Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis

Dongjie He
The University of New South Wales, Sydney, Australia

Jingbo Lu
The University of New South Wales, Sydney, Australia

Jingling Xue
The University of New South Wales, Sydney, Australia

Abstract

Existing whole-program context-sensitive pointer analysis frameworks for Java, which were open-sourced over one decade ago, were designed and implemented to support only method-level context-sensitivity (where all the variables/objects in a method are qualified by a common context abstraction representing a context under which the method is analyzed). We introduce Qilin as a generalized (modern) alternative, which has been open-sourced on GitHub, to support the current research trend on exploring **fine-grained context-sensitivity** (including variable-level context-sensitivity where different variables/objects in a method can be analyzed under different context abstractions at the variable level), **precisely**, **efficiently**, and **modularly**. To meet these **four** design goals, Qilin is developed as an imperative framework (implemented in Java) consisting of a fine-grained pointer analysis kernel with parameterized context-sensitivity that supports on-the-fly call graph construction and exception analysis, solved iteratively based on a new carefully-crafted incremental worklist-based constraint solver, on top of its handlers for complex Java features.

We have evaluated Qilin extensively using a set of 12 representative Java programs (popularly used in the literature). For method-level context-sensitive analyses, we compare Qilin with Doop (a declarative framework that defines the state-of-the-art), Qilin yields logically the same precision but more efficiently (e.g., 2.4x faster for four typical baselines considered, on average). For fine-grained context-sensitive analyses (which are not currently supported by open-source Java pointer analysis frameworks such as Doop), we show that Qilin allows seven recent approaches to be instantiated effectively in our parameterized framework, requiring additionally only an average of 50 LOC each.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Pointer Analysis, Fine-Grained Context Sensitivity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.30

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.6`

Funding Supported by ARC Grants DP180104069 and DP210102409.

Acknowledgements We thank all the reviewers for their constructive comments.

1 Introduction

Pointer analysis, which approximates statically the possible run-time objects that may be pointed to by a variable in a program, is the basis of nearly all the other static program analyses. There are many significant applications, including call graph construction [23, 1, 38], program understanding [46, 36], bug detection [34, 55, 27, 10], security analysis [4, 11, 13], compiler optimization [9, 47], and symbolic execution [52, 21, 51].

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 30; pp. 30:1–30:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For object-oriented languages, context-sensitive pointer analyses are the most common class of precise pointer analyses [48, 32, 23, 42, 44]. Existing (whole-program) pointer analysis frameworks for Java, such as Doop [8], Wala [16], Jchord [34], and Paddle [24], were all designed and implemented over a decade ago by supporting *method-level context-sensitivity* only. For these traditional frameworks, when a method is analyzed under a given context abstraction (e.g., a k-limited context string [41] with k being fixed for the program), all the variables/objects in the method are analyzed uniformly under that given context abstraction.

For the past decade, Doop has been the most widely used context-sensitive pointer analysis framework for Java [42, 22, 43, 48, 49, 26, 17, 18, 19]. Doop encodes a pointer analysis declaratively by using Datalog (a logic-based language) to define pointer-related relations (in terms of Datalog rules) and a Datalog engine to infer the points-to facts. Thus, its performance is largely determined by the Datalog engine used and can be also sensitive to both automatic and manual optimizations applied to the Datalog rules. For example, a recent Datalog engine porting effort for replacing LogicBlox with Soufflé [20, 39] has boosted its performance by up to 4x [3]. When it was released in 2009 [8], Doop was shown to outperform Paddle [24] (the then state-of-the-art framework with the core of its pointer analysis algorithm performed in Datalog declaratively but the rest coded in Java imperatively) significantly. In addition, Doop was then argued to cost less human effort in implementing different pointer analyses with different flavors of context-sensitivity as they can all be specified modularly as variations on a common code base.

Currently, one emerging research trend is shaping the future of research on context-sensitive pointer analyses. To analyze large programs more scalably with more flexible efficiency/precision trade-offs, context-sensitivity is becoming increasingly more fine-grained. Method-level context-sensitivity can now be selective [43, 19, 25] (with only a subset of methods in the program being analyzed context-sensitively) or partial [30, 14] (with only a subset of variables/objects in a method being analyzed context-sensitively). In the future, pointer analysis frameworks are expected to support *variable-level context-sensitivity*, which allows different variables/objects in a method to be analyzed under different context abstractions, in order to enlarge the space of efficiency/precision trade-offs made. As for the option of extending Doop to support such fine-grained context-sensitivity, we have made significant efforts, but its resulting performance can often be disappointing due to possibly poor join orders selected by its underlying Soufflé Datalog engine [39, 20] used. Understandably, while the authors of [43, 19, 25] implemented trivially their selective method-level context-sensitivity in Doop and observed the desired efficiency/precision trade-offs, the authors of [30, 14] had to settle with some in-house implementations of their partial method-level context-sensitivity in Soot [23] imperatively in order to achieve the expected efficiency/precision trade-offs.

We introduce Qilin, a modern framework (implemented imperatively in Java) for supporting Java pointer analyses with (1) **fine-grained context-sensitivity**, (2) **precisely**, (3) **efficiently**, and (4) **modularly**. How to support (1) subject to (2) – (4) is nontrivial both scientifically and engineering-wise and was not done before. For example, achieving (3) and (4) requires Qilin to adopt new scientific approaches to specify and conduct pointer analysis when different variables/objects in a method are analyzed under different context abstractions imperatively. Achieving (2) requires Qilin to handle the full semantic complexity of Java, involving huge engineering efforts. Given that Doop has been tuned for supporting method-level context-sensitivity for over a decade, can Qilin outperform Doop while achieving the same precision? In addition, can Qilin support a variety of fine-grained context-sensitivity well? Qilin addresses these challenges, making the following contributions:

- QILIN represents the first imperative Java pointer analysis framework for supporting fine-grained context-sensitivity, precisely, efficiently and modularly.
- QILIN achieves its efficiency and modularity by decoupling the analysis logic for a given analysis algorithm from its implementation in the following novel way:
  - QILIN includes a pointer analysis kernel (supporting both on-the-fly call graph construction and on-the-fly exception analysis) with parameterized fine-grained context-sensitivity, allowing different flavors (i.e., granularities) of fine-grained context-sensitivity to be specified (i.e., instantiated) modularly as variations on a common code base (even in the fully imperative setting).
  - QILIN includes a new incremental worklist-based constraint solver that has been generalized in a non-trivial manner from a traditional incremental worklist-based constraint solver for supporting context-insensitive pointer analyses [23]. As existing solvers are limited to method-level context-sensitivity, we have crafted our solver carefully in order to support fine-grained context-sensitive pointer analyses efficiently.
- QILIN covers the same complex Java features and semantic complexities (e.g., reflection, native code, threads, etc.) as DOOP, delivering the same analysis precision.
- QILIN anchors around it a tool suite consisting of not only all method-level context-sensitive analyses supported by DOOP but also a wide range of fine-grained analyses.
- QILIN is evaluated with a set of 12 representative Java benchmarks and applications (popularly used in the literature). For method-level context-sensitive analyses, QILIN (which is currently single-threaded) is 2.4x faster than DOOP (running with 8 threads under its best thread configuration) for four typical baselines considered on average while achieving exactly the same precision. Unlike DOOP, QILIN supports effectively fine-grained context-sensitive analyses, by enabling seven recent approaches to be instantiated in its parameterized framework with an average of 50 LOC being added only.

QILIN is designed to be an open-source project (released at `https://github.com/QiLinPTA/QiLin/`), consisting of currently about 20.3 KLOC in Java (including 4.7 KLOC only at its core for performing parameterized pointer analysis). As a highly-configurable pointer analysis framework, QILIN provides benefits for both researchers and end users. For researchers, QILIN can help them both experiment with new ideas more quickly than if they have to conduct their own in-house implementations of their pointer analysis algorithms as in [14, 28, 15] and evaluate their ideas by making an apples-to-apples comparison against the state of the art in the same framework. For end users, QILIN can help them build their client application tools, such as bug detectors and program verifiers, by choosing some existing configured pointer analyses that are best suited to their needs. We plan to grow and maintain this open-source pointer analysis framework on GitHub to provide a common framework for researchers and practitioners to design, implement and evaluate different analyses for Java programs.

The rest of this paper is organized as follows. Section 2 provides some background knowledge and motivates this work. Section 3 introduces our QILIN framework. In Section 4, we demonstrate how to create, i.e., instantiate a number of fine-grained context-sensitive pointer analyses modularly in QILIN. Section 5 provides an extensive evaluation of QILIN. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Background and Motivation

We discuss context-sensitive pointer analyses by first reviewing method-level context-sensitivity and then motivating its generalization to variable-level context-sensitivity.

2.1 Method-Level Context-Sensitivity

Traditionally, context-sensitive approaches analyze a method separately under different calling contexts that abstract its different run-time invocations. Under such *method-level context-sensitivity*, whenever a method is analyzed for a given context, all its (local) variables and (allocated) objects are qualified by, i.e., analyzed under that context. For object-oriented languages, two common context abstractions are callsites (callsite-sensitivity [40]) and receiver objects, or precisely, their allocation sites (object-sensitivity [32, 33]). The two other variations are type-sensitivity [42] and hybrid sensitivity [22].

To tame the combinatorial explosion of contexts encountered in practice, a context is often represented by a k-limited context string, i.e., a sequence of k context elements [41]. Under k-limiting, two representative forms of context-sensitivity for object-oriented languages are: (1) *k-callsite-sensitivity* [40], which distinguishes the contexts of a method by its k-most-recent callsites, and (2) *k-object-sensitivity* [32, 33]), which distinguishes the contexts of a method by its receiver's k-most-recent allocation sites.

Despite k-limiting, the context explosion problem still occurs frequently in analyzing large programs [42, 48, 19], causing context-sensitive pointer analyses to be inefficient even when they are scalable (for usually only small values of k). Instead of applying k-limiting uniformly (with a fixed value of k) to all the methods (i.e., all the variables/objects) in the program, researchers have recently demonstrated that making context-sensitivity more fine-grained can lead to more flexible efficiency/precision trade-offs and better scalability. As a result, method-level context-sensitivity can now be selective [43, 19, 25] (with a subset of methods in the program being analyzed context-sensitively) and partial [30, 14] (with a subset of variables/objects in a method being analyzed context-sensitively).

2.2 Variable-Level Context-Sensitivity

In the future, variable-level context-sensitivity (that includes naturally method-level context-sensitivity as a special case) can be investigated by using Qilin. Under such *fine-grained context-sensitivity* in its full generality, different variables/objects in a method can be analyzed under different contexts (e.g., different k-limited context strings with different values of k). This will significantly enlarge the space of context-sensitive pointer analyses that researchers and practitioners can experiment with in order to achieve the most flexible efficiency/precision trade-offs and best scalability possible for their pointer analysis problems considered.

```
1 class A { Object f; }
2 class B {
3   Object foo(Object x, A a) {
4     a.f = x;
5     Object t = a.f;
6     System.out.print(t);
7     return x; }
8 }
```

```
 9 void main() {
10   Object o1 = new Object(); // O1
11   B b1 = new B(); // B1
12   A a1 = new A(); // A1
13   Object v1 = b1.foo(o1, a1);
14   Object o2 = new Object(); // O2
15   B b2 = new B(); // B2
16   Object v2 = b2.foo(o2, a1);}
```

Figure 1 A motivating example program.

2.3 Example

Consider a simple program given in Figure 1, where `foo()` is called twice, once on receiver `B1` in line 13 and once on receiver `B2` in line 16. If the program is analyzed context-insensitively, these two calls will be conflated, and consequently, the parameter `x` will also be conflated, preventing the analysis from proving that the two calls actually return two distinct objects (i.e., `O1` and `O2`). As a result, `v1` and `v2` are concluded to point to both `O1` and `O2` conservatively.

However, a context-sensitive analysis that distinguishes the two calls will be able to infer that `v1` points to `O1` only and `v2` points to `O2` only. Without loss of generality, let us consider 1-object-sensitivity [32, 33], under which these two calls will be distinguished by its two different receiver objects, `B1` and `B2`, used. Thus, under method-level context-sensitivity, `foo()` is analyzed twice, with its four variables `this`, `x`, `a` and `t` being analyzed once under context [`B1`] and once under context [`B2`]. As the contexts of its parameter `x` are distinguished under the two calls, `v1` is found to point to `O1`, i.e., the object pointed to by `x` under [`B1`], and similarly, `v2` is found to point to `O2`, i.e., the object pointed to by `x` under [`B2`].

However, applying method-level context-sensitivity to `foo()` in this program is overkill. Under fine-grained context-sensitivity, we can conduct 1-object-sensitive analysis to `foo()` exactly as before except that we only need to distinguish its parameter `x` context-sensitively. Note that all the variables/objects in `main()` are naturally context-insensitive. By analyzing only `x` in this program context-sensitively, the resulting fine-grained analysis will be faster while losing no precision at all for all its variables/objects.

