Incremental Program Analysis in the Wild: An Empirical Study on Real-World Program Changes

Xizao Wang, Xiangrong Bin, Lanxin Huang, Shangqing Liu, Jianhua Zhao, Lei Bu†
State Key Laboratory for Novel Software Technology, Nanjing University, China
{wangxiz,xrbin,lanxinhuang}@smail.nju.edu.cn, {shangqingliu,zhaojh,bulei}@nju.edu.cn

***Abstract*—Incremental program analysis (IPA) has gained increasing attention as an effective approach for maintaining up-to-date analysis results by leveraging previously computed results in response to program changes. Consequently, a variety of IPA algorithms and tools have been proposed. However, their empirical performance in practical, real-world scenarios remains insufficiently investigated. To address this gap, this study presents a comprehensive examination of the current state-of-the-art in IPA evaluation. Specifically, we identify two key limitations: (1) the lack of standardized benchmarks reflecting real-world program changes, and (2) the inadequacy and imbalanced distribution of evaluation metrics.**

To overcome these challenges, we propose an automated pipeline for constructing real-world program change benchmarks and develop a unified incremental evaluation framework for systematically evaluating IPA tools. Using the proposed evaluation pipeline, we constructed large-scale benchmarks of real-world program changes—sourced from 4,084 commits across 20 Java projects—and systematically evaluated two IPA tools for Java. The results demonstrate that, although incremental analysis substantially improves efficiency compared to exhaustive analysis, existing IPA tools exhibit inconsistencies and markedly higher peak memory consumption. Finally, we distill practical insights from our findings to inform future research and development in the field of incremental program analysis.

***Index Terms*—incremental program analysis, empirical study**

I. Introduction

Incremental program analysis (IPA) [1], [2], [3], [4] has emerged as a promising approach for enhancing the efficiency of program analysis. Compared to other approaches with similar objectives, such as selective analysis [5], [6], [7], parallel processing [8], [9], and demand-driven analysis [10], [11], IPA is distinguished by its core principle of analyzing only the portions of code that have changed since the last analysis, thus avoiding redundant reanalysis of the entire codebase. This incremental nature is particularly beneficial in CI/CD pipelines [12], [1], where changes are typically frequent and localized. By enabling timely identification of the impact of changes, IPA facilitates the rapid resolution of issues and sustained maintenance of high code quality [13], [14].

A wide range of IPA techniques have been developed to support various program analysis algorithms, including type checking [15], [16], [17], data flow analysis [4], [18], symbolic execution [19], [20], [21], call graph construction [22], [1], and pointer analysis [23], [3], [24]. Empirical evaluations reported across these works consistently confirm the superior efficiency of incremental analysis over its exhaustive counterpart, particularly in terms of analysis time.

Despite the notable efficiency improvements reported for IPA techniques in the literature, existing IPA work suffers from several important limitations. A particularly salient issue is the predominant reliance on *small-scale, synthetic program changes* in their evaluations [3], [11], [2], [25]. For instance, Zwaan *et al*. [17] applied their framework on only three commits from each of three selected Java applications, while Lu *et al*. [23] evaluated their approach by randomly removing classes, methods, or statements. Although such setups facilitate controlled and repeatable experimentation, they fall short of reflecting the complex and heterogeneous nature of real-world software evolution. In industrial-scale development, program changes often involve extensive refactoring, intricate interdependencies, diverse coding practices, and continuous integration of new features and fixes—factors that pose substantial challenges to IPA techniques. To meaningfully advance the state of the art, there is a pressing need for large-scale empirical evaluations based on real-world program changes. Such studies are essential to provide a nuanced and realistic understanding of the effectiveness and limitations of existing IPA techniques in practical development environments.

This paper aims to address critical gaps identified in state-of-the-art research on incremental program analysis. We begin with conducting a comprehensive literature analysis of the state-of-the-art evaluation practices for existing IPA work. Given the substantial differences in program analysis among different programming languages and the need for a unified evaluation for existing IPA work, the literature analysis focuses specifically on IPA work for Java. To this end, we reviewed 25 IPA works for Java published in top-tier PL and SE venues since the inception of Java in 1996 (*i.e*., 1996–2025).

The literature analysis yields two key observations. First, although most works (84%) incorporate real-world projects in their evaluations, the majority (84%) rely heavily on synthetic program changes. This trend is primarily due to the lack of standardized benchmarks constructed for IPA evaluation. Second, the evaluation metrics employed in these works are insufficient and unevenly distributed. While all works assess efficiency, the primary advantage of IPA, only 32% examine memory overhead, and 20% consider correctness.

These observations underscore two key limitations in the field of incremental program analysis: *the lack of large-scale benchmarks capturing real-world program changes*, and *the*

† Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00194

need for a comprehensive evaluation framework that accommodates diverse and meaningful evaluation metrics.

