# The ART of Sharing Points-to Analysis

Reusing Points-to Analysis Results Safely and Efficiently

SHASHIN HALALINGAIAH, IIT Madras, India and University of Texas at Austin, USA
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
V. KRISHNA NANDIVADA, IIT Madras, India

Data-flow analyses like points-to analysis can vastly improve the precision of other analyses, and enable powerful code optimizations. However, whole-program points-to analysis of large Java programs tends to be expensive – both in terms of time and memory. Consequently, many compilers (both static and JIT) and program-analysis tools tend to employ faster – but more conservative – points-to analyses to improve usability. As an alternative to such trading of precision for performance, various techniques have been proposed to perform precise yet expensive fixed-point points-to analyses ahead of time in a static analyzer, store the results, and then transmit them to independent compilation/program-analysis stages that may need them. However, an underlying concern of safety affects all such techniques – can a compiler (or program analysis tool) trust the points-to analysis results generated by another compiler/tool?

In this work, we address this issue of trust in the context of Java, while accounting for the issue of performance. We propose **ART**: Analysis-Results Representation Template – a novel scheme to efficiently and concisely encode results of flow-sensitive, context-insensitive points-to analysis computed by a static analyzer for use in any independent system that may benefit from such a precise points-to analysis. ART also allows for fast regeneration of the encoded sound analysis results in such systems. Our scheme has two components: (i) a producer that can statically perform expensive points-to analysis and encode the same concisely, (ii) a consumer that, on receiving such encoded results (called ARTwork), can regenerate the points-to analysis results encoded by the ARTwork if it is deemed "safe". The regeneration scheme completely avoids fixed-point computations and thus can help consumers like static analyzers and JIT compilers to obtain precise points-to information without paying a prohibitively high cost. We demonstrate the usage of ART by implementing a producer (in Soot) and two consumers (in Soot and the Eclipse OpenJ9 JIT compiler). We have evaluated our implementation over various benchmarks from the DaCapo and SPECjvm2008 suites. Our results demonstrate that using ART, a consumer can obtain precise flow-sensitive, context-insensitive points-to analysis results in less than (average) 1% of the time taken by a static analyzer to perform the same analysis, with the storage overhead of ART representing a small fraction of the program size (average around 4%).

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Just-in-time compilers**; *Dynamic analysis*.

Additional Key Words and Phrases: Java program analysis, staged analysis, points-to analysis, IDFA

**ACM Reference Format:**
Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. 2024. The ART of Sharing Points-to Analysis: Reusing Points-to Analysis Results Safely and Efficiently. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 363 (October 2024), 27 pages. https://doi.org/10.1145/3689803

Authors' Contact Information: Shashin Halalingaiah, IIT Madras, Department of CSE, Chennai, India and University of Texas at Austin, Austin, USA, shashin@cs.utexas.edu; Vijay Sundaresan, IBM Canada Lab, Markham, Canada, vijaysun@ca.ibm.com; Daryl Maier, IBM Canada Lab, Markham, Canada, maier@ca.ibm.com; V. Krishna Nandivada, IIT Madras, Department of CSE, Chennai, India, nvk@iitm.ac.in.


ACM 2475-1421/2024/10-ART363
https://doi.org/10.1145/3689803

## 1 Introduction

One of the salient features of Java is portability. Programs written in Java are compiled once by their respective static compilers to a platform-independent intermediate language (bytecode). Once statically compiled, these programs may then be analyzed by independent program analysis tools, or executed on platform-specific Java Virtual Machines (JVMs) that use just-in-time (JIT) compilers to generate optimized native code specific to the target platform. Such two-step compilation processes bring in unique opportunities and interesting challenges.

For example, the time spent in JIT compilation is considered a part of the execution time of the program as a whole, as the JIT compilation is performed at runtime. Thus, it is essential that the time spent in JIT compilation is not prohibitively high as this could render such JIT compilers unusable in practice. A direct impact of such a restriction is that all the JIT compilers in popular Java virtual machines (like Eclipse OpenJ9 [25], the HotSpot JVM [42], Jikes RVM [2], among others) avoid complex fixed-point-based iterative data-flow analyses (for example, inter-procedural points-to analysis) and instead utilize imprecise analyses in the form of approximations and heuristics. Any optimizations performed by the JIT compilers based on such imprecise points-to analysis results tend to be less effective. In summary, these JIT compilers end up sacrificing analysis precision for execution efficiency.

There have been prior works that make precise points-to analyses results available to the JIT compiler without negatively impacting its performance, by using various staged analysis [1, 46, 47, 53]. In most such techniques, a producer (static compilation) stage bears the cost of expensive points-to analysis and makes the results of the analysis available to the consumer (JIT compiler); this information may even be transmitted over the wire. The consumer, in turn, simply reads the obtained results, and uses them when needed after making any necessary (non-expensive) adaptations. However, owing to the remote and independent nature of the consumer, two important challenges exist:

1. *Ensuring safety* – given that the virtual machine is an independent (and possibly remote) system, how does it ensure that the results of an obtained expensive points-to analysis are safe to use? Note that the analysis results may have been tampered by a malicious producer. Since the requirement of sound analysis results is non-negotiable for JIT compilers, production JVMs cannot "just trust" the static analysis results as they are. Since there is no existing way for such a consumer to trust the analysis results, at this time, it is not common in practice for a production JIT compiler to consume static analysis results from outside. Further, if such a JIT compiler wants to consume externally-sourced static analysis results, then having some sort of inexpensive verification scheme is essential.
2. *Ensuring transmission efficiency* – for a large program (where an expensive points-to analysis has the most benefit), the artifacts of an analysis can potentially be huge. Since any such artifacts have to be transmitted to the JVM along with the bytecode, it effectively increases the size of the executable and is thus considered an overhead that impacts portability. Further, reading large-sized artifacts may impact the performance of the JIT compiler negatively.

Similar to the impact on JIT compilers, the precision of many useful static analyses and optimizations (for example, Function Inlining [12], Stack Allocation [57], Common Sub-expression Elimination [38], and so on) can be directly improved by employing more precise points-to analyses. In a typical static analysis workflow, points-to analysis may need to be run on a program as a pre-pass. Attempting to use a precise fixed-point-based points-to analysis in each of these analyses may result in an unacceptable analysis-time overhead. Inspired by the staged compilation discussion above, one can envisage a scenario where a producer performs an expensive analysis once and stores the results in a persistent store. These results may be read by later consumers (independent

```
1   void foo() {
2       A a = new A();
3       a.f = new F1();
4       A b = new B();
5       b.f = new F2();
6       A c = new C();
7       c.f = new F2();
8       while(*) { // unknown condition.
9           c.f = b.f;
10          b.f = a.f;  } // end loop
11      ...  } // end foo
```

Fig. 1. A synthetic Java program whose points-to analysis necessitates fixed-point computation due to the presence of a loop.

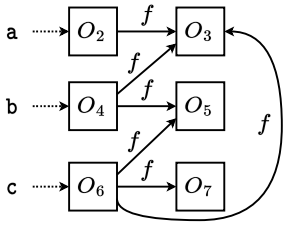



Fig. 2. Points-to graph representing the fixed-point points-to information at Line 8 of the program snippet in Fig. 1.

static analysis/compiler tools) to obtain/use the analysis results efficiently. However, the same concerns of safety and transmission efficiency hold in this scenario as well.

We now use an example to illustrate these challenges by using an optimizing compiler as a consumer. Consider the Java program snippet shown in Fig. 1. Analyzing the code manually, we can see that after the loop body is executed, at Line 11, the field `c.f` may point-to one of the three different objects (of two different types – `F1` or `F2`). Determining this fact precisely requires precise points-to analysis results, which in turn requires expensive fixed-point computations. To avoid such performance overheads in the consumer, say we use staged analysis: we can use a static analyzer to perform expensive points-to analysis (for example, flow-sensitive) and transmit the results to the consumer. Such a flow-sensitive analysis would rightly indicate that at Line 11, `c.f` may point to objects of two types. Say, these analysis results were tampered with to indicate that at Line 11, `c.f` points to a single object. If the consumer uses this tampered information, then it may lead to optimizations or analysis-results that are not semantics-preserving. For example, if there is a call statement `c.f.bar()` after Line 11, then the call may be erroneously inlined by the function inliner of the optimizing compiler.

We can also see the challenges related to transmission efficiency: a naive scheme that supports flow-sensitive points-to analysis, and transmits the flow-sensitive information at each program-point, will lead to potentially high space and time overheads. These issues of safety and transmission efficiency can also be observed in programs with function calls and recursion.

In the context of inter-procedural analysis, considering the scalability (space and time) issues arising in context-sensitive analysis [48], context-insensitive analysis provides a middle ground, compared to a fully intra-procedural analysis. In this article, we propose a scheme to encode context-insensitive points-to analysis results for Java programs to enable inter-stage transmission in a manner that addresses both the issues discussed above. We add flow-sensitivity to maintain increased precision [3, 53] without much performance trade-offs. We term our proposed scheme ART (Analysis-Results Representation Template).

At its core, given a flow-sensitive, context-insensitive points-to analysis result generated by a producer (say, a static analyzer), ART prescribes a small subset of representative information that is enough to encode the results of the whole analysis for efficient transmission to a consumer. We call this encoded information the ARTwork for the given analysis and program. In addition, the information selected for encoding by ART is such that it allows the consumer to not only establish

the safety of the received ARTwork, but also very efficiently regenerate the complete points-to analysis results encoded by this received ARTwork, if found safe.

There have been works that augment code in ways that enable verification of the code's safety during execution. TAL (Typed Assembly Language [37]) is one such example where assembly code is type-checked to certify the code, which can be used in systems where code must be checked (before execution) for untrusted and potentially malicious behavior. Similarly, PCC (Proof-carrying code [40]) is a novel mechanism where a host system can ascertain whether it is safe to execute a program provided by an untrusted source. In PCC, the producer of the untrusted code must supply – along with the code – a safety proof that attests to the code's adherence to a previously agreed upon safety policy. These works have led to further research targeting different languages and domains [4, 13, 45]. Starting from Java 6, the JVMs support StackMapTable [31] attributes, using which the static compiler can communicate type information to JVMs, which can be modularly verified and used to perform type checking of Java applications. Our technique of ART is similar in spirit, wherein ART uses a fundamental property of the underlying (points-to) analysis to establish its safety at the consumer end, but novel in that it applies the safety reasoning to points-to analysis results. To the best of our knowledge there is no prior work that caters to sharing of points-to analysis results in a safe and efficient manner.

The design of ART is based on the underlying properties of fixed-point computations. Given an ARTwork, for an analysis $A$ and program $P$, the consumer makes exactly one pass over the statements of the whole-program and regenerates the complete points-to analysis results at each program-point by using the information present in the ARTwork, thereby avoiding any fixed-point computation. In the process, it establishes the safety of the ARTwork, by checking that the received ARTwork is consistent with the results regenerated by the consumer. We define the notion of safety for any ARTwork and give a guarantee that for a given program $P$ and a points-to analysis $A$, the results regenerated from a safe ARTwork matches the underlying fixed-point results.

To summarize the different approaches discussed above: compared to the performance-centric systems (that avoid expensive fixed-point computation based analyses) and precision-centric systems (that perform expensive fixed-point computations to ensure high precision), the staged analysis supported systems discussed before [1, 46, 47, 53] provide both precision and efficiency, but at the cost of soundness. In contrast, ART helps us obtain highly precise points-to results, without performing any fixed-point computation, while guaranteeing the soundness of the obtained results.

