	20,862	161,222	136
	Collection	11,366,316	57.4	70,520,064	3,266	8,657	20,523	146,770	122
	Zipper^e	11,848,840	59.8	90,933,963	3,223	8,599	20,499	148,037	3,052
	Scaler	11,117,708	56.4	73,208,882	3,211	8,486	20,374	145,953	680
	Baton (Unity)	9,817,684	50.4	62,570,596	2,834	8,352	20,142	143,503	5,965
briss	CI	34,335,976	136.6	370,495,823	4,904	6,297	26,582	176,785	233
	Collection	9,454,391	38.7	101,712,404	3,416	5,373	25,706	153,414	151
	Zipper^e	8,202,040	33.7	85,183,405	3,158	5,306	25,537	151,550	217
	Scaler	9,294,471	38.1	101,195,279	3,428	5,323	25,652	152,761	1,265
	Baton (Unity)	7,653,188	31.5	79,503,155	2,955	5,147	25,473	150,749	1,726
columba	CI	86,528,734	336.2	745,538,811	4,889	6,669	31,476	195,930	771
	Collection	8,837,513	41.8	109,500,746	2,842	4,774	25,709	122,187	218
	Zipper^e	21,510,036	100.4	264,213,761	3,444	5,213	26,051	140,637	837
	Scaler	9,986,400	47.2	122,951,398	2,897	4,799	25,729	123,330	295
	Baton (Unity)	6,658,750	32.1	85,365,064	2,393	4,522	25,157	118,803	358

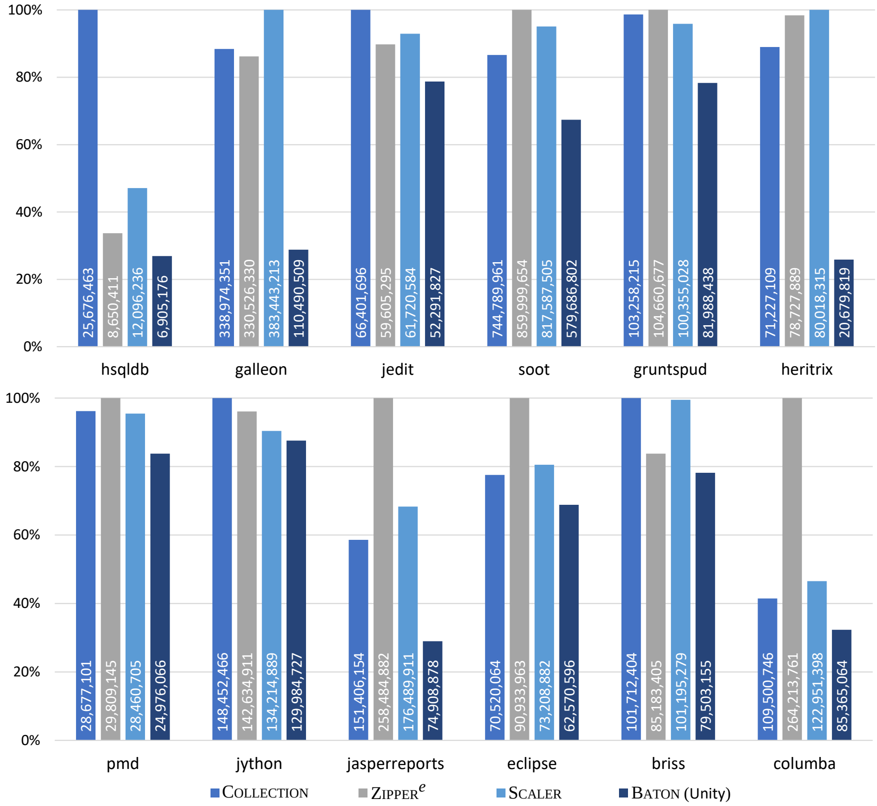

Fig. 7. May-alias variable pairs (`Aliases`), normalized to highest. Baton (`Unity`) consistently finds a significant proportion of previously-reported alias pairs to be spurious.