Motivated by the aforementioned observations and in response to the key limitations in existing IPA work, we have developed an automated benchmark construction pipeline and a unified incremental evaluation framework that integrates real-world program changes with comprehensive, multi-metric evaluation. The benchmark construction pipeline combines change mining, build automation, and change statistics to systematically filter and compile program revisions that satisfy predefined criteria. To address the challenges of evaluating heterogeneous IPA tools—each with distinct input/output formats and runtime dependencies—we designed a modular, containerized incremental evaluation framework. This design ensures standardization and automation of the evaluation process across tools and metrics. The current implementation of the framework supports three core evaluation metrics: *efficiency*, *memory overhead* and *correctness*, thereby facilitating a more holistic and comparable assessment of IPA tools.

Leveraging the proposed benchmark construction pipeline and incremental evaluation framework, we conducted a systematic empirical study of existing IPA tools using the large-scale benchmarks of real-world program changes. Specifically, we constructed large-scale benchmarks comprising 4,084 commits from 20 open-source Java projects hosted on GitHub and automated their build processes. Within our evaluation framework, we manually containerized two representative IPA tools for Java (Reviser [4] and JavaDL [26]) to assess them across multiple evaluation metrics. Our experimental results reveal that incremental program analysis delivers substantial efficiency gains over exhaustive analysis for the vast majority (over 95%) of program changes. However, we also observe partial inconsistencies in analysis results and consistently elevated memory consumption, with peak usage exceeding twice that of exhaustive analysis in the worst cases. Based on these results, we distill eight key findings and articulate six actionable lessons to inform and guide future research and development efforts in incremental program analysis.

In summary, we make the following contributions:

- We present the first comprehensive literature analysis of IPA evaluation practices and identify two key limitations in the current landscape: the lack of standardized benchmarks of real-world program changes and the imbalanced use of evaluation metrics across IPA works.
- We developed an automated benchmark construction pipeline and a unified incremental evaluation framework to provide the necessary infrastructure for systematically evaluating IPA tools using real-world program changes.
- We constructed large-scale benchmarks comprising 4,084 real-world commits from 20 open-source Java projects, enabling a comprehensive evaluation of IPA tools. This study reports eight key findings and six actionable lessons to inform and advance future IPA research and development. All code and data are publicly available[1].

[1]https://sites.google.com/view/inceval

II. State of the Art

In this section, we conduct a comprehensive and systematic literature analysis of the current state-of-the-art in the evaluation of incremental program analysis.

Before delving into the SOTA analysis, we first provide a brief overview of the general workflow of IPA. In general, an IPA tool requires an initial exhaustive analysis to produce exhaustive analysis results. IPA then reuses existing results and only re-analyzes the code affected by changes for subsequent commits, thereby updating the results efficiently. For IPA evaluation, the input typically consists of a series of commits. The IPA tool operates according to the workflow described above, while the exhaustive analysis sequentially processes each commit. Evaluation metrics (*e.g.*, efficiency) are then employed to compare IPA against exhaustive analysis.

A. Research Questions

Our objective is threefold: 1) to identify and categorize commonly used benchmarks in existing research; 2) to dissect techniques for obtaining program changes; 3) to review evaluation metrics, assessing their effectiveness in evaluating performance of incremental program analysis. This investigation aims to uncover strengths and limitations of the current state-of-the-art, thereby providing guidance for the evaluation methodology presented later in this paper.

We aim to answer the following research questions (RQs):

- **RQ1.1 Benchmarks**: What benchmarks are used in the evaluation of existing IPA work?
- **RQ1.2 Changes**: How are the program changes for IPA obtained in the evaluation of existing work?
- **RQ1.3 Metrics**: What evaluation metrics are used in the evaluation of existing work?

B. Methodology

To answer these questions, we conducted a manual literature analysis in the field of IPA for Java. It is difficult to quickly and accurately find all relevant work through keyword searches, so we chose to start with the latest papers from top-tier PL/SE venues, determine whether they meet the criteria based on their content and evaluations, then gradually trace back to earlier work based on their related work and references, and repeat this process until no additional relevant work meeting the criteria is identified. More specifically, we began with papers published in the past three years (2023–2025) from venues including PLDI, POPL, OOPSLA, ECOOP, ICSE, ASE, FSE, ISSTA, TOPLAS, TOSEM, and TSE, and traced back to 1996 (a cutoff year chosen for the inception of Java) and our paper selection criteria were as follows: 1) the work must be related to IPA for Java; 2) the work must include experimental evaluations targeting IPA. Ultimately, we collected 25 papers, which cover all relevant work meeting the criteria since the inception of Java, and we believe these works are sufficient to reflect the current state-of-the-art in the field of IPA for Java.

For each research question, we manually assign labels to every publication by carefully reviewing the evaluation section of the paper, then group similar labels into categories, and

assign category names. Finally, we use the size of each category (*i.e.*, the number of publications) to indicate the relevance and answer the research questions.

C. Results

An overview of these results is presented in Table I. For each research question, we classify its results into several categories, accompanied by the number and percent of publications. It should be noted that the sum of results for all categories under each research question is not equal to 100%. Figure 1 presents the Venn diagrams illustrating the relationships among the categories of each research question.

