Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics

Tian Tan
State Key Laboratory for Novel Software Technology
Nanjing University, China
tiantan@nju.edu.cn

Yue Li*
State Key Laboratory for Novel Software Technology
Nanjing University, China
yueli@nju.edu.cn

ABSTRACT

Static analysis is a mature field with applications to bug detection, security analysis, program understanding, optimization, and more. To facilitate these applications, static analysis frameworks play an essential role by providing a series of fundamental services such as intermediate representation (IR) generation, control flow graph construction, points-to/alias information computation, and so on. However, although static analysis has made great strides and several well-known frameworks have emerged in this field over the past decades, these frameworks are not that easy to learn and use for developers who rely on them to create and implement analyses. In that sense, it is far from trivial to build a developer-friendly static analysis framework, because compared to the knowledge required for static analysis itself, we have significantly less knowledge designing and implementing static analysis frameworks.

In this work, we take a step forward by discussing the design trade-offs for the crucial components of a static analysis framework for Java, and select the designs by following the HGDC (Harnessing the Good Designs of Classics) principle: *for each crucial component of a static analysis framework, we compare the design choices made for it (possibly) by different classic frameworks such as Soot, Wala, Doop, SpotBugs and Checker, and choose arguably a more appropriate one; but if none is good enough, we then propose a better design.* These selected or newly proposed designs finally constitute Tai-e, a new static analysis framework for Java, which has been implemented from scratch. Tai-e is novel in the designs of several aspects like IR, pointer analysis and development of new analyses, etc., leading to a developer-friendly (easy-to-learn and easy-to-use) analysis framework. To the best of our knowledge, this is the first work that systematically explores the designs and implementations of various static analysis frameworks for Java. We expect it to provide useful materials and viewpoints for building better static analysis infrastructures, and we hope that it could draw more attentions of the community to this challenging but tangible topic.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

*Corresponding author

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598120

KEYWORDS

static analysis, framework design and implementation, Java

ACM Reference Format:
Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598120

1 INTRODUCTION

Static analysis is a well-studied technique that has been successfully applied to many applications like bug detection [8, 12, 50], security analysis [4, 32, 43, 54], code optimization [60, 64, 74], program understanding [42, 66] and verification [19, 31, 53], and its effect has translated into real benefit for a substantial number of research work and industry products [14, 15, 56]. To facilitate these applications (by implementing specific analysis algorithms), static analysis frameworks play an essential role by providing a series of fundamental services such as intermediate representation (IR) generation, control flow graph construction, points-to/alias information computation, and more. However, despite impressive progress of static analysis, and this field has seen several popular frameworks in the last decades [10, 24, 52, 63, 71, 72], they are not that easy to learn and use for developers who rely on them to create and implement analyses. In other words, it is far from trivial to build a developer-friendly static analysis framework, as compared to the knowledge required for static analysis itself, we have significantly less knowledge designing and implementing static analysis frameworks. This is a challenging problem, as framework design is mostly a trade-off among different goals such as simplicity, usability and efficiency (one is often implemented at the expense of another), and it will take a lot of labor and intelligence to implement; additionally, it is also hard to evaluate its effectiveness due to its subjective nature. As a result, a decade ago, the authors of Soot [71] wrote the following words in their retrospective paper [33]:

> *"We have noticed that it is difficult to publish framework papers... We encourage conferences to accept more framework papers."*

Until now, despite still very few, there is work that attempts to offer some lessons learned for improving (or adding) certain facilities for their analysis frameworks after using them for a period of time [33, 58], but none addresses the root of our problem — we still lack a systematic view to examine the quality of a static analysis framework from the perspective of developers who rely on the framework to create new analysis. To tackle this problem, in this paper, we take one step forward by discussing some of the key design trade-offs for the following crucial components that a static analysis framework (for Java) is supposed to provide.

- *Program Abstraction.* It needs to provide an abstraction model, including IR, class hierarchy, etc., to represent all program elements that are ready for various static analyses to acquire;
- *Fundamental Analyses.* It is supposed to support fundamental facilities to allow analysis developers to work with analysis-friendly structures such as control flow and call graphs to implement classic graph-based algorithms, and to make use of abstracted memory information of input programs such as points-to/alias relations to build sophisticated analyses;
- *New Analysis Development.* It ought to offer a mechanism for developing and integrating any new analysis, including clients like bug detectors and security analyzers as well as more basic ones like exception and reflection analyses;
- *Multiple Analyses Management.* It should provide a standardized approach to managing multiple analyses (e.g., configuring their dependencies or cooperating their outputs) when they are required to work together for certain analysis task.

In addition, for the above components, we argue for the most appropriate design by following the HGDC (Harnessing the Good Designs of Classics) principle: *Given any component, we compare the design choices made for it (possibly) by different classic frameworks such as Soot [71], Wala [72], Doop [10], SpotBugs [63], and Checker [52], and select arguably a more appropriate one; but if none is good enough, we then propose a better design.* These selected or newly proposed designs together constitute Tai-e, a new static analysis framework for Java, which has been implemented from scratch, and built with great care; these efforts finally contribute to a developer-friendly, i.e., easy-to-learn and easy-to-use framework.

We consider Soot, Wala, Doop, SpotBugs and Checker as they are all classic frameworks for Java with histories spanning more than a decade. Soot and Wala, as two *general* analysis frameworks, have drawn the attention of numerous academics, and as the centerpieces of a general analysis framework, the systems of pointer analysis and data-flow analysis are crucial and should be specifically explored; Doop is the state-of-the-art of the former, while SpotBugs (the successor of FindBugs [23]) and Checker are the representatives of the latter. Specifically, this work makes the following contributions.

1. We present the first work that systematically explores the designs and implementations of various classic static analysis frameworks for Java, and discuss their rationalities for different crucial analysis components, providing useful materials and viewpoints for building better static analysis infrastructures.

2. We introduce Tai-e, a developer-friendly static analysis framework for Java, which is built from scratch, following the HGDC principle. In addition to the *integration novelty* stemming from HGDC, Tai-e has its specific novel designs. For examples:

- Tai-e offers a usage-friendly IR for developing analysis: compared to the IRs of Soot and Wala, it enables to produce more succinct code for implementing static analysis algorithms, and makes it easier to understand its underlying intents.
- Tai-e provides an effective pointer analysis system that is more extensible and efficient than other frameworks, making it easier to create various new pointer analysis algorithms.
- Tai-e introduces a novel analysis plugin system to easily develop and integrate new analysis (that interacts with pointer analysis) like taint analysis and exception analysis, etc.

The primary goal of Tai-e is to be developer-friendly, and the scheme to determine whether Tai-e achieves this goal is to evaluate feedback from analysis developers. Below, we list some examples.

> An established professor wrote to us: *"My students told me that it is very smooth to write code on Tai-e and Tai-e is significantly better than Soot in usability. "*

> A senior static-analysis engineer from a famous IT company wrote to us: *"To me, Tai-e's design model is worth learning for many analysis tools. I found it can help junior analysis developers quickly understand analysis algorithms by debugging and reading the framework code; senior developers can also quickly integrate their analysis module based on Tai-e's pluggable analysis system. The overall analysis process of Tai-e is clear and controllable."*

> A netizen wrote publicly in a technical blog: *"In retrospect, Tai-e is undoubtedly excellent in the ability of pointer analysis and algorithm design, and the entire framework design of Tai-e and its plugin-style programming, and the design thinking of other aspects, go far beyond Soot..."*

In addition, we conduct a survey from 32 graduate students (whose fields are SE and PL) to compare Tai-e, Soot (or Wala) for evaluating which framework is more developer-friendly, and 16 survey reports are received. Briefly, all 16 respondents agree that Tai-e is easier to learn and use than Soot and Wala, despite the fact that seven of them have previously used Soot or Wala. Moreover, we ask them to implement the same reflection analysis on different frameworks, and they spent on average 29 hours on Tai-e while at least 49 hours on Soot/Wala (many of them claimed that functionality, debugging, and other challenges prevented them from completing the task on Soot/Wala, so the real time spent should be longer). Some of their detailed feedback is given at the end of each section, and more evaluation results are discussed in Section 6.