In this paper, we describe ART for flow-sensitive, context-insensitive points-to analysis of Java programs. However, we believe that the discussed concepts can be extended to flow-sensitive, context-sensitive points-to analysis, as well as, any flow-sensitive, context-(in)sensitive iterative data-flow analysis, for any Java-like languages.

**Contributions**

- We propose ART, an efficient scheme for a producer to concisely encode the results of whole program flow-sensitive, context-insensitive points-to analysis, for Java programs; this encoded result is termed ARTwork.
- We propose a scheme that a consumer may employ to efficiently regenerate the originally encoded whole-program points-to analysis results from the given ARTwork, while ensuring safety. Our scheme is accompanied by a correctness argument.
- We demonstrate the usage of ART by instantiating two consumers (one in the Soot framework, and one in the Eclipse OpenJ9 JIT compiler) that share a common producer (implemented in Soot).
- We evaluate our implementation over various benchmarks from the DaCapo and SPECjvm2008 suites. The evaluation shows that ART enables a consumer to obtain and gainfully use previously unattainable precise flow-sensitive points-to analysis results without making the consumer's execution time prohibitively high, while addressing the issues of safety, and transmission efficiency.

## 2 Background

In this section, we provide a brief description of some background material relevant to this paper.

*Recursive call-site.* Consider a call-graph with cycles (due to recursion). A call-site that corresponds to an edge in the cycle of a call-graph, is a recursive call-site. For example, if `main` calls `foo`, `foo` calls `bar`, and `bar` calls `foo`. The call-sites of the latter two calls are considered recursive call-sites.

*Points-to Analysis.* Points-to analysis is a program-analysis technique that can establish which storage locations (or objects) are pointed to by which pointers (or reference variables). We use a points-to graph (similar to Thakur and Nandivada [53]) to represent the points to information. A points-to graph $G(V, E)$ consists of (i) a set $V$ of nodes representing the variables and abstract objects in the program; and (ii) a set $E \subseteq (V \times V) \cup (V \times \texttt{Fields} \times V)$ of edges representing the points-to relationships among the nodes in the program. An edge $(a, O_x)$ from a reference variable $a$ to a node $O_x$ in a points-to graph implies that the variable $a$ may point to the object $O_x$. Similarly, an edge $(O_x, f, O_y)$ from node $O_x$ to $O_y$ with a label $f$ implies that $O_x.f$ may point to $O_y$. Statically, we represent an object on the heap by an abstract-object $O_l$ where $l$ is a label indicating the line number of the program where the object is allocated.

In this paper, we use a pictorial representation of points-to graphs to illustrate points-to information at various program statements. In a points-to graph, we use dotted lines to represent edges between a reference variable and a heap object, and solid lines to represent edges between heap objects. For example, Fig. 2 shows the points-to graph at Line 8 of the program-snippet shown in Fig. 1, after completing the fixed-point points-to analysis.

*Iterative Data Flow Analysis (IDFA).* An iterative data-flow analysis is defined over lattice $\mathcal{L}$, and a set of flow-equations (or, transfer functions) that establishes the relationship between data-flow values. Flow-sensitive context-insensitive points-to analysis is typically encoded as an IDFA, where the goal is to compute the points-to graph after each statement. At each iteration of analysis, the information flowing *in* to a node (called the IN-value of that node) is transformed, by applying the flow-equation of the node, into information flowing *out* of it (called the OUT-value of that node). This evaluation of flow-equations continues in an iterative manner until the points-to information at each node stabilizes (that is, reaches a *fixed-point*). For a program $P$, we say that the result $R$ of a points-to analysis $A$ is the fixed-point result for $A$, if $R$ satisfies all the flow-equations of $A$. Naturally, given an analysis $A$ and a program $P$, there may be many fixed-point results for $P$ that individually satisfy all the flow-equations of $A$. However, $P$ will have a single *least fixed-point result* for $A$. In the context of points-to analysis in this paper, the lattice is the power-set of abstract objects, the $\perp$ element is represented by the set of all the abstract objects, $\top$ is represented by the empty set, and the meet operator is given by the set union operator.

In context-insensitive analysis, the summary of points-to information flowing into a procedure is known as the IN-flow (or IN-summary) of the procedure; and upon completion of analysis of a procedure, the summary of the information flowing out of it back to the call-site is called its OUT-flow (or OUT-summary). At a call-site, computing the information flowing in to a callee varies based on the underlying analysis. In general, it involves taking a projection of the IN-value of the call-site with respect to the portion of heap reachable from the actual arguments of the function call. We encode this process by a macro `project-in`. Similarly, we use a macro `project-out` to propagate the points-to information from the `Exit` node of any procedure back to the call-site. We note that the specific mechanism of doing this does not weigh in on our technique, and so we do not delve into it in this paper.

*Comparison of points-to information.* Given two instances of points-to information $I_1$ and $I_2$ represented by points-to graphs $G_1$ and $G_2$ respectively, we say: that $I_1$ is "equal to" or "matches" $I_2$ if $G_1 = G_2$. Similarly, we say that $I_1$ "subsumes" $I_2$ (represented as $I_1 \succcurlyeq I_2$) if $G_2$ is a subgraph of $G_1$.

## 3 ART: Analysis-Results Representation Template for Java Programs

As discussed in Section 1, since the producer and the consumer of points-to analysis results can be independent systems, the consumer needs a way to check that the obtained analysis results are sound with respect to the program under consideration. In this section, we discuss an encoding scheme called ART to (i) efficiently encode a summary of the points-to analysis computed by a producer, and (ii) quickly regenerate the sound analysis results represented by the encoding (if the encoding is found to be safe) in a consumer. In this scheme, the encoding is efficient in that it avoids sending the complete analysis results, but includes minimal information that can be used by the consumer to regenerate the analysis results (termed as the *efficiency-goal of the producer*). Similarly, at the consumer site, the analysis results are generated quickly in the sense that it does not require any fixed-point computation (called the *efficiency-goal of the consumer*). We term any given instance of ART for a program $P$ and analysis $A$ as the ARTwork for $P$ and $A$.

Before we present our proposed scheme, we first state our basic assumptions about the producer and the consumer: For a given program $P$ and a fixed-point computation based points-to analysis $A$, (i) the consumer needs the results for $A$, but cannot afford to compute it from scratch, (ii) the producer makes available an ARTwork purported to be that for $P$ and $A$, and (iii) the consumer can access that ARTwork. In Section 5, we discuss a relaxation of this assumption, where the producer and consumer may not refer to the same points-to analysis. Further, the consumer expects the following two soundness and completeness guarantees: (i) unsound analysis results will never be marked as sound, and (ii) sound analysis results will always be marked as sound.

For ease of exposition, we will first discuss ART in the context of intra-procedural flow-sensitive points-to analysis and then extend it to handle inter-procedural points-to analysis (in Section 3.4). In this paper, we will use a notion of safety given by the following definition.

*Definition 3.1.* Given an intra (inter) procedural iterative-data-flow points-to analysis $A$, and a program $P$, we say that an intra (inter) procedural ARTwork is *safe* for $P$, with respect to $A$, if it encodes a sound intra (inter) procedural points-to analysis result of $P$ satisfying the transfer functions of $A$.

### 3.1 Design of Intra-procedural ART

During intra-procedural flow-sensitive points-to analysis, it is well understood that computing the points-to information for statements inside loops involve fixed-point computation and hence is more expensive (compared to statements outside the loops). On the other hand, for a statement that is not inside a loop, its OUT can be computed, without needing any fixed-point computation, if we have the OUT of its topologically sorted predecessors. We use this understanding to design ART.

The intuition behind the design of ART is that it should only carry information that would otherwise be expensive to compute, and the consumer must be able to use this information to regenerate the encoded analysis. To understand the minimal information that needs to be included in ART, consider the results realized at the end of a flow-sensitive points-to analysis for a procedure. These results include the OUT for each statement. Say we have the control-flow graph (CFG) [38] of the procedure and the OUT of the `Entry` node of the procedure is also given. The first instruction of a basic-block is termed a leader [38]. Consider a basic-block, whose leader is a loop-header. We can avoid storing the point-to information of all the statements in the basic-block and recompute them without needing any fixed-point computations, if we have the fixed-point OUT of the loop-header. We term such a basic-block whose leader is an `Entry` node or a loop-header as a *key* basic-block. For the non-key basic-blocks the IN points-to information of their leaders can be computed simply by taking the meet of the OUTs of their respective predecessors. This IN information can be used

to compute the OUT values of the leaders of these non-key basic-blocks; thus the OUT values for such leaders or the statements in their corresponding basic-blocks need not be stored.

To summarize, the OUT information of the leaders of the key basic-blocks can be used to generate the points-to information for all the statements, without needing any fixed-point computation (the exact scheme of regeneration will be discussed in Section 3.2).

Since OUT of the `Entry` node is equal to its IN, which in intra-procedural analysis is initialized to $\perp$ (see Section 2) it need not be stored. In Section 3.4, we will revisit this point when we discuss how ART deals with inter-procedural analysis.

We use the discussion above to identify the single type of points-to information that ART needs to carry in order to encode intra-procedural points-to analysis:

• **Loop-Invariants:** A loop-invariant encodes the fixed-point OUT of a loop-header. We denote the collection of all such loop-invariants using a map $\mathcal{I}_{loop} : L \rightarrow G$, where $L$ is the set of labels of all the loop-headers and $G$ is the set of all possible OUTs.

**Example.** For the program snippet shown in Fig. 1, ART would need to encode only the OUT values corresponding to the loop-header $s_8$. For each program $P$, the list of such encodings (program-points and the corresponding OUT values) is called an instance of ART, represented as $ART\langle P\rangle$. We use $ART\langle P\rangle.\mathcal{I}_{loop}$ to refer to the loop-invariant map of $ART\langle P\rangle$. The contents of $ART\langle P\rangle.\mathcal{I}_{loop}[s_8]$ for the program snippet in Fig. 1 are shown in Fig. 2.

We would like to highlight that while the traditional data-flow analyses [38] generate the most precise fixed-point result, in general, there can be many fixed-point results (and hence loop-invariants) for a given loop. And each of them can be a valid instance of ART encoding for that loop. Thus a program can have many valid instances of ART, each of which encodes/corresponds to a sound point-to analysis result for that program.

### 3.2 Regeneration Of Sound Intra-procedural Analysis

We will now present a technique to regenerate the sound intra-procedural points-to analysis for a program $P$, using the given instance $ART\langle P\rangle$, such that the generated points-to results match the points-to results encoded by $ART\langle P\rangle$. For ease of understanding, in this section, we assume that the $ART\langle P\rangle$ is safe (Definition 3.1). In Section 3.3, we discuss how to verify the safety of the given $ART\langle P\rangle$.

Fig. 3 presents the algorithm for regeneration of sound intra-procedural points-to analysis for a procedure $M$ in $P$, given $ART\langle P\rangle$; we use $ART\langle P, M\rangle$ to represent the information encoded in $ART\langle P\rangle$ for the program points within $M$. Our regeneration scheme follows the steps that the producer would have followed to generate the encoded points-to information, while making sure that we do not incur any fixed-point computations.