1) RQ1.1 Benchmarks: For this research question, we investigate the benchmarks used in the evaluation of existing IPA work, as benchmarks are essential for assessing the performance and effectiveness of IPA techniques. Existing benchmarks can be broadly classified into three categories: *Existing Non-IPA Benchmarks* (**C1.1**), *Real-World Programs* (**C1.2**), and *Synthetic Benchmarks* (**C1.3**).

Non-IPA benchmarks represents benchmarks originally designed for exhaustive analysis evaluation or other purposes, such as DaCapo [27] and SPECjvm. Real-world programs comprises real-world open-source or commercial projects, which are manually selected for IPA evaluation. In fact, Non-IPA benchmarks are also real-world projects, we classified them separately only to distinguish them from manually selected projects. Synthetic benchmarks includes small and synthesized programs specifically created for IPA evaluation.

The most prevalent category is *Real-World Programs* (**C1.2**), which uses manually selected real-world projects as benchmarks and is attributed to 56% of all publications. 36% of all publications belong to category **C1.1**, in which existing Non-IPA benchmarks are used to evaluate IPA. Projects in these two categories are all real-world projects, and together they account for 84% of all publications. This indicates that real-world projects are considered crucial for evaluating IPA techniques, as they provide realistic scenarios and challenges that reflect actual software development practices.

The last category is *Synthetic Benchmarks* (**C1.3**), which is attributed to only 24% of all publications and uses synthesized programs as benchmarks. Among them, two publications [28], [17] simultaneously used both synthetic and real-world programs as benchmarks. The use of synthetic programs aimed to facilitate manual validation of the correctness of the result, while the other four publications (16%) [29], [30], [15], [31] exclusively used synthetic programs as benchmarks.

Finding 1: In addressing RQ1.1, we found that the benchmarks used in existing IPA evaluation fall into three categories. 84% of all publications use real-world projects, which come from either selected programs (**C1.2**, 56%) or Non-IPA benchmarks (**C1.1**, 36%), to evaluate IPA. Synthetic benchmarks (**C1.3**) are the least prevalent, used only in 24% of all publications; two publications combined synthetic and real-world programs to ease manual verification, while four others (16%) used only synthetic programs.

TABLE I: Categories for each research question with their respective number and percent of publications.

Research Question	Category	#Pubs	%Pubs
RQ1.1: Benchmarks	**C1.1** Non-IPA Benchmarks	9	36%
	C1.2 Real-World Programs	14	56%
	C1.3 Synthetic Benchmarks	6	24%
RQ1.2: Changes	**C2.1** Commit-derived Changes	6	24%
	C2.2 Synthetic Changes	21	84%
RQ1.3: Metrics	**C3.1** Efficiency	25	100%
	C3.2 Memory Overhead	8	32%
	C3.3 Correctness	5	20%

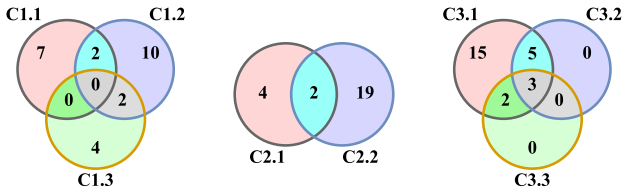


Fig. 1: The Venn diagrams for categories of the three research questions: **RQ1.1** Benchmarks (**C1.1–C1.3**), **RQ1.2** Changes (**C2.1–C2.2**), and **RQ1.3** Metrics (**C3.1–C3.3**).

2) RQ1.2 Changes: This research question aims to investigate how program changes for IPA evaluation are obtained in existing work. The existing approaches for obtaining program changes can be broadly classified into two categories: *Commit-derived Changes* (**C2.1**) and *Synthetic Changes* (**C2.2**). Commit-derived changes are extracted directly from version control systems (*e.g.*, git) by analyzing commit histories. This approach captures real-world modifications made by developers, providing authentic scenarios for IPA evaluation. Synthetic changes encompasses changes which are created by manually or programmatically modifying programs to simulate various types of changes.

Of all the publications, only six (24%) incorporated real-world commits to capture program changes (**C2.1**). Among these, JavaDL [26] tracks changed files and computes program changes at the AST granularity. Statix [17] implements incrementalization at the compilation unit granularity. Reviser [4] and SHARP [24] operate at the statement level, while IncCHA [1] aggregates changes at the method level. Krainz *et al.* [28] employs diff graphs for incremental pointer analysis.

In contrast, the majority (84%) of all publications resorted to synthetic changes (**C2.2**) for IPA evaluation. Specifically, 76% used synthetic changes exclusively. The primary reason for this trend is the lack of standardized benchmarks of real-world program changes, as stated by Szabó *et al.* [32]:

> "Unfortunately, there is no standard benchmark for incremental program changes available, so we chose to synthesize program changes that are likely to affect the analysis results…"

Among these publications, some introduce changes at the code level (e.g., adding or deleting statements) [33], [34],

[3], [23], [35], [36], while others directly synthesize changes on intermediate representations (IRs) required by specific analyses—for example, adding or deleting nodes and edges in some intermediate code graphs [37], [38].