3. We release Tai-e as an open-source framework to offer a platform to develop new analyses *with low cost of framework learning and analysis implementation* (https://github.com/pascal-lab/Tai-e). Additionally, an educational version of Tai-e was developed, on top of which eight assignments are carefully designed for systematically training developers to implement various static analysis techniques to analyze real Java programs (https://tai-e.pascal-lab.net/en/intro/overview.html). This educational version has now attracted lecturers from 26 universities (for teaching purpose), leaders of the program analysis R&D teams from 13 companies (for training their software engineers), and students from 104 universities (for doing Tai-e assignments on our online judgment platform).

2 PROGRAM ABSTRACTION

A static analysis framework needs to offer an abstraction model of programs, including IR, type system, class hierarchy, etc., to represent all program elements for static analysis to conveniently obtain. Due to limited space, below we elaborate on some key design choices made by Tai-e, Soot and Wala for IR (the most important abstraction model) and explain why we arrive at making them.

Soot and Wala have their specific IRs, and the IR of Tai-e is largely inspired by these two frameworks with notable improvements for helping analysis developers implement more concise analysis code and better understand the underlying intents of IR.

```
1  // Soot
2  void processBinary(AssignStmt assign) {
3      Value rightOp = assign.getRightOp();
4      // "Abstract" statement introduces conditional checks
5      if (rightOp instanceof BinopExpr) {
6          BinopExpr binopExpr = (BinopExpr) rightOp;
7          // "Top" returned type needs casting
8          Local lhs = (Local) assign.getLeftOp();
9          Value op1 = binopExpr.getOp1();
10         Value op2 = binopExpr.getOp2();
11         // obtain name of op1
12         if (op1 instanceof Local) {
13             Local var1 = (Local) op1;
14             String name1 = var1.getName();
15         }
16         // obtain type of op1
17         Type type1 = op1.getType();
18         // obtain constant value of op2
19         if (op2 instanceof Constant) {
20             Constant val2 = (Constant) op2;
21         }
22     }
23 }
24 // WALA
25 void processBinary(SSABinaryOpInstruction binary, IR ir) {
26     int lhs = binary.getDef();
27     // getUse() returns an int value as index to access the
28     // info of the corresponding operand: lines 32, 37-39
29     int op1 = binary.getUse(0);
30     int op2 = binary.getUse(1);
31     // obtain name of op1
32     String[] name1 = ir.getLocalNames(binary.iIndex(), op1);
33     // obtain type of op1
34     TypeInference ti = TypeInference.make(ir, true);
35     TypeAbstraction type1 = ti.getType(op1);
36     // obtain constant value of op2
37     SymbolTable symbolTable = ir.getSymbolTable();
38     if (symbolTable.isConstant(op2)) {
39         Object val2 = symbolTable.getConstantValue(op2);
40     }
41 }
42 // Tai-e
43 void processBinary(Binary binary) {
44     Var lhs = binary.getLValue();
45     BinaryExp binaryExp = binary.getRValue();
46     Var op1 = binaryExp.getOperand1();
47     Var op2 = binaryExp.getOperand2();
48     // obtain name of op1
49     String name1 = op1.getName();
50     // obtain type of op1
51     Type type1 = op1.getType();
52     // obtain constant value of op2
53     if (op2.isConst()) {
54         Literal val2 = op2.getConstValue();
55     }
56 }
```

Figure 1: A case depicts how a binary statement is handed in Soot, Wala and Tai-e respectively based on their IRs.

In brief, Tai-e enhances developer-friendliness from multiple perspectives, including: (1) the design of IR, e.g., Tai-e distinguishes between different types of assign statements in contrast to Soot; (2) associated API designs, e.g., Tai-e utilizes concrete return types for expression retrieval and avoids the use of integers as indexes to represent variables, which differs from Soot and Wala, respectively; and (3) the organization and accessibility of program elements such as values, types, and names, e.g., unlike Wala, Tai-e centralizes all variable-related information into a single interface, rather than distributing them into different interfaces. Thanks to (1) and (2), Tai-e enables developers to write more concise code, e.g., avoiding unnecessary conditional checks and downcasts when using various expressions and statements; additionally, (3) enables developers to easily learn Tai-e's functionalities, and allows them to create analyses more fluently. Below we take binary statement as a representative example to explain the above design intentions.

Figure 1 depicts this example and shows how a binary statement (e.g., $x = y + z$), as a parameter of method `processBinary`, is handled in Soot, Wala and Tai-e respectively.

Soot represents all statements that have "=" operator inside as `AssignStmt` and does not explicitly distinguish the concrete types of assign statements, e.g., new, load, store, unary, binary statements, etc. (but Wala and Tai-e do). Although utilizing fewer statement types makes IR simpler, it may introduce many unnecessary conditional type checks. For example, as shown in Figure 1, unlike Wala and Tai-e, the declared type of the `assign` parameter (line 2) has to be `AssignStmt` as `AssignStmt` is the lowest-level interface in the class hierarchy of Soot to represent a binary statement (Wala uses `SSABinaryOpInstruction` (line 25) and Tai-e uses `Binary` (line 43) to represent a binary statement respectively); as a result, conditional check to the right operand of `assign` is required (line 5) in Soot to ensure the casting (line 6) to be safe.

In addition, as Soot always returns `Value` (the highest-level interface in the class hierarchy) to represent data, e.g., local, constant, expression, reference, etc., further casting is needed to determine the right type of data. Line 8 shows one such case: the type of the left operand of a binary statement is a local variable (this is a pre-knowledge of Soot, but if users do not know it, a conditional type check is also required here), but as Soot declares `Value` as the returned type of `getLeftOp()`, casting to `Local` is needed due to type constraint. As another example (not shown in code), Soot provides `<Value getCondition()>` of class `IfStmt` to obtain the conditional expression of an `if` statement; that means Soot still returns `Value` even if it knows that the condition of an `if` statement must be a type of `ConditionExpr` (can be validated via its code), which is a subtype of `Value`, resulting in another case of unnecessary casting. The above cases imply that distinction in IR design may appear subtle, but its effect is often profound.

Wala does not have this problem and it adopts a different strategy to represent local variables. For instance, `getUse()` in line 30 returns an int value (`op2`) as the index to access the information of the corresponding operand: to obtain the constant value stored in this operand, `op2` is used as index to lookup a symbol table (line 39) which can be retrieved from `ir` (line 37); in other words, unlike Soot (line 10) and Tai-e (line 47) that directly hold the operand values, Wala adopts the above int-index strategy to obtain the related values, which is less straightforward to understand and may become more difficult to debug when variable information is needed.

Besides, possibly due to the above design, different from Soot (lines 11–17) and Tai-e (lines 48–51) that uniformly obtain information from operands directly, Wala has to use other interfaces to obtain the name and type of an operand, via `ir` (line 32) and `TypeInference` (lines 34–35) respectively, increasing learning costs.

Tai-e avoids the above issues of Soot and Wala as reflected in lines 43–56. Now we invite readers to view the code of Figure 1 in its entirety to experience how the same binary statement is handled differently based on different IRs. Given the binary statement example in Figure 1, and the fact that a program is often made of many different types of statements and expressions, we could anticipate considerable benefits afforded by Tai-e's IR for writing more concise and understandable analysis code.

Moreover, Tai-e introduces a few new IR designs to make it more accessible for certain analyses. For example, to facilitate pointer analysis, Tai-e associates each variable v with its related statements in its IR, and once v's value is changed during analysis, developers can directly and conveniently retrieve all the related statements of v via IR to take further actions. Below, we list some feedback about the IRs of these frameworks from different developers.

"Tai-e's IR is concise and easy to understand while Soot's is over-complicated." "The IR APIs provided by Tai-e is self-explainable."

"In comparison to Tai-e, the modeling to statements and variables in Soot is not straightforward, producing some ungraceful logic implementations when traversing the graph. "

"Compared to Soot, the designs for the data structures of Tai-e's IR are more intuitive. E.g., I can directly retrieve the related information of a statement via Tai-e's interface for statements (no need to collect information by taking the effort to find other interfaces)."