3 Designing the Qilin Framework

Figure 2 gives the architecture of Qilin, built on top of Soot [53], for supporting fine-grained context-sensitive pointer analysis for Java programs. Currently, Qilin's toolbox (①) includes not only all the method-level context-sensitive pointer analyses that are supported by Doop [8] but also a range of recently proposed representative fine-grained analyses, as will be elaborated in Section 4. In this section, we describe how Qilin is designed to support fine-grained context-sensitivity, precisely, efficiently and modularly in terms of its four major components depicted at ② – ⑤. In Section 3.1, we explain how we parameterize variable-level context-sensitivity to allow different flavors of context-sensitive analyses to be specified modularly (②). We then formalize our parameterized pointer analysis in Section 3.2 (③) and introduce our new incremental worklist-based constraint solver for solving it efficiently in Section 3.3 (④). In Section 3.4, we describe how Qilin covers complex Java features in order to support pointer analyses precisely for real-world Java programs (⑤).

It should be pointed out that in Qilin, all the pointer analyses (including what is provided in its toolbox (①)) are instantiated (concretized) in terms of ② as variations on a common code base consisting of ③ – ⑤, even though Qilin is implemented imperatively in Java.

3.1 Parameterized Context-Sensitivity

In Qilin, as depicted at ② in Figure 2, context-sensitivity for a given analysis is defined by a set of three parameters, a context constructor, a context selector, and a heap abstractor, each of which can be instantiated to support different flavors (i.e., granularities) of context-sensitivity from the method level to the variable level.

The *context constructor*, denoted `Cons`, is used to create the contexts required for analyzing a method in the traditional manner. Therefore, this parameter alone will be sufficient to specify different flavors of method-level context sensitivity considered traditionally, including

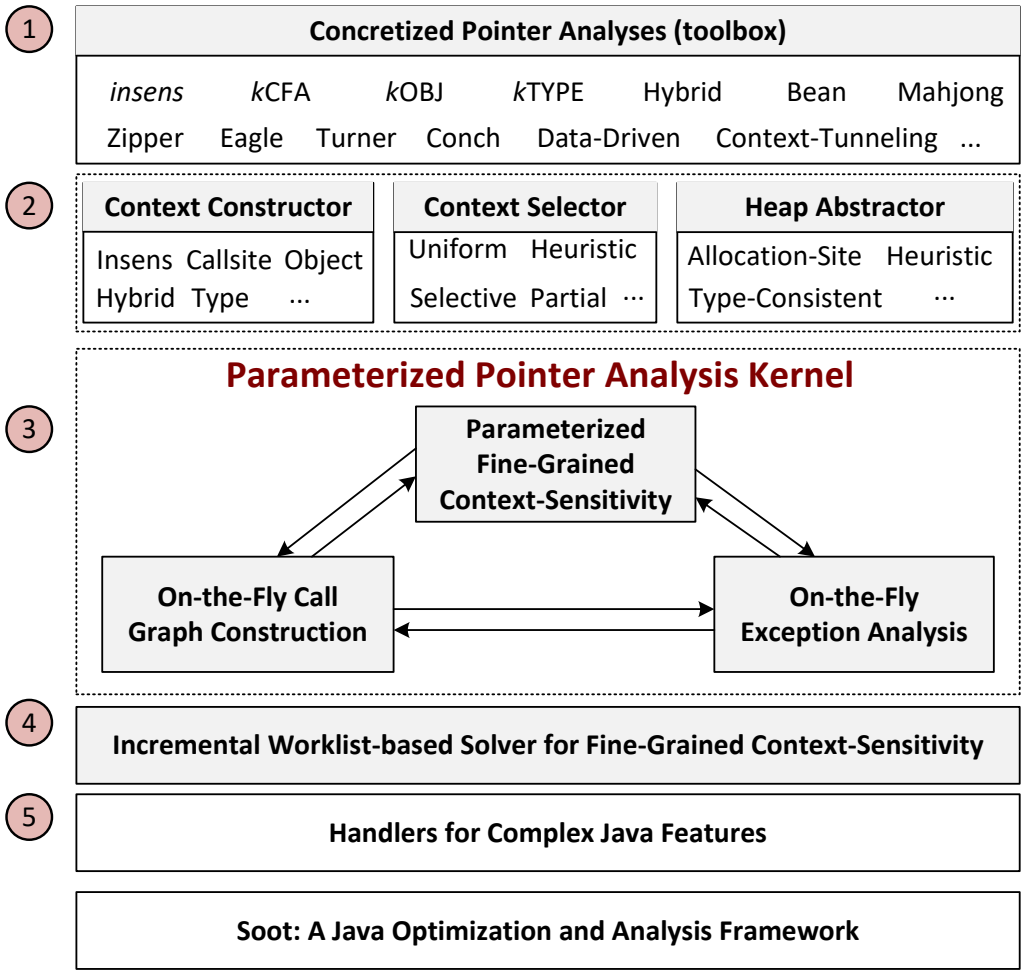


Figure 2 The architecture of Qilin.

"Insens", "CallSite", "Object", "Type", and "Hybrid", which will be instantiated in Section 4.1. The *context selector*, denoted `Sel`, is used to define the contexts required for analyzing variables/objects (based on the contexts of their containing methods specified by `Cons`) to support fine-grained context-sensitivity, including "Uniform", "Heuristic", "Selective", and "Partial", which will be instantiated in Section 4.2. The *heap abstractor*, denoted `HeapAbs`, is used to define an abstraction of the objects in the heap, including "Allocation Site", "Heuristic", and "Type-Consistent", which will be instantiated in Section 4.3.

As context-sensitivity is parameterized (②), an understanding about its actual instantiations is not needed now in order to understand its other three components (③ – ⑤), which will be described in turn below.

3.2 Parameterized Pointer Analysis

We describe our parameterized pointer analysis (③ in Figure 2) that supports on-the-fly call graph construction [23] and exception analysis [7] by considering a simplified subset of Java, with only eight kinds of labeled statements given in Table 1. Note that "x = **new** T(...)" in standard Java is modeled as "x = **new** T; x.`<init>`(...)", where `<init>`(...)

is the corresponding constructor invoked. The control flow statements are not considered because Qilin supports only context-sensitive analyses just as Doop [8]. In our formalism (for simplicity and without loss of generality), every method is assumed to have one return statement "**return** *ret*", where *ret* is its *return variable*, and one special catch statement "**catch** *eret*" for catching all throwable objects thrown out of the method.

Table 1 Eight kinds of statements analyzed by Qilin.

Kind	Statement	Kind	Statement
NEW	$l : x = \textbf{new}\ T$	ASSIGN	$l : x = y$
STORE	$l : x.f = y$	LOAD	$l : x = y.f$
THROW	$l : \textbf{throw}\ x$	CATCH	$l : \textbf{catch}\ y$
CALL	$l : x = a_0.f(a_1, ..., a_r)$	RETURN	$l : \textbf{return}\ ret$

Let $\mathbb{V}$, $\mathbb{H}$, $\mathbb{T}$, $\mathbb{M}$, $\mathbb{F}$, and $\mathbb{L}$ be the domains for representing sets of variables, heap abstractions, types, methods, field names, and statements (identified by their labels), respectively. Let $\mathbb{C}$ be the universe of contexts. The following auxiliary functions are used:

- $\texttt{PTS} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \rightarrow \wp(\mathbb{H} \times \mathbb{C})$
- $\texttt{MethodOf} : \mathbb{L} \rightarrow \mathbb{M}$
- $\texttt{Stmt} : \mathbb{M} \rightarrow \wp(\mathbb{L})$
- $\texttt{MethodCtx} : \mathbb{M} \rightarrow \wp(\mathbb{C})$
- $\texttt{VirtualCallDispatch} : \mathbb{M} \times \mathbb{H} \rightarrow \mathbb{M}$
- $\texttt{ExceptionDispatch} : \mathbb{L} \times \mathbb{H} \rightarrow \mathbb{L}$
- $\texttt{Cons} : \mathbb{H} \times \mathbb{C} \times \mathbb{L} \times \mathbb{C} \rightarrow \mathbb{C}$
- $\texttt{Sel} : (\mathbb{V} \cup \mathbb{H}) \times \mathbb{C} \mapsto \mathbb{C}$
- $\texttt{HeapAbs} : \mathbb{L} \times \mathbb{T} \mapsto \mathbb{H}$

where `PTS` records the points-to information found context-sensitively for a variable or an object's field, `MethodOf` gives the method containing a statement, `Stmt` returns the statements in a method, and `MethodCtx` maintains the contexts used for analyzing a method. As the pointer analysis is conducted together with both the call graph construction and exception analysis performed on the fly, we use `VirtualCallDispatch` to resolve a virtual call to a target method based on the dynamic type of the receiver object, and `ExceptionDispatch` to resolve a throwable statement to a catch statement by tracing the exception-catch links [7]. `Cons`, `Sel` and `HeapAbs` are three significant parameters described in Section 3.1. `Cons` describes how to construct a new context for a method, `Sel` selects some context elements from the context of a method to form a new context for a variable declared (or object allocated) in the method, and `HeapAbs` defines the heap abstraction for an object. We will discuss their instantiations for supporting different analysis algorithms in Section 4.

Figure 3 gives six rules used for formalizing our parameterized pointer analysis that supports both call graph construction and exception analysis on the fly. In [New], $o \in \mathbb{H}$ is an abstract heap object created by `HeapAbs`. Rules [Assign], [Load] and [Store] for handling assignments, loads and stores, respectively, are standard. In [Throw], a throwable object o pointed by variable x is dispatched to its corresponding catch trap along the exception-catch links [7]. In [Call], a call to an instance method $x = a_0.f(a_1, ..., a_r)$ is analyzed. Here, $this^{m'}$, $p_i^{m'}$, and $ret^{m'}$ are the "this" variable, i-th parameter, and return variable of m', respectively, where m' is a target method resolved. In the conclusion of this rule, $ctx' \in \texttt{MethodCtx}(m')$ reveals how the contexts of a method are maintained. Initially, the contexts of all the entry methods are set to be empty, e.g., $\texttt{MethodCtx}(\text{"}\texttt{main}\text{"}) = \{[\,]\}$. To simulate Java's run-time

$$\frac{l : x = \text{new } T \quad o = \texttt{HeapAbs}(l, T) \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m)}{(o, \texttt{Sel}(o, ctx)) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad [\textsc{New}]$$

$$\frac{l : x = y \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m)}{\texttt{PTS}(y, \texttt{Sel}(y, ctx)) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad [\textsc{Assign}]$$

$$\frac{l : x = y.f \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad (o, htx) \in \texttt{PTS}(y, \texttt{Sel}(y, ctx))}{\texttt{PTS}(o.f, htx) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx))} \quad [\textsc{Load}]$$

$$\frac{l : x.f = y \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad (o, htx) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx))}{\texttt{PTS}(y, \texttt{Sel}(y, ctx)) \subseteq \texttt{PTS}(o.f, htx)} \quad [\textsc{Store}]$$

$$\frac{\begin{array}{c} l : \textbf{throw } x \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \\ (o, htx) \in \texttt{PTS}(x, \texttt{Sel}(x, ctx)) \quad l' : \textbf{catch } y = \texttt{ExceptionDispatch}(l, o) \end{array}}{(o, htx) \in \texttt{PTS}(y, \texttt{Sel}(y, ctx))} \quad [\textsc{Throw}]$$

$$\frac{\begin{array}{c} l : x = a_0.f(a_1, ..., a_r) \quad m = \texttt{MethodOf}(l) \quad ctx \in \texttt{MethodCtx}(m) \quad l' : \textbf{throw } t_l \in \texttt{Stmt}(m) \\ (o, htx) \in \texttt{PTS}(a_0, \texttt{Sel}(a_0, ctx)) \quad m' = \texttt{VirtualCallDispatch}(f, o) \quad ctx' = \texttt{Cons}(o, htx, l, ctx) \end{array}}{\begin{array}{c} ctx' \in \texttt{MethodCtx}(m') \quad (o, htx) \in \texttt{PTS}(this^{m'}, \texttt{Sel}(this^{m'}, ctx')) \\ \forall i \in [1, r] : \texttt{PTS}(a_i, \texttt{Sel}(a_i, ctx)) \subseteq \texttt{PTS}(p_i^{m'}, \texttt{Sel}(p_i^{m'}, ctx')) \\ \texttt{PTS}(eret^{m'}, \texttt{Sel}(eret^{m'}, ctx')) \subseteq \texttt{PTS}(t_l, \texttt{Sel}(t_l, ctx)) \quad \texttt{PTS}(ret^{m'}, \texttt{Sel}(ret^{m'}, ctx')) \subseteq \texttt{PTS}(x, \texttt{Sel}(x, ctx)) \end{array}} \quad [\textsc{Call}]$$

Figure 3 Rules for defining Qilin's parameterized pointer analysis.

semantic for re-throwing the exception objects (not handled in m') caused by its special catch statement "**catch** $eret'$", we associate a unique throw statement, $l' : \textbf{throw } t_l$, with the callsite l, where t_l is a special local variable in m for receiving these exception objects.