Finding 2: In the vast majority of publications (84%), IPA evaluation is based on synthetic changes (**C2.2**), while only 24% adopt real-world commit-derived program changes (**C2.1**). This is mainly attributed to the current lack of standardized benchmarks specifically designed for IPA.

3) RQ1.3 Metrics: This research question focuses on the evaluation metrics employed in existing IPA work. Evaluation metrics serve as quantitative criteria for measuring the effectiveness and efficiency of IPA techniques. Existing metrics can be broadly classified into three categories: *Efficiency* (**C3.1**), *Memory Overhead* (**C3.2**), and *Correctness* (**C3.3**).

Efficiency measures the time taken by the IPA tool to analyze program changes, typically compared to exhaustive analysis time. This metric is crucial for assessing the practical usability of IPA techniques, as one of their primary goals is to provide timely feedback to developers. Memory overhead assesses the additional memory consumption incurred by the IPA tool during analysis, often in relation to the memory usage of exhaustive analysis, as incremental analysis may require maintaining additional state information typically. Correctness evaluates whether the results produced by the IPA tool are consistent with those of exhaustive analysis, ensuring that incremental updates do not compromise accuracy.

Table I reveals a notable disparity in evaluation metric adoption across existing IPA work. IPA efficiency (**C3.1**) is the most prominent, as all works (100%) report it, underscoring strong emphasis on computational efficiency and practical demand for timely updates of analysis results.

By contrast, memory overhead (**C3.2**) is addressed in only 8 (32%) of works, despite being a critical concern for IPA tools in practice, as they typically incur higher memory overhead than exhaustive tools. For instance, Liu et al. [3] observed 140GB peak memory usage, while Szabó et al. [2] reported IPA memory overhead up to $4.5\times$ that of exhaustive tools. This limited attention points to a potential gap in evaluating IPA scalability, especially for large codebases where memory constraints are a major bottleneck.

It is important to note that correctness (**C3.3**) is the most overlooked metric, appearing in only 5 publications (20%) [1], [3], [4], [34], [22]. This is concerning because correctness is fundamental to ensuring that IPA results align with those from exhaustive re-analysis, *i.e.*, consistency. Such alignment is a critical factor for the reliability of IPA in real-world applications. While some works [2], [37], [38] have theoretically analyzed the correctness of incremental algorithms, from an implementation perspective, algorithms do not always satisfy their assumptions nor are they always implemented correctly. Thus, empirical evaluation of correctness is equally essential.

Finding 3: Evaluation metrics in existing IPA work show a pronounced focus (100% of publications) on efficiency (**C3.1**), driven by the need for timely feedback, while memory overhead is addressed in only 32% of works (**C3.2**), potentially limiting scalability for large codebases. Critically, correctness is the most overlooked metric, appearing in only 20% of works (**C3.3**).

D. Discussion

Our analysis of the current state-of-the-art in incremental program analysis evaluation sheds light on several notable aspects of existing evaluation methodologies.

First, we observe that the majority of existing IPA works (84%) utilize real-world projects for evaluation (**C1.1**, **C1.2**), underscoring the importance of real-world projects in IPA evaluation. However, due to the lack of standardized benchmarks of program changes, only 24% of these works employ changes derived from real-world commits (**C2.1**), with most relying on synthesized changes (**C2.2**). This heavy dependence on synthetic changes reveals a significant gap between controlled research settings and the complexities of real-world software evolution, which may compromise the generalizability of IPA techniques in practical development scenarios.

Lesson 1: Diverse program changes derived from commits in real-world projects are essential for the systematic evaluation of IPA work. Currently, however, there is a lack of large-scale standardized benchmarks of real-world program changes in the field of incremental program analysis.

Furthermore, the evaluation metrics adopted in existing IPA research exhibit a notable disparity. Although all works prioritize efficiency (**C3.1**, 100%) which is motivated by the demand for timely feedback, memory overhead (**C3.2**) and correctness (**C3.3**) are addressed in only 32% and 20% of the works, respectively. The metric distribution indicates that existing research prioritizes speed over memory overhead and correctness, potentially compromising scientific rigor for immediate practicality. To advance IPA toward more robust real-world adoption, there is a clear need to balance evaluation efforts across efficiency, memory overhead, and correctness.

Lesson 2: The imbalanced distribution of evaluation metrics highlights the urgent need for evaluation of memory overhead and correctness, underscoring the necessity of a unified and balanced evaluation of existing IPA work across multiple evaluation metrics.

Overall, we observe key gaps between IPA research and empirical evaluation, resulting from both a lack of real-world benchmarks and an imbalanced distribution of evaluation metrics. The over-reliance on synthetic program changes implies that many IPA techniques may fail to handle the complexity of real-world development scenarios. Similarly, the imbalanced metrics reveal that measurements of usability in real-world software development scenarios remain insufficient.