"You can hardly directly get the information via many of Wala's APIs (for IR, class hierarchy, etc.). It often requires multiple APIs to be used jointly, resulting in lengthy code; but you can basically get the target information directly via various of Tai-e's APIs."

"Tai-e offers a simpler representation of IR statements. Compared with other analysis tools, we can implement analysis algorithms based on Tai-e's IR with more concise code, which also increases the readability and facilitates code maintenance for team members."

3 FUNDAMENTAL ANALYSES

Static analysis approximates how abstracted data flows along the control structure of a program according to the language semantics and runtime environment. Accordingly, a static analysis framework should offer fundamental facilities to produce such control structures like control flow and call graphs (which are analysis-friendly structures that enable classic graph-based analysis algorithms) to develop various data flow analyses; besides, we need a pointer analysis to compute abstractions of the possible values/relations of pointer variables (points-to/alias information) in a program that are required by many other fundamental analyses and clients [60, 65]. In this section, we introduce what design choices are made by different frameworks for these two fundamental facilities: *pointer analysis* and *control/data flow analysis*, and explain how Tai-e is inspired by and differs from classic frameworks in designing them.

3.1 Pointer Analysis (Alias Analysis)

"Pointer analysis is one of the most fundamental static program analyses, on which virtually all others are built." [36]. Soot offers a pioneer (context-insensitive) pointer analysis system for Java called Spark that is highly optimized and runs very fast [35], and Wala also implements a pointer analysis system with context-sensitivity enhancement [65]. Doop [10] is another classic pointer analysis framework that is full of clever and useful designs. Unlike Soot, Wala and Tai-e which are imperative (implemented in Java), Doop is fully declarative and implemented in Datalog, and it is considered as the mainstream platform to implement and compare different newly proposed pointer analysis algorithms for Java in the last decade [13, 25–28, 30, 37–39, 61, 62, 68–70]. All of these frameworks implement the same Andersen-style algorithm [2] as the core of

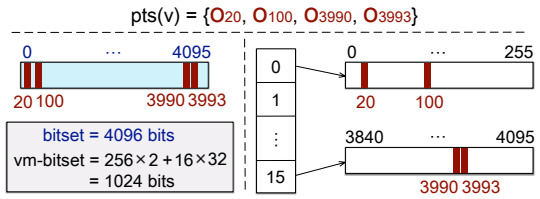

Figure 2: A case for regular (2^{12} = 4096) bitsets and (one-level) virtual memory-like sparse bit sets (vm-bitset) used in Tai-e.

pointer analysis; however, different choices are made by them for the following key points that need to be considered when designing a pointer analysis system for Java:

- a *representation* of points-to information
- a *context manager* for handling context sensitivity[1]
- a *heap manager* for modeling heap objects
- a *solver* for propagating points-to information

Due to limited space, the first two design points, where Tai-e differs more from other frameworks in them, are discussed below.

Representation of Points-to Information. Pointer analysis requires a uniform data structure to effectively represent the points-to set associated with each variable in a program. Both Spark and Wala adopt a hybrid points-to set: when the size of set is less than certain value, they use array to store the pointed-to objects; otherwise, a regular bit set is considered to represent the points-to set. Tai-e follows this hybrid approach, but the bit set is designed differently.

Assume a regular bit set uses 2^{12} = 4096 bits to represent 4096 pointed-to objects stored in the points-to set of any variable that points to them. Figure 2 depicts a case that variable v has a points-to set $\{O_{20}, O_{100}, O_{3990}, O_{3993}\}$, and the four objects inside are presented by 1 (in red color) and the other bits remain 0 in the 4096-bit set (see left-hand side of the figure). It is not hard to see that many bits are wasted when there are only a few pointed-to objects. To address this issue, Tai-e adopts a scheme called "virtual memory"-like sparse bit set [11] to represent points-to sets, as shown in Figure 2. In this case (see right-hand side of the figure), 16 integers (each occupies 32 bits) are used as pointers for referring to maximally sixteen 256-bit sets (also 4096 bits in total). Since there are only four objects to represent in this case, two 256-bit sets are enough to store them, totaling $1024 = 2 \times 256 + 16 \times 32$ (space for storing pointers) bits instead of the 4096 bits required by regular bit set.

In practice, to save more space, instead of the one-level page table used in the above example, Tai-e adopts two-level page table (like the concept in virtual memory) for referring to objects, and the table size is dynamically determined according to the number of pointed-to objects. Compared to regular bit set (used in Soot and Wala), the "virtual memory"-like sparse bit set in Tai-e helps save on average 23% (up to 40%) memory for context-sensitive pointer analysis. When we constrain the memory size to a limited one (like 8G for a laptop), this approach helps Tai-e scale for more benchmarks. Like work for C language [6], more attempts for designing new bit sets are encouraged for Java. Note that as Doop is declarative, its points-to set representation is not accessible to analysis developers, and different Datalog engines may use different representations [3, 10].

[1]Unlike pointer analysis for C/C++, context sensitivity is much more practically useful than flow sensitivity in improving precision for Java [60]

Context Manager. Context sensitivity is the most widely used approach to improving precision of Java pointer analysis [60, 65], and we need a strategy to manage various context-sensitivity variants (call-site- [59], object- [47, 48] and type-sensitivity [61]) with different context lengths, for both method calls and heap objects.

Soot does not have an effective context-sensitive pointer analysis system: Spark is context-insensitive; `Paddle` is a BDD-based context-sensitive pointer analysis [34] of Soot and it has been shown to be noticeably less efficient than Doop [10] and hasn't been maintained for years. Doop offers a set of elegant rules to deal with context sensitivity. However, due to the limitation of Datalog, for each context length for the combination of method calls and heap objects, developers have to write a separate implementation for context-sensitive analysis, resulting in redundant code.

In contrast, Tai-e is imperative and it can easily treat context length as an input parameter to the same implementation of context-sensitive analysis. Wala only provides context management for method calls, and its heap contexts directly inherit from the ones selected for the method that includes the allocation site of the heap object. Compared to Wala, Tai-e offers more flexible context management: developers can specify the contexts for both method calls and heap objects, e.g., 3-call-site (or 2-object) sensitivity for method calls and 1-call-site (or 1-object) sensitivity for heap objects.

Tai-e also provides facilities to develop selective context sensitivity (now a hot research topic of Java pointer analysis) that scales for large and complex Java programs with good precision. Now many state-of-the-art selective pointer analyses like Zipper [37], Zipper^e [39], Scaler [38] and Mahjong [70] have been implemented in Tai-e, serving as a uniform pointer analysis framework to compare and develop new context-sensitivity approaches.

Below, we list some feedback (from different developers) for Tai-e's pointer analysis system compared to Soot's and Wala's, which are all imperative that most developers are familiar with (their efficiency and completeness are evaluated in Section 6).

> *"Unlike Tai-e, the code of Wala's pointer analysis is highly coupled with other analyses, which makes the code hard to read."*

> *"In Tai-e's pointer analysis framework, the abstractions to the key concepts of pointer analysis (e.g., pointers, objects, contexts, allocation sites, method calls, etc.,) is easy to understand and utilize, while the same concepts in Soot's is hard to comprehend." "It is easier to design my pointer analysis in Tai-e than in Soot"*

> *"It is really easy to extend and implement new pointer analysis based on Tai-e's pointer analysis system, and by contrast, Soot's extension is not good."*

> *"Regarding pointer analysis, the design of Tai-e's points-to sets is very clear, but Soot's is strange. It is hard for me to understand when reading its code even though it can be eventually understood."*

> *"Compared to other related tools, Tai-e integrates various state-of-the-art top-down pointer analysis algorithms, on which high-precision call graphs can be generated. This is crucial for enterprises to do accurate code change impact analysis and testing, etc."*

3.2 Control/Data Flow Analysis

The algorithms for building control/data flow analyses (e.g., control flow graph creation and various data flow analyses like live variables analysis) are standard and well understood in the field of compilers. However, as the providers that support fundamental facilities to build these analyses, static analysis frameworks may adopt different strategies in design details. We next introduce how Tai-e, Soot, Wala, SpotBugs and Checker make choices for certain key design points to ease the development of control/data flow analyses.