3.3 A High-Performance Incremental Worklist-based Solver

Doop [8] is declarative and uses a Datalog engine, e.g., Soufflé [20, 39] to compute the points-to facts for a pointer analysis. As Qilin is imperative, we have developed a new high-performance worklist-based constraint solver (depicted at ④ in Figure 2), which is currently single-threaded, for performing fine-grained context-sensitive pointer analyses (including variable-level context-sensitive pointer analyses in its full generality). When designing and implementing this constraint solver, we leverage an incremental worklist algorithm [23] suggested originally for resolving Andersen's context-insensitive pointer analysis [2]. However, we would like to stress that the basic algorithm used in our incremental worklist-based constraint solver is new, since variable-level context-sensitive pointer analyses require some points-to facts to be propagated in a way that does not exist traditionally before, as highlighted in Theorem 1. We will prove its correctness in Theorem 2 and illustrate its key part in supporting fine-grained context-sensitivity by using two examples (Tables 2 and 3).

Given a program P, we write `EntryOf(P)` to represent the set of its entry methods (including `main()`). To compute the points-to set $\texttt{PTS}(p, c)$ iteratively, where $p \in \mathbb{V} \cup \mathbb{H} \times \mathbb{F}$ and $c \in \mathbb{C}$, according to the rules given in Figure 3, we use four additional sets to represent four other kinds of context-sensitive facts that are also computed iteratively: (1) `PAG` (containing the currently discovered constraints expressed in terms of the assign, load and store edges in a PAG (Pointer Assignment Graph [23]), (2) `CALL` (containing the currently discovered call statements), (3) `THROW` (containing the currently discovered throw statements), and (4) `RM` (containing the (transitively) reachable methods found from `EntryOf(P)` so far).

For each of these five sets, denoted IS, we represent it as an *incremental set* [23] by dividing it into an "old" part ($\texttt{IS}_{\text{old}}$) and a "new" part ($\texttt{IS}_{\text{new}}$), so that $\texttt{IS} = \texttt{IS}_{\text{old}} \cup \texttt{IS}_{\text{new}}$, denoted also $\langle \texttt{IS}_{\text{old}}, \texttt{IS}_{\text{new}} \rangle$. At some point during the current iteration of an incremental worklist algorithm, `FlushNew(IS)` is called to flush $\texttt{IS}_{\text{new}}$ into $\texttt{IS}_{\text{old}}$, meaning that $\texttt{IS}_{\text{old}} \leftarrow \texttt{IS}_{\text{old}} \cup \texttt{IS}_{\text{new}}$ and $\texttt{IS}_{\text{new}} \leftarrow \varnothing$ are performed sequentially in that order. For notational convenience, we will write $S \overset{\cup}{\leftarrow} T$ as a shorthand for $S \leftarrow S \cup T$, where S and T are sets.

When computing the points-to facts iteratively for a pointer analysis, generalizing from method-level to variable-level context-sensitivity introduces one additional subtlety as summarized below, affecting the design of an incremental worklist-based constraint solver.

▶ **Theorem 1.** *Let method m (containing variable v) be analyzed under a new context c. Then $\mathit{PTS}(v, \mathit{Sel}(v,c))_{\text{old}} = \varnothing$ always holds under method-level context-sensitivity but $\mathit{PTS}(v, \mathit{Sel}(v,c))_{\text{old}} \neq \varnothing$ may hold under variable-level context-sensitivity.*

Proof. Let c' be a context under which m was analyzed earlier. Under method-level context-sensitivity, $\texttt{Sel}(v,c) = c$ and $\texttt{Sel}(v,c') = c'$. As $c \neq c'$, $\texttt{Sel}(v,c) \neq \texttt{Sel}(v,c')$. Hence, $\texttt{PTS}(v, \texttt{Sel}(v,c))_{\text{old}} = \varnothing$ (as $\langle v, \texttt{Sel}(v,c) \rangle$ is new and has never been analyzed before). Under variable-level context-sensitivity, $\texttt{Sel}(v,c) = \texttt{Sel}(v,c')$ may hold. Thus, $\texttt{PTS}(v, \texttt{Sel}(v,c))_{\text{old}} \neq \varnothing$ can hold if $\langle v, \texttt{Sel}(v,c) \rangle$ has been already analyzed earlier. ◀

Our incremental worklist-based constraint solver, given in Algorithm 1, takes a program P as input and applies the rules in Figure 3 to compute PTS as output, by performing both call graph construction and exception analysis on the fly. During the initialization (lines 1-5), RM is initialized with the entry methods in $\texttt{EntryOf}(P)$. `ProcessStmts` (lines 46-62) is called to initialize PTS, PAG, THROW and CALL with the points-to facts, three kinds of PAG edges, throw statements, and call statements found in these entry methods, respectively. Note that in line 61, l' : **throw** t_l is used for handling the exception objects thrown at callsite l, as discussed in Section 3.2. At this stage, the worklist W contains all the variables initialized to point to all the newly created objects (lines 48-50 ([New]) and lines 22-25).

The main loop (lines 6-20) discovers the points-to facts in the program iteratively. During each iteration, W contains a set of context-sensitive pointers p (variables or object fields) whose newly found points-to facts in $\texttt{PTS}(p)_{\text{new}}$ need to be propagated to their successors in PAG (i.e., $\texttt{PAG}_{\text{old}} \cup \texttt{PAG}_{\text{new}}$). Due to Theorem 1, however, for some context-sensitive pointer q that no longer appears in W, such that $o \in \texttt{PTS}(q)_{\text{old}}$ and f is a field of o, we may also need to propagate the points-to facts in $\texttt{PTS}(q)_{\text{old}}$ and/or $\texttt{PTS}(o.f)_{\text{old}}$ to their newly found successors in $\texttt{PAG}_{\text{new}}$ or from some newly found predecessors of $o.f$ in $\texttt{PAG}_{\text{new}}$ to $\texttt{PTS}(o.f)_{\text{new}}$.

During each iteration of the main loop (lines 6-20), we remove one such pointer *curr* from W and perform a four-step points-to fact propagation in the current iteration as follows:

- **Step 1: Resolving Direct Constraints (lines 8-11).** We resolve direct constraints by applying two rules in Figure 3: [Assign] (lines 8-9) and [Throw] (lines 10-11). In handling an outgoing assign edge at *curr*, we propagate the new points-to facts in $\texttt{PTS}(\mathit{curr})_{\text{new}}$ to the successor of *curr* along this outgoing assign edge (lines 22-25). In handling a throw for *curr*, a thrown object is passed to the variable in its exception handler (lines 26-29).
- **Step 2: Resolving Indirect Constraints (lines 12-15).** We resolve indirect constraints by applying also two rules in Figure 3: [Load] (lines 12-13) and [Store] (lines 14-15). In handling load and store edges (lines 30-37), new assign edges are introduced in PAG (lines 33 and 37) to make their underlying assignment semantics explicit.
- **Step 3: Collecting New Constraints (lines 16-19).** We discover new reachable methods by first calling `HandleCall` to analyze the calls with *curr* as their receiver variable ([Call]) and then adding the new constraints for these new methods by calling

Algorithm 1 Qilin's incremental worklist-based constraint solver (implemented using incremental sets [23]) for supporting pointer analyses with fine-grained context-sensitivity.

Input: P // Input program
Output: PTS // Points-to Sets
$\forall (v, c) \in \mathbb{V} \times \mathbb{C} : \texttt{PTS}(v, \texttt{Sel}(v, c)) \leftarrow \langle \varnothing, \varnothing \rangle$
$\texttt{THROW} \leftarrow \texttt{CALL} \leftarrow \texttt{PAG} \leftarrow \langle \varnothing, \varnothing \rangle$
$\texttt{W} \leftarrow \varnothing$
$\texttt{RM} \leftarrow \langle \varnothing, \{\langle m_e, [\,] \rangle \mid m_e \in \texttt{EntryOf}(P)\} \rangle$
ProcessStmts ()
while $\texttt{W} \neq \varnothing$ **do**
  $curr \leftarrow \texttt{Poll(W)}$ // $curr \in (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$
  /* Step 1: Resolving Direct Constraints */
  for $curr \xrightarrow{\text{assign}} \langle x, c \rangle \in \texttt{PAG}$ **do**
    $\texttt{PropPTS}(\langle x, c \rangle, \texttt{PTS}(curr)_{\text{new}})$
  for $\langle l : \textbf{throw } y, c \rangle \in \texttt{THROW}, s.t., curr = \langle y, \texttt{Sel}(y, c) \rangle$ **do**
    $\texttt{HandleThrow}(l, curr, \text{“new”})$
  /* Step 2: Resolving Indirect Constraints */
  for $curr \xrightarrow{\text{load[f]}} \langle x, c \rangle \in \texttt{PAG}$ **do**
    $\texttt{HandleLoad}(curr \xrightarrow{\text{load[f]}} \langle x, c \rangle, \text{“new”})$
  for $\langle x, c \rangle \xrightarrow{\text{store[f]}} curr \in \texttt{PAG}$ **do**
    $\texttt{HandleStore}(\langle x, c \rangle \xrightarrow{\text{store[f]}} curr, \text{“new”})$
  /* Step 3: Collecting New Constraints */
  for $\langle l : x = a_0.f(\cdots), c \rangle \in \texttt{CALL}, s.t., curr = \langle a_0, \texttt{Sel}(a_0, c) \rangle$ **do**
    $\texttt{HandleCall}(l : x = a_0.f(\cdots), c, \text{“new”})$
  ProcessStmts()
  FlushNew(PTS($curr$))
  /* Step 4: Activating New Constraints */
  ActivateConstraints()
return PTS
Function PropPTS($\langle x, c \rangle, s$):
  if $\exists\ \langle o, htx \rangle \in s \setminus \texttt{PTS}(x, c)$ **then**
    $\texttt{PTS}(x, c)_{\text{new}} \xleftarrow{\cup} s \setminus \texttt{PTS}(x, c)$
    $\texttt{W} \xleftarrow{\cup} \{\langle x, c \rangle\}$
Function HandleThrow($l, \langle y, c \rangle, i$):
  for $\langle o, htx \rangle \in \texttt{PTS}(y, c)_i$ **do**
    $l' : \textbf{catch } x \leftarrow \texttt{ExceptionDispatch}(l, o)$
    $\texttt{PropPTS}(\langle x, \texttt{Sel}(x, c) \rangle, \{\langle o, htx \rangle\})$
Function HandleLoad($\langle y, c \rangle \xrightarrow{\text{load[f]}} \langle x, c' \rangle, i$):
  for $\langle o, htx \rangle \in \texttt{PTS}(y, c)_i$ **do**
    if $\langle o.f, htx \rangle \xrightarrow{\text{assign}} \langle x, c' \rangle \notin \texttt{PAG}$ **then**
      $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle o.f, htx \rangle \xrightarrow{\text{assign}} \langle x, c' \rangle\}$
Function HandleStore($\langle y, c \rangle \xrightarrow{\text{store[f]}} \langle x, c' \rangle, i$):
  for $\langle o, htx \rangle \in \texttt{PTS}(x, c')_i$ **do**
    if $\langle y, c \rangle \xrightarrow{\text{assign}} \langle o.f, htx \rangle \notin \texttt{PAG}$ **then**
      $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle y, c \rangle \xrightarrow{\text{assign}} \langle o.f, htx \rangle\}$
Function HandleCall($l : x = a_0.f(a_1, \cdots, a_r), c, i$):
  for $\langle o, htx \rangle \in \texttt{PTS}(a_0, \texttt{Sel}(a_0, c))_i$ **do**
    $m' \leftarrow \texttt{VirtualCallDispatch}(f, o),\ c' \leftarrow \texttt{Cons}(o, htx, l, c)$
    $\texttt{PropPTS}(\langle this^{m'}, \texttt{Sel}(this^{m'}, c') \rangle, \{\langle o, htx \rangle\})$
    $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle ret^{m'}, \texttt{Sel}(ret^{m'}, c') \rangle \xrightarrow{\text{assign}} \langle x, \texttt{Sel}(x, c) \rangle\}$
      $\cup\ \{\langle eret^{m'}, \texttt{Sel}(eret^{m'}, c') \rangle \xrightarrow{\text{assign}} \langle t_l, \texttt{Sel}(t_l, c) \rangle\}$
      $\cup\ \{\langle a_i, \texttt{Sel}(a_i, c) \rangle \xrightarrow{\text{assign}} \langle p_i, \texttt{Sel}(p_i, c') \rangle \mid i \in [1, r]\}$
    $\texttt{RM}_{\text{new}} \xleftarrow{\cup} \{\langle m', c' \rangle\} \setminus \texttt{RM}$
Function ProcessStmts():
  for $\langle m, c \rangle \in \texttt{RM}_{\text{new}}$ **do**
    for $l : x = \textbf{new } T \in \texttt{Stmt}(m)$ **do**
      $o = \texttt{HeapAbs}(l, T)$
      $\texttt{PropPTS}(\langle x, \texttt{Sel}(x, c) \rangle, \{\langle o, \texttt{Sel}(o, c) \rangle\})$
    for $l : x = y \in \texttt{Stmt}(m)$ **do**
      $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle y, \texttt{Sel}(y, c) \rangle \xrightarrow{\text{assign}} \langle x, \texttt{Sel}(x, c) \rangle\} \setminus \texttt{PAG}$
    for $l : x = y.f \in \texttt{Stmt}(m)$ **do**
      $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle y, \texttt{Sel}(y, c) \rangle \xrightarrow{\text{load[f]}} \langle x, \texttt{Sel}(x, c) \rangle\} \setminus \texttt{PAG}$
    for $l : x.f = y \in \texttt{Stmt}(m)$ **do**
      $\texttt{PAG}_{\text{new}} \xleftarrow{\cup} \{\langle y, \texttt{Sel}(y, c) \rangle \xrightarrow{\text{store[f]}} \langle x, \texttt{Sel}(x, c) \rangle\} \setminus \texttt{PAG}$
    for $l : \textbf{throw } x \in \texttt{Stmt}(m)$ **do**
      $\texttt{THROW}_{\text{new}} \xleftarrow{\cup} \{\langle l : \textbf{throw } x, c \rangle\}$
    for $l : x = a_0.f(a_1, \cdots, a_r) \in \texttt{Stmt}(m)$ **do**
      $\texttt{CALL}_{\text{new}} \xleftarrow{\cup} \{\langle l : x = a_0.f(a_1, \cdots, a_r), c \rangle\}$
      $\texttt{THROW}_{\text{new}} \xleftarrow{\cup} \{\langle l' : \textbf{throw } t_l, c \rangle\}$
  FlushNew(RM)
Function ActivateConstraints():
  while $\texttt{CALL}_{\text{new}} \neq \varnothing$ **do**
    for $\langle l : x = a_0.f(a_1, \cdots, a_r), c \rangle \in \texttt{CALL}_{\text{new}}$ **do**
      $\texttt{HandleCall}(l : x = a_0.f(a_1, \cdots, a_r), c, \text{“old”})$
    FlushNew(CALL)
    ProcessStmts()
  for $\langle l : \textbf{throw } y, c \rangle \in \texttt{THROW}_{\text{new}}$ **do**
    $\texttt{HandleThrow}(l, \langle y, \texttt{Sel}(y, c) \rangle, \text{“old”})$
  FlushNew(THROW)
  for $\langle y, c \rangle \xrightarrow{\text{load[f]}} \langle x, c' \rangle \in \texttt{PAG}_{\text{new}}$ **do**
    $\texttt{HandleLoad}(\langle y, c \rangle \xrightarrow{\text{load[f]}} \langle x, c' \rangle, \text{“old”})$
  for $\langle y, c \rangle \xrightarrow{\text{store[f]}} \langle x, c' \rangle \in \texttt{PAG}_{\text{new}}$ **do**
    $\texttt{HandleStore}(\langle y, c \rangle \xrightarrow{\text{store[f]}} \langle x, c' \rangle, \text{“old”})$
  for $\langle y, c \rangle \xrightarrow{\text{assign}} \langle x, c' \rangle \in \texttt{PAG}_{\text{new}}$ **do**
    $\texttt{PropPTS}(\langle x, c' \rangle, \texttt{PTS}(y, c)_{\text{old}})$
  FlushNew(PAG)
Function FlushNew(IS):
  $\langle \texttt{IS}_{\text{old}}, \texttt{IS}_{\text{new}} \rangle \leftarrow \langle \texttt{IS}_{\text{old}} \cup \texttt{IS}_{\text{new}}, \varnothing \rangle$