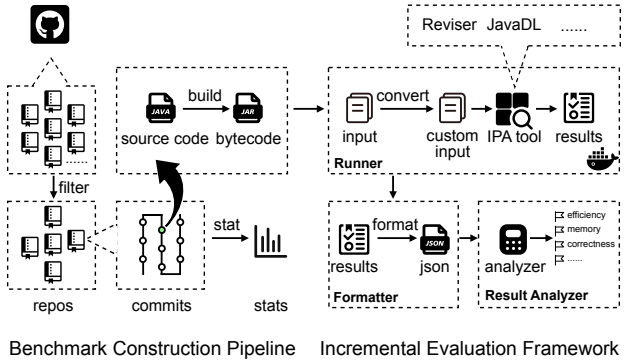


Fig. 2: The overview of the evaluation pipeline.

This analysis highlights two pressing imperatives for the IPA research community: constructing large-scale benchmarks of program changes that mirror real-world software evolution, and establishing comprehensive incremental evaluation frameworks that mandate multi-metric evaluation (efficiency, memory overhead, correctness, *etc*.). The lessons drawn here call for a paradigm shift toward more rigorous, representative evaluation practices that align research goals with the demands of real-world software development.

III. Methodology

This section details the comprehensive methodology we designed to enable large-scale evaluation of IPA techniques, focusing on real-world program changes and spanning multiple metrics. Specifically, we describe the design of two core components: an automated benchmark construction pipeline, which systematically collects and constructs real-world program changes from open-source repositories; and an incremental evaluation framework, which standardizes and automates the evaluation of diverse IPA tools across multiple metrics. Figure 2 shows the overview of these two components, which together form the backbone of our evaluation methodology.

A. Benchmark Construction Pipeline

Our methodology begins with automating benchmark construction. As highlighted in Section II-D, the lack of large-scale benchmarks of real-world program changes is a key limitation in existing IPA research. To address this, we have developed an automated pipeline that systematically collects real-world program changes from open-source Java projects. This pipeline integrates three key components: change mining, build automation, and change statistics. In what follows, we first describe each component of the general pipeline and then present the constructed benchmarks.

1) Change Mining: How to automatically obtain a large number of real-world program changes is a fundamental challenge in constructing benchmarks for IPA evaluation. To address this, we have designed a three-stage filtering process that automatically mines real-world projects to obtain a large number of program changes which represent real-world meaningful code modifications and are suitable for IPA evaluation.

The first stage is to select suitable open-source projects. Most real-world open-source projects use version control systems to manage their commit history, providing a rich and authentic data source for real-world program changes. As the largest open-source project hosting platform, GitHub hosts numerous projects that serve as an invaluable resource for our benchmark construction. We adopt open-source Java projects hosted on GitHub as the source of program changes. More specifically, we query projects using the GitHub API to obtain suitable Java projects. We can filter projects based on various criteria, such as the number of stars, forks, commits, and the primary programming language. This allows us to prioritize active and well-maintained repositories.

The second stage is to extract the suitable commit history from a selected project. We extract commit history using a single-chain approach (ensuring adjacent commits follow each other sequentially) to avoid artificial changes from cross-branch commits and preserve authentic, real-world development sequences. Notably, we implement additional filtering to retain only those commits compatible with the required Java version. This step is critical because open-source projects naturally evolve over their maintenance lifecycle, often upgrading to newer Java versions to leverage enhanced language features or performance improvements. Such upgrades, while beneficial for the project itself, can create incompatibilities with our target IPA tools, which are constrained to specific older versions. By explicitly extracting commits that align with the required Java version, we ensure that our benchmarks remain relevant and usable for evaluating these IPA tools.

The third stage is to filter meaningful program changes from an extracted commit history. We filter out trivial commits, such as whitespace and comment-only changes, as they are meaningless to IPA tools. We also filter out commits which do not change any Java project files (.java, pom.xml, *etc*.), as we focus on Java IPA techniques. We include changes to config files like pom.xml because they often involve dependency updates that can significantly impact the analysis results.

2) Build Automation: Some IPA works [15], [17] operate directly at the source code or AST level, allowing direct input of source code corresponding to each commit. In contrast, other works [4], [1] operate at the bytecode level or on custom intermediate representations (IRs), necessitating compilation of the source code corresponding to each commit. Build automation is therefore critical for large-scale evaluation.

As is well recognized in software engineering, build automation is a critical step in large-scale development practices. Yet it poses notable challenges in our setup, especially when handling projects of varying versions maintained by different organizations. Specifically, three key factors complicate build automation: the diversity of build tools, conflicting requirements for JDK versions, and the complexity of dependencies.

Since build tasks are not our focus (we only require the final compiled bytecode), we can filter projects based on a specific build tool (maven, gradle, *etc*.) and then use the

TABLE II: Characteristics of the benchmarks for real-world program changes. The File and LOC columns list the number of `.java` files and number of lines of java code in the latest commit for each project. The Commit and Commit^{F} columns report the numbers of total and filtered commits for each project. The last 8 columns present the statistics on changed files and LOC in the program changes composed of filtered commits, including the minimum, maximum, average, and median values.