Our scheme ensures that the consumer analyzes each statement of $M$ exactly once. For an iterative data-flow analysis like points-to analysis, a statement needs to be reanalyzed only when its IN-value changes during the course of the analysis (which happens only if the OUT of at least one of its predecessors changes). We avoid the reanalysis by ensuring that the fixed-point OUTs of all the predecessors of each statement are obtained before the statement is analyzed. This required ordering can be ensured by visiting each basic-block and each statement within the basic-block in order by ignoring the back-edges (Loops starting at Lines 3 and 5, Fig. 3). As discussed in Section 3.1 for the Entry node of $M$, the OUT-value is set to $\perp$. The fixed-point OUTs of the leaders of the key basic-blocks are obtained from ART (Line 7). For each such statement $s$, $\text{OUT}[s]$ can be computed given the OUTs of all its predecessors and the relevant transfer function $f_s$ (Line 8). Note that since there is an agreement between the producer and the consumer on the specific points-to analysis to use, both of them use the same transfer function $f_s$, for each statement $s$. The method

```
1  Function regenIntra(M, ART⟨P, M⟩)
2      List B ← topologically sorted list of basic-blocks of M
3      foreach B ∈ B do
4          List S ← statements of B in program order
5          foreach s ∈ S do
6              if s is the Entry node of M then OUT[s] ← IN[s]; // = ⊥
7              else if s is leader of key basic-block then OUT[s] ← ART⟨P, M⟩.I_loop[s];
8              else OUT[s] ← f_s( ⨅_{p∈preds(s)} OUT[p]);
9              checkForIntraSafety(s, M, OUT)
10     return OUT
```

Fig. 3. Regeneration of intra-procedural points-to analysis using ART

call `checkForIntraSafety` will be used to check the safety of the obtained ART and is discussed in Section 3.3.

**Complexity.** In our proposed scheme, we process each statement exactly once, that is $O(N)$, where $N$ is size of the program. If the cost of any transfer function is $O(g(N))$, then the complexity of our regeneration scheme is $O(N \times g(N))$. In contrast, a typical flow-sensitive, context-insensitive points-to analysis may process the statements up to $O(N^3)$ times and incurs a cost of $O(N^3 \times g(N))$.

**Example.** Consider the snippet shown in Fig. 1. We use $s_i$ to refer to the CFG node for line $i$ and $f_i$ to denote the transfer function used by the producer to perform flow-sensitive intra-procedural points-to analysis, for $s_i$. The transfer functions used by the producer can be classified into three categories based on the type of the node: (i) the `Entry` node: OUT[`Entry`] = $\perp$. (ii) a node $s_i$ with a single predecessor $p_i$: OUT$[s_i]$ = $f_i$(OUT$[p_i]$). (iii) a node $s_i$ with multiple predecessors $\mathcal{P}$: OUT$[s_i] = f_i(\sqcap_{p \in \mathcal{P}}$OUT$[p])$. For example, for the snippet in Fig. 1, we use the constraint, OUT$[s_8] \leftarrow f_8$(OUT$[s_7] \sqcap$ OUT$[s_{10}]$). Note that the constraints to generate OUTs for the statements at Lines 8-10 require a fixed-point computation to obtain a solution.

Once the OUTs for all the statements have been computed, in our suggested scheme the producer will emit OUT$[s_8]$ as a loop-invariant in $ART\langle P,$ `foo`$\rangle$, as shown in Fig. 2. The consumer invokes `regenIntra` with the procedure `foo` and $ART\langle P,$ `foo`$\rangle$ as inputs.

The consumer first initializes OUT[`entry`] to $\perp$. Except for OUT$[s_8]$, the constraints to compute the remaining OUTs are exactly the same as that used by the producer. For $s_8$, the consumer simply uses the constraint: OUT$[s_8] \leftarrow f_8(\mathcal{I}_{loop}[s_8])$. It can be seen that none of the constraints used by the consumer have cyclic dependencies and hence need no further fixed-point computation.

Thus, with the assistance of ART, the consumer has regenerated intra-procedural points-to analysis results for the whole procedure without needing any fixed-point computations; this matches the results generated by the producer.

### 3.3 Safety of Intra-procedural ART

The process of regeneration presented in Section 3.2 assumed that $ART\langle P, M\rangle$ obtained by the consumer is safe. We now will discuss the scenarios where $ART\langle P, M\rangle$ may have been *tampered* with (by the producer or a malicious middle agent) and how we can establish the safety of $ART\langle P, M\rangle$.

Without loss of generality, let us assume that $ART\langle P\rangle$ is shared via a file. Any tampering of the file that breaks the expected syntactic structure is easily caught. Hereafter, we only focus on that type of tampering which still encodes valid points-to information, but not the least fixed-point points-to analysis information for $P$. Thus, we can identify two ways in which $ART\langle P, M\rangle$ can be

```
1 Function checkForIntraSafety(s, M, OUT)
2     foreach s′ ∈ succs(s) such that (s, s′) is a CFG back-edge do
          // s′ is a key-basic-block leader and s′ has already been analyzed
3         prevOut = OUT[s′] // was earlier set to ART⟨P, M⟩.I_loop[s′]
4         newOut = f_s′( ⨅_{p∈preds(s′)} OUT[p])
5         assert prevOut = newOut
```

Fig. 4. Checking Safety of Intra-procedural ART

tampered with: non-conservative and conservative, as described below. Let $\alpha = ART\langle P\rangle$ be the encoding of the least fixed-point points-to analysis results for $P$; say the points-to results were obtained using an analysis $A$.

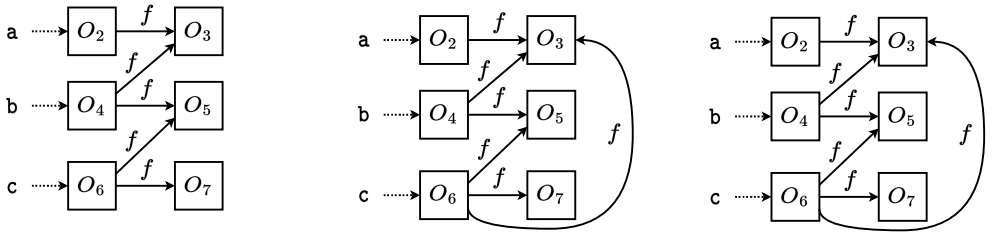
• **non-conservative tampering.** Let $\alpha'$ be a modified version of $\alpha$ such that $\alpha'$ does not encode any fixed-point result for $A$. Then $\alpha'$ no longer encodes any sound intra-procedural fixed-point analysis result for $P$. In such cases, we say that $\alpha$ has been non-conservatively tampered with, to yield $\alpha'$. A consumer using such a tampered instance of ART in the regeneration process will obtain points-to analysis results that do not match any fixed-point result for $P$. Thus, by Definition 3.1, any non-conservatively tampered ART is unsafe. An example of non-conservative tampering would be the removal of a node or an edge in a points-to graph that represents the least fixed-point value.

• **conservative tampering.** Any other form of tampering is considered conservative tampering. A consumer using such a tampered instance of ART in the regeneration process will obtain points-to analysis results that is more conservative than the least fixed-point solution, but still sound (that is, an over-approximation).

We first provide a sketch of our proposed scheme to identify if, for a given procedure $M$ of a program $P$, $ART\langle P, M\rangle$ given to a consumer (to regenerate the points-to analysis information) is safe. Our scheme leverages a fundamental property of fixed-point computations in data-flow analysis. Consider a loop in the method $M$ of program $P$, where the point-to analysis algorithm of the producer reached a fixed-point, after processing the loop body $i$ number of times. Processing the loop-body $i + k$ ($k \geq 1$) times will not alter the computed points-to information for any statement in that loop.

We apply a similar reasoning for establishing the safety of $ART\langle P, M\rangle$. Say, $ART\langle P, M\rangle.\mathcal{I}_{loop}[s]$ is the OUT value of a loop-header labeled $s$ encoded in $ART\langle P, M\rangle$. Note that $ART\langle P, M\rangle.\mathcal{I}_{loop}[s]$ is supposed to be the fixed-point value of OUT$[s]$ in the producer. Thus, in the consumer, by initializing OUT$[s]$ to $ART\langle P, M\rangle.\mathcal{I}_{loop}[s]$, processing each statement of the loop-body once, and eventually re-computing OUT$[s]$ after considering all the predecessors of $s$ (including the ones via the back-edges) should not change the value of OUT$[s]$.

In Fig. 4, we use the above intuition to define the `checkForIntraSafety()` method used in Fig. 3. We process each control-flow successor $s'$, of $s$, such that $(s, s')$ is a back-edge. Based on the intuition presented above, we now re-compute OUT of $s'$ considering all the predecessors of $s'$ (which includes the newly computed OUT$[s]$) and compare it with the previously computed OUT of $s'$ (*prevOut*). Given that *prevOut* was initially set to $ART\langle P, M\rangle.\mathcal{I}_{loop}[s']$ (which is supposed to be a fixed-point OUT value), it should not change after this re-computation. We assert the same in Line 5. If this assertion fails, it means that $ART\langle P, M\rangle.\mathcal{I}_{loop}[s']$ was not a fixed-point value. This implies that $ART\langle P, M\rangle$ is not safe and does not encode a sound points-to analysis for procedure $M$, in program $P$.




(a) OUT[$s_8$]← $ART\langle P, M\rangle.\mathcal{I}_{loop}[s_8]$ (tampered by removing $(O_6, O_3)$). (b) Generated OUT[$s_9$], OUT[$s_{10}$]. (c) Value of *newOut*.

Fig. 5. Values computed by `regenIntra` for the loop-header and the loop-body of Fig. 1.

The discussion of the algorithm in Fig. 3 so far assumed that for any loop-header $s$, in a procedure $M$ of program $P$, the $\mathcal{I}_{loop}[s]$ entry exists. But in case the tampering involves the deletion of this entry, then the consumer would not find such an entry in the ARTwork. In such a scenario, the consumer infers a default value for $ART\langle P, M\rangle.\mathcal{I}_{loop}[s]$ (= IN[$s$]), at Line 7; not explicitly shown. Thus, if the actual fixed-point OUT[$s$] differs from IN[$s$] then our above-discussed procedure will detect the tampering. And in case they do not differ, then it implies that despite the omission the ARTwork is safe to use; in Section 4, we will use this observation to optimize the encoding of ART.

**Example.** Let the ARTwork in Fig. 2 be non-conservatively tampered by removing the edge $(O_6, O_3)$. Fig. 5 shows the steps taken by `regenIntra` (Fig. 3) for the loop-header and the loop-body in Fig. 1: The algorithm sets OUT[$s_8$] to the tampered ART (Fig. 5a). Fig. 5b shows the OUT computed for statement $s_9$ (and $s_{10}$). After computing OUT[$s_{10}$], `checkForIntraSafety()` (Fig. 4) processes the back-edge $(s_{10}, s_8)$ and finds that the value of *newOut* (Fig. 5c) does not match *prevOut* (Fig. 5a). This leads to the assertion failure at Line 5, and establishes that the given ARTwork has been non-conservatively tampered with and is thus unsafe to use.

Note: It is simply a coincidence in the above example that *newOut* matches the non-tampered ARTwork (Fig. 2). In general, this may not hold. For example, say the tampering removed $(O_4, O_3)$ and $(O_6, O_5)$ edges in addition to $(O_6, O_3)$, then *newOut* will contain $(O_4, O_3)$ and $(O_6, O_5)$, but not $(O_6, O_3)$.

### 3.4 Design of Inter-procedural ART

We will now extend the intra-procedural ART introduced in Section 3.1 to handle inter-procedural (context-insensitive) points-to analysis. As discussed in Section 2, during context-insensitive points-to analysis, a procedure $M$ may be analyzed more than once till it reaches fixed-point. Recall that our efficiency-goals (Section 3) require that we process each statement exactly once during the regeneration of the points-to results by the consumer. By extension, this also requires that we analyze any given method exactly once, while ensuring that the computed/generated point-to information at each program-point in that method is valid across all the calls to that method. Hereafter, we use points-to IN/OUT information to indicate the context-insensitive points-to IN/OUT information.