`ProcessStmts`. In lines 16-17, we process every call $\langle l : x = a_0.f(a_1, \cdots, a_r), c\rangle$ in `CALL` (i.e., both $\texttt{CALL}_{\text{old}}$ and $\texttt{CALL}_{\text{new}}$), where $curr = \langle a_0, \texttt{Sel}(a_0, c)\rangle$. In handling such a call (lines 38-45), new assign edges are introduced (lines 42-44) for modeling parameter passing, and in addition, new reachable methods are recorded in $\texttt{RM}_{\text{new}}$ (line 45). Note that $this^{m'}$ is handled (line 41) differently from the other parameters $p_1, ..., p_r$ (lines 42-44). A receiver object in a_0 flows only to the method dispatched on itself while the objects pointed to by the other arguments $a_1, ..., a_r$ flow to $p_1, ..., p_r$, respectively, for all the methods dispatched with a_0 as its receiver variable. For the new reachable methods just found, new constraints are added with `PTS`, `PAG`, `THROW` and `CALL` being updated (lines 46-62). Finally, in line 19, $\texttt{PTS}(curr)_{\text{new}}$ is flushed into $\texttt{PTS}(curr)_{\text{old}}$ as $curr$ has been processed.

- **Step 4: Activating New Constraints (line 20).** `ActivateConstraints` (lines 63-78) is called to initiate the points-to fact propagation across the new constraints in $\texttt{CALL}_{\text{new}}$, $\texttt{THROW}_{\text{new}}$ and $\texttt{PAG}_{\text{new}}$ found in Steps 2 – 3 by using the points-to facts in the "*old*" parts of the relevant pointers involved in these constraints. In lines 64-68, we process the calls in $\texttt{CALL}_{\text{new}}$ in turn by discovering more new reachable methods at every call $\langle l : x = a_0.f(a_1, \cdots, a_r), c\rangle$ in $\texttt{CALL}_{\text{new}}$ (by using $\texttt{PTS}(a_0, \texttt{Sel}(a_0, c))_{\text{old}}$ (line 66)) and adding the new constraints for the new reachable methods found (line 68) just after $\texttt{CALL}_{\text{new}}$ is flushed into $\texttt{CALL}_{\text{old}}$ (line 67). According to Theorem 1, lines 64-75 (shaded in blue) are needed to support variable-level context-sensitivity, as demonstrated by two examples below. Note that lines 76-77 are needed even in a context-insensitive analysis in order to support the on-the-fly call graph construction (among others) by activating the points-to propagation from the "old" part of an argument to its corresponding parameter in a newly found callee across its argument-to-parameter assign edge (line 44).

▶ **Theorem 2.** *Algorithm 1 computes the context-sensitive points-to information in a program exactly according to the rules given in Figure 3.*

Proof. We prove that Algorithm 1 computes the points-to facts according to the pointer analysis algorithm given in Figure 3 in the same manner. Thus, once a fixed point is reached, Algorithm 1 produces exactly the same points-to facts as the rules given in Figure 3.

- We first argue that during each iteration of Algorithm 1, one context-sensitive pointer $n \in (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$ is removed from `W` and the objects in $\texttt{PTS}(n)_{\text{new}}$ are handled in exactly the same manner as in Figure 3: Step 1 handles [Assign] (lines 8-9), and [Throw] (lines 10-11). Step 2 handles [Load] (lines 12-13), and [Store] (lines 14-15). Step 3 handles [Call] (lines 16-17) and extends `PAG` with the newly reachable methods (line 18). Whenever an object allocation statement is visited, [New] is handled immediately (lines 48-50). Steps 2 and 3 serve only to add the newly discovered assign edges (constraints) to `PAG` without performing the actual points-to fact propagation. Step 4 activates these new constraints (lines 76-77). To support variable-level context-sensitivity according to Theorem 1, lines 64-75 (in blue) are added to activate also the newly reachable calls in $\texttt{CALL}_{\text{new}}$, throw statements in $\texttt{THROW}_{\text{new}}$, and loads/stores in $\texttt{PAG}_{\text{new}}$.
- We then argue that at the start of each iteration of Algorithm 1, $\forall n \in (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$: $n \in \texttt{W} \iff \texttt{PTS}(n)_{\text{new}} \neq \emptyset$. "$\Longrightarrow$" is trivial by noting that n can only be added into `W` in line 25 and $\texttt{PTS}(n)_{\text{new}} \neq \emptyset$ due to line 23. We prove "$\Longleftarrow$" by induction. Initially, only the LHS of each allocation statement in the entry methods is added into `W` (line 5 and lines 48-50). Thus, "$\Longleftarrow$" holds. Given the induction hypothesis that "$\Longleftarrow$" holds at the start of the i-th iteration, we prove that "$\Longleftarrow$" still holds at the start of the $(i+1)$-th iteration. During the i-th iteration, only $curr$ is removed from `W` at the start of the iteration and its points-to facts in $\texttt{PTS}(curr)_{\text{new}}$ have been flushed in line 19 after they have been handled. All the pointers that are added into `W` during this iteration must be added by `PropPTS`, ensuring that their new points-to facts are not empty.

By combining the two proof steps above, we conclude that for every n in $(\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C}$, Algorithm 1 handles n in exactly the same manner as in Figure 3 for all the objects in $\texttt{PTS}(n)$. As a result, Algorithm 1 produces exactly the same points-to facts as Figure 3. ◀

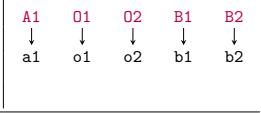
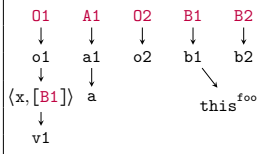
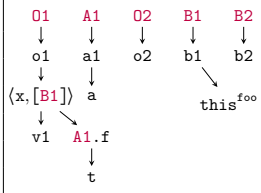
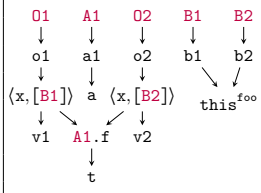
Below we use two small example programs to illustrate how our worklist-based constraint solver (Algorithm 1) works in supporting pointer analyses with variable-level context-sensitivity. In addition, we will also highlight the two subtleties involved (one in each example) in designing this new constraint solver for computing the points-to facts iteratively.

In the first example, we explain how our constraint solver works in computing the points-to facts for the program in Figure 1, by applying the 1-object-sensitive pointer analysis with variable-level context-sensitivity discussed in Section 2.3 under which only the parameter `x` of `foo()` is analyzed context-sensitively. We would like to stress the significance of Theorem 1 by highlighting the necessity of lines 64-75 (shaded in blue) in supporting variable-level context-sensitivity. As `foo()` is called under two different receiver objects in lines 13 and 16, `x` will be qualified by either `[B1]` or `[B2]`. Every other variable/object p, which is analyzed context-insensitively, is identified by $\langle p, [\,] \rangle$, which will be abbreviated to p for brevity. Therefore, whenever we write $\texttt{PTS}(p)$ without providing a context, we mean $\texttt{PTS}(p, [\,])$.

Table 2 traces one particular execution of Algorithm 1, showing how `PTS`, `RM`, `CALL`, `PAG`, and `W` are updated incrementally in a total of 17 iterations (with its initialization assumed to start at 0). For this simple program. `THROW` is not relevant. To save space, we have segmented these 17 iterations into six groups. For each group, we start with `W` being given in the preceding group at the beginning of its first iteration and produce the results obtained for `PTS`, `RM`, `CALL`, `PAG`, and `W` at the end of its last iteration. For `PTS`, we list explicitly both the "old" and "new" parts for all its variables and fields. For `RM`, `CALL`, and `PAG`, we list only their "new" parts as their "old" parts can be read-off easily from the earlier iterations given.

- **Iteration 0**. Initially, we perform the initialization (lines 1-5) by taking `main()` as the only entry of the program. Then we compute the points-to information iteratively during all the iterations of the `while` loop in line 6 (i.e., Iterations 1–16).
- **Iterations 1-3**. We start with the worklist `W` given at Iteration 0 and then obtain the updated results as shown after `o1`, `o2` and `a1` have been processed during these three iterations. At this point, $\texttt{CALL}_{\text{new}}$ has already been flushed into $\texttt{CALL}_{\text{old}}$ so that $\texttt{CALL} = \langle \texttt{CALL}_{\text{old}}, \texttt{CALL}_{\text{new}} \rangle = \langle \{\langle \texttt{line 13}, [\,] \rangle, \langle \texttt{line 16}, [\,] \rangle\}, \emptyset \rangle$.
- **Iteration 4**. We start with *curr* = "`b1`", where $\texttt{PTS(b1)}_{\text{new}} = \{\texttt{B1}\}$, indicating that we are just about to analyze `foo()` under context `[B1]` (invoked in line 13 in the program). In Step 3, a total of five edges are added to $\texttt{PAG}_{\text{new}}$. The three new assign edges are introduced for modeling parameter passing for this particular call (two for the two parameters `x` and `a`, and one for the return variable `x`). In addition, the store edge and the load edge are introduced for representing the two statements in `foo()`. At the end of Step 3, $\texttt{PTS(b1)}_{\text{new}}$ is flushed into $\texttt{PTS(b1)}_{\text{old}}$. In Step 4, the propagation into the two parameters, $\langle \texttt{x}, [\texttt{B1}] \rangle$ and `a`, from their corresponding actual arguments is made (lines 76-77).
- **Iterations 5-10**. We propagate the newly found points-to information by processing `a`, $\langle \texttt{x}, [\texttt{B1}] \rangle$, and $\texttt{this}^{\texttt{foo}}$ in `W` and the others added later to `W` iteratively.
- **Iteration 11**. We start with *curr* = "`b2`", where $\texttt{PTS(b2)}_{\text{new}} = \{\texttt{B2}\}$, so that we can analyze `foo()` under context `[B2]` (invoked in line 16 in the program). We proceed exactly as when `foo()` is analyzed under context `[B1]` at iteration 4 except that we no longer need to introduce $\texttt{a1} \xrightarrow{\text{assign}} \texttt{a}$ and $\texttt{a} \xrightarrow{\text{load[f]}} \texttt{t}$ in Step 3, since both edges already exist in `PAG`. Before Step 4 starts, $\texttt{PTS(b2)}_{\text{new}}$ has already been flushed into $\texttt{PTS(b2)}_{\text{old}}$. In Step 4,

Table 2 Tracing a particular execution of Algorithm 1 in applying the fine-grained 1-object-sensitive pointer analysis discussed in Section 2.3 to the program given in Figure 1.