Control Flow Analysis. Building control flow graph (CFG) is the major task of control flow analysis, and despite its basic algorithm is standard, its effectiveness for facilitating users to build analysis varies. We use two examples to explain.

Edge categories. Different from Tai-e and SpotBugs, Soot, Wala and Checker do not categorize edges of CFG, such as `IF_TRUE`, `IF_FALSE`, and `CAUGHT_EXCEPTION`, etc. These well-categorized edge information will ease to develop certain analyses like path-sensitive and branch-correlated analysis, or perform exception-specific handling. Compared to SpotBugs, Tai-e provides additionally useful edge information, e.g., it labels the `switch` edges with case values, and the `exception` edges with concrete exception types. Note that developers can also parse out the edge information in Soot and Wala by resolving the related nodes and IR in their analyses, but doing so would be inconvenient and not easy to use. While Checker does not categorize edges, it does categorize its CFG nodes (basic blocks) into types such as `ConditionalBlock` and `ExceptionBlock`, allowing developers to retrieve equivalent edge information through the APIs of these specific blocks.

Exception handling. Regarding CFG for Java, an essential factor that affects its effectiveness is how to statically resolve Java exception, and it is seen as the most critical source of incompleteness in program analysis that developers indicated should not be overlooked [14]. There are explicit and implicit exceptions: the former is thrown by `throw` statement and is caught by the corresponding `catch` statement if their types are matched, and the latter is thrown implicitly by JVM. A complete CFG should consider both of them, but in many cases, there is possibly a huge number of implicit exceptional control flows in a program and they do not interact much with the normal flows (explicit exceptional control flows do as logics are often implemented in their `catch` body), and thus consider them in CFG may reduce analysis precision (and usability). Unlike Wala, SpotBugs and Checker, Tai-e and Soot distinguish explicit and implicit exceptional control flows and allow users to decide which ones should be added to a CFG. Moreover, Tai-e implements state-of-the-art exception analysis [9, 29] that resolves exception significantly more precise (sometimes also more complete) than others, and offers the analysis as an option for constructing CFG.

Data Flow Analysis. To implement an analysis, developers typically follow the interfaces provided by a data flow analysis system to specify (1) data facts abstraction and initialization, (2) the transfer function for approximating different statements, and (3) the meet or join operation for merging data facts at control flow confluences, while making their analyses monotonic and safe-approximated. Below, we take two examples to illustrate how different designs may lead to different perceptions of use for developers.

Data facts initialization. Unlike Tai-e, SpotBugs, Checker and Soot, Wala does not allow to initialize data facts in the analysis; instead, it puts the related API in the solver, and thus every time we write a new analysis that needs different initialization, we have

to additionally implement a new solver to override the API that is responsible for initializing data facts. We argue that an elegant design is to have just one solver to drive multiple data flow analyses, so that developers only need to focus on the implementations of their analyses (no need to know the details of solver).

Edge transfer functions. Unlike Tai-e, SpotBugs and Wala, Soot and Checker do not explicitly support edge transfer functions. Edge transfer functions differ from node transfer functions in that they allow distinct data facts to be sent to various successors of a particular node along the edges between these nodes, utilizing branch information (e.g., fact D is propagated when expression E's value is true) to create more effective analysis. When the body of edge transfer function is left empty, it is regarded as an identity function to directly propagate the OUT fact of its source node to the IN fact of its target node (i.e., at this point, the analysis will change back to the normal one where only node transfer function is in charge). However, in Soot, to leverage branch information, developers need to extend a special analysis called `BranchedFlowAnalysis` and implement the logics for both edges and nodes in its node transfer function, which is inconvenient and a bit cumbersome in design. While Checker does not directly support edge transfer functions, it distinguishes the results of node transfers into different kinds, such as those for the `then` and `else` branches. The solver will propagate the `then` (`else`) result to the corresponding successor along the `then` (`else`) branch, enabling developers to achieve the same functionality as edge transfer functions. This method may require handling edge-related facts in node transfer functions, coupling the analysis logic of node and edge transfer functions.

Checker stands apart from the other frameworks we discuss here in that it enhances the type system of Java by enabling developers to write qualifiers (essentially Java annotations) for types, and perform analysis through type qualifier inference [52]. This scheme has proved effective in practice [1, 5], and we may investigate how to use annotations to assist in static analysis in Tai-e going forward.

Control/data flow analysis is very mature and as discussed above, what we have improved in Tai-e is mainly to make it more convenient to develop sophisticated control/data-flow analyses, or to make the CFG more complete. So far, we received its feedback from only two developers, and the reason why they felt easier to implement control/data flow analysis on Tai-e is due to the better design of Tai-e's IR. We expect more feedback for this system in the future.

4 NEW ANALYSIS DEVELOPMENT

A static analysis framework should offer mechanisms to incorporate new analyses, from intraprocedural to interprocedural ones, and Tai-e supports such facilities to conveniently develop and integrate new analysis, as indicated by this feedback from a developer:

> *"When developing a new analysis, ideally, developers should be able to rapidly identify their needed interfaces, data structures and graphs etc., provided by the framework. In Soot, there are various types of CFGs, etc., and it is not easy to understand their relations and the purpose or usage of certain pieces of code. However, I think that most novice developers can quickly find out the interfaces/classes, etc., that they need in Tai-e."*

Due to space constraints, in the rest of this section, we only introduce how Tai-e and other frameworks design for developing a very important class of analyses that need to interact with pointer analysis [60, 65], e.g., fundamentals like reflection analysis [40, 41] and exception analysis [9, 29] as well as clients like bug finders [12, 50] and security analyzers [4, 20, 43].

Past Work. Doop naturally supports such interactive analysis and is able to yield elegant implementation, benefiting from Datalog's declarative ability. However, Doop is also limited by Datalog in implementing analysis that requires non-set-based lattices [45, 67], and it is hard to optimize specific analyses as Datalog solver adopts analysis-independent data structures and execution strategy [22]. As a result, imperative frameworks that facilitate such interactive analysis are in high demand. As representatives, Soot lacks this backing, whereas Wala does.

Wala provides a scheme to add new analysis that interacts with pointer analysis, but in a limited way. Briefly, developers need to implement an interface called `ContextSelector` to specify related call sites (of certain APIs) which the new analysis aims to model based on the points-to results; for example, to analyze reflective call `v = c.newInstance()`, developers encode `ContextSelector` to identify this call site and retrieve the `Class` objects, say `C0`, that are pointed to by `c` via pointer analysis. Then, developers need to implement `ContextInterpreter` to generate different fictitious but effect-equivalent IRs (e.g., `v = new T(); v.<init>()`: allocating an object and calling its constructor in this case) according to the resolved types of `C0` (say `T`). Then these generated IRs are fed back to pointer analysis to continue the resolution for this reflective call.

This scheme is straightforward to understand but we argue that it has some limitations to facilitate interactions with pointer analysis.

- First, for certain analyses, it is insufficient to interact with pointer analysis by only giving developers a way to concentrate on and resolve related call sites; capabilities for monitoring the points-to information of specified variables are required. For example, in exception analysis, if the points-to set of variable `e` in `throw e` is changed, the points-to set of the corresponding `catch` variable should also be updated.
- Second, in many situations, it is simpler to update call graph edges or points-to sets directly, which can prevent the creation of excessive amounts of fictitious IR code and the need to invoke the solver to reanalyze the created code. Thus, a framework should additionally offer a mechanism to perceive changes for any variables of a program in order to accommodate more analyses; besides, it should provide a way to directly adjust the points-to results and call graph edges for an easier or more effective engagement with pointer analysis.

Helm et al. [22] present an approach to collaborating various analyses on the fly, even with different analysis lattices. However, this approach is too complex to adopt for addressing our problem, because many constraints must be put in place by developers, including encoding the rules that govern how control engine should interpret and communicate the results of one analysis to others.