We first present the intuition behind our proposed approach: Assuming that there is no recursion, using the intra-procedural ART scheme discussed before, given the IN-summary of a procedure $M$, we can regenerate the OUT information for each statement in $M$ (without needing any fixed-point computation), provided we have the OUT information for each call-site in $M$. Thus, if we have the IN-summary of every method in the program and we process the methods of the program, in the bottom-up order of the call-graph (leaves first), then we can meet our efficiency-goal of the consumer: for each method $M$, the OUT information at each program-point in $M$ (along with

```
1  void main() {
2    ...
3    A a = new A();
4    a.f = new A();
5    a.f.f = new A();
6    A b = a.f;
7    foo(b, a);  ... }
8  void foo(A q, A p) {
9    if(p != null) {
10     A r = p.f;
11     if(*) { p.f = null; }
12     else { p.f = q; return; }
13     foo(q, r);
14     .../*no effect on heap*/ } }
```

Fig. 6. A synthetic Java program snippet whose points-to analysis necessitates fixed-point computation due to the presence of function calls and recursion.

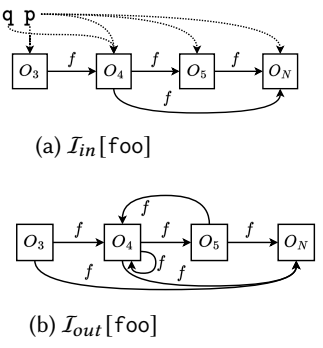



Fig. 7. Inter-procedural ART for the program snippet shown in Fig. 6. Here $O_N$ is the abstract object representing the null object.

the OUT summary of $M$) can be computed by processing the statements in $M$ exactly once. But considering the fact that computing the IN-summary for any method $M$ in a context-insensitive analysis involves fixed-point computation, we propose to encode the IN-summary of each non-recursive method in ART.

The above discussed scheme to efficiently compute the OUT for each statement of a function given its IN-summary, encounters a challenge in the case of recursive functions. This is because the OUT of the recursive call-sites (see Section 2) in the recursive methods won't be available even if we process the call-graph nodes in the bottom-up order. This challenge can be addressed if the context-insensitive OUT-summary of each recursive method can be stored (along with the context-insensitive IN-summary) and made available to the consumer to compute the OUT of the corresponding recursive call-site.

We use these two intuitions (for recursive and non-recursive procedures) to propose the addition of the following two types of points-to information to ART, to support context-insensitive inter-procedural points-to analysis of any program $P$.

• **$M_{in}$-Invariants**: An $M_{in}$-Invariant encodes the context-insensitive IN-summary of a procedure. We denote the collection of all such in-invariants using a map $\mathcal{I}_{in} : \mathcal{M} \rightarrow G$, where $\mathcal{M}$ is the set of all the procedures in $P$ and $G$ is the set of all possible context-insensitive IN-summaries.

• **$M_{out}$-Invariants**: An $M_{out}$-Invariant encodes the context-insensitive fixed-point OUT-summary of a recursive procedure. We denote the collection of all such out-invariants using a map $\mathcal{I}_{out} : \mathcal{M}_{rec} \rightarrow G$, where $\mathcal{M}_{rec}$ is the set of all recursive procedures (direct and indirect) and G is the set of all possible context-insensitive OUT-summaries.

Note that even though both the IN and OUT-summaries of the recursive procedures are present in the ART of any program $P$, a consumer cannot skip processing these recursive procedures during the regeneration process, since it still needs to (i) compute the OUT for each statement in those methods, and (ii) verify the given IN and OUT-summaries.

**Summary of the design of ART.** In order to support flow-sensitive, context-insensitive inter-procedural points-to analysis, ART needs to carry three types of points-to information: (a) loop-invariants, (b) in-invariants, (c) out-invariants

```
1  Function regenInter(M, ART⟨P⟩)
2      List B ← topologically sorted list of basic-blocks of M
3      foreach B ∈ B do
4          List S ← statements of B in program order
5          foreach s ∈ S do
6              if s is the Entry node of M then OUT[s] ← IN[Entry]; // = ART⟨P⟩.I_in[M]
7              else if s is leader of key basic-block then OUT[s] ← ART⟨P⟩.ART⟨P, M⟩.I_loop[s];
8              else if s is a call-site then
9                  List T ← targets resolved for s
10                 foreach t ∈ T do
11                     checkForInSafety(s, t, ART⟨P⟩, OUT)
12                     if t has already been analyzed then continue
13                     if s is a non-recursive invocation of t then
14                         OUT[t] ← regenInter(t, ART⟨P⟩)
15                     else // read from ART to avoid fixed-point computation.
16                         OUT[t] ← ART⟨P⟩.I_out[t]
17                 OUT[s] ← ⨅_{t∈T} OUT[t] // As per the underlying points-to-analysis; see Section 2
18             else
19                 OUT[s] ← f_s( ⨅_{p∈preds(s)} OUT[p])
20                 checkForIntraSafety(s, M, OUT)
21     if M is a recursive method then checkForOutSafety(M, OUT);
22     return OUT
```

Fig. 8. Regeneration of inter-procedural points-to analysis using ART

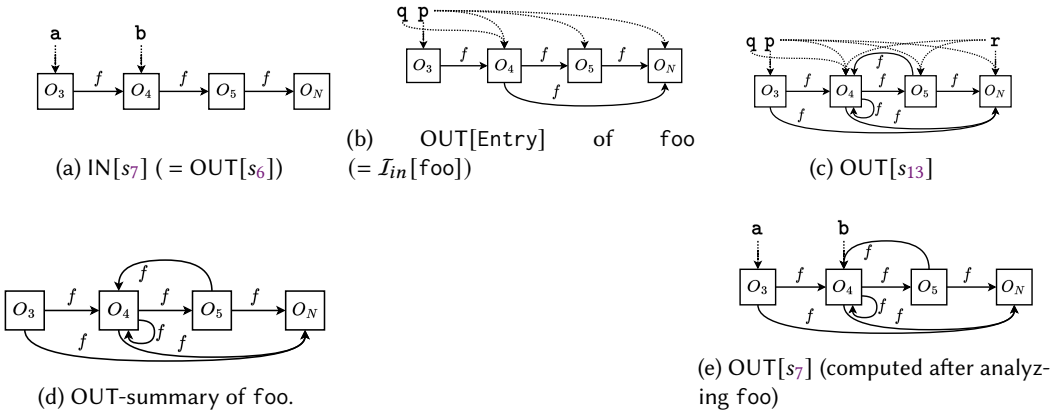
**Example.** Fig. 6 shows a program snippet with function calls and recursion. The contents of $ART\langle P\rangle$ for this program snippet are shown in Fig. 7. For this program, inter-procedural ART will encode the context-insensitive IN-summary of `foo`. In addition, since `foo` is recursive, it will also encode the context-insensitive fixed-point OUT-summary of `foo`.

### 3.5 Regeneration Of Sound Inter-procedural Analysis

We will now extend the analysis regeneration scheme discussed in Section 3.2, to regenerate the sound inter-procedural points-to analysis for a program $P$, using the given instance $ART\langle P\rangle$, such that the generated points-to results match the inter-procedural points-to analysis encoded by $ART\langle P\rangle$. For ease of understanding – similar to the discussion of intra-procedural regeneration in Section 3.2, we assume that $ART\langle P\rangle$ is safe (Definition 3.1). In Section 3.6, we discuss how to verify the safety of the given inter-procedural $ART\langle P\rangle$.

Fig. 8 presents the algorithm for regeneration of sound context-insensitive inter-procedural points-to analysis for the whole program $P$ starting from an entry procedure $M$, given $ART\langle P\rangle$. The presented algorithm builds upon the intra-procedural regeneration algorithm presented in Fig. 3 by addressing regeneration in the presence of method calls. We now discuss some of the main differences between the inter-procedural regeneration scheme and the intra-procedural one.

*Handling the* `Entry` *nodes.* Since we are now regenerating inter-procedural points-to analysis, IN of the `Entry` node will not be $\perp$; it is instead initialized with $ART\langle P\rangle.\mathcal{I}_{in}[M]$ (Line 6).




(a) IN[$s_7$] ( = OUT[$s_6$])

(b) OUT[Entry] of foo (= $\mathcal{I}_{in}$[foo])

(c) OUT[$s_{13}$]

(d) OUT-summary of foo.

(e) OUT[$s_7$] (computed after analyzing foo)

Fig. 9. Progression of regenInter for program snippet in Fig. 6.

*Handling the call-sites.* If the statement $s$ is a call-site, we resolve the targets of the invocation (Line 9). For each such target $t$, if $t$ has already been analyzed, we do not re-analyze it. Otherwise, we need to obtain its OUT-summary so we can compute the OUT information for $s$. If $s$ is a non-recursive call-site, we recursively invoke regenInter on $t$ to analyze it and obtain the OUT-summary. On the other hand, if $s$ is a recursive call-site, computing the OUT-value for $s$ will involve a fixed-point computation. The consumer avoids this by using the information encoded in $ART\langle P\rangle$, assuming it is safe (Line 16). After we have obtained the OUT for each target, we compute OUT[$s$] by taking the meet of the OUT-summary of each of the targets (Line 17). For the rest of the statements, regenInter follows the same steps as regenIntra. The method calls checkForInSafety and checkForOutSafety will be used to check the safety of the obtained inter-procedural ARTwork and are discussed in Section 3.6.

**Complexity.** The complexity of regeneration of sound inter-procedural analysis is exactly the same as that of the intra-procedural analysis (Section 3.2).

**Example.** For the program snippet shown in Fig. 6, given the ARTwork shown in Fig. 7, in Fig. 9 we show some of the important steps taken by regenInter to regenerate the points-to analysis results; we mainly focus on how we compute the points-to information in the presence of function calls. The regeneration process begins at the main procedure, and processes each statement. Fig. 9a shows the value of IN[$s_7$](= OUT[$s_6$]). To compute OUT[$s_7$], regenInter needs to process the function foo and obtain its OUT-summary. To do so, regenInter first sets the OUT of the Entry node of foo to $\mathcal{I}_{in}$[foo] (Fig. 9b), obtained from the given ART instance (Fig. 7a) and then processes the statements of foo. At statement $s_{13}$, which is a recursive call, regenInter uses $\mathcal{I}_{out}$[foo] to obtain the OUT-summary of foo and uses it to compute OUT[$s_{13}$] (shown in Fig. 9c). On reaching the Exit node of foo, regenInter will return the OUT-summary of foo (Fig. 9d) and use it to compute the OUT[$s_7$] (Fig. 9e) in the main function.

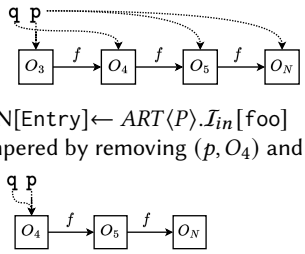
### 3.6 Safety of Inter-procedural ART

The process of regeneration presented in Section 3.5 assumed that $ART\langle P\rangle$ obtained by the consumer is safe. We will now discuss the scenarios where $ART\langle P\rangle$ may have been *tampered* with and how we can establish the safety of $ART\langle P\rangle$. Since the two additional points-to information carried by inter-procedural ART ($M_{in}$-Invariants and $M_{out}$-Invariants) are fixed-point values, the discussion

```
1 Function checkForInSafety(s, T, ART⟨P⟩, OUT)
    // at call-site s, check that ART⟨P⟩ subsumes the IN
    flow to the target callee T
2   IN[s] ← ⨅_{p∈preds(s)} OUT[p]
3   in_T ← project-in(IN[s], T)
4   assert ART⟨P⟩.I_in[T] ⩾ in_T

5 Function checkForOutSafety(M, OUT)
6   assert ART⟨P⟩.I_out[M] = OUT[M]
```

Fig. 10. Checking Safety of Inter-procedural ART




(a) IN[Entry]← $ART\langle P\rangle.\mathcal{I}_{in}$[foo] (tampered by removing $(p, O_4)$ and $(O_4, O_N)$).