Scalability. Collection, Zipper^e and Scaler can analyze all 13 programs within time budget, demonstrating their extremely good scalability. As for Baton (`Unity`), even though it chooses the *most precise configuration* for all methods selected by all approaches in Baton, it is still able to scale for 12 out of 13 hard-to-analyze programs as shown in Table 1, validating the insights of Section 3.2. The only exception is `h2`, which will be handled by Baton (`Relay`) as shown in Section 6.2. Although our focus is precision, it is worth pointing out that Baton (`Unity`) is always more precise but not always slower. For `galleon`, `heritrix`, `soot`, `jasperreports` and `columba`, Baton (`Unity`) is faster than at least one of the state-of-the-art systems. For `galleon` and `soot`, Baton (`Unity`) is even faster than all other analyses including `CI`! Such surprising performance advantages stem from the significant precision improvements of Baton (`Unity`).

6.2 RQ2. Precision and Scalability of Baton (Relay).

We evaluate how Baton (`Relay`), as the second punch of Baton, performs in practice when Baton (`Unity`) is not scalable. With default settings of Zipper^e and Scaler, Baton (`Unity`) can scale for all programs except for `h2` (Section 6.1). However, to thoroughly evaluate Baton (`Relay`), we need to examine more such cases. Zipper^e and Scaler are tunable, i.e., they both provide options to tune the analysis for different precision and efficiency trade-offs. Specifically, Zipper^e can be configured by a percentage value, PV, which corresponds to the threshold for identifying scalability-threat

Table 2. Precision and performance metrics for context-insensitive CI and selective context-sensitive pointer analyses guided by reconfigured Zipper^e, Scaler, and each pass of Baton (Relay). For all numbers, lower is better.

Program	Analysis	VarPts	AvgPts	Aliases	FCasts	PCalls	RMtds	CEdges	Time (s)
eclipse	CI	24,841,670	123.4	261,199,033	4,190	9,197	20,862	161,222	136
	Zipper^e	11,848,840	59.8	90,933,963	3,223	8,599	20,499	148,037	3,052
	Scaler	10,975,564	55.7	72,760,368	3,170	8,448	20,352	145,412	1,396
	Baton-p1 (Relay-o1)	11,366,316	57.4	70,520,064	3,266	8,657	20,523	146,770	122
	Baton-p2 (Relay-o1)	10,484,964	53.2	65,384,654	3,003	8,534	20,426	145,849	3,592
	Baton-p3 (Relay-o1)	9,865,071	50.7	62,549,586	2,823	8,335	20,127	143,046	3,171
briss	CI	34,335,976	136.6	370,495,823	4,904	6,297	26,582	176,785	233
	Zipper^e	5,966,496	24.6	80,559,736	3,016	5,209	25,440	150,826	1,899
	Scaler	8,830,552	36.4	97,300,729	3,298	5,190	25,541	151,224	4,706
	Baton-p1 (Relay-o1)	9,454,391	38.7	101,712,404	3,416	5,373	25,706	153,414	151
	Baton-p2 (Relay-o1)	5,279,719	21.8	77,196,053	2,918	5,147	25,432	150,621	2,976
	Baton-p3 (Relay-o1)	5,178,884	21.4	75,371,504	2,807	5,028	25,378	149,864	6,205
pmd	CI	7,713,272	57.8	86,502,490	2,948	4,183	15,254	104,457	67
	Zipper^e	2,760,540	21.1	29,809,145	2,153	3,634	14,908	93,516	386
	Scaler	2,282,726	17.5	26,945,008	2,080	3,498	14,839	92,439	2,230
	Baton-p1 (Relay-o1)	2,765,814	21.1	28,677,101	2,237	3,684	14,967	94,750	74
	Baton-p2 (Relay-o1)	2,558,733	19.6	26,892,448	2,072	3,616	14,891	93,164	1,827
	Baton-p3 (Relay-o1)	2,142,179	16.5	24,966,489	1,958	3,462	14,802	91,764	6,164
jedit	CI	16,874,067	98.0	224,287,546	3,382	4,749	21,006	118,426	104
	Zipper^e	2,771,120	16.6	51,199,984	2,148	3,937	20,350	96,946	3,055
	Scaler	3,059,304	18.3	56,958,326	2,228	3,909	20,386	97,154	1,520
	Baton-p1 (Relay-o1)	3,895,227	23.1	66,401,696	2,516	4,152	20,562	99,574	81
	Baton-p2 (Relay-o1)	2,627,200	15.7	49,384,180	2,113	3,930	20,346	96,763	3,887
	Baton-p3 (Relay-o1)	2,575,332	15.4	48,120,051	2,022	3,845	20,327	96,463	4,347
h2	CI	3,566,057	30.1	66,745,220	1,866	3,946	14,192	95,541	49
	Zipper^e	1,571,052	13.5	26,790,820	1,418	3,646	13,903	89,292	254
	Scaler	1,368,866	11.8	26,113,035	1,373	3,605	13,850	88,268	1,832
	Baton-p1 (Relay-o1)	1,647,387	14.1	26,677,699	1,523	3,654	13,935	89,361	63
	Baton-p2 (Relay-o1)	1,480,200	12.7	22,915,240	1,384	3,632	13,867	88,959	2,284
	Baton-p3 (Relay-o2)	1,238,765	10.7	22,264,106	1,303	3,593	13,821	88,027	2,004
gruntspud	CI	23,261,792	112.5	301,389,233	3,583	5,703	24,887	148,874	171
	Zipper^e	5,601,249	28.0	79,231,338	2,232	4,629	24,010	116,972	1,370
	Scaler	6,928,500	34.4	99,005,787	2,470	4,671	24,166	119,136	4,546
	Baton-p1 (Relay-o1)	8,099,569	40.1	103,258,215	2,636	4,820	24,210	120,835	137
	Baton-p2 (Relay-o2)	5,525,251	27.6	78,100,575	2,199	4,625	24,001	116,892	1,456
	Baton-p3 (Relay-o2)	5,356,169	26.8	76,524,253	2,113	4,541	23,973	116,558	3,962