To practically facilitate the development of new analysis that needs to interact with pointer analysis, in Tai-e, we introduce a simple yet effective method called *analysis plugin system*. Currently a dozen analyses in Tai-e are built on top of this system, including fundamentals like reflection and exception analyses, clients like taint analysis, utility tools like analysis timer and constraint checker

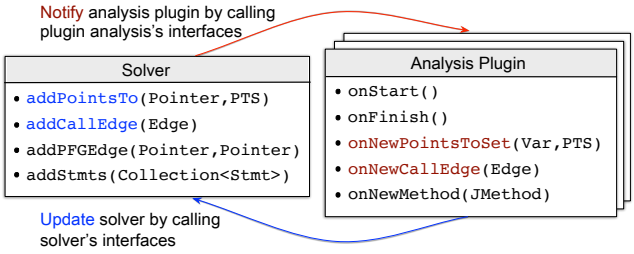

Figure 3: Overview of Tai-e's analysis plugin system for developing new analyses that interacts with pointer analysis, which can be seen as an instance of the *observer* pattern.

(for debugging), modern language feature handling like lambda expression and method reference analyses, runtime environment modeling like native code and thread modeling, and so on.

In summary, current frameworks either lack an imperatively interactive mechanism (Soot and Doop), have overly complex mechanisms ([22]), or have limited capabilities (Wala). In contrast, Tai-e's plugin system offers a comprehensive range of interactive features, including the ability to add new points-to sets, call graph edges, and generate IRs. Its simple interface design consists of only one interface and a few APIs to implement, making it easy to learn and use. These features allow developers to implement sophisticated analyses without requiring knowledge of the implementation details of pointer analysis. Below we explain its basic idea.

Basic Idea. We explain how this analysis plugin system works. As shown in Figure 3, this system includes a pointer analysis solver and a number of user-defined analyses that communicate with it. Each of these analyses is referred to as an analysis plugin that needs to implement interface `Plugin` of Tai-e. The interactions between pointer analysis solver and analysis plugin are carried out by calling each other's APIs. The *core* APIs of `Solver` and `Plugin` are highlighted in blue and red, respectively. *The* `Solver` *APIs have been implemented in Tai-e, and developers only need to implement the related APIs of* `Plugin` *to develop new analysis.* The additional auxiliary APIs are optional and designed to make it easier to create specific functionalities; for example, `addStmts` of `Solver` can be called to simulate the effect of specific call sites, which is similar to the generated-IR approach of Wala mentioned above.

Let us illustrate the basic working mechanism that drives those core APIs. Assuming you are implementing the `onNewPointsToSet` method of an analysis `Plugin`, this means whenever an interested variable's (parameter `Var`) points-to set (parameter `PTS`) is changed (i.e., it points to more objects), you need to encode your logic to reflect the side effect made by this change; the final consequence of such an effect, from the perspective of pointer analysis, is to modify the points-to set of any related pointers or to add call graph edges at related call sites. Accordingly, in the implementation of `onNewPointsToSet`, you should call the `addPointsTo` or `addCallEdge` methods of the `Solver` to alert it of these changes.

Conversely, during each analysis iteration, the `Solver` will automatically invoke the `onNewPointsToSet` and `onNewCallEdge` methods of every `Plugin` to notify them of any changes to the variables' points-to sets or call graph edges, respectively. As a result, to add a new analysis that interacts with pointer analysis, developers just need to implement a few methods of `Plugin` in accordance with the requirement, as previously described.

```
1  class TaintAnalysis implements Plugin {
2    Set<Pair<Var, Var>> transferVars;
3    void onNewCallEdge(Edge edge) {
4      if edge.target is source:
5        o = heapModel.getMockObj("TaintObj", edge.cs)
6        solver.addPointsTo(edge.cs.lhsV, o)
7      if edge.target is transfer:
8        foreach (from, to) in getTransfer(edge.target, edge.cs):
9          transferTaint(solver.getPointsToSet(from), to)
10         record (from, to) in transferVars
11   }
12   void onNewPoinstToSet(Var v, PTS pts) {
13     foreach (v, to) in transferVars:
14       transferTaint(pts, to)
15   }
16   void transferTaint(PTS pts, Var to) {
17     foreach o in pts:
18       if o.desc is "TaintObj":
19         solver.addPointsTo(to, o)
20   }
21   void onFinish() {
22     foreach (sink, param_i) in sinks:
23       foreach callsite cs in solver.getCallersOf(sink):
24         foreach o in solver.getPointsToSet(cs.args[param_i]):
25           if o.desc is "TaintObj":
26             report(o, cs)
27   }
28 }
```

Figure 4: Plugin of taint analysis (pseudocode version).

Case Study. To better understand how to build new analyses on top of this plugin system, let us look at an example of taint analysis (its pseudocode is depicted in Figure 4). We recommend following the figure together with our text explaining the scenario below.

The three lines of code below outlines a typical case for taint analysis: sensitive data from an object, say `o1`, returned by `x.source()`, is combined with an object, say `o3`, pointed to by `s3`, returned by `s2.concat(s1)`. Taint analysis reports that line 3 contains a leak of `o1`'s sensitive data as `o3` is used as an argument by a sink method.

```
1 String s1 = x.source();
2 s3 = s2.concat(s1);
3 y.sink(s3);
```

As taint analysis is developed as an analysis plugin, it needs to implement some of `Plugin`'s methods in Figure 3. In our case, as shown in Figure 4, two core methods `onNewCallEdge` (line 3) and `onNewPointsToSet` (line 12), and an auxiliary method `onFinish` (line 21) are considered.

Let us first examine `onNewCallEdge`. Assuming we are handling call site `1: String s1 = x.source()` in the code snippet above, and for this call site, the pointer analysis solver has resolved a new call graph edge from edge source (which is a call site denoted as `edge.cs`), i.e., call site 1, to edge target, i.e., the dispatched method `source`; then the solver notifies the taint analysis plugin and passes this new edge to parameter `edge` in line 3, by calling plugin's `onNewCallEdge` method (the solver code is omitted).

Then the plugin code checks what the `edge.target` is. If the target is a sensitive source method, denoted as `source` (line 4) (the `source`, `transfer` and `sink` methods are specified in the configuration file of taint analysis), a mocked taint object is created (line 5) and the plugin updates the solver that the left-hand side variable of this call site (`edge.cs.lhsV`), namely `s1` (of the above code), should point to the taint object that was just created (line 6).

If the target is a transfer method (line 7), e.g., the `concat` method at call site `2: s3 = s2.concat(s1)`, by which taint objects could be transferred from its parameters (e.g., `s1`) to other variables at the call site (e.g., `s3`) [20], for each of such transfer relation, denoted as `(from,to)` in line 8, the taint objects that are pointed to by `from` should be added in the points-to set of `to` (line 9 and lines 16 – 19). As a result, `s3` now points to the taint object transferred from `s1`. Actually, this transfer relation is also recorded in a data structure called `transferVars` (line 10), which is equivalent to add a taint-relevant flows-to edge summarized for transfer method.

Then the code of `onNewPointsToSet` (line 12) is easy to understand: whenever a new taint object flows to parameter `v`, and `v` is also the `from` variable recorded in `transferVars` mentioned above, the corresponding flows-to variable `to`, should also point to this taint object (line 14) for correctness. `onFinish` (line 21) will be called by solver after it reaches the fixed point. For each sink method and its sensitive parameter `param_i` (line 22), `onFinish` identifies all its call sites `cs` and checks if any taint objects flow to `param_i` (lines 23 – 25), and if they do, it reports this taint information.

We will discuss the usefulness of this analysis plugin system (with developers' feedback) in RQ2 of Section 6.

5 MULTIPLE ANALYSES MANAGEMENT

In many circumstances, an analysis depends on the outcomes of others, and it is helpful if the framework can provide a mechanism to coordinate multiple analyses. Below, we briefly discuss two crucial issues: how to configure an analysis and its dependencies, and how to save the outcomes of one analysis and access them in another?