(b) computed $in_{\texttt{foo}}$ for the call at $s_{13}$

Fig. 11. Safety verification of $M_{in}$-Invariants for the procedure foo, shown in Fig. 6.

on conservative and non-conservative tampering from Section 3.3 also holds here. To recall, any non-conservatively tampered ARTwork is unsafe.

We note that of the two types of points-to information carried in inter-procedural ART, one of them ($M_{in}$-Invariant) is an IN-value and the other ($M_{out}$-Invariant) is an OUT-value. As a result, the technique for establishing the safety of each of them has subtle differences – while still using the property of fixed-point values discussed in Section 3.3. We will now discuss our techniques for establishing the safety of inter-procedural ART.

**Establishing Safety of $M_{in}$-Invariants**. We start with an intuition. Recall from Section 3.4 that an $M_{in}$-Invariant encodes the context-insensitive IN-summary of a procedure $M$. This means that for a procedure $M$, $\mathcal{I}_{in}[M]$ reflects (*subsumes*) the IN-summary of $M$ at each and every one of its call-sites. As a result, processing $M$ by setting the IN of $M$'s Entry node to $\mathcal{I}_{in}[M]$ will result in the regeneration of points-to information that is valid at each and every one of $M$'s call-sites. In contrast, consider a non-conservatively tampered (see Section 3.3) $ART\langle P\rangle$, where the tampering is performed with respect to the $\mathcal{I}_{in}[M]$ entry. In such a case, using $ART\langle P\rangle.\mathcal{I}_{in}[M]$ (at Line 6, Fig. 8) will result in the generation of points-to information (of $M$) that does not correspond to the fixed-point context-insensitive points-to information of $M$. To establish the safety of $ART\langle P\rangle.\mathcal{I}_{in}[M]$, we make use of the property of fixed-point values discussed in Section 3.3. In the producer, if $ART\langle P\rangle.\mathcal{I}_{in}[M]$ reached a fixed-point after analyzing $M$ at all of its call-sites, then processing *any* of those call-sites again in the consumer should not result in any change to the IN-summary of $M$. We use this intuition to define the checkForInSafety method shown in Fig. 10. On encountering a call-site $s$ invoking a target procedure $T$, checkForInSafety first computes $in_T$ by invoking project-in (see Section 2) on IN[$s$] (Line 3). It then asserts that the information carried in $ART\langle P\rangle.\mathcal{I}_{in}[M]$ *subsumes* $in_M$ (Line 4). This subsumption check is natural (in contrast to an equals check) because $ART\langle P\rangle.\mathcal{I}_{in}[M]$ is supposed to be the meet of the incoming relevant points-to information from all the call-sites (not just the call-site $s$). If the assertion succeeds, it means that processing $M$ by using $ART\langle P\rangle.\mathcal{I}_{in}[M]$ in regenInter will result in the regeneration of points-to information that is valid at call-site $s$. If the assertion fails, it means that $ART\langle P\rangle.\mathcal{I}_{in}[M]$ does not reflect the context-insensitive IN-summary of $M$.

The discussion so far assumed that for any procedure $M$ of program $P$, the $\mathcal{I}_{in}[M]$ entry exists. But in case the tampering involves the deletion of this entry, then the consumer would not find such an entry in the ARTwork. In such a scenario, on encountering a call-site $s$ invoking $M$, regenInter sets a default value for $ART\langle P\rangle.\mathcal{I}_{in}M$ (= project-in(IN[$s$], $M$)), before the assertion at Line 4, in

Fig. 10; not explicitly shown. Thus, in case the fixed-point context-insensitive OUT-summary of $M$ differs from this default value then our above discussed procedure will detect the tampering. And in case they do not differ, then it implies that despite the omission, the ART instance is safe to use; in Section 4, we will use this observation to optimize the encoding of ART.

**Example.** Consider the ARTwork shown in Fig. 7 for the program $P$ shown in Fig. 6. As an instance of non-conservative tampering, Fig. 11a shows $ART\langle P\rangle.\mathcal{I}_{in}[\texttt{foo}]$ with edges $(p, O_4)$ and $(O_4, O_N)$ removed from Fig. 7a. When `regenInter` uses such a tampered ART to analyze `foo` and reaches the call-site $s_{13}$, it computes $in_{\texttt{foo}}$ at Line 3 of `checkForInSafety` as shown in Fig. 11b; we rediscover the previously removed edge $(p, O_4)$. This causes the assertion on Line 4 to fail, since $ART\langle P\rangle.\mathcal{I}_{in}[\texttt{foo}]$ does not encode this information. In other words, the tampered $ART\langle P\rangle.\mathcal{I}_{in}[\texttt{foo}]$ does not reflect the IN-summary of `foo`, at this call-site, hence unsafe to use.

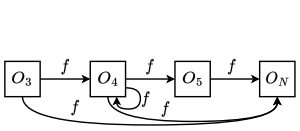
**Establishing Safety of $\mathbf{M_{out}}$-Invariants**. We start with an intuition. Recall from Section 3.4 that an $\mathbf{M_{out}}$-Invariant encodes the fixed-point context-insensitive OUT-summary of a recursive procedure $M_{rec}$. This means that for a recursive procedure $M_{rec}$, $\mathcal{I}_{out}[M_{rec}]$ can be used to compute sound fixed-point OUT values of *any* of its call-sites. In contrast, consider a non-conservatively tampered (see Section 3.3) $ART\langle P\rangle$, where the tampering is performed with respect to the $\mathcal{I}_{out}[M_{rec}]$ entry. In such a case, using $ART\langle P\rangle.\mathcal{I}_{out}[M_{rec}]$ (at Line 16, Fig. 8) to compute the OUT value of a recursive call-site $s$ (in a procedure $M$) will result in the generation of points-to information that does not correspond to the fixed-point context-insensitive points-to information of $M$. The technique to establish the safety of $ART\langle P\rangle.\mathcal{I}_{out}[M_{rec}]$ also makes use of the property of fixed-point values discussed in Section 3.3. Thus, if $ART\langle P\rangle.\mathcal{I}_{out}[M_{rec}]$ represents the fixed-point OUT-summary of $M_{rec}$ obtained after analyzing $M_{rec}$ using its context-insensitive IN-summary in the producer, then processing $M_{rec}$ using the same IN-summary ($ART\langle P\rangle.\mathcal{I}_{in}[M_{rec}]$) again in the consumer should not change the OUT-summary of $M_{rec}$. We use this intuition to define the `checkForOutSafety` method shown in Fig. 10.

For each recursive procedure $t$, when the OUT-summary is regenerated by `regenInter` (when the `Exit` node of $t$ is analyzed by `regenInter`), `checkForOutSafety` asserts that the regenerated OUT-summary of $t$ matches $ART\langle P\rangle.\mathcal{I}_{out}[t]$ (Line 6). This assertion is important as for any recursive procedure $t$, a recursive call-site (see Section 2) invoking $t$ must have been processed by `regenInter` before processing the `Exit` node of $t$, and `regenInter` must have used $ART\langle P\rangle.\mathcal{I}_{out}[t]$ as the required OUT-summary of $t$ (at Line 16, Fig. 8), at that call-site.

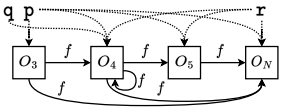
If the assertion succeeds, it means that the OUT-summary of $t$ regenerated by `regenInter` matches the fixed-point context-insensitive OUT-summary of $t$. Further, this OUT-summary of the procedure $t$ can be used to compute the fixed-point OUT points-to information at all of its call-sites. If the assertion fails, it means that $ART\langle P\rangle.\mathcal{I}_{out}[t]$ does not reflect the context-insensitive OUT-summary of $t$.

Similar to the discussion about tampering via deletion of an ART entry for the IN-summary, consider the case where the OUT-summary of a recursive procedure $M_{rec}$ has been deleted as part of tampering. In such a scenario, on encountering a recursive call-site $s$ invoking $M_{rec}$, `regenInter` sets a default value for $ART\langle P\rangle.\mathcal{I}_{out}M_{rec}$ (= $ART\langle P\rangle.\mathcal{I}_{in}M_{rec}$), at Line 16 in Fig. 8; not explicitly shown. Thus, in case the fixed-point context-insensitive IN-summary of $M$ differs from this default value then our above discussed procedure will detect the tampering. And in case they do not differ, then it implies that despite the tampering, the ART instance is safe to use; in Section 4, we will use this observation to optimize the encoding of ART.

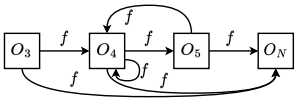
**Example.** Consider the ART instance shown in Fig. 7 for the program $P$ shown in Fig. 6. As an instance of non-conservative tampering, Fig. 12a shows $ART\langle P\rangle.\mathcal{I}_{out}[\texttt{foo}]$ with the edge $(O_5, O_4)$ removed from Fig. 7b. When `regenInter` uses such a tampered ART at the call-site $s_{13}$ (that recursively invokes `foo`), the computed OUT$[s_{13}]$ is shown in Fig. 12b. When the `Exit` node of




(a) $ART\langle P\rangle.\mathcal{I}_{out}$[foo] tampered by removing edge $(O_5, O_4)$




(b) OUT[$s_{13}$] obtained with tampered $ART\langle P\rangle.\mathcal{I}_{out}$[foo]




(c) OUT-summary of foo computed by regenInter ($\neq ART\langle P\rangle.\mathcal{I}_{out}$[foo])

Fig. 12. Safety Verification of $M_{out}$-Invariants for program in Fig. 6.

foo is eventually processed by regenInter and the OUT-summary of foo is computed (shown in Fig. 12c), we observe that the edge $(O_5, O_4)$ is rediscovered. This causes the assertion at Line 6 (Fig. 10) to fail, since $ART\langle P\rangle.\mathcal{I}_{out}$[foo] does not encode this information. In other words, the tampered $ART\langle P\rangle.\mathcal{I}_{out}$[foo] does not reflect the fixed-point context-insensitive OUT-summary of foo, and is hence unsafe to use. It is interesting to note that the edge $(O_5, O_4)$ is missing in OUT[$s_{13}$], which means that it does not correspond to fixed-point points-to information at $s_{13}$.

### 3.7 Correctness

We now present a correctness argument of ART in the following context: the consumer has been given an ARTwork, from which the consumer has to either regenerate sound points-to analysis results, or report a safety violation. The consumer uses the algorithm in Fig. 8 to regenerate the points-to analysis results from a given ARTwork. Without the loss of generality, we also assume that the transfer functions used by the consumer are monotonic in nature (monotonically increases).

Informally, the correctness argument can be summarized in three points: (i) if there is no tampering then consumer will regenerate results of same precision as the producer, (ii) if the tampering is non-conservative then the consumer will detect the tampering, and (iii) if the tampering is conservative, then the consumer will regenerate a sound, yet an over-approximation of the non-tampered result. The following two lemmas and corollary formalize the same.