Iters	$\mathbb{V}$	PTS_{old}	PTS_{new}	RM_{new}	CALL_{new}	PAG_{new}	W	Points-to Fact Propagation
0	o1 o2 b1 b2 a1	∅ ∅ ∅ ∅ ∅	{O1} {O2} {B1} {B2} {A1}	{⟨main, [ ]⟩}	{⟨line 13, [ ]⟩, ⟨line 16, [ ]⟩}	∅	{o1, o2, b1, b2, a1}	
1-3	o1 o2 b1 b2 a1	{O1} {O2} ∅ ∅ {A1}	∅ ∅ {B1} {B2} ∅	∅	∅	∅	{b1, b2}	SAME AS ABOVE
4	o1 o2 b1 b2 a1 this^{foo} ⟨x, B1⟩ a	{O1} {O2} {B1} ∅ {A1} ∅ ∅ ∅	∅ ∅ ∅ {B2} ∅ {B1} {O1} {A1}	{⟨foo, [B1]⟩}	{⟨line 7, [B1]⟩}	{o1 $\xrightarrow{\text{assign}}$ ⟨x, [B1]⟩, a1 $\xrightarrow{\text{assign}}$ a, ⟨x, [B1]⟩ $\xrightarrow{\text{assign}}$ v1, ⟨x, [B1]⟩ $\xrightarrow{\text{store[f]}}$ a, a $\xrightarrow{\text{load[f]}}$ t}	{b2, a, ⟨x, [B1]⟩, this^{foo}}	
5-10	o1 o2 b1 b2 a1 v1 this^{foo} ⟨x, [B1]⟩ a A1.f t	{O1} {O2} {B1} ∅ {A1} {O1} {B1} {O1} {A1} {O1} {O1}	∅ ∅ ∅ {B2} ∅ ∅ ∅ ∅ ∅ ∅ ∅	∅	∅	{⟨x, [B1]⟩ $\xrightarrow{\text{assign}}$ A1.f, A1.f $\xrightarrow{\text{assign}}$ t}	{b2}	
11	o1 o2 b1 b2 a1 v1 this^{foo} ⟨x, [B1]⟩ ⟨x, [B2]⟩ a A1.f t	{O1} {O2} {B1} {B2} {A1} {O1} {B1} {O1} ∅ {A1} {O1} {O1}	∅ ∅ ∅ ∅ ∅ ∅ {B2} ∅ {O2} ∅ {} ∅	{⟨foo, [B2]⟩}	{⟨line 7, [B2]⟩}	{o2 $\xrightarrow{\text{assign}}$ ⟨x, [B2]⟩, ⟨x, [B2]⟩ $\xrightarrow{\text{assign}}$ v2, ⟨x, [B2]⟩ $\xrightarrow{\text{store[f]}}$ a ⟨x, [B2]⟩ $\xrightarrow{\text{assign}}$ A1.f}	{this^{foo}, ⟨x, [B2]⟩}	
12-16	o1 o2 b1 b2 a1 v1 v2 this^{foo} ⟨x, [B1]⟩ ⟨x, [B2]⟩ a A1.f t	{O1} {O2} {B1} {B2} {A1} {O1} {O2} {B1, B2} {O1} {O2} {A1} {O1, O2} {O1, O2}	∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅	∅	∅	∅	∅	SAME AS ABOVE

the points-to information of $\langle$x, [B2]$\rangle$ is updated (lines 76-77). In addition, we discover $\langle$x, [B2]$\rangle \xrightarrow{\text{assign}} \langle$A1.f, [ ]$\rangle$ from $\langle$x, [B2]$\rangle \xrightarrow{\text{store[f]}} \langle$a, [ ]$\rangle$ in lines 74-75 (Theorem 1), since PTS(a, [ ])$_{\text{old}}$ = $\langle$A1, [ ]$\rangle$. Otherwise, O2 in PTS(t) will be missed unsoundly.

- **Iterations 12-16**. After these iterations have been done, a fixpoint will be reached.

In Algorithm 1, its `while` loop in line 64 is needed for supporting a number of different flavors of context-sensitivity. Next, we use a second example to explain its necessity for supporting k-callsite-based context-sensitive pointer analysis (i.e., kCFA [41]) with variable-level context-sensitivity. This loop is needed to avoid a rare case under which W = ∅ but CALL$_{\text{new}}$ ≠ ∅, implying that the new reachable methods in CALL$_{\text{new}}$ must still need to be analyzed to look for the new points-to facts despite the fact that the worklist W is empty.

In this second example given in Figure 4, three classes, A, B, and C, are defined, where `create()` and `wcreate()` in B and `wcreate2()` in C are used to establish a chain of method calls in the program, where `wcreate2()` is a wrapper method for `wcreate1()`, which is a wrapper method for `create()`. In `main()`, O1 is first allocated, then stored into `a1.f`, and finally, retrieved from `a1.f` and assigned to `v1`. Similarly, O2 is first allocated, then stored into `a2.f`, and finally, retrieved from `a2.f` and assigned to `v2`.

```
1 void main() {
2   C c1 = new C(); // C1
3   C c2 = c1;
4   A a1 = c1.wcreate2(); // c1
5   A a2 = c2.wcreate2(); // c2
6   Object o1 = new Object(); // O1
7   Object o2 = new Object(); // O2
8   a1.f = o1;
9   a2.f = o2;
10  Object v1 = a1.f;
11  Object v2 = a2.f;
12 }
13
14 class A { Object f; }
15 class B {
16   A create() {
17     A r1 = new A(); // A1
18     return r1; }
19   A wcreate() {
20     A r2 = this.create(); // c4
21     return r2;
22 }}
23 class C {
24   A wcreate2() {
25     B b1 = new B(); // B1
26     A r3 = b1.wcreate(); // c3
27     return r3;
28 }}
```

Figure 4 An example for illustrating the necessity of the `while` loop in line 64 of Algorithm 1 for supporting callsite-based context-sensitive pointer analyses with variable-level context-sensitivity.

Let us analyze this program by using 4CFA with variable-level context-sensitivity, under which `r1`, `r2`, `r3` and A1 are context-sensitive but all the remaining variables and objects are context-insensitive. In particular, as is usually done in practice [22, 14, 25], the length of a context for `r1`, `r2` and `r3` is limited by 4 and the length of a context for A1 is limited by 3.

If we apply our constraint solver (Algorithm 1) to solve this particular 4CFA-style pointer analysis for this program, its points-to facts will be computed soundly as desired. In particular, `v1` will be found to point to O1 and `v2` will be found to point to O2. However, if we apply our constraint solver with its line 64 being deleted, then the resulting modified constraint solver will be unsound. For this particular program, `a2` will be found point to no object at all, and consequently, that `v2` will also be regarded as pointing to no object at all.

Table 3 traces a particular execution of this modified constraint solver (in 15 iterations):

- **Iterations 0-2**. During the initialization, i.e., at Iteration 0 (lines 1-5), `main()` is the only entry method for the program. By processing its statements, we end up adding two new calls (for its lines 4-5) to CALL$_{\text{new}}$, five new edges (for its lines 3 and 8-11) to PAG$_{\text{new}}$ and {c1, o1, o2} (for its lines 2 and 6-7) to W. At iterations 1-2, o1 and o2 are handled by just flushing PTS($o1$)$_{\text{new}}$ and PTS($o2$)$_{\text{new}}$ into PTS($o1$)$_{\text{old}}$ and PTS($o2$)$_{\text{old}}$, respectively.

Table 3 Tracing a particular execution of Algorithm 1 *with its* `while` *loop in line 64* being deleted in applying 4CFA with variable-level context-sensitivity to the program given in Figure 4, under which all the variables and objects except `r1`, `r2`, `r3`, and `A1` are context-insensitive.

Iters	$\mathbb{V}$	PTS_{old}	PTS_{new}	RM_{new}	CALL_{new}	PAG_{new}	W
0-2	c1 o1 o2	∅ {O1} {O2}	{C1} ∅ ∅	{⟨main, [ ]⟩}	{⟨line 4, [ ]⟩, ⟨line 5, [ ]⟩}	{c1 $\xrightarrow{\text{assign}}$ c2 o1 $\xrightarrow{\text{store[f]}}$ a1 o2 $\xrightarrow{\text{store[f]}}$ a2 a1 $\xrightarrow{\text{load[f]}}$ v1 a2 $\xrightarrow{\text{load[f]}}$ v2}	{c1}
3-13	c1 o1 o2 c2 $\text{this}^{\text{wcreate2}}$ b1 $\text{this}^{\text{wcreate}}$ $\text{this}^{\text{create}}$ ⟨r1, [c4, c3, c1]⟩ ⟨r2, [c3, c1]⟩ ⟨r3, [c1]⟩ a1 ⟨A1.f, [c4, c3, c1]⟩ v1	{C1} {O1} {O2} ∅ {C1} {B1} {B1} {B1} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {O1} {O1}	∅ ∅ ∅ {C1} ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅	{⟨wcreate2, [c1]⟩ ⟨wcreate, [c3, c1]⟩ ⟨create, [c4, c3, c1]⟩}	{⟨line 26, [c1]⟩ ⟨line 20, [c3, c1]⟩}	{⟨r3, [c1]⟩ $\xrightarrow{\text{assign}}$ a1 ⟨r2, [c3, c1]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c1]⟩ ⟨r1, [c4, c3, c1]⟩ $\xrightarrow{\text{assign}}$ ⟨r2, [c3, c1]⟩ o1 $\xrightarrow{\text{assign}}$ ⟨A1.f, [c4, c3, c1]⟩ ⟨A1.f, [c4, c3, c1]⟩ $\xrightarrow{\text{assign}}$ v1}	{c2}
14	c1 o1 o2 c2 $\text{this}^{\text{wcreate2}}$ b1 $\text{this}^{\text{wcreate}}$ $\text{this}^{\text{create}}$ ⟨r1, [c4, c3, c1]⟩ ⟨r2, [c3, c1]⟩ ⟨r3, [c1]⟩ a1 ⟨A1.f, [c4, c3, c1]⟩ v1	{C1} {O1} {O2} {C1} {C1} {B1} {B1} {B1} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {⟨A1, [c4, c3, c1]⟩} {O1} {O1}	∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅	{⟨wcreate2, [c2]⟩ ⟨wcreate, [c3, c2]⟩}	{⟨line 26, [c2]⟩ ⟨line 20, [c3, c2]⟩}	{⟨r3, [c2]⟩ $\xrightarrow{\text{assign}}$ a2 ⟨r2, [c3, c2]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c2]⟩}	

- **Iterations 3-13**. We propagate the newly found points-to information by processing `c1` and others added later to `W` during these few iterations iteratively while keeping `c2` always in `W`. This particular execution order is possible, since the items (i.e., unprocessed pointers) in `W` are processed non-deterministically. At the end of Iteration 13, the points-to information for all the variables except `a2` and `v2` has been obtained.
- **Iteration 14**. We start with *curr* = "`c2`", where $\texttt{PTS(c2)}_{\text{new}}$ = {C1}, indicating that we are just about to analyze `wcreate2()` under context [c2] (invoked in line 5 in the program). In Step 3, ⟨r3, [c2]⟩ $\xrightarrow{\text{assign}}$ a2 is added to PAG_{new} and a new call in line 26 in the program invoked under context [c2] is discovered and added into CALL_{new}. We no longer add $\texttt{this}^{\texttt{wcreate2}}$ and `b1` to `W` as they do not point to any new points-to fact discovered. In Step 4, we handle the new call (line 26 in the program under [c2]) in CALL_{new} with $\texttt{PTS}(b1)_{\text{old}}$ = {B1} (lines 65-66) and find a new reachable method `wcreate()` under context [c3, c2]. Finally, we call `ProcessStmts` (line 68) to process the statements in this newly reachable method, establish ⟨r2, [c3, c2]⟩ $\xrightarrow{\text{assign}}$ ⟨r3, [c2]⟩, and discover one more call (line 20 in the program) under context [c3, c2] (highlighted in red). In the modified constraint solver, the `while` loop in line 64 (Algorithm 1) has been deleted. As `W` = ∅, this new call will not be processed during the next iteration. As a result, `a2` and `v2` will be concluded not to point to any object unsoundly. However, our constraint solver, with this `while` loop being used in line 64, works soundly.