Configure Analysis and its Dependencies. Wala has no explicit management for multiple analyses. In Soot, to add a new analysis, developers need to hard-code to add his implemented `Transformer` into Soot, while Tai-e supports registering *all* new analyses (and their dependencies) in the framework through one configuration file and then drive them via reflection automatically, enabling code decoupling. In addition, before running an analysis in Soot, users must explicitly list any dependent analyses (including those that depend on the dependent ones) in the command. This approach is cumbersome and prone to mistakes for users who are unfamiliar with the framework. In Tai-e, the dependency resolution is automatic by analyzing configuration files, ensuring the correctness of execution order for all dependent analyses (if each analysis is configured correctly); besides, this approach frees up developers to concentrate on the specification of their own analysis, and saves their effort of writing command options when running an analysis. Compared to SpotBugs, Tai-e is more flexible in resolving analysis dependencies, by supporting conditional logics to describe analysis options and dependencies. In summary, in order to facilitate simple usage, maintenance, and troubleshooting in terms of configuring and conducting analyses, Tai-e strives to guide users to modify code or configurations as little as possible.

Store/Access Analysis Results. Storing and accessing analysis results may seem like a minor concern that doesn't need to be discussed, but a good design, despite the fact that it may not look technical, can nevertheless produce a favorable user experience. Unlike Tai-e and SpotBugs, Wala and Soot do not have a uniform mechanism to manage analysis results (Soot only stores the results of some analyses like pointer analysis in the singleton instance of `Scene`). In SpotBugs, users need to remember different methods and arguments to obtain related results for various kinds of analyses. In Tai-e, users only need to remember one method `getResult(id)` (`id` is the analysis name in configuration) for all types of analyses, including method-, class- and program-level analyses. The straightforward user interface for accessing analysis results benefits from the fact that Tai-e automatically stores analysis results in different locations according to various types of analyses. As a result, users no longer have to bother trying to memorize complicated methods and specify additional arguments to retrieve results.

Below, we list two feedback from different developers about the capability of multi-analysis management in analysis frameworks.

> *"Compared to Tai-e, Soot lacks the management of different analyses. So new developers are unclear about what analyses have been provided by Soot, and developers in Soot's community can hardly share their implemented analyses with each other; besides, there are dependencies between analyses that, if not clarified, can cause many issues to development and testing."*

> *"Tai-e provides many built-in analyses and it is easy to combine different analyses in Tai-e, but with Wala, you need to manually engage with different analyses to combine them."*

6 EVALUATION

- **RQ1**: As the primary goal, is Tai-e really developer-friendly (i.e., easy to learn and use) compared to the other general Java static analysis frameworks, Soot and Wala?
- **RQ2**: As Tai-e's key component, is the analysis plugin system really helpful from the view of analysis developers?
- **RQ3**: What are the main shortcomings of Tai-e, and what are the threats to validity of the study?
- **RQ4**: Although efficiency is not Tai-e's primary objective, it would be helpful to know how well it compares in terms of analysis speed to other related state-of-the-art frameworks.

To address RQ1 to RQ3, we carried out a user study as outlined in Section 1. Below, we first provide background information on the study and then explore the four research questions respectively.

6.1 Background of Study

We offered a free six-hour static analysis tutorial to graduate students in our department. As part of the tutorial, we assigned a task: implementing a reflection analysis using Tai-e and Soot/Wala. We taught the analysis algorithms but did not provide instructions on how to implement them on these frameworks. Participation in this assignment was voluntary, and not mandatory for students.

We also conducted a survey among the participating students, and obtained feedback from 16 out of 32 students (the survey questions are provided in the following sections). Prior to the survey, we informed the students to provide honest feedback about their experience implementing the analysis on different frameworks. We told them that their feedback would be helpful, and we may consider integrating their code into Tai-e if it meets our high standards. This served as encouragement for them to complete the time-consuming assignment and participate in the survey. The students were not informed that we would be evaluating these frameworks in our paper, and they were not compensated for filling out the survey.

Out of the 16 respondents who provided feedback, five were from our group and 11 were from other groups, working on research topics such as program analysis, AI for SE, and system software. Prior to the study, three respondents were familiar with Soot/Wala only, one was familiar with Tai-e only, and four were familiar with both frameworks. They had experience in developing various types of analyses on these frameworks, like security analyzers, bug detectors, pointer analysis, and data-flow analysis. The remaining eight respondents were unfamiliar with either framework.

6.2 RQ1: Is Tai-e Developer-Friendly?

Developer-friendliness is very crucial since it reveals whether or not developers would want to create their analyses utilizing a framework and whether doing so would allow them to save effort on both learning and using a framework. Is Tai-e really developer-friendly? Actually, the comments throughout the paper made by different analysis developers provide some assistance in answering the question. Below, we present more findings from the study.

"Q: When developing your analysis on Soot/Wala or Tai-e, which one is more developer-friendly? Please give at least three reasons for why one is better than the other. "

For the survey question above, all 16 respondents replied that Tai-e is more developer-friendly than Soot/Wala. The top four reasons that contribute to Tai-e's developer-friendliness feature is that (1) Tai-e's framework structure, code and its API designs are easier to understand and use; (2) Tai-e's IR is more intuitive and simpler to use, enabling more concise implementations; (3) Tai-e's analysis plugin system is simple yet quite effective to use; (4) Tai-e has a more powerful pointer analysis system. In Sections 2 and 3, we provided examples of feedback for (2) and (4) respectively. Below, we give examples for (1) and will cover (3) in RQ2.

"Tai-e's design patterns, code style and the ability of extension is much better than Soot."

"The readability of Tai-e's code is good while Soot's is not. So Tai-e's code is significantly easier to understand than Soot's."

"Compared to Wala, Tai-e is more akin to a dependable static analysis framework for Java. It features a variety of beautiful OOP designs that make it simple to extend and overload code."

"I get the impression from Soot that "it is best to utilize only the functions it provides, do not construct the functions you require," whereas Tai-e is more conducive to programmers developing new static analysis. Thus compared to Tai-e, in my opinion, Soot is more of an analysis tool than a framework, as it lacks the features that a good framework oughts to have, including maintainability, code readability, and extensibility."

To quantitatively examine the developer-friendly feature of a static analysis framework, the most straightforward approach is to evaluate how much time it spends to develop a new analysis on different frameworks. Hence, we request the respondents to implement a set of functionalities for an important fundamental analysis, reflection analysis [41, 43] on both Tai-e and Soot/Wala (Soot or Wala are randomly assigned to the respondents who are unfamiliar with either), and leave the following question to answer:

"Q: How long does it take to develop the analysis on Tai-e and Soot/Wala, respectively? Specifically, what is the time taken for (a) being familiar with the framework, (b) thinking and designing your analysis on the basis of the framework, and (c) coding, debugging and testing, respectively?"

For the total time, they spent on average 29 hours on Tai-e while at least 49 hours on Soot/Wala (4 hours for (a), 10 hours for (b), and 15 hours for (c) on Tai-e, while 12 hours for (a), 15 hours for (b) and 22 hours for (c) on Soot/Wala). Students who have prior experience with a particular framework tend to complete tasks more efficiently. For instance, those who were familiar with Tai-e beforehand took 18 hours on average to complete the task, while those who were already acquainted with Soot/Wala took 24 hours to finish. Note that many of them claimed that functionality, debugging, and other challenges prevented them from completing the task on Soot/Wala, so the real time spent should be longer on Soot/Wala.

6.3 RQ2: Is Tai-e's Plugin System Helpful?

As explained in Section 4, although Doop-like declarative framework is nature to implement interactive analysis, the capability is limited by its Datalog language and underlying engine when developing or optimizing a wide range of analyses. Thus it is necessary to offer an effective system to support interactive analysis (with pointer analysis) imperatively. Now we examine whether Tai-e's analysis plugin system is practically helpful in the view of analysis developers by asking the following question in the survey:

"Q: Compared to Soot/Wala, do you find the analysis plugin system of Tai-e to be helpful or not? Why?"

All 16 respondents acknowledged that Tai-e's analysis plugin system is helpful. Nine of them in particular find it to be very helpful. We only list a portion of their remarks due to space constraints.