methods, and Scaler is parameterized by a number called *total scalability threshold* (TST), which corresponds to a bound on the overall points-to size. For both analyses, using higher thresholds (PV and TST) would generally improve precision but reduce efficiency. Thus, we change their settings in Baton by increasing PV and TST, to make them more precise, and accordingly, more expensive, until Baton (Unity) no longer scales but each individual selective approach can still terminate within the time budget for all the programs in Table 1.

As a result, Table 2 shows six programs that fulfill our experimental requirements. As the baseline, CI is still included. The results of Collection are the same as the ones of Baton-p1 (Relay-o1) (since Baton (Relay) runs Collection in its first pass as described in Section 5), thus we use the latter to represent Collection. For each program, we list the analysis results of the reconfigured Zipper^e, Scaler, and the three passes (Section 5) of the accordingly-reconfigured Baton (Relay).

Precision. Theorem 4.6 proves that the last pass of Relay is more precise than any individual selective approach in the framework, and Table 2 validates this in practice: Baton (Relay) obtains better precision than Zipper^e and Scaler (and also Collection) for *all* precision metrics for *all* programs. Figure 8 shows graphically the precision improvement for the Aliases metric.

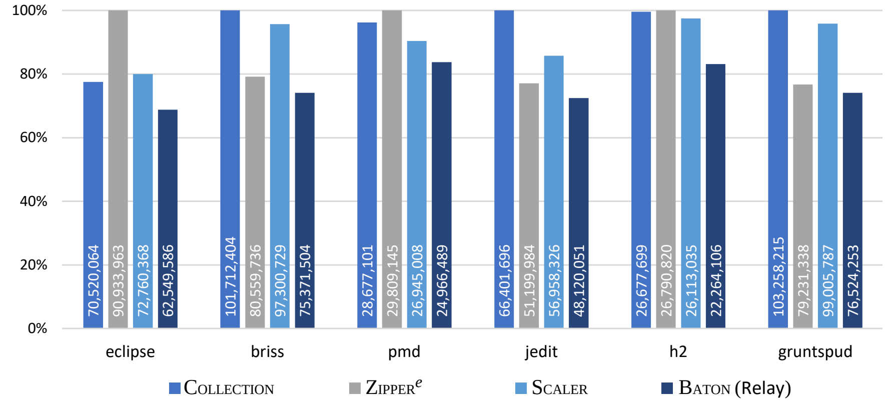

Fig. 8. May-alias variable pairs (Aliases), normalized to highest. Baton (Relay) consistently improves over the *best* past algorithm. Importantly, the best algorithm varies per benchmark. On average, Baton (Relay) improves the precision of Collection, Zipper^e, and Scaler by 20%, 13% and 16% respectively.