Project	File	LOC	Commit	Commit^{F}	File$^{\Delta}$				LOC$^{\Delta}$			
					Min	Max	Avg	Median	Min	Max	Avg	Median
checkstyle	5,561	359,866	14,956	300	1	202	4	1	1	4,595	100	15
commons-bcel	646	45,884	2,807	501	1	474	7	1	1	6,020	110	11
commons-io	513	51,884	4,924	189	1	207	5	1	1	4,038	127	11
commons-lang	492	93,768	7,873	228	1	30	2	1	1	3,240	68	8
dependency-check	484	50,181	10,600	151	1	95	5	1	1	53,621	504	11
easyexcel	533	27,625	1,196	144	1	298	10	1	1	25,030	465	8
error-prone	1,994	281,110	6,794	344	1	295	7	2	1	23,200	258	61
fastjson	3,119	186,876	3,983	43	1	186	7	1	1	12,678	376	8
google-java-format	80	17,739	855	160	1	134	7	3	2	7,728	203	54
gson	259	36,073	2,108	299	1	282	8	2	1	47,143	959	34
javaparser	1,916	212,316	9,701	64	1	143	11	2	1	35,131	1,738	67
jedis	768	114,325	2,563	123	1	31	5	2	1	3,046	131	22
jsoup	174	34,305	2,251	215	1	34	3	2	1	2,364	87	19
jsqlparser	662	58,294	2,459	125	1	556	11	2	1	9,489	167	18
junit4	471	31,232	2,515	269	1	60	2	1	1	1,348	54	10
lettuce	1,254	104,247	2,546	279	1	227	8	3	1	5,448	207	31
maven	2,853	188,268	15,569	136	1	196	5	1	1	1,680	79	23
opennlp	1,195	84,978	2,295	257	1	492	6	2	1	9,776	189	22
pysonar2	112	11,365	552	159	1	183	6	1	1	12,016	275	15
redisson	1,932	232,371	10,419	98	1	51	4	2	1	4,922	150	49
Avg	1,251	111,135	5,348	204	1	209	6	2	1	13,626	312	25

corresponding build tool to automate and standardize the build process. This choice streamlines the build process by leveraging the standardized dependency management and compilation workflow of build tools. Furthermore, recognizing that analysis tools may support different Java versions, we enable filtering of commits based on the Java version specified in a build configuration file. This ensures that the build environment aligns with the requirements of the analysis tools.

Finally, recognizing that analysis tools require not only code from a repository but also code from third-party libraries and the JDK, we augment each commit with detailed metadata. Specifically, for every commit, we record the file paths of third-party JAR packages which are retrieved via the dependency resolution process of build tool. Additionally, we identify the relevant JDK version based on the build configuration and record the corresponding paths to JDK runtime JARs.

Even with a standardized build process in place, some commits may still fail to build, primarily due to issues such as unresolvable legacy dependencies or inherent compilation errors within a commit. To address this, our pipeline features an adaptive skipping mechanism: when a commit cannot be built successfully, it is automatically bypassed, and the pipeline proceeds to the next valid revision in the commit history. This valid revision is then treated as the new baseline version for subsequent IPA evaluation. This approach ensures build failures do not disrupt the overall evaluation process, enabling us to preserve both the integrity of our benchmarks and the authenticity of real-world program change sequences.

3) Change Statistics: To gain deeper insights into program changes, we conduct an automated change statistics within the pipeline, measuring them across multiple dimensions. At the file level, we record the number of files added, deleted, or modified in each commit, offering a high-level view of the change scope. At a finer granularity, we calculate lines of code (LOC) inserted and deleted using `git diff`. These metrics help quantify the impact of individual commits.

4) Constructed Benchmarks: Based on this automated benchmark construction pipeline, we have constructed the first large-scale benchmarks of real-world program changes. The project selection criteria for the constructed benchmarks are as follows: *I. Language Specification*: We restricted selections to open-source Java projects hosted on GitHub; *II. Activity and Maintenance Thresholds*: We imposed two quantitative thresholds (repositories must have at least 1,000 stars and 500 commits) to guarantee sufficient activity and maintenance quality; *III. Tool Compatibility and Organizational Focus*: Given that the evaluated IPA tools in Section IV have Java version constraints, we prioritized Java 7-compatible projects (necessitating a focus on historical ones) and thus targeted repositories from well-established organizations (Apache, Google, *etc.*), whose Java projects typically have long-term maintenance histories and sustained activity; *IV. Maven-Based Java Projects*: We restricted selections to Maven-based projects for two key reasons: first, the two IPA tools we evaluated require Java 7-compatible builds, most such projects use Ant or Maven, but newer Gradle-based ones typically need Java 8+; second, Maven ensures consistent dependency management and build options (facilitating automated building/evaluation), whereas Ant-based projects often have inconsistent configurations that hinder the automation of benchmark construction.

The constructed benchmarks cover 20 projects and 4,084 filtered commits, with details in Table II. Columns 2–4 present latest project statistics, confirming the projects are generally large-scale and well-maintained. Column 5 lists the number of filtered commits. Each pair of consecutive commits in a project forms a program change, and such a large number of changes is sufficient for adequate IPA evaluation. The last 8 columns present the change statistics of the filtered commits: most changes are relatively small, but the scale of real-world changes can vary significantly. This reflects the characteristics of real-world changes, in stark contrast to synthetic changes.