Lemma 3.2. *Given a sound iterative-data-flow points-to analysis A, if $R_{prod}$ represents the sound points-to analysis results (for A) encoded by the producer in the* ARTwork $\Psi$*, then given $\Psi$, the consumer will (i) successfully regenerate points-to analysis results (no safety violation error) (ii) and these results will match $R_{prod}$, if the consumer uses the same lattice and transfer functions as the producer.*

Lemma 3.3. *Given a sound iterative-data-flow points-to analysis A, and a program P, if $\Psi$ is a non-conservatively tampered points-to analysis results (for the analysis A and program P), then the consumer will detect the violation of the safety.*

Corollary 3.4. *Given a sound iterative-data-flow points-to analysis A, an* ARTwork $\Psi$*, and a program P, our proposed scheme will infer a fixed-point solution for the transfer functions in A, if $\Psi$ is safe for P with respect to A, and declare $\Psi$ to be unsafe, otherwise.*

See the technical report [23] for a proof sketch, and Section 5 for a discussion on the possible scenario that the consumer and producer use different points-to analysis schemes. .

## 4 Optimizations

While the techniques discussed in Section 3 ensure that ART carries only information that is necessary and sufficient for efficient regeneration of the original sounds points-to analysis, we have identified four opportunities to improve the efficacy of ART (in terms of reducing the storage size without compromising on the precision). We discuss these optimizations below.

• *Encoding of Loop-Invariants.* For loops whose body does not contain any statements that affect the heap (for example, in the case of arithmetic loops), the fixed-point OUT-value of the loop-header (and all statements in the loop-body) will be equal to the IN-value of the loop-header. Thus, the emitted ART instance may be optimized to not carry loop-invariants for such loop-headers. During the regeneration phase, if any loop-header $i$ has no entry for $\mathcal{I}_{loop}[i]$ in the given ART instance, the consumer will simply use the IN-value of the loop-header as $\mathcal{I}_{loop}[i]$. We can easily identify such loops by iterating over the loop-body and checking for the absence of reference type instructions.
• *Encoding of $M_{in}$-Invariants.* Consider a procedure $M$, called from a set $\mathcal{S}$ of call-sites in a program $P$. Say, for each $s \in \mathcal{S}$, `project-in`(IN[$s$], $M$) is identical, where IN[$s$] represents the fixed-point IN value for $s$. That is, `project-in`(IN[$s$], $M$) is equal to the context-insensitive IN-summary of $M$. Some of the common scenarios where we encounter such procedures include, procedures invoked only once in a program, static procedures with no arguments, and so on. For such procedures, ART can be optimized by avoiding the overhead of carrying the respective $M_{in}$-Invariants, and the consumer will use the default value as discussed in Section 3.6, where we discussed how we handle tampering by deletion of $M_{in}$-Invariant entries.
• *Encoding of $M_{out}$-Invariants.* Consider a recursive procedure $M_{rec}$, in a program $P$. If the context-insensitive OUT-summary of $M_{rec}$ does not differ from its context-insensitive IN-summary, then for all such procedures, ART can be optimized by avoiding the overhead of carrying the $M_{out}$-Invariants, and the consumer will use the default value as discussed in Section 3.6, where we discussed how we handle tampering by deletion of $M_{out}$-Invariant entries.
• *Avoiding Duplicate Information.* When two ART entries refer to the same points-to information, we can avoid transmitting duplicate information by transmitting just the unique value and using references to the unique entry, wherever required.

## 5 Discussion

In this section, we discuss some interesting features and observations in the context of ART.

**Handling Tampered** ARTwork**.** For simplicity, we assume that if a consumer deduces that the input ARTwork is tampered with (and unsafe to use, see Sections 3.3), discards all the regenerated points-to analysis results and continues the JIT compilation or analysis like it would, in the absence of our technique. Such a deduction could be because of tampered ARTwork, or the consumer using a points-to analysis whose flow functions do not "match" that of the producer. ARTwork may also be conservatively tampered in a way that it now encodes *overly* imprecise results. As discussed in Section 3.3, ART will admit such results, as they are still sound. A consumer may decide to use some heuristics to ignore such possible highly imprecise results. For example, one such heuristic can be a tunable parameter that sets a "usability threshold", in the form of the size of any points-to set, or an upper limit on the maximum size of the encoded ARTwork. We leave the exploration of such heuristics as a future work.

**No** ARTwork **transmitted to the consumer.** A consumer (JIT compiler) capable of using ARTwork may get a program, without any ARTwork accompanying it. In such a scenario, the consumer can easily recognize the complete absence of ARTwork and not attempt the regeneration process at all. Instead, the consumer continues as it would in the absence of our technique.

**Dynamic features of Java and ART.** Java allows the programs to change during program execution using features like dynamic class loading (DCL) and hot code replace (HCR). In such a scenario, we have to analyze the ARTwork accompanying the newly loaded code, obtain the points-to information for the newly loaded code, and establish the overall safety of the ARTwork of the whole program. Efficiently maintaining points-to results in the presence of such dynamic features is an interesting future work.

**Consumer and producer using a different analysis.** The regeneration techniques proposed in Sections 3.2 and 3.5 assumed that the consumer and the producer use the same analysis (that is, the transfer functions and lattice). In the event that the consumer does not know the details about the analysis used by the producer and instead uses an arbitrary points-to analysis, we make the following two observations on the outcome (i) if the constraints used by the consumer during regeneration are more precise than the constraints used by the producer to generate ARTwork, then our technique will regenerate sound points-to analysis for the program; (ii) if the constraints used by the consumer are less precise (that is, more conservative) than the constraints used by the producer, then our technique may identify it as a case of tampering, since the assertions in Figures 4 and 10 may fail. This is expected, as the ARTwork generated by the producer may not include the additional information introduced by the more conservative analysis performed by the consumer.

**Threat model.** Given a program $P$ and an ARTwork $X$, there are two activities that can lead to $X$ being considered tampered with: $X$ has been tampered with, or $P$ has been tampered with. This leads to three scenarios that we consider as part of the threat model: (i) $X$ is tampered with, but $P$ is not. (ii) $P$ is tampered with, but $X$ is not. (iii) Both $X$ and $P$ are tampered with. All these scenarios can be represented by a single scenario that $X$ is tampered with respect to $P$, which is the scenario addressed in this manuscript.

**Encoding of ARTwork.** We use a simple scheme to encode the ARTwork. For each type of information (as discussed in Section 3.4) encoded in ARTwork, we emit the corresponding points-to graph (structure discussed in Section 2). We represent local variables by using stack slot indices. We represent each abstract object as a tuple containing the method and the program location where the object was allocated.

**Handling Composability.** Analysis of large real world Java applications with libraries presents two interesting directions of work related to composability. (1) Owing to the large sizes and resulting scalability issues with analysing real world Java applications with libraries, researchers have proposed analyzing them separately in a modular way [10], and these modular results have been composed[8, 9, 28, 34, 44, 58] to realize the analysis results for the whole program. Note that this composition may involve expensive computation. Our proposed scheme can be extended in a way the producer sends the ARTworks for the modular results and the consumer first composes them and then uses the proposed scheme to validate the composed results. The ARTwork for the modular results would have to be extended to maintain the summary information of all the arguments (including the receiver) at all the call-sites, for all the possible calls to the unavailable methods. An important challenge that needs to be addressed in this space is that the composition at the consumer may involve expensive computations and hence may not be suitable where the consumer cannot afford to pay a high cost for obtaining+verifying the analysis results. (2) The runtime libraries may not be available (and different than the libraries present) during static analysis: Thakur and Nandivada [53] present a scheme in which the application and the runtime libraries are analyzed independently to produce partial analysis results (without being conservative). These two partial analysis results can be combined during the runtime, without losing precision. Extending the idea of our proposed ART framework to partial analysis results and combining them safely would be quite interesting, but beyond the scope of this paper.

## 6 Implementation and Evaluation

We have implemented our proposed scheme of ART for Java programs in three parts: (i) the producer component as an extension to the Soot bytecode optimization framework [56], and (ii) a consumer component (termed RegenPTA) in the form of a Soot-based static analysis that attempts to obtain points-to analysis results for a Java program without paying the cost of fixed-point computations, and (iii) a second consumer component in the Eclipse OpenJ9 JIT compiler [25], which currently

Table 1. Static Details of the benchmarks used, storage overheads for ART and whole analysis results.

| | Benchmark | .class size (MB) | #analyzed methods | Artifact Size (KB) | | Overhead (%) | |
|---|---|---|---|---|---|---|---|
| | | | | naive | ART | naive | ART |
| 1. | sunflow | 1.2 | 908 | 275 | 78 | 22.38 | 6.34 |
| 2. | lusearch | 1.6 | 975 | 396 | 88 | 24.17 | 5.37 |
| 3. | luindex | 1.3 | 1280 | 252 | 64 | 18.9 | 4.81 |
| 4. | avrora | 1.5 | 2022 | 489 | 108 | 31.8 | 7.03 |
| 5. | antlr | 1.2 | 1324 | 1119 | 241 | 91.06 | 19.61 |
| 6. | fop | 1.9 | 377 | 66 | 14 | 3.39 | 0.71 |
| 7. | pmd | 2.0 | 2103 | 381 | 96 | 18.60 | 4.68 |
| 8. | compress | 0.47 | 466 | 98 | 19 | 20.40 | 3.96 |
| 9. | sparse | 0.47 | 480 | 93 | 18 | 19.32 | 3.74 |
| 10. | sor | 0.47 | 480 | 94 | 18 | 19.53 | 3.74 |
| 11. | fft | 0.47 | 485 | 94 | 18 | 19.40 | 3.72 |
| 12. | montecarlo | 0.47 | 485 | 93 | 17 | 19.40 | 3.54 |
| 13. | lu | 0.47 | 487 | 95 | 18 | 19.66 | 3.72 |
| | geomean | 0.85 | 728 | 157.89 | 33.72 | 18.02 | 4.22 |

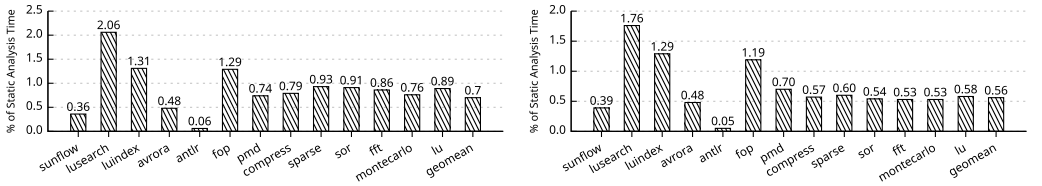
neither performs nor can afford to perform precise, fixed-point based, points-to analysis due to the performance considerations. The producer uses an extension to VASCO [41] to obtain flow-sensitive, context-insensitive points-to information and generate the ARTwork for each input application.

As is the common practice, we use the popular TamiFlex [6] tool for resolving reflective calls. However, note that our proposed scheme is not restricted by the presence of reflection: given any points-to analysis approach (augmented by Tamiflex or not), our scheme can be used to generate the corresponding ARTwork (by the producer) and regenerate the matching points-to information (by the consumer).

To experimentally evaluate the proposed ART scheme, similar to the many prior works [3, 26, 53], we used DaCapo and SPECjvm suites. We chose 13 benchmarks: (i) SUNFLOW, LUSEARCH, LUINDEX and AVRORA from the DaCapo 9.12-MR1-bach suite [5]; (ii) ANTLR, FOP and PMD from the DaCapo 2006-10-MR2 suite; and (iii) COMPRESS, SPARSE, SOR, FFT, MONTECARLO, and LU from SPECjvm2008 [50]. The rest of the excluded benchmarks could not be statically analyzed – either by Soot or by TamiFlex[1]. Our evaluation was performed on a Dell Precision 7920 server, which is a 2.3GHz Intel(R) Xeon(R) Gold 5218 CPU system with 64GB of main memory, running Ubuntu 20.04.1 LTS. Table 1 shows some of the static characteristics about the benchmarks used for evaluation; Column 3 shows the size of the portion of each benchmark analyzed by Soot (varied between 0.47 to 2.0 MB) and Column 4 shows the number of statically analyzed methods (varied between 466 to 2103) in each benchmark.