We can also observe from Table 2 that Baton (Relay) improves the precision in a stable, cumulative way, pass after pass (Baton-p1, -p2 and -p3). For 4 out of 6 programs, the precision achieved by the second pass of Baton (Relay), i.e., Baton-p2, already outperforms the other analyses.

Scalability. For all six programs in Table 2, for which the reconfigured Baton (Unity) fails to scale, Baton (Relay) is scalable, by dividing the scalability burden of Baton (Unity) into different passes. The beneficial effects of on-the-fly filtering, based on the results of the previous pass, can be seen in the experiment. For instance, one can observe that, for gruntspud, Baton-p3 (Relay-o2) spends 3,962s when the corresponding Scaler analysis requires 4,546s. This is entirely due to the precision transmitted from passes 1 and 2, which can help filter many spurious object flows during the analysis of pass 3, making the analysis reach fixed point more quickly. Although the purpose of filtering is to improve precision, and not scalability, when the time saved by the improved precision exceeds the time cost of filtering itself, such results arise.

Pass Ordering. We conduct further experiments to examine whether the pass ordering of Relay affects precision and scalability. The present Relay runs Collection, Zipper^e and Scaler, in this order, for each pass. In the extra experiments, we have two new orderings: (1) Collection, Scaler and Zipper^e and (2) Scaler, Zipper^e and Collection. We run Relay for all the programs in Table 2 under these two orderings. For scalability, as expected, Relay-o2 is still able to scale for all the programs, keeping its scalability promise. For Relay-o1, the Scaler-related pass fails to scale for briss under the new orderings: the context-sensitivity configuration in Relay-o1 for Scaler is relatively expensive in terms of analysis efficiency, but when running Zipper^e before Scaler, as in the original ordering, the former helps prune away many spurious objects, which makes the latter more scalable by reaching fixed point faster. This instance suggests that a pass ordering with relatively fast and precise analyses placed first can lead to a more scalable Relay-o1, thus avoiding the fallback to the less-precise Relay-o2. Again, the pass ordering does not affect the scalability capacity of Relay-o2, as explained in Section 4.4, and validated above.

For precision, as expected, Relay outperforms the other analyses in all precision metrics under both new orderings. (For the aforementioned briss case, note the fallback to Relay-o2.) This further validates the Relay precision guarantee. Regarding the final precision produced by Relay under the original and the two new orderings, although one may perform better than the others for different programs, the precision difference among them is very minor: less than 0.1% on average. Therefore, although the pass ordering may affect the final precision of Relay, as explained in Section 3.3, the experimental results show that such impact is very minor in practice. (The precision guarantee of Relay proved in Theorem 4.6 remains unaffected.)

Other Choices for Precision Improvements. When Baton (Unity) fails to scale, one may wonder whether some other schemes are better choices than Relay for improving precision. This may include: (1) downgrading the context-sensitivity variants for the overlapped methods reported by Zipper^e, Scaler and Collection (e.g., choose the faster 2type rather than 2obj if both are reported), (2) simply intersecting the points-to results produced by Zipper^e, Scaler and Collection, and (3) further tuning Zipper^e or Scaler for better precision by increasing their provided thresholds.

We conduct further experiments and find that none of the above schemes is better than Baton (Relay) for improving precision. For (1), this scheme still fails to scale for jedit, and for the remaining 5×7=35 precision metrics, Baton (Relay) outperforms it in almost all (33 out of 35) cases. For (2), Baton (Relay) achieves better precision for all 42 metrics. For (3), when increasing the current threshold of Zipper^e (used in Table 2) by only 1%, (e.g., from 16% to 17% for gruntspud), Zipper^e is already not scalable for several programs; when increasing the threshold of Scaler from current 60M to a high value 100M, Scaler generally becomes significantly slower (e.g., 6,856s for briss), but Baton (Relay) (with the current 60M) still outperforms it in precision for virtually all (40 out of 42) metrics. Most importantly, no matter how precise an input selective approach becomes, Baton (Relay) can always leverage it for better precision.