"Tai-e's plugin system is very helpful. This design shields me from the complexity of the underlying implementation of the pointer analysis, achieves separation of concerns, and allows me to focus on my own analysis. Soot has no such system, and before building the analysis, you have to understand the details of its pointer analysis, which is a great burden."

"Tai-e's analysis plugin system is very helpful, and Wala has no mechanism to inject customized objects in pointer analysis, and its scheme by implementing ContextSelector and ContextInterpreter to generate IR is limited for some analyses. So when implementing certain rules, one needs to interact with Wala's pointer analysis, which demands to understand its implementation details."

"Tai-e's plugin system has three benefits. First, it is time consuming to understand the principles of various analyses, but through this plugin system, we can focus more on our own analyses, simplifying the analysis development. Second, it is beneficial for both code extension and bug location to wrap additional analysis functionality in a plugin. Third, it greatly decreases the amount of code you have to create, which saves a lot of time and effort and somewhat raises the level of code quality. "

"According to my experience on developing analysis for enterprise applications, Tai-e's plugin system allows me to very conveniently add my logics to the process of pointer analysis, and meanwhile, it enables my analysis as well as the downstream security analysis to interact naturally. Additionally, during testing, developers can easily compose their needed functionalities by using this system. "

6.4 RQ3: Flaws of Tai-e and Threats to Validity

We show the two flaws of Tai-e that were most frequently brought up in respondents' feedback: (1) Tai-e's documentation is not enough, and (2) the strength of Tai-e's ecosystem is insufficient. E.g.,

"The main flaw of Tai-e is that it contains little documentation (including tutorials, JavaDoc, etc.). Although its advantages (e.g., self-explainable APIs) greatly compensate for this disadvantage, it would be more preferable with more informative documentation."

"Likely because Tai-e was recently created, its ecosystem is not as good as Soot's and has fewer projects available for reference."

Actually, there is no third most common negative feedback, but a few feedbacks about specific API design issues of Tai-e. For example:

"Why is `Visitor` *in the pointer analysis solver of Tai-e a private class? If the plugin system can override/extend* `visitXXX()` *methods, it would be very helpful."*

We acknowledge these flaws and as a new static analysis framework, we are actively creating a variety of analysis applications on Tai-e. We will enrich its documentation with more project examples, and continue to improve its API designs based on user feedback.

Threats to Validity of the Study.

- The participating students' feedback may be influenced by a bias towards the teacher's work, i.e., Tai-e.

We should acknowledge that there is a potential threat to the fairness of the study due to the participants coming from the same department as the teacher. To mitigate this issue, we requested that students provide honest feedback based on their experience. Furthermore, as this was a voluntary, non-credit-bearing tutorial without compensation, we expect that any bias would have a minimal impact on the results.

- This study may have been more persuasive, if more analysis developers from various backgrounds had participated in, and more of them were able to finish the survey.
- If more participants could implement various analyses (not just reflection analysis) on those frameworks, the results for the question about evaluating how much time is spent on different frameworks would be more convincing.

However, completing such a time-consuming study to compare different analysis frameworks requires a lot of work, and it is already not easy to get qualified feedback from 16 (out of 32) analysis developers. Moreover, the task becomes harder when the participants were asked to implement reflection analysis (a relatively sophisticated analysis) on unfamiliar frameworks. So if we ask participants to implement more analysis, their willingness to finish the survey would be significantly reduced.

- The time required to implement reflection analysis on different frameworks shows a discernible trend, but its precise numerical values should be interpreted with caution.

This is because the time taken to implement the analysis on each framework is influenced by several factors that are challenging to measure in the study, such as programming proficiency, familiarity with reflection analysis and the framework, and code quality, which vary among different students. Therefore, the time cost results merely indicate a trend that Tai-e can potentially facilitate or accelerate the development of specific analyses.

6.5 RQ4: Is Tai-e Efficient?

Although efficiency is not Tai-e's primary goal, it would be helpful to know how well it performs in analysis speed compared to other relevant state-of-the-art frameworks. This is because, despite the framework being developer-friendly, developers may still not want to base their analysis on a framework whose provided underlying analysis runs very slowly. As pointer analysis serves as the basis on which virtually all other analyses are built [36], its analysis speed is important for all its clients. In the rest of this section, we first evaluate the performance of pointer analysis and then the data-flow analysis, the other fundamental component of Tai-e. We consider all standard Java DaCapo benchmarks [7] plus several large real-world applications that are often used in recent literature [27, 39, 44, 68]. Due to limited space, we only show the summarized results.

Table 1: Completeness and efficiency of pointer analysis.

Tool	Recall		Context Insensitivity			2-obj	2-call
	#reach	#edges	Time (s)	#reach	#edges	Time (s)	Time (s)
Tai-e	95.9%	91.3%	32.8	19,517	145,903	503.0	978.7
Qilin	90.5%	83.5%	47.2	18,927	143,503	522.2	2145.4
Doop	78.1%	68.4%	113.6	14,934	109,486	334.8	2202.3
Soot	81.3%	73.2%	41.4	15,376	130,337	N/A	N/A

Pointer Analysis. Table 1 shows the average results for each program by running the pointer analysis of Tai-e, Qilin [21] (a recently released imperative pointer analysis tool), Doop and Soot (Spark), respectively (Wala is not shown as unlike others, it does not accept the output of dynamic reflection analysis as its input, and it runs noticeably slower than others in context sensitivity; we found that Doop is not taking full advantage of the reflection information, and thus its recall results are not good for these cases).

Analysis's speed heavily depends on its completeness (e.g., code coverage), as it typically takes longer time to analyze more code. As a result, we conduct recall experiments to record the amounts of real reachable methods and call graph edges that are dynamically collected when running the benchmarks; accordingly, their recall rates are shown in the columns of #reach and #edges under "Recall", indicating how many real methods or call graph edges that are dynamically reachable, are successfully over-approximated by pointer analysis. So higher recall rate implies better analysis completeness.

In summary, Tai-e achieves better recall than all other frameworks for all cases (all programs and clients) while is able to run faster than others for virtually all cases in both context-insensitive and context-sensitive settings (For context sensitivity, we consider two widely used approaches: object- and call-site sensitivity with context length of two, denoted as 2-obj and 2-call in the table, respectively). That means although Tai-e analyzes more code, it still runs faster than other pointer analysis frameworks. Such good performance benefits from Tai-e's well treatment to language features like reflection resolution, native code modeling, etc., and various optimizations to its pointer analysis implementations.

Table 2: Efficiency of live variable analysis.

Tool	#Classes	#Methods	Time (s)	#Methods/s
Soot	674	6,073	0.14	33,109
Tai-e	674	6,074	0.42	11,952
Wala	369	3,017	0.89	3,684
SpotBugs	2,324	20,510	6.11	2,542

Data Flow Analysis. Table 2 shows the average results by running live variable analysis, which is the only data flow analysis that is provided by all of the four frameworks. Note that Checker was excluded from the comparison because the DaCapo benchmarks only offer bytecode, whereas Checker can only analyze source code. SpotBugs analyzes more classes and methods as it resolves all the code included in a Jar file, while the other frameworks resolves a class only when it is referenced by other classes. Hence, to fairly measure the efficiency, we add the last column #Methods/s to denote how many methods are analyzed per second for each program. Tai-e is more efficient than Wala and SpotBugs, but less efficient than Soot. After inspection, we found that Soot applies several sophisticated optimizations to its data flow analysis, but Tai-e does not (as a data flow analysis typically already runs extremely fast).

7 RELATED WORK

The most pertinent work has been compared and discussed throughout the paper. Below we discuss additional related work.

Lam et al. [33] give a retrospective of Soot by summarizing its main features, major changes (e.g., it consolidated singletons to support multiple runs, and supported to analyze incomplete programs), and future directions (e.g., to build faster startup and enhance interprocedural analysis). In addition, they mention some difficulties in developing Soot and suggest some desirable features for future compiler frameworks.

Schubert et al. [58] describe the lessons from building a data flow analysis framework for C/C++ [57], some of which are specific to framework's features while others are more general. For example, it would be beneficial for a framework to offer means like instrumentation to help debug analysis-related bugs. This seems useful and we may consider developing similar approach in Tai-e.