Finding 4: From Table II, based on the average and median values in the change statistics, we observe that most changes are relatively small in scale, which aligns with the motivation behind IPA, suggesting that IPA may provide better efficiency in real-world program changes. However, the minimum and maximum values reveal that the scale of real-world changes can vary significantly, and a large program change poses challenges to the effectiveness of IPA techniques on real-world software development.

B. Incremental Evaluation Framework

In this section, we present a unified incremental evaluation framework for the systematic evaluation of diverse IPA tools for Java. Specifically, we first examine the challenges associated with evaluating existing IPA tools, then provide a detailed description of the architectural design underlying the incremental evaluation framework.

Given the diversity of analysis tools, their input and output requirements vary significantly. As noted earlier, we focus on IPA tools for Java. In terms of input formats, some IPA tools [15], [26], [17] accept source code or AST directly, while others [4], [1], [33] require compiled bytecode; still others [38], [37] demand specialized IRs such as custom code graphs. Output formats are similarly diverse, with different IPA tools generating analysis results in varied types and formats. Furthermore, the runtime environments required by these tools differ substantially, even to the point of incompatibility. These incompatibilities complicate concurrent evaluation, as tools configured with specific runtime dependencies may interfere with one another during execution.

Lesson 3: Existing IPA tools exhibit substantial interface heterogeneity, posing significant challenges for unified evaluation to understand and compare the actual effectiveness of the IPA tools. An IPA tool should strive to provide an end-to-end infrastructure, which takes source code as input and generates analysis results in a standard format. Notably, the requirement for source code input does not mandate that the IPA tool analyzes it directly; the tool may convert source code into its required input format internally.

To address the challenges posed by varying input/output formats and runtime environments across IPA tools, we present a modular, containerized incremental evaluation framework designed to standardize and automate their evaluation. This evaluation framework provides an abstraction for establishing a fair and equitable environment to evaluate the performance of various IPA tools. For an IPA tool to be evaluated within this framework, it must be adapted to a containerized format that aligns with the specification of the framework.

The evaluation framework is organized into three main modules: runner, formatter, and result analyzer. First, the runner module manages the initiation and execution of a containerized IPA tool. For each containerized IPA tool, the runner module instantiates a container, mounts the benchmarks into the container, and executes the IPA tool in both exhaustive and incremental modes within the container. Specifically, we run exhaustive analysis on each commit and incremental analysis on each program change (*i.e.*, a pair of consecutive commits) of the benchmarks. In this work, we observed that many IPA tools lack a standalone exhaustive counterpart. To enable comparative analysis, we treated the initial execution of an IPA tool as the exhaustive analysis.

Following this, the formatter module plays a pivotal role in normalizing heterogeneous analysis outputs into a unified format, ensuring seamless integration with the downstream result analyzer module. We adopt JSON as the standardized result format, leveraging its language-agnostic nature and schema flexibility to accommodate diverse analysis outputs. This standardization eliminates parsing ambiguities and facilitates robust statistical analysis, enabling direct comparisons of evaluation metrics. Lastly, the result analyzer collects the results from all the IPA tools and provides meaningful metrics.

Importantly, the evaluation framework realizes a high degree of automation by enabling the systematic integration of existing IPA tools, thereby ensuring reproducibility. Its core objective is to provide a standardized and replicable evaluation framework for multi-metric evaluation (focusing on efficiency, memory overhead, and correctness in this work). Specifically, the evaluation framework has successfully integrated two IPA tools as detailed in Section IV, and it can be readily extended to incorporate new IPA tools in the future.

IV. Evaluation

This section presents a systematic evaluation of existing IPA tools based on the large-scale benchmarks we constructed. First, we introduce the experimental subjects used in the evaluation, then present the research questions we aim to answer, followed by the results and findings, and finally discuss the lessons learned and threats to validity. The evaluation was conducted on an Ubuntu 20.04.6 LTS with an Intel Xeon Gold 6240 CPU (2.6GHz, 75 threads) and 256GB RAM. We repeated the evaluation 3 times to report the average data.

A. Experimental Subjects

We start the introduction with the selection of experimental subjects from existing IPA works. We have discussed the evaluation methodologies for existing IPA work in Section II. Here, we delve deeper into their implementation and availability details. The selection adheres to the following criteria:

1) *Focus on IPA work for Java*;
2) *Availability of source code or reusable artifacts*;
3) *Support for real-world program changes*.

The 25 works discussed in Section II satisfy the first criterion. The second criterion emphasizes the availability of tools. Clearly, the evaluation cannot proceed without available tools, as reproducing tools from papers is not the goal of our work. By prioritizing works with available tools, 15 works remain after excluding works that do not meet the criterion. The last criterion is crucial for evaluating IPA work on real-world program changes; as discussed in Section II, most work fails to meet it. Finally, only two works (Reviser and JavaDL) satisfy all the criteria.