7 RELATED WORK

To our knowledge, Unity-Relay is the first technique that systematically exploits and leverages *multiple* selective context-sensitivity approaches for yielding high precision for Java pointer analysis. As experimental results show, Unity-Relay is able to produce a pointer analysis with a precision that is well beyond existing approaches for hard-to-analyze programs. Although the ideas of Unity and Relay have not been applied to pointer analysis before, for Relay, the high-level idea of its precision filtering is similar to the reduced product approach in abstract interpretation [Cousot and Cousot 1979]: different domains (e.g., an integer interval domain and a parity domain) can be used to improve the precision of another in one analysis; however, in Relay, there is only one domain (the mapping from pointers to heap objects) and the precision filtering process is divided into multiple analyses (passes). The high-level idea of staged analyses in Relay has also been adopted in static analysis, e.g., in staged verification [Fink et al. 2008], faster verifiers run in early stages which reduce the workload for later, more precise, stages; however, in Relay, neither the time cost nor the precision of the analysis in each stage are comparable. Relay is also different in how it utilizes the results from a previous stage to incorporate with the analysis in a later stage. (Such handling also differs from staged optimization [Philipose et al. 2002].) Thus, although Relay shares some similarities with the high-level ideas of previous work, technical specifics are different.

We have discussed Zipper^e [Li et al. 2020], Scaler [Li et al. 2018b] and Collection [WALA 2018] in Section 5. Below we review other related work about selective context-sensitive pointer analysis.

Oh et al. [2014] present a principled approach to resolving given queries for C programs by selective context sensitivity. In particular, the approach uses a pre-analysis to estimate the impact of context sensitivity on the precision of the subsequent main analysis. In Oh et al. [2015], this approach is extended to include selective flow sensitivity by following the same principle to demonstrate its generality.

Smaragdakis et al. [2014] introduce introspective analysis, which relies on several manually-selected metrics (e.g., the maximum field points-to set per object) to define heuristics for determining which methods are supposed to be analyzed context-sensitively. Similar parameterized heuristics are also leveraged by Hassanshahi et al [Hassanshahi et al. 2017] to decide whether object sensitivity should be applied to certain methods.

Like Scaler [Li et al. 2018b], context elements may also vary for different methods in the work of Wei and Ryder [2015] and Thakur and Nandivada [2020]. Differently, in Wei and Ryder [2015], the relationship between context-sensitivity variants and methods is obtained by a machine-learning approach and domain knowledge. In the work of Thakur and Nandivada [2020], object sensitivity is further added to the methods that have been assigned call-site sensitivity, in a special form of context abstraction proposed earlier by Thakur and Nandivada [2019].

Data-driven approaches [Jeon et al. 2019; Jeong et al. 2017] assign to each method an appropriate context length by learning heuristics based on various program elements (e.g., some Java keywords or specific statements) expressed in a disjunctive form. Although the learning phase is relatively heavy, it helps in making good precision and efficiency trade-offs for pointer analysis. However, as well-known weaknesses, machine-learning approaches may behave unpredictably for new inputs, and their learned results, although useful, are usually difficult to explain. Still, this does not prevent the ability of `Unity-Relay` to take advantage of learned results (based on opaque insights from machine-learning algorithms) to gain good precision, just as it can for any other individual selective approach.

Lu and Xue [2019] present a scheme to selectively apply context sensitivity on the granularity of variables (rather than methods) based on CFL-reachability [Reps 1997]. The general idea of `Unity-Relay` also works for this granularity by considering which portion of a program should be analyzed by what context-sensitivity variants at the variable level (see the circles in Figures 2 and 3 as selected variables).

It is worth pointing out that `Unity-Relay` is not in competition with the above approaches, but instead complements them by taking advantages of their insights (through consuming their outputs) to yield more precise results.

When we understand selective context sensitivity more generally as "applying context sensitivity discriminately, in order to save analysis cost" it becomes similar to ideas also explored in demand/client-driven pointer analyses [Guyer and Lin 2003; Liang and Naik 2011; Liu et al. 2019; Späth et al. 2019; Späth et al. 2016; Sridharan and Bodík 2006; Sridharan et al. 2005; Sui and Xue 2016; Wang et al. 2017; Zhang et al. 2014]: these reduce the analysis cost by computing only the results which are necessary to answer specific queries at given program locations.