Sadowski et al. [56] summarize the lessons from building static analysis tools at Google. They advise integrating static analysis into workflow as early as possible, and do the analysis checks as compiler errors if possible (otherwise, developers often ignore analysis results). Integrating Tai-e's individual analyses into developers' workflows may need to modify the basic infrastructure of Tai-e, but that is an interesting subject that merit further investigation.

The study from Facebook [15] highlights the value of interprocedural analysis for identifying deep bugs and security vulnerabilities. Tai-e's analysis plugin system is specifically designed to make it easier to develop a variety of sophisticated interprocedural analyses that interact with pointer analysis to address the issues in [15] such as precise virtual-call resolution and static value-flow tracking.

Some studies evaluate static analysis tools from the view of users' needs [14, 16, 49]. For instance, a good static analysis tool should produce high-quality warning messages that offer information on what might be wrong, why it should be fixed, and how it could be fixed, have low false positives of analysis results, and support the integration of user knowledge, and more.

`Chord` [50] is a static analysis framework for Java that is written in Java and Datalog with `bddbddb` [73] (a BDD-based implementation of Datalog) serving as the Datalog solver. As explained in Section 4, despite being elegant when implementing various analyses, Datalog has limited expression capacity and optimization potential. `Chord` is particularly known for its capability to detect concurrent errors such as data races, and we will develop these clients imperatively (rather than declaratively) in Tai-e.

`OPAL` [17] is a static analysis framework for Java written in Scala. The collaborative analysis approach [22] is implemented in `OPAL`; unfortunately, as explained in Section 4, the approach is too complex to be effective for our problem: developing analysis that interacts with pointer analysis. In addition, the same authors [55] conduct interesting study to assess the soundness of call graphs produced by call-graph algorithms (e.g., CHA) in various frameworks, emphasizing the importance of effectively handling language features.

We discuss TAJS [24], a classic static analysis framework for JavaScript and Node.js [18, 46, 51], as certain designs of Tai-e are inspired by it. One is the regression testing and the other is the initial idea of Tai-e's analysis plugin system, where the solver-plugins structure resembles TAJS's monitor approach, despite that their goals, methodologies and APIs are fundamentally different. For examples, their monitor approach is primarily used to gather analysis results and perform statistics such as recording the times a statement is accessed, or timing the amount of time an analysis takes to complete for each statement, and it lacks the fine-grained interactive interfaces of Tai-e's analysis plugin system. In addition, it requires that monitor interface implementations should not have side effects on the analysis state prior to the monitor scan; on the contrary, Tai-e's interface implementation for analysis plugin is to have side effect on the pointer analysis by invoking methods as explained in Section 4.

8 CONCLUSIONS

Although static analysis has made great strides and several popular analysis frameworks have emerged over the past decades, these frameworks are not that easy to learn and use for developers who rely on them to build analyses, as it is far from trivial to build a developer-friendly static analysis framework. This paper takes a step forward by systematically comparing and discussing the design trade-offs for the crucial components of a static analysis framework for Java, following the `HGDC` principle.

Our efforts are highly labor- and intelligence-intensive as for each design point, we must study and comprehend the code of those large and intricate frameworks full of complex analysis algorithms and implementations (this may be one of the primary causes of the paucity of papers on the design of static analysis frameworks). But such efforts are worthwhile as they aid in the creation of Tai-e, a developer-friendly static analysis framework for Java, and we have shown throughout the article that it works well in attaining its goal. We expect this work to provide useful materials and perspectives for building better static analysis infrastructures, and we will actively and constantly contribute to Tai-e by developing and incorporating more analyses and clients in the future.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by the Natural Science Foundation of China under Grant Nos. 62025202 and 62002157, and the Fundamental Research Funds for the Central Universities under Grant No. 020214380102. We also thank the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM '12)*. IEEE Computer Society, USA, 14–23. https://doi.org/10.1109/SCAM.2012.28

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. University of Copenhagen.

[3] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. 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* (Barcelona, Spain) *(SOAP 2017)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

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

[5] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: practical type-based null safety for Java. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 740–750. https://doi.org/10.1145/3338906.3338919

[6] Mohamad Barbar and Yulei Sui. 2021. Compacting Points-to Sets through Object Clustering. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 159 (oct 2021), 27 pages. https://doi.org/10.1145/3485547

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

[8] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1572272.1572274

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

[11] Bruce K.Haddon. 2010. Virtual-memory-like sparse bit set. https://github.com/brettwooldridge/SparseBitSet/blob/master/SparseBitSet.pdf.

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

[13] Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. 2021. Accelerating Program Analyses in Datalog by Merging Library Facts. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 77–101. https://doi.org/10.1007/978-3-030-88806-0_4

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

[15] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

[16] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. https://doi.org/10.1109/TSE.2020.3004525

[17] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614630

[18] André Takeshi Endo and Anders Møller. 2020. NodeRacer: Event Race Detection for Node.js Applications. In *Proc. IEEE International Conference on Software Testing, Verification, and Validation (ICST)*.

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

[20] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *PACMPL* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

[21] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[22] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 184–196. https://doi.org/10.1145/3368089.3409765

[23] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA) *(OOPSLA '04)*. ACM, New York, NY, USA, 132–136. https://doi.org/10.1145/1028664.1028717

[24] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*. Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

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

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

[27] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (nov 2020), 30 pages. https://doi.org/10.1145/3428247

[28] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. *PACMPL* 6, POPL, Article 58 (jan 2022), 29 pages. https://doi.org/10.1145/3498720

[29] George Kastrinis and Yannis Smaragdakis. 2013. Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *Compiler Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60.

[30] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 423–434. https://doi.org/10.1145/2491956.2462191

[31] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2022. Accumulation Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:30. https://doi.org/10.4230/LIPIcs.ECOOP.2022.10

[32] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*. 10:1–10:27. https://bodden.de/pubs/ksa+18crysl.pdf

[33] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*.

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

[35] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[36] Ondrej Lhotak, Yannis Smaragdakis, and Manu Sridharan. 2013. Pointer Analysis (Dagstuhl Seminar 13162). *Dagstuhl Reports* 3, 4 (2013), 91–113. https://doi.org/10.4230/DagRep.3.4.91

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

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

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

[40] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing Reflection Resolution for Java. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 27–53. https://doi.org/10.1007/978-3-662-44202-9_2

[41] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2 (2019), 7:1–7:50. https://doi.org/10.1145/3295739

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

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

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

[45] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

[46] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proc. 32nd European Conference on Object-Oriented Programming (ECOOP)*.

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

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

[49] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/3533767.3534374

[50] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. ACM, 308–319. https://doi.org/10.1145/1133981.1134018

[51] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.js Applications. In *Proc. 30th International Symposium on Software Testing and Analysis (ISSTA)*.

[52] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. ACM, 201–212. https://doi.org/10.1145/1390630.1390656

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

[54] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-Sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659

[55] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 251–261. https://doi.org/10.1145/3293882.3330555

[56] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (mar 2018), 58–66. https://doi.org/10.1145/3188720

[57] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*. Springer, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

[58] Philipp Dominik Schubert, Ben Hermann, Eric Bodden, and Richard Leer. 2021. Into the Woods: Experiences from Building a Dataflow Analysis Framework for C/C++. In *SCAM '21: IEEE International Working Conference on Source Code Analysis and Manipulation (Engineering Track)*.

[59] Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph. D. Dissertation. Carnegie Mellon University.

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

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

[62] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 485–495. https://doi.org/10.1145/2594291.2594320

[63] SpotBugs. 2018. Find bugs in Java Programs. https://spotbugs.github.io.

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

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

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

[67] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (oct 2018), 29 pages. https://doi.org/10.1145/3276509

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

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

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

[71] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*. IBM, 13. https://doi.org/10.5555/781995.782008

[72] WALA. 2006. Watson Libraries for Analysis. http://wala.sf.net.

[73] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[74] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/2491956.2462159

Received 2023-02-16; accepted 2023-05-03