3.4 Handling Complex Language Features

We have closely modeled the handling of complex Java features and semantic complexities after the logic in Doop [8] (as depicted at ⑤ in Figure 2), so that Qilin achieves exactly the same precision for a program as Doop except for a few tool-specific variables introduced.

We have modeled the system/main thread groups and main thread to identify a variety of the entry methods of a Java program (line 4 of Algorithm 1), Java's reference objects (e.g., WeakReference and SoftReference) and reference queues, and class initialization. For example, JVM will register reference objects to reference queues by calling `Finalizer.register()` so that `finalize()` methods can be invoked. In addition, JVM will initialize classes/interfaces by calling their static initializers, `<clinit>()`. To handle such implicit calls by JVM, we dynamically inject static calls into the body of a `FakeMain()` method (regarded as an entry of the program) to simulate their behavior during the pointer analysis for a given Java program.

To model a native method, we have designed a *handler* to simulate its semantics by generating a method body in Jimple [54], the IR (Intermediate Representation) for Soot. Currently, Qilin handles the same set of native methods (e.g., `thread:start()`, `DoPrivileged()` and `clone()`) supported by Doop in exactly the same way.

As for Java reflection, Doop can handle it either statically or dynamically (by relying on the reflective targets found by Tamiflex [6]). We have taken the latter approach since it has been used exclusively in the pointer analysis community in the past few years [48, 25, 30, 14]. Qilin's reflection handler supports exactly the same set of the most commonly used Java reflection APIs (e.g., `ClassForName()` and `ClassNewInstance()`) as in Doop.

Qilin, as in Doop, handles cast and assignment compatibility by using the declared type of a variable to filter out type-incompatible pointed-to objects during the pointer analysis. As is standard, arrays are considered monolithic (without distinguishing their elements). In particular, we filter out type-incompatible objects stored in an array by using the declared type of its elements instead of `java.lang.Object`.

In Qilin, we handle static fields and static methods in the standard manner. As global variables, static fields are analyzed context-insensitively. A static method `m()` is modeled as a special instance method by just interpreting a call to `m()` as `this.m()` and proceeding as if it were an instance method defined in `java.lang.Object`. As a result, static methods can be analyzed uniformly under all flavors of context-sensitivity except for hybrid context-sensitivity [22], under which static and virtual calls are distinguished.

4 Using the Qilin Framework

We first describe a few significant instantiations of `Cons` (Section 4.1), `Sel` (Section 4.2), and `HeapAbs` (Section 4.3), respectively. We then combine these instantiations to obtain the pointer analyses provided in Qilin's toolbox depicted at ① in Figure 2 (Section 4.4).

Given a context $c = [e_1, \cdots, e_n]$ and a context element e_0, we write $e_0 \mathbin{+\!\!+} c$ to denote $[e_0, e_1, \cdots, e_n]$, and $\lceil c \rceil_k$ for $[e_1, \cdots, e_k]$ (i.e., c restricted to its prefix of length k).

4.1 Context Constructors

We instantiate $\texttt{Cons}(o, htx, l, ctx)$ used in [Call] (Figure 3) to define five common types of contexts for a method ("Insens", "Callsite", "Object", "Type", and "Hybrid" listed at ② in Figure 2). Note that in our framework, k-limiting will be applied by `Sel`.

- Insens. For a context-insensitive pointer analysis [2, 23], all methods are analyzed under the same fixed empty context (without distinguishing their calling contexts):

$$\texttt{Cons}_{\text{INSENS}}(o, htx, l, ctx) = [] \tag{1}$$

- Callsite. A callsite-sensitive pointer analysis [40], known also as *control-flow analysis* (CFA) [41], uses a callsite l as a context element. Therefore, the context constructor is:

$$\texttt{Cons}_{\text{CFA}}(o, htx, l, ctx) = l +\!\!+ ctx \tag{2}$$

- Object. An object-sensitive pointer analysis [32, 33] uses a receiver object o as a context element. Thus, the context constructor simply becomes:

$$\texttt{Cons}_{\text{OBJ}}(o, htx, l, ctx) = o +\!\!+ htx \tag{3}$$

- Type. A type-sensitive pointer analysis [42], which is a more scalable but less precise alternative of an object-sensitive pointer analysis, resorts to the class type containing the method where a receiver object o is allocated, denoted as $\texttt{TypeContg}(o)$. Thus, we have:

$$\texttt{Cons}_{\text{TYPE}}(o, htx, l, ctx) = \texttt{TypeContg}(o) +\!\!+ htx \tag{4}$$

- Hybrid. A hybrid pointer analysis [22] distinguishes static and dynamic call sites:

$$\texttt{Cons}_{\text{HYB}}(o, htx, l, ctx) = \begin{cases} o +\!\!+ htx & l \notin \texttt{SC} \\ \texttt{car}(ctx) +\!\!+ l +\!\!+ \texttt{cdr}(ctx) & l \in \texttt{SC} \end{cases} \tag{5}$$

where SC is the set of all static call sites in the program. Here, `car` and `cdr` are standard, with `car` pulling the first element of a list and `cdr` returning the list without the `car`. For a non-static callsite, the new context is constructed identically as in Equation (3). For a static callsite, the new context also includes the invocation site, which has been shown to be effective in improving the precision of pointer analysis [22].

4.2 Context Selectors

It is simple to instantiate a context selector `Sel` to support both method-level and variable-level context-sensitivity, including "Uniform", "Heuristic", "Selective", and "Partial" listed at ② in Figure 2. Given a calling context for a method, a context selector `Sel` picks some of its context elements to define the contexts for the variables/objects in the method by applying k-limiting [41], where k can vary across the variables/objects in the same method.

- Uniform. To support traditional method-level context-sensitive pointer analyses that rely on k-limited context abstractions [40, 32, 42, 22], a "uniform" context selector is used:

$$\texttt{Sel}_{\mathcal{U}}(ctx, e) = \begin{cases} \lceil ctx \rceil_k & e \in \mathbb{V} \\ \lceil ctx \rceil_{hk} & e \in \mathbb{H} \end{cases} \tag{6}$$

where the local variables (objects allocated) in a method adopt its calling context ctx uniformly under k-limiting (hk-limiting). In practice, $hk = k - 1$ is often used.

- Heuristic. For efficiency reasons at little loss of precision, the objects of certain types from an empirically determined set, $\mathcal{T}$, are usually analyzed context-insensitively as in Doop [8] and Wala [16], where the following definition of $\mathcal{T}$ is the most popularly used:

$$\mathcal{T} = \{\texttt{StringBuffer}, \texttt{StringBuilder}, \texttt{Throwable}\} \tag{7}$$

Let $\texttt{SubTypeOf}(\mathcal{T})$ be the set including the types in $\mathcal{T}$ and their subtypes. Let $\texttt{TypeOf}(o)$ be the dynamic type of an object o. A "*heuristic*" context selector is given by:

$$\texttt{Sel}_{\mathcal{H}}(ctx, e) = \begin{cases} [\,] & e \in \mathbb{H} \wedge \texttt{TypeOf}(e) \in \texttt{SubTypeOf}(\mathcal{T}) \\ \texttt{Sel}_{\mathcal{U}}(ctx, e) & \text{otherwise} \end{cases} \tag{8}$$

- Selective. Under "*selective*" method-level context-sensitivity [43, 25], only a subset of (precision-critical) methods, CSML, in the program is selected to be analyzed context-sensitively:

$$\texttt{Sel}_{\mathcal{S}}(ctx, e) = \begin{cases} [\,] & \texttt{MethodOf}(e) \notin \text{CSML} \\ \texttt{Sel}_{\mathcal{U}}(ctx, e) & \texttt{MethodOf}(e) \in \text{CSML} \end{cases} \tag{9}$$

- Partial. Under "*partial*" method-level context-sensitivity [30, 14], only a subset of (precision-critical) variables/objects in the program, CPML, is selected to be analyzed context-sensitively:

$$\texttt{Sel}_{\mathcal{P}}(ctx, e) = \begin{cases} [\,] & e \notin \text{CPML} \\ \texttt{Sel}_{\mathcal{U}}(ctx, e) & e \in \text{CPML} \end{cases} \tag{10}$$

The power of QILIN goes beyond existing fine-grained context-sensitive pointer analyses [43, 25, 30, 14]. In our pointer analysis framework, different variables/objects can be analyzed completely independently under different context abstractions, thus providing support for fine-grained context selectivity in its full generality.

4.3 Heap Abstractors

We can instantiate $\texttt{HeapAbs}(l, T)$ in [NEW] (Figure 3) to define a range of heap abstractions used, including "Allocation-Site", "Heuristic", and "Type-Consistency" listed at ② in Figure 2.

- Allocation-Site. This represents the most widely used heap abstraction:

$$\texttt{HeapAbs}_{\mathcal{A}}(l, T) = O_l \tag{11}$$

where O_l is an abstract object created at the allocation site identified by its label l.

- Heuristic. In practice, for efficiency reasons at little loss of precision, the objects of a particular type may be distinguished per dynamic type (instead of per object). As a result, we obtain the following "*heuristic*" heap abstractor (with $\mathcal{T}$ given in Eq. (7)):

$$\texttt{HeapAbs}_{\mathcal{H}}(l, T) = \begin{cases} O_T & T \in \texttt{SubTypeOf}(\mathcal{T}) \\ O_l & \text{otherwise} \end{cases} \tag{12}$$

- Type-Consistency. For the "*type-consistent*" heap abstraction proposed in [48], we have:

$$\texttt{HeapAbs}_{\mathcal{T}}(l, T) = \texttt{rep}(S(O_l)) \tag{13}$$

where $S(O_l)$ is the equivalence class containing the objects that are type-consistent as O_l and $\texttt{rep}(S(O_l))$ is its representative. In other words, all the allocation sites are divided into equivalence classes so that those in the same equivalence class are not distinguished.

4.4 Qilin's Toolbox

Qilin, as shown at ① in Figure 2, includes already a rich set of pointer analyses for supporting (1) *insens* (Andersen's context-insensitive analysis) [23], (2) all common flavors of method-level context-sensitivity: kCFA (k-callsite-sensitivity) [40], kOBJ (k-object-sensitivity) [32, 33], kTYPE (k-type-sensitivity) [42], S-kOBJ (hybrid k-object-sensitive analysis) [22], and (3) many flavors of fine-grained context-sensitivity, enabled by different pre-analyses (for defining `Sel` and `HeapAbs`): Bean [49], Mahjong [48], Zipper [25], Eagle [30], Turner [14], Conch [15], Data-Driven [19], and Context-Tunneling [17].

Table 4 lists a subset of these analyses (evaluated below) and their instantiations. Given two context selectors s_1 and s_2, we define $\texttt{Min}(s_1, s_2) = \lambda\ (ctx, e).$ **if** $|s_1(ctx, e)| \leqslant |s_2(ctx, e)|$ **then** $s_1(ctx, e)$ **else** $s_2(ctx, e)$. Each analysis is specified by a triple [`Cons`, `Sel`, `HeapAbs`]. Z-kOBJ, E-kOBJ, and T-kOBJ are the versions of kOBJ performed with fine-grained context-sensitivity prescribed by Zipper [25], Eagle [30], and Turner [14], respectively.

Table 4 A subset of pointer analyses instantiated in Qilin.

Pointer Analysis	Instantiation (Parameterization)
insens [23]	[Eq. (1), Eq. (8), Eq. (12)]
kCFA [40]	[Eq. (2), Eq. (8), Eq. (12)]
kOBJ [32, 33]	[Eq. (3), Eq. (8), Eq. (12)]
S-kOBJ [22]	[Eq. (5), Eq. (8), Eq. (12)]
Z-kOBJ [25]	[Eq. (3), `Min`(Eq. (9), Eq. (8)), Eq. (12)]
E-kOBJ [30]	[Eq. (3), `Min`(Eq. (10), Eq. (8)), Eq. (12)]
T-kOBJ [14]	[Eq. (3), `Min`(Eq. (10), Eq. (8)), Eq. (12)]

5 Evaluation

We have implemented Qilin as a standalone tool in Java in 20.3 KLOC (including 4.7 KLOC at its core) that operates on the Jimple IR [54] of Soot (version `4.2.1`) [53]. Qilin (including a micro-benchmark suite consisting of ≈100 unit test cases) has been open-sourced and maintained at `https://github.com/QiLinPTA/QiLin`.

Our evaluation aims to show that Qilin has met all its four design goals by answering the following four questions positively:

- **RQ1.** Is Qilin precise in terms of the precision achieved against the state-of-the-art?
- **RQ2.** Is Qilin efficient in terms of the analysis time taken against the state-of-the-art?
- **RQ3.** Is Qilin modular in allowing its common codebase to be shared?
- **RQ4.** Is Qilin effective in supporting fine-grained context-sensitive pointer analyses?