8 CONCLUSIONS

Pointer analysis is hard to scale with good precision for large and complex Java programs. This challenge needs to be urgently addressed, as such programs dominate real practice. To address the problem, we present the simple and practical `Unity-Relay` framework for systematically exploiting a set, S, of selective context-sensitivity approaches, to produce highly-precise pointer analysis results for hard-to-analyze Java programs. Its first punch, `Unity`, fully unleashes the precision potential of all approaches in S with good scalability. When `Unity` fails, the second punch, `Relay`, can be confidently relied upon to reap precision in a stable and accumulative way. `Unity-Relay` has soundness and precision guarantees relative to the individual selective approaches it combines. Furthermore, if each approach in S is scalable, users can expect that `Unity-Relay` is also able to scale by its last defender of scalability, `Relay-o2`: a technique guaranteed to (at worst) match the scalability profile of individual approaches in S. We validate these properties in practice via extensive experiments in Section 6.

The Unity-Relay proof-of-concept tool, Baton, has been demonstrated to be very effective: Baton achieves the best precision for all precision metrics and clients for all evaluated programs. It is the first time that these levels of precision are obtained for these hard-to-analyze programs.

Baton is merely one instantiation of the Unity-Relay framework. In the future, we expect more instantiations to unleash the power of more effective selective context-sensitivity approaches (including ones not yet foreseen).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported in part by Key-Area Research & Development Program of Guangdong Province (Grant #2020B010164003), Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant #BK20202001), and National Natural Science Foundation of China (Grants #61932021, #62025202, #62002157), and by the Hellenic Foundation for Research and Innovation (project DEAN-BLOCK). The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*. 241–250. https://doi.org/10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 363–374. https://doi.org/10.1145/1542476.1542517

Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 332–343. https://doi.org/10.1145/2970276.2970347

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269âĂŞ282. https://doi.org/10.1145/567752.567778

Pratik Fegade and Christian Wimmer. 2020. Scalable Pointer Analysis of Data Structures Using Semantic Models. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 39âĂŞ50. https://doi.org/10.1145/3377555.3377885

Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 465–484.

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

Samuel Z. Guyer and Calvin Lin. 2003. Client-Driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 214âĂŞ236.

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 13–18. https://doi.org/10.1145/3088515.3088519

Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 13:1–13:41. https://doi.org/10.1145/3293607

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

Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *PACMPL* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. https://doi.org/10.1145/2931098

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. https://doi.org/10.1145/2462156.2462191

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

Ondrej Lhoták and Laurie J. Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3923)*, Alan Mycroft and Andreas Zeller (Eds.). Springer, 47–64. https://doi.org/10.1007/11688839_5

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

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proc. 12th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 129–140. https://doi.org/10.1145/3236024.3236041

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

Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

Percy Liang and Mayur Naik. 2011. Scaling Abstraction Refinement via Pruning. *SIGPLAN Not.* 46, 6 (June 2011), 590âĂŞ601. https://doi.org/10.1145/1993316.1993567

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 6 (March 2019), 31 pages. https://doi.org/10.1145/3293606

Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association.

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

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 1–11. https://doi.org/10.1145/566172.566174

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

Myungho Lee Minseok Jeon and Hakjoo Oh. 2020. Learning Graph-based Heuristics for Pointer Analysis without Hand-crafting Application-Specific Features. *Proc. ACM Program. Lang.* OOPSLA (2020).

Mayur Naik, Alex Aiken, and aley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

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

2594291.2594318

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2015. Selective X-Sensitive Analysis Guided by Impact Pre-Analysis. *ACM Trans. Program. Lang. Syst.* 38, 2, Article 6 (Dec. 2015), 45 pages. https://doi.org/10.1145/2821504

Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2002. Towards Automatic Construction of Staged Compilers. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 113âĂŞ125. https://doi.org/10.1145/503272.503284

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 925–935. https://doi.org/10.1109/ICSE.2012.6227127

Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming* (Port Washington, New York, USA) *(ILPS '97)*. MIT Press, Cambridge, MA, USA, 5âĂŞ19.

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

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. https://doi.org/10.1145/2594291.2594320

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

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

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 112–122. https://doi.org/10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. https://doi.org/10.1145/1094811.1094817

Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. *Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity (Artifact)*. https://doi.org/10.5281/zenodo.5491895

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. https://doi.org/10.1145/3062341.3062360

Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC*

2019). ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359

Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27âĂŞ38. https://doi.org/10.1145/3377555.3377902

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

WALA. 2018. Watson Libraries for Analysis. http://wala.sf.net.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. *SIGPLAN Not.* 39, 6 (June 2004), 131âĂŞ144. https://doi.org/10.1145/996893.996859

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

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 239âĂŞ248. https://doi.org/10.1145/2594291.2594327