We present an evaluation to empirically establish the following six research questions: (RQ1) How does the regeneration technique compare to the complete analysis in terms of time and precision? (RQ2) Is the proposed scheme able to detect tampering of the generated ARTwork? (RQ3) What is the cost of regenerating flow-sensitive, context-insensitive analysis during JIT compilation? (RQ4) What is the storage overhead of ARTwork? (RQ5) What is the effect of the optimizations proposed in Section 4 on the size of ARTwork? (RQ6) What performance benefits can be realized by regenerating analysis results using ARTwork?

[1]Our prototype implementation makes use of Soot, along with the `play-in` and `play-out` agents of Tamiflex, the combination of which has documented issues with large applications [18–20, 35, 36]. To compensate, where possible, we have instead used corresponding benchmarks from an older version of the benchmark suite.




(a) Consumer: RegenPTA (b) Consumer: OpenJ9 JIT compiler

Fig. 13. Time for regeneration in the two consumers as compared to the time taken by the producer to perform the complete analysis.

### 6.1 Evaluation of the Consumer Components

We study the consumer components of our technique with respect to efficiency and safety of the regeneration process.

**RQ1.** *How does the regeneration technique compare to complete analysis in terms of time and precision?* To answer this question, we compare our regeneration scheme in the consumers against the complete analysis of the producer. Figures 13a and 13b show the time taken to regenerate the flow-sensitive, context-insensitive points-to analysis results by our regeneration technique in RegenPTA and OpenJ9, respectively, as a percentage of the time taken by the producer to perform the complete analysis. The figure shows that our technique regenerates points-to analysis results by paying an extremely small fraction of the time taken to perform the complete analysis by the producer (0.06% to 2.06%, geomean 0.70% for RegenPTA; and 0.05% to 1.76%, geomean 0.56% for OpenJ9). Note: since the base compilation times are different, the percentage overheads across the two consumers differs.

As an addition, to show that the points-to analysis results regenerated by the consumer is of equal precision as the results encoded by ART, we compared the OUT-summary regenerated by the consumers for each procedure with the OUT-summary of the same procedure as computed by the producer. These OUT-summaries were found to match for all the procedures analyzed for each program, thus showing that the regenerated points-to analysis results are of equal precision to the complete analysis.

**RQ2.** *Is the proposed scheme able to detect tampering of the generated* ARTwork*?* To empirically study the safety of the regenerated results, we tested the ability of our consumer component to identify ARTwork that is unsafe to use (that is, non-conservatively tampered). To do this, we randomly identified 10 different elements of ARTwork for each benchmark, and non-conservatively tampered them manually. This process consisted of two steps (1) randomly picking a non-trivial element of ARTwork (i.e., one that isn't solely composed of *summary* points-to information), and (2) non-conservatively tampering it. Examples of the performed non-conservative tampering include replacing an object $O_i$ in the points-to set of a variable or a field, with another object $O_j$ ($i \neq j$), reducing the cardinality of a points-to set, and removing a variable or field from a points-to map. We found that for all such tampered ARTwork, the consumer successfully identified the tampering.

**RQ3.** *What is the cost of regenerating flow-sensitive, context-insensitive analysis during JIT compilation?* We have found the regeneration scheme is extremely fast (owing to the underlying linear-time algorithm). For example, on average, it takes a couple of milliseconds per procedure across all the benchmarks. To understand its performance relative to the existing JIT compilation time, Fig. 14 shows the per-method average time taken by the regeneration process (includes the time to read the ARTwork from disk and parse it) as a percentage of the on-average per-method JIT compilation time. We see that we are able to obtain precise points-to information (typically a very expensive analysis) as a small fraction (15-35%) of the JIT compilation time. We believe that this is reasonable,

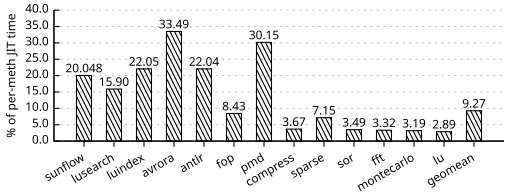



Fig. 14. Per-method cost of regeneration in comparison to the per-method cost of JIT compilation.

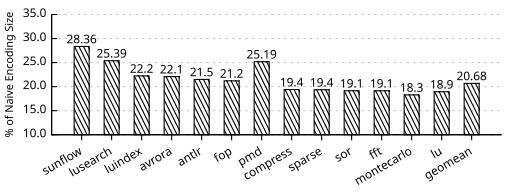



Fig. 15. Size of ARTwork as a percentage of size of naive encoding.

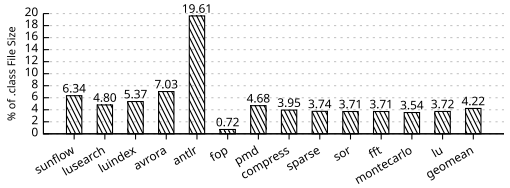



Fig. 16. Overhead of ARTwork in comparison to size of `.class` files.

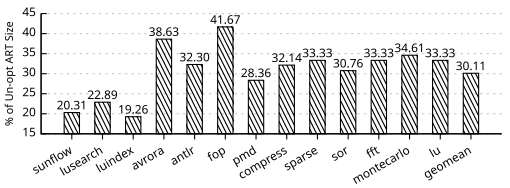



Fig. 17. Reduction in size of optimized ARTwork in comparison to size of un-optimized ARTwork.

considering the fact that JIT compilation itself takes a very small fraction of the total execution time and the possible utility of the regenerated precise points-to analysis results. We believe that our proposed technique clearly overcomes the drawbacks of both the alternatives (of using fast, but highly imprecise heuristics, or precise, but unacceptably slow analysis, as discussed in Section 1), while ensuring the analysis is safe to use.

### 6.2 Evaluation of the Producer Components

**RQ4.** *What is the storage overhead of* ARTwork*?* To study the efficiency of our proposed encoding of ARTwork, we define a *naive* encoding as a dump of the OUT values at each program-point for the whole program, and consider this encoding as a baseline for storage overhead. Fig. 15 shows the size of the ARTwork of each benchmark, as a percentage of the size of this naive encoding. The figure shows that the size of ARTwork, on average, is less than 21% of the size of the naive encoding; with the maximum size of ARTwork being 28.36% of the naive encoding for any benchmark.

**Note.** The comparison between the sizes of ARTwork and naive encoding was performed after compressing the artifacts of each encoding. Uncompressed, this comparison is observed to favor ART even more. For example, in the case of AVRORA, uncompressed ART was 1.7MB in size – which is only 10% of the uncompressed naive encoding at 17MB. Compression tends to make this comparison less dramatic due to the presence of similar points-to information at different program-points in the naive encoding, which gets compressed better. However, using such a naive scheme will naturally lead to higher file I/O time because of the handling of larger files.

We also studied the overhead of our encoding of ARTwork with respect to the size of the .class files analyzed. Fig. 16 shows the overhead of ARTwork as a percentage of the .class file size. We find that the overhead is around 4% of .class files size, on average.

**RQ5.** *What is the effect of the optimizations proposed in Section 4 on the size of* ARTwork*?* Fig. 17 shows the storage size of the (optimized) ARTwork as a percentage of the size of the ARTwork obtained without applying the optimizations discussed in Section 4. We observe that the optimized ARTwork is on average, approximately 70% of the size of the unoptimized version (leading to 30% reduction, on average, in the size of the ARTwork).

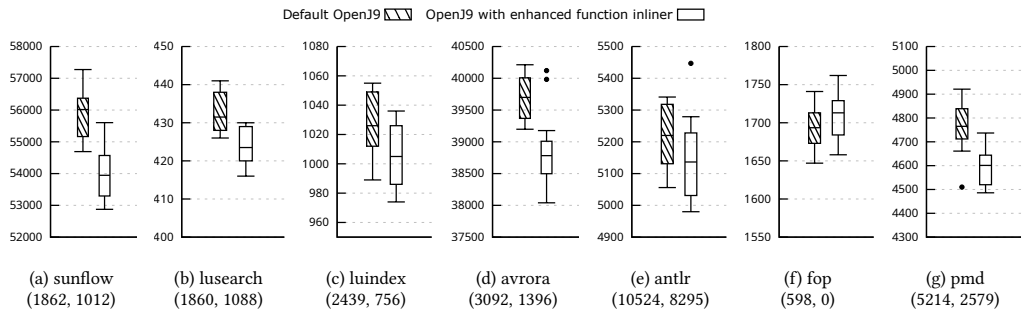



Fig. 18. Comparison of the execution times (in ms) on default OpenJ9 and OpenJ9 with the enhanced inliner. The caption subtext shows #monomorphic-calls-identified and #inlining-guards-avoided.

We have also observed that the regeneration time using the optimized ARTwork is, on average, approximately 92% of the regeneration time using unoptimized ARTwork (leading to on average 8% reduction in time taken for regeneration); we skipped the graph for brevity. We find the proposed optimizations are effective in reducing the size of the ARTwork and reducing the regeneration time.

### 6.3 Evaluation of the Usability of Regenerated Analysis Results

**RQ6.** *What performance benefits can be realized by regenerating analysis results using* ARTwork*?* As discussed in the answer to RQ1, our first consumer RegenPTA is able to very efficiently regenerate the points-to analysis results encoded by ARTwork provided to it after establishing its safety. Due to its high efficiency and the modular nature of Soot analyses, RegenPTA can be used by any Soot pass (analysis/optimization) that needs precise points-to analysis results, without paying the cost of fixed-point computations; Fig. 13a gives an indication of the benefits that can be accrued.

Next, to study the impact of efficiently realizing precise points-to analysis results in the Eclipse OpenJ9 JIT compiler, our second consumer, we have used its function inliner as a client and studied the effect. Since fixed-point based points-to analysis is considered expensive, OpenJ9 currently uses the following technique to perform function inlining (i) it first uses the information from a runtime profiler to identify the probabilities of invocation of each potential target of a virtual call, and then (ii) if the probability associated with invoking a particular target is sufficiently high, OpenJ9 inlines that particular target guarded by a runtime check on the type of the receiver of the virtual call. If the type of the receiver at runtime does not match the type that defined the inlined target, then OpenJ9 falls back to the original virtual call. Therefore, in OpenJ9, each inlined call-site takes the form of a conditional statement – with the inlined method forming the body of 'then-branch', and the original virtual call the body of the 'else-branch'. In case of *monomorphic* calls, as there will always be a singular invocation target, the evaluation of the associated guards is redundant. Removing the corresponding condition-checking code and the else-branch of the conditional-statement can improve the program performance. We leverage this intuition to answer RQ6: we used the points-to analysis results regenerated using our technique to identify such monomorphic calls, inlined those call-sites without any guards, and then studied the impact of such an optimization on the overall execution time. For this experiment, we found that SPECjvm offered very few opportunities for applying this optimization and the gains/overheads were negligible. Thus, we only show the details for the DaCapo benchmarks.

Fig. 18 plots the execution times of the benchmarks using the default OpenJ9, and the OpenJ9 with the enhanced function inliner. With each benchmark we also mention (in the caption) the number of monomorphic calls identified and the number of inlining guards avoided by the enhanced

inliner. We see that we were able to use the points-to analysis results regenerated by our technique to identify a large number of monomorphic calls across the benchmarks evaluated. Note that for any given program, the number of inlining guards avoided due to this optimization can be less than the total number of monomorphic calls identified. This is because OpenJ9 may not find all the monomorphic calls profitable to be inlined (decided by OpenJ9 based on factors like caller size, callee size, and so on), and hence inlining is not even attempted for such non-profitable calls.