We report and analyze our results by focusing on the seven representative analyses listed in Table 4, *insens*, 1CFA, 2OBJ and S-2OBJ (with method-level context-sensitivity) and E-2OBJ, T-2OBJ and Z-2OBJ (with fine-grained context-sensitivity). We address **RQ1** and **RQ2** by comparing Qilin with Doop [8] (using a recent stable version `4.24.0` with Soufflé Datalog engine 1.5.1 [20]) in supporting *insens*, 1CFA, 2OBJ and S-2OBJ. Note that Doop has been tested with Soufflé 1.5.1 and Soufflé 2.0.2, but Doop is slower under Soufflé 2.0.2 than under Soufflé 1.5.1 in our evaluation. During this process, we have fixed a number of bugs in Doop that may have caused its unsoundness as also reported earlier [37]. We address **RQ3** and **RQ4** by considering how Qilin supports E-2OBJ, T-2OBJ and Z-2OBJ (among others).

We have used a large Java library (`JDK1.6.0_45`) and 12 popular Java programs (including 9 benchmarks from DaCapo 2006 [5] and 3 Java applications, `checkstyle`, `JPC` and `findbugs`), which are frequently used for evaluating pointer analysis algorithms in the literature [43,

22, 48, 25, 14, 28, 15]. We have excluded `jython` and `hsqldb` since their context sensitive analyses do not scale due to overly conservative handling of Java reflection [50]. By using DaCapo 2006 as in these earlier papers, we are able to evaluate these earlier algorithms in Qilin with reference to the results reported earlier. We have carried out all the experiments on an eight-core Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM.

5.1 RQ1: Precision

As shown in Table 5, Qilin delivers exactly the same precision as Doop for *insens*, 1cfa, 2obj and S-2obj, since both tools (1) use the same logic points-to definitions (Section 3.3), and (2) cover the same complex Java features identically (Section 3.4). The precision of a pointer analysis is measured in terms of four common metrics [42, 48, 30, 25]: (1) *#call-edges*: the number of call graph edges discovered, (2) *#fail-cast*: the number of type casts that may fail, (3) *#poly-calls*: the number of polymorphic calls discovered, and (4) *#avg-pts*: the average number of objects pointed by a variable, i.e., the average points-to set size by considering only the variables in the Java methods (i.e., excluding all tool-specific temporary variables introduced, and consequently, the native methods summarized).

For a total of 12 programs × 4 analyses = 48 configurations evaluated, Qilin yields the same results as Doop for all the four metrics. In addition, we have also validated that both produce exactly the same points-to sets for all the variables considered. Thus, Qilin represents a modern framework for supporting precise pointer analyses for large Java programs.

Note that Qilin and Doop may introduce a few different temporary variables in modeling native methods and certain language constructs. The differences in their points-to facts will not affect the points-to information computed for the variables in a Java method.

5.2 RQ2: Efficiency

Table 5 also compares Qilin with Doop in terms of the efficiency of *insens*, 1cfa, 2obj, and S-2obj achieved. The time budget for running an analysis on a program is 12 hours. The analysis time of a program is an average of 3 runs. Qilin currently uses a single-threaded constraint solver while Doop uses a multi-threaded Datalog engine, Soufflé. According to [3], Soufflé delivers its maximum performance at 4 or 8 threads. While Doop defaults to 4 threads, we have used 8 threads to enable it to achieve slightly better performance. Note that the analysis time of a program under Doop is given as the analysis time spent by its Datalog engine only (without including the time spent by its fact generator, which is claimed to be amortizable across a number of analyses applied to the same program).

For the same 12 programs × 4 analyses = 48 configurations evaluated, Qilin outperforms Doop substantially, with the speedups ranging from 0.9x (for `checkstyle` under 2obj) to 6.3x (for `xalan` also under 2obj). Note that Qilin is slightly slower than Doop only under 2obj in analyzing `checkstyle`. The overall average speedup achieved by Qilin over Doop for all the four analyses across the 12 programs is 2.4x. This increases to 2.9x when Doop switches from 8 to 4 threads and 5.1x when Doop switches to a single thread.

As Qilin achieves exactly the same precision as Doop (Section 5.1), its high performance is attributed to our new incremental worklist-based constraint solver, which runs significantly faster than Soufflé [39, 20]. Thus, this work confirms (for the first time) that an imperative framework (implemented in Java) that relies on a well-crafted constraint solver can outperform a declarative counterpart that relies on a (multi-threaded) Datalog engine, despite that Qilin is currently single-threaded and Doop has been carefully optimized in over one decade. Qilin's high efficiency is expected to provide significant performance benefits for its client applications, such as call graph construction tools [23, 1, 38], bug detection tools [34, 55, 27, 10] and compiler optimization techniques[9, 47].

Table 5 The efficiency and precision of Qilin and Doop in supporting *insens*, 1cfa, 2obj, and S-2obj. For all metrics (except speedups of Qilin over Doop in blue), smaller is better.

Program	Metrics	*insens*		1cfa		2obj		S-2obj	
		Doop	Qilin	Doop	Qilin	Doop	Qilin	Doop	Qilin
antlr	Time (s)	27	11 (**2.5x**)	61	24 (**2.6x**)	116	55 (**2.1x**)	90	46 (**2.0x**)
	#call-edges	57472	57472	56226	56226	51319	51319	51318	51318
	#fail-casts	1127	1127	930	930	511	511	439	439
	#poly-calls	1987	1987	1933	1933	1643	1643	1642	1642
	#avg-pts	36.498	36.498	32.106	32.106	9.055	9.055	8.945	8.945
bloat	Time (s)	20	11 (**1.8x**)	85	30 (**2.9x**)	1503	788 (**1.9x**)	1450	800 (**1.8x**)
	#call-edges	67856	67856	65689	65689	56837	56837	56836	56836
	#fail-casts	2088	2088	1891	1891	1316	1316	1244	1244
	#poly-calls	2344	2344	2171	2171	1714	1714	1713	1713
	#avg-pts	52.992	52.992	47.391	47.391	15.387	15.387	15.287	15.287
chart	Time (s)	51	19 (**2.7x**)	174	41 (**4.3x**)	411	222 (**1.9x**)	435	370 (**1.2x**)
	#call-edges	86806	86806	84116	84116	72805	72805	72801	72801
	#fail-casts	2563	2563	2207	2207	1348	1348	1183	1183
	#poly-calls	2732	2732	2614	2614	2068	2068	2067	2067
	#avg-pts	64.751	64.751	52.949	52.949	5.796	5.796	5.541	5.541
eclipse	Time (s)	115	37 (**3.1x**)	482	132 (**3.7x**)	8556	4701 (**1.8x**)	8493	4266 (**2.0x**)
	#call-edges	183288	183288	178585	178585	162934	162934	162876	162876
	#fail-casts	5114	5114	4732	4732	3648	3648	3542	3542
	#poly-calls	10738	10738	10455	10455	9773	9773	9718	9718
	#avg-pts	137.322	137.322	62.446	62.446	16.115	16.115	14.944	14.944
fop	Time (s)	29	9 (**3.3x**)	51	17 (**3.1x**)	52	25 (**2.1x**)	48	22 (**2.2x**)
	#call-edges	40558	40558	39285	39285	34424	34424	34424	34424
	#fail-casts	914	914	710	710	396	396	315	315
	#poly-calls	1223	1223	1156	1156	842	842	842	842
	#avg-pts	25.526	25.526	20.469	20.469	4.399	4.399	4.262	4.262
luindex	Time (s)	12	9 (**1.3x**)	28	15 (**1.9x**)	38	25 (**1.6x**)	34	23 (**1.5x**)
	#call-edges	39809	39809	38529	38529	33643	33643	33642	33642
	#fail-casts	923	923	727	727	396	396	324	324
	#poly-calls	1294	1294	1228	1228	935	935	934	934
	#avg-pts	20.807	20.807	16.290	16.290	4.480	4.480	4.325	4.325
lusearch	Time (s)	13	9 (**1.4x**)	31	17 (**1.9x**)	73	37 (**2.0x**)	67	36 (**1.9x**)
	#call-edges	43153	43153	41841	41841	36525	36525	36524	36524
	#fail-casts	1035	1035	831	831	411	411	332	332
	#poly-calls	1505	1505	1432	1432	1133	1133	1132	1132
	#avg-pts	22.418	22.418	17.625	17.625	4.461	4.461	4.299	4.299
pmd	Time (s)	35	14 (**2.6x**)	103	30 (**3.5x**)	104	53 (**2.0x**)	94	56 (**1.7x**)
	#call-edges	69713	69713	67899	67899	60030	60030	60029	60029
	#fail-casts	2273	2273	2026	2026	1416	1416	1333	1333
	#poly-calls	2989	2989	2871	2871	2390	2390	2389	2389
	#avg-pts	37.331	37.331	31.764	31.764	6.036	6.036	5.947	5.947
xalan	Time (s)	39	11 (**3.6x**)	68	22 (**3.1x**)	4046	638 (**6.3x**)	823	405 (**2.0x**)
	#call-edges	54147	54147	52657	52657	46856	46856	46855	46855
	#fail-casts	1305	1305	1058	1058	601	601	516	516
	#poly-calls	2101	2101	2010	2010	1657	1657	1656	1656
	#avg-pts	29.968	29.968	24.529	24.529	6.037	6.037	5.924	5.924
checkstyle	Time (s)	64	16 (**4.0x**)	146	36 (**4.1x**)	6308	7148 (**0.9x**)	5338	5535 (**1.0x**)
	#call-edges	80291	80291	77881	77881	67285	67285	67276	67276
	#fail-casts	1941	1941	1680	1680	1117	1117	1023	1023
	#poly-calls	2778	2778	2655	2655	2241	2241	2234	2234
	#avg-pts	47.925	47.925	39.536	39.536	8.048	8.048	7.706	7.706
JPC	Time (s)	32	21 (**1.5x**)	127	40 (**3.2x**)	211	132 (**1.6x**)	264	192 (**1.4x**)
	#call-edges	95055	95055	91661	91661	81465	81465	81429	81429
	#fail-casts	2254	2254	1894	1894	1357	1357	1208	1208
	#poly-calls	4960	4960	4840	4840	4282	4282	4275	4275
	#avg-pts	45.533	45.533	32.175	32.175	6.045	6.045	5.841	5.841
findbugs	Time (s)	54	20 (**2.8x**)	198	48 (**4.1x**)	3887	1644 (**2.4x**)	3856	1593 (**2.4x**)
	#call-edges	106065	106065	102352	102352	88107	88107	88107	88107
	#fail-casts	3457	3457	3000	3000	2058	2058	1968	1968
	#poly-calls	4534	4534	4308	4308	3679	3679	3679	3679
	#avg-pts	64.510	64.510	53.842	53.842	9.102	9.102	9.052	9.052

Table 6 Human effort required in integrating fine-grained context-sensitive analysis into Qilin.

Pointer Analysis	Source Code (#LOC)	Supporting Code (#LOC)
Bean [49]	355	41
Mahjong [48]	666	51
Data-Driven [19]	1062	39
Zipper [25]	1474	35
Context-Tunneling [17]	1151	29
Eagle [30]	357	75
Turner [14]	769	75
Conch [15]	1230	55

5.3 RQ3: Modularity

Table 7 The efficiency and precision of E-2obj, T-2obj, and Z-2obj for performing *k*obj under fine-grained context-sensitivity enabled by Eagle [30], Turner [14], and Zipper [25], respectively. The speedups of each analysis (in blue) is computed with 2obj (shown in Table 5) as the baseline.

Metrics	Program	E-2obj	T-2obj	Z-2obj	Program	E-2obj	T-2obj	Z-2obj
Time (s)		38 (**1.4x**)	18 (**3.1x**)	31 (**1.8x**)		27 (**1.4x**)	19 (**1.9x**)	19 (**1.9x**)
#call-edges		51319	51319	51505		36525	36525	36720
#fail-casts	antlr	511	511	532	lusearch	411	411	441
#poly-calls		1643	1643	1666		1133	1133	1162
#avg-pts		9.055	9.067	9.492		4.461	4.473	5.071
Time (s)		577 (**1.4x**)	302 (**2.6x**)	663 (**1.2x**)		38 (**1.4x**)	25 (**2.1x**)	31 (**1.7x**)
#call-edges		56837	56837	57059		60030	60030	60180
#fail-casts	bloat	1316	1316	1339	pmd	1416	1416	1447
#poly-calls		1714	1714	1746		2390	2390	2412
#avg-pts		15.387	15.403	16.381		6.036	6.045	6.441
Time (s)		165 (**1.3x**)	119 (**1.9x**)	45 (**4.9x**)		366 (**1.7x**)	297 (**2.1x**)	285 (**2.2x**)
#call-edges		72805	72805	73243		46856	46856	47005
#fail-casts	chart	1348	1348	1395	xalan	601	601	618
#poly-calls		2068	2068	2094		1657	1657	1680
#avg-pts		5.796	5.809	6.474		6.037	6.055	6.550
Time (s)		2929 (**1.6x**)	1904 (**2.5x**)	2000 (**2.4x**)		3776 (**1.9x**)	2813 (**2.5x**)	1882 (**3.8x**)
#call-edges		162934	162934	163176		67285	67285	67511
#fail-casts	eclipse	3648	3648	3707	checkstyle	1117	1117	1144
#poly-calls		9773	9773	9834		2241	2241	2278
#avg-pts		16.115	16.305	16.413		8.048	8.152	9.257
Time (s)		17 (**1.5x**)	12 (**2.1x**)	13 (**1.9x**)		100 (**1.3x**)	75 (**1.8x**)	54 (**2.4x**)
#call-edges		34424	34424	34615		81465	81465	81741
#fail-casts	fop	396	396	421	JPC	1357	1357	1393
#poly-calls		842	842	868		4282	4282	4337
#avg-pts		4.399	4.416	4.977		6.045	6.062	6.514
Time (s)		18 (**1.4x**)	11 (**2.3x**)	14 (**1.8x**)		925 (**1.8x**)	179 (**9.2x**)	160 (**10.3x**)
#call-edges		33643	33643	33833		88107	88107	88172
#fail-casts	luindex	396	396	422	findbugs	2058	2058	2091
#poly-calls		935	935	960		3679	3679	3687
#avg-pts		4.480	4.494	5.065		9.102	9.139	9.300

Qilin supports a variety of context-sensitive pointer analyses that can all be specified modularly as variations on a common code base with their context-sensitivity parameterized by `Cons`, `Sel`, and `HeapAbs`. We use #LOC, the number of LOC required in integrating a pointer analysis algorithm into Qilin, to measure the modularity of our framework in supporting the design and implementation of new algorithms. While #LOC is not equivalent

to the amount of engineering efforts involved, a small #LOC needed indicates that our framework is highly modular. For the four traditional method-level analyses, *insens*, *k*CFA, *k*OBJ and S-*k*OBJ, listed in Table 4 and evaluated above, their parameterization requires 15, 43, 40 and 61 LOC, respectively, totaling only 98 LOC with the commonalities factored out.

In addition, QILIN also accommodates well a range of recently proposed fine-grained context-sensitive pointer analyses [49, 48, 19, 17, 25, 30, 14], as demonstrated in Table 6. For each analysis, the second column lists the number of LOC required for defining the context-sensitivity proposed (in the form of a pre-analysis), and the third column gives the number of LOC required for parameterizing it in QILIN (requiring only an average of 50 LOC each). We have integrated all these seven analyses into QILIN except the machine learning phases used in DATA-DRIVEN [19] and Context-Tunneling [17].

5.4 RQ4: Fine-Grained Context-Sensitivity

QILIN is expected to represent a common framework in which different pointer analysis algorithms can be designed and evaluated effectively. Table 7 compares the efficiency and precision of E-2OBJ, T-2OBJ, and Z-2OBJ (the three fine-grained variations of 2OBJ listed in Table 4) enabled by EAGLE [30], TURNER [14], and ZIPPER [25], respectively, with all the parameterization efforts given in Table 6 (in terms of #LOC added). The speedups of each of the three analyses is computed with 2OBJ (shown in Table 5) as the baseline.

Precision-wise, our results are consistent with those reported earlier in [30, 25, 14]. Specifically, E-2OBJ always preserves the precision of 2OBJ in theory, T-2OBJ preserves the precision of *#call-edges*, *#fail-cast*, and *#poly-calls* but not *#avg-pts* in practice, and finally, Z-2OBJ loses precision by design in general as it has caused *#call-edges*, *#fail-cast*, *#poly-calls* and *#avg-pts* to increase by 3.6%, 1.6%, 0.4% and 8.7%, respectively.

Efficiency-wise, our results are also consistent with those reported earlier in [30, 25, 14] in the sense that E-2OBJ, T-2OBJ, and Z-2OBJ are faster than 2OBJ. Specifically, ZIPPER (the least precise) achieves the highest speedups, ranging from 1.2x (for `bloat`) to 10.3x (for `findbugs`) with an average of 3.0x, TURNER achieves slightly lower speedups, ranging from 1.8x (for `JPC`) to 9.2x (for `findbugs`) with an average of 2.8x, and finally, EAGLE (the most precise) achieves the lowest speedups, ranging from 1.3x (for `JPC`) to 1.9x (for `checkstyle`) with an average of 1.5x. However, the relative speedups of E-2OBJ, T-2OBJ, and Z-2OBJ over 2OBJ reported here are not expected to be exactly the same as those reported in [30, 14] due to different experimental settings used (for the purposes of validating different design hypotheses). One difference is particularly noteworthy: E-2OBJ and Z-2OBJ are compared by parameterizing E-*k*OBJ and Z-*k*OBJ as in Table 4 in [14], but E-*k*OBJ as [Eq. (3), Eq. (10), Eq. (11)] and Z-*k*OBJ as [Eq. (3), Eq. (9), Eq. (11)] instead in [14, 30]. Specifically, the objects that are instantiated from `StringBuilder` and `StringBuffer` as well as `Throwable` (including its subtypes) are merged per dynamic type and then analyzed context-insensitively in [14, 25] but not in [30]. This again highlights the significance for the research community to share QILIN as a common open-source framework to design and evaluate different fine-grained analyses in future work.

6 Related Work

We review the past work on context-sensitive pointer analyses for Java by focusing on representative open-source frameworks developed and the recent research trend on exploring fine-grained context-sensitivity, by placing QILIN again in its research context.

Pointer Analysis Frameworks. Existing frameworks, which were originally designed and implemented to support method-level context-sensitivity, fall into three categories: (1) imperative, e.g., Spark [23] and Wala [16] (implemented in Java), (2) declarative, e.g., Doop [8] (coded in Datalog on top of a Datalog engine, e.g., Soufflé [20, 39]), and (3) hybrid, e.g., Jchord [34] and Paddle [24] (with the core of a pointer analysis algorithm performed in Datalog declaratively but the rest coded in Java imperatively). In contrast, Qilin is a new framework designed to support variable-level context-sensitivity (by subsuming existing traditional frameworks as a special case since all the variables/objects in a method can only be analyzed traditionally by using exactly the same context abstraction).

In the past decade or so, Doop has been the state of the art for supporting traditional pointer analyses with method-level context-sensitivity. As a fully-declarative framework, Doop is highly scalable, enabling complex and precise context-sensitive pointer analyses to be developed efficiently in the past [8, 42]. Whether an imperative alternative can outperform Doop in terms of efficiency while achieving the same precision remains to be unknown for years in the pointer analysis community. In this paper, we show that Qilin, as an imperative framework, can outperform Doop substantially while also allowing all traditional pointer analyses to be specified precisely and modularly as in Doop.

Fine-Grained Context-Selectivity. To scale context-sensitive analyses further for large codebases, how to explore a significantly larger space of efficiency/precision tradeoffs by moving from method-level to fine-grained context-sensitivity has received increasing interest. In the past few years, method-level context-sensitivity has been made (1) selective (by analyzing only a subset of methods context-sensitively via exploiting user-supplied hints [43], machine learning [19], and pattern-matching [25], and (2) selective (by analyzing only a subset of variable/objects context-sensitively via exploiting CFL (Context-Free Language) reachability [28, 30, 14, 29]. In the near future, we envisage to see more pointer analyses with variable-level context-sensitivity to be developed.

However, existing pointer analysis frameworks [23, 16, 8, 34, 24] were all designed for supporting method-level context-sensitivity. As described in Section 1, we have made significant efforts in extending Doop to support fine-grained context-sensitivity, but to no avail. We see two limitations from our preliminary investigation: First, the number of rules for supporting fine-grained analyses increases drastically relative to the Doop baseline since a fine-grained analysis relies on more configurable parameters (in addition to the context length k). Second, the performance of the extended Doop version is rather disappointing due to possibly poor join orders selected by its underlying Datalog engine [39, 20] used.

In this paper, we have designed and implemented Qilin on top of Spark [23] for supporting fine-grained context-sensitivity. In our previous work [28, 30, 14, 29], we have also introduced a few in-house implementations (which can be seen as some precursors of Qilin) for supporting only partial context-sensitivity (under which variables/objects and methods can be analyzed either context-sensitively or context-insensitively). These implementations are designed to support specific pointer analysis techniques with different design choices and settings for handling different language features, which limit them from being used as a general-purpose framework. To the best of our knowledge, Qilin represents the first framework that supports all such fine-grained analyses precisely, efficiently and modularly.

Iterative Constraint Solving via Difference Propagation. Many program analyses, such as pointer analysis, exploit the idea of difference propagation [12, 35, 23] when resolving their constraints towards a fixed-point solution efficiently. For example, Sridharan et al. [45] present

a difference-propagation-based pointer analysis algorithm for object-oriented programs. In their algorithm, every time when an edge $x \xrightarrow{\text{assign}} y$ needs to be handled, their algorithm needs to compute $\delta = \texttt{PTS}(x) - \texttt{PTS}(y)$ and then propagates δ to $\texttt{PTS}(y)$, which can be highly expensive. In contrast, Qilin's incremental worklist-based algorithm (Algorithm 1), which is is extended from Spark [23], computes only $\delta_{\text{new}} = \texttt{PTS}(x)_{\text{new}} - \texttt{PTS}(y)_{\text{new}}$ and then propagates δ_{new} to $\texttt{PTS}(y)_{\text{new}}$. As a result, our constraint solver is more efficient. In addition, the semi-naïve evaluation, an efficient evaluation strategy used by many existing Datalog engines [20, 31], refines the naïve (chaotic iteration) strategy to avoid redundant work by exploiting also the idea of difference propagation. The speedups achieved by Qilin over Doop are attributed to our incremental worklist algorithm, which may exhibit better join orders than the ones automatically selected by Doop's underlying Datalog engine [20].

7 Conclusion and Future work

We have introduced Qilin as the first open-source framework (to be released soon) for supporting fine-grained context-sensitive pointer analyses (including the traditional ones as special cases) for Java, precisely, efficiently and modularly. Developing such a production-quality framework involves a lot of technical and engineering efforts. We will maintain and grow this open-source project actively on GitHub to support further research on pointer analysis and a variety of other static analyses for Java (and possibly other object-oriented programming languages). Several immediate future research/engineering activities that can be carried out in Qilin include (1) parallelizing its constraint solver to lift its performance further, (2) covering more native methods and Java reflection APIs, (3) supporting more Java features in JDK8 and above, and (4) experimenting with the design and implementation of novel variable-level context-sensitive pointer analysis algorithms. Finally, as an open-source project, Qilin is also expected to be driven by community contribution.

References

1 Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

2 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

3 Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting Doop to Soufflé: A tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3088515.3088522.

4 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

5 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

6 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Honolulu, HI, USA, 2011. IEEE.

7 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

8 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

9 Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 416–425, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1168857.1168908.

10 Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. SMOKE: Scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82, New York, NY, USA, 2019. IEEE. doi:10.1109/ICSE.2019.00025.

11 Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3092703.3092729.

12 Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *European Symposium on Programming*, pages 90–104. Springer, 1998.

13 Neville Grech and Yannis Smaragdakis. P/Taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. doi:10.1145/3133926.

14 Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, pages 18:1–18:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

15 Dongjie He, Jingbo Lu, and Jingling Xue. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–91. IEEE, 2021.

16 IBM. WALA: T.J. Watson Libraries for Analysis, 2020. URL: http://wala.sourceforge.net/.

17 Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

18 Minseok Jeon, Myungho Lee, and Hakjoo Oh. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

19 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017.

20 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

21 Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 774–784, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338906.3338936.

22 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434, New York, NY, USA, 2013. Association for Computing Machinery.

23 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

24 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), October 2008. doi:10.1145/1391984.1391987.

25 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

26 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 129–140, New York, NY, USA, 2018. Association for Computing Machinery.

27 Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 359–373, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192390.

28 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46, 2021.

29 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability. In *International Static Analysis Symposium*, pages 261–285. Springer, 2021.

30 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

31 Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices*, 51(6):194–208, 2016.

32 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. Association for Computing Machinery.

33 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

34 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery.

35 David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12. IEEE, 2003.

36 Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, pages 361–369, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3196321.3197546.

37 Jyoti Prakash, Abhishek Tiwari, and Christian Hammer. Effects of program representation on pointer analyses – an empirical study. *Fundamental Approaches to Software Engineering*, 12649:240, 2021.

38 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for Java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 474–486, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2950290.2950312.

39 Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892226.

40 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

41 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

42 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery.

43 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery.

44 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

45 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, Berlin, Heidelberg, 2013.

46 Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, New York, NY, USA, 2007. Association for Computing Machinery.

47 Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, New York, NY, USA, 2013. IEEE. doi:10.1109/CGO.2013.6494978.

48 T. Tan, Y. Li and J. Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery.

49 Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

50 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–277, New York, NY, USA, 2017. Association for Computing Machinery.

51 David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 197–208, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3409698.

52 David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180251.

53 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010.

54 Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

55 Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337, New York, NY, USA, 2018. IEEE. doi:10.1145/3180155.3180178.