Since the performance of Java applications varies significantly [17], across different JVM invocations and iterations, to obtain steady-state performance for Fig. 18, we used the following evaluation methodology: for each benchmark, we conducted 30 runs, where each run consisted of 99 warmup iterations, and the following iteration was used to measure the execution time. The runs for each benchmark were interleaved to account for systemic bias and impact due to different JVM invocations. As can be seen, the enhanced OpenJ9 is able to reduce execution times for all benchmarks where a non-zero number of inlining guards were avoided. Considering that this improvement is on top of the plethora of other optimizations that OpenJ9 already has, we argue that the gains are significant.

In case of FOP, we found that though the enhanced inliner did not reduce any guards, we did not observe much degradation. This attests to the extremely low overheads of regeneration as discussed in the context of RQ3.

*Overall summary.* Our study shows that the points-to analysis results regenerated using ARTwork are not only usable in practice, but can also result in measurable performance improvements.

## 7 Related Work

**Staged Analysis.** There have been prior works that have attempted to reduce JIT compilation overheads due to expensive analyses by leveraging the multi-stage nature of Java compilation. For example Ali [1], Serrano et al. [46], Sharma et al. [47] have proposed schemes to perform expensive whole-program analyses/optimizations statically for Java, which are then leveraged at runtime to obtain complete analysis/optimized code. Similarly, the PYE framework [53] performs points-to analysis in a static analyzer and sends along partial summaries representing the analysis to the JIT compiler, where it is augmented with information available at runtime to make it more precise. We note that all these schemes are affected by the same challenges safety and transmission costs discussed in Section 1, thereby severely impacting their practical usage. Our proposed technique of ART addresses both these issues and can be the first step towards making staged analysis practical.

**Points-to Analysis.** There have been many papers on points-to/alias analysis, depicting various dimensions of analysis such as flow-sensitivity, context-sensitivity, path-sensitivity, and so on [14, 44, 51, 53, 58], various avenues for speeding up [43, 52, 54], and performing optimization specific analysis [32, 39, 53], and so on. For our implementation, we have extended the analysis implemented in VASCO [41] to build our static analyzer. Many recent static analysis tools [15, 16, 24, 30, 33] employ SPARK [29] (which is flow- and context-insensitive, to ensure scalability) as their points-to analysis solution. Deploying ART in these tools can improve their precision by offering flow-sensitive points-to analysis, with little additional cost. Over the years, there has been a rich body of work championing flow-sensitive, context-insensitive points-to analyses [11, 55, 58, 59]. Rather than serve as a replacement for these techniques, ART in fact promotes their adoption by ameliorating the concern of scalability in tools that wish to employ them.

**Declarative points-to analyses.** Datalog-based points-to analyses (like Doop [7]) are convenient due to the declarative nature of their analysis rules. While this paper discusses ART in the context of IDFA-based points-to analysis, ART isn't restricted by an IDFA-based producer. We believe that ART can be deployed even in systems where the producer employs a declarative

points-to analysis (see prior work [22, 27, 49]) provided the consumer possesses an equivalent set of transfer functions to apply during the regeneration process.

**Interactive Proof Systems.** A keen reader will note that ART is similar, in spirit, to interactive proof systems like zero-knowledge protocols [21] (ZKPs). While an interesting parallel, we note that they differ fundamentally in both domain and purpose of application. ZKPs are employed in cryptography to convey an awareness (i.e., a proof) of knowledge without actually giving away the knowledge, an interaction motivated by privacy. In contrast, ART is not motivated by privacy, but with efficiently sharing points-to information, along with a proof of its correctness.

## 8 Conclusion and Future Work

This article proposes a scheme called ART that helps Java compilers (both static and JIT) and program analysis tools – termed *consumers* – obtain highly precise (flow-sensitive, context-insensitive) points-to analysis results in an efficient and safe manner; where such analysis results were hitherto difficult to attain due to performance and safety concerns. Using ART, a consumer can obtain very precise points-to analysis results without prohibitively impacting compilation (or analysis) time or compromising on the safety of the analysis results. We show, via a detailed evaluation, that ART enables the generation of precise and trusted analysis results in consumer systems in a minute fraction of the time (<1%, on average) required to perform the complete analysis, while adding a very small space-overhead (around 4%, on average) to the bytecode. It would be interesting to extend the proposed ideas to support other dimensions of analyses (for example, call-site-based context-sensitivity, object-sensitivity, path-sensitivity, and so on) and the nuances of other non-Java like languages. For example, supporting context-sensitivity necessitates an efficient encoding in ART to uniquely identify each calling context; and languages like C and C++ may need interesting augmentations to ART to support features like pointer arithmetic, casting and function pointers. We leave such explorations as future work.

## Acknowledgments

This research was partially supported by IBM CAS projects 1101 and 1156. We also thank Manas Thakur from IIT Bombay for the fruitful discussions during the early stages of this work.

## References

[1] Karim Ali. 2014. *The Separate Compilation Assumption.* Ph. D. Dissertation. University of Waterloo, Waterloo, Canada. https://plg.uwaterloo.ca/~olhotak/pubs/thesis-karim-phd.pdf

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417. https://doi.org/10.1147/sj.442.0399

[3] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. 2024. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. *Proc. ACM Program. Lang.* 8, PLDI, Article 159 (Jun 2024), 24 pages. https://doi.org/10.1145/3656389

[4] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 57 (sep 2022), 69 pages. https://doi.org/10.1145/3477579

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[6] 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* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

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

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[10] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 159–178. http://dl.acm.org/citation.cfm?id=647478.727794

[11] Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 665–687. https://doi.org/10.1007/978-3-642-31057-7_29

[12] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[13] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. 2019. Proof-Carrying Smart Contracts. In *Financial Cryptography and Data Security*, Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–338. https://doi.org/10.1007/978-3-662-58820-8_22

[14] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-Scale Exhaustive Points-to Analysis for Java in under a Minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 535–551. https://doi.org/10.1145/2814270.2814307

[15] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. LiveDroid: Identifying and Preserving Mobile App State in Volatile Runtime Environments. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 160 (nov 2020), 30 pages. https://doi.org/10.1145/3428228

[16] Kostas Ferles, Benjamin Sepanski, Rahul Krishnan, James Bornholt, and Işil Dillig. 2022. Synthesizing Fine-grained Synchronization Protocols for Implicit Monitors. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 67 (apr 2022), 26 pages. https://doi.org/10.1145/3527311

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[18] Github. 2015. *Problem using Booster and the play-out agent.* Github. https://github.com/secure-software-engineering/tamiflex/issues/1

[19] Github. 2018. *FATAL ERROR while using Play In Agent Tamiflex Issue #940.* Github. https://github.com/soot-oss/soot/issues/940

[20] Github. 2021. *PlayOutAgent fails with OpenJ9 JDK · Issue #11 · secure-software-engineering/tamiflex.* Github. https://github.com/secure-software-engineering/tamiflex/issues/11

[21] S Goldwasser, S Micali, and C Rackoff. 1985. The Knowledge Complexity of Interactive Proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

[22] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-scalable Static Analysis with Heap Snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/3213846.3213860

[23] Shashin Halalingaiah, Vijay Sundaresan, Daryl Maier, and V. Krishna Nandivada. 2024. *The ART of Sharing Points-to Analysis (Extended Abstract).* Technical Report. IIT Madras. http://cse.iitm.ac.in/~krishna/preprints/oopsla24/oopsla-ea-24.pdf

[24] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 256 (oct 2023), 30 pages. https://doi.org/10.1145/3622832

[25] IBM. 2017. *Eclipse OpenJ9.* IBM. https://github.com/eclipse/openj9

[26] Md. Jahidul Islam, A.T.M Mizanur Rahman, and Sohel Rana. 2023. Performance Analysis of Modern Garbage Collectors using Big Data Benchmarks in the JDK 20 Environment. In *5th International Conference on Sustainable Technologies for Industry 5.0 (STI)*. IEEE, 1–6. https://doi.org/10.1109/STI59863.2023.10464900

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

[28] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

[29] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. 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

[31] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional, USA. https://docs.oracle.com/javase/specs/jvms/se8/html/

[32] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying Dereference Safety via Expanding-Scope Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/1390630.1390657

[33] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 46 (jul 2021), 46 pages. https://doi.org/10.1145/3450492

[34] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A Framework For Efficient Modular Heap Analysis. *Found. Trends Program. Lang.* 1, 4 (Jan. 2015), 269–381. https://doi.org/10.1561/2500000020

[35] Misc. 2015. *Problem using Booster and the play-out agent.* https://groups.google.com/g/tamiflex-discuss/c/ZQIagfvlOsI

[36] Misc. 2019. *Soot-list Soot Dacapo Issue.* https://mailman.cs.mcgill.ca/pipermail/soot-list/2019-February/009071.html

[37] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 527–568. https://doi.org/10.1145/319301.319345

[38] Steven S. Muchnick. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers an imprint of Elsevier, Amsterdam. https://dl.acm.org/doi/10.5555/286076

[39] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate Interprocedural Null-Dereference Analysis for Java. In *31st International Conference on Software Engineering*. IEEE, Canada, 133–143. https://doi.org/10.1109/ICSE.2009.5070515

[40] George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712

[41] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) *(SOAP '13)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/2487568.2487569

[42] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1. https://www.usenix.org/conference/jvm-01/java-hotspot%E2%84%A2-server-compiler

[43] Girish Maskeri Rama, Raghavan Komondoor, and Himanshu Sharma. 2018. Refinement in object-sensitivity points-to analysis via slicing. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 142:1–142:27. https://doi.org/10.1145/3276512

[44] Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–215. https://doi.org/10.1007/978-3-540-30579-8_14

[45] Christopher Schwaab, Ekaterina Komendantskaya, Alasdair Hill, František Farka, Ronald P. A. Petrick, Joe Wells, and Kevin Hammond. 2019. Proof-Carrying Plans. In *Practical Aspects of Declarative Languages*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, Cham, 204–220. https://doi.org/10.1007/978-3-030-05998-9_13

[46] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. 2000. Quicksilver: A Quasi-Static Compiler for Java. *SIGPLAN Not.* 35, 10 (oct 2000), 66–82. https://doi.org/10.1145/354222.353176

[47] Shamik Sharma, Anurag Acharya, and Joel Saltz. 1998. *Deferred Data-Flow Analysis.* Technical Report. University of California at Santa Barbara, USA. https://cs.ucsb.edu/index.php/research/tech-reports/1998-03

[48] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. https://doi.org/10.1561/2500000014

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

[50] SPECjvm. 2008. *SPECJVM2008*. SPEC. https://www.spec.org/jvm2008/

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

[52] 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 (CC)*. ACM, Washington, DC, USA, 135–146. https://doi.org/10.1145/3302516.3307359

[53] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (jul 2019), 37 pages. https://doi.org/10.1145/3337794

[54] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3377555.3377902

[55] Hamid A. Toussi and Abbas Rasoolzadegan. 2014. Flow-sensitive points-to analysis for Java programs using BDDs. In *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 380–386. https://doi.org/10.1109/ICCKE.2014.6993367

[56] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13. https://dl.acm.org/doi/10.5555/781995.782008

[57] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. *SIGPLAN Not.* 34, 10 (oct 1999), 187–206. https://doi.org/10.1145/320385.320400

[58] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

[59] Jongwook Woo, Jehak Woo, I. Attali, D. Caromel, J.-L. Gaudiot, and A.L. Wendelborn. 2001. Alias Analysis for Java with Reference-set Representation. In *Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001*. IEEE, 459–466. https://doi.org/10.1109/ICPADS.2001.934854

Received 2024-04-06; accepted 2024-08-18