Reviser [4] incrementalizes the analyses implemented in the IFDS framework [39]. JavaDL [26] is a Datalog-based declarative specification language for bug pattern detection in Java code. JavaDL seamlessly supports both exhaustive and incremental evaluation from the same detector specification. In this work, we manually containerized these two IPA tools with non-trivial effort to integrate them into our incremental evaluation framework, enabling systematic multi-metric evaluation on real-world program changes.

Finding 5: Only two existing IPA works for Java provide tools that can be evaluated on real-world projects with relatively low effort. Other works either do not offer available IPA tools or require substantial cost for implementations to make them deployable in real-world projects.

B. Research Questions

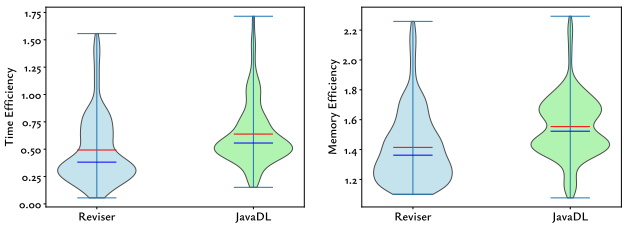
This work aims to systematically evaluate the performance of IPA tools in multiple metrics: efficiency, memory overhead, and correctness. The selection of these metrics is based on our analysis of existing IPA work in Section II, which reveals that these metrics are the most commonly used for evaluating IPA techniques. Furthermore, these metrics are critical for assessing the practical usability of IPA tools in real-world software development scenarios. In what follows, we outline the research questions around these metrics.

- **RQ2.1 Efficiency:** How does the speedup of an IPA compare to its exhaustive one?
- **RQ2.2 Memory Overhead:** How does the memory overhead of an IPA compare to that of its exhaustive one?
- **RQ2.3 Correctness:** How consistent are the results of an IPA compared to those of its exhaustive one?

To answer the research questions, we collected data during both the benchmark construction and incremental evaluation phases. Specifically, we collected benchmark information during benchmark construction as shown in Table II and data for both incremental and exhaustive analysis, including analysis time, memory usage, and analysis results.

C. Results

In this section, we present the results of the systematic evaluation. Due to the massive scale of the raw data and space restrictions, we primarily present the distributions of relative performance results between incremental and exhaustive analysis, then conduct a high-level analysis of the results.



(a) Time Efficiency (b) Memory Efficiency

Fig. 3: Distributions of time efficiency and memory efficiency of incremental analysis by comparing its analysis time and memory usage with those of exhaustive analysis.

1) RQ2.1 Efficiency: To answer this research question, we compare the performance of both analyses when processing the same input. Specifically, we collect the analysis time (T_i) of the incremental analysis on a commit and the analysis time (T_f) of the exhaustive analysis on the same commit. We then calculate the efficiency improvement as $TE = T_i/T_f$. A smaller *TE* indicates a greater efficiency improvement. *TE* less than 1 indicates that incremental analysis is faster, while greater than 1 indicates that exhaustive analysis is faster.

Figure 3a visualizes the time efficiency of each IPA tool. It illustrates the distribution of *TE* over all program changes, including the minimum, maximum, mean, and median values of *TE*. It can be observed that the *TE* values of both IPA tools are less than 1 for the majority (over 95%) of program changes in the benchmarks. This indicates that incremental analysis achieves speedup over exhaustive analysis for these program changes, which means that these tools have kept their primary promise as IPA tools in most real-world scenarios.

However, we also observed that both IPA tools exhibited slower performance than exhaustive analysis in a small number of program changes. We sampled the program changes that led to this slowdown and found that they mainly fell into two categories: one involved large-scale code changes (as shown in Finding 4), such as refactoring or extensive code rewrites, which could cause the IPA tool to reanalyze a significant amount of code; the other involved complex dependency changes, such as cross-module dependency changes or even external dependency changes, which could lead to the IPA tool requiring more time to resolve dependencies and analyze the code. This indicates that while incremental analysis can provide efficiency improvements in most cases, it may still be slower than exhaustive analysis in certain specific scenarios.

Finding 6: Most program changes in the benchmarks indicate that the IPA tools achieved acceleration compared to exhaustive analysis. However, in a small subset of changes, incremental analysis exhibited performance degradation, either due to extensive reanalysis or complex dependency resolution required by the tools.

2) RQ2.2 Memory Overhead: The second research question focuses on IPA tools' memory overhead. To evaluate this, we compared incremental analysis and exhaustive analysis memory usage on identical inputs: collecting incremental (M_i) and exhaustive (M_f) memory usage for the same commit. We define memory efficiency *ME* as M_i/M_f: smaller *ME* indicates lower overhead, while $ME > 1$ indicates incremental analysis has greater overhead than exhaustive analysis.

Figure 3b shows the memory efficiency distribution of the two IPA tools across all program changes in the benchmarks. Both tools exhibit higher incremental analysis memory overhead than exhaustive analysis across all changes: Reviser averages 1.41, JavaDL 1.55. This confirms a IPA tool typically incur higher memory overhead than exhaustive analysis.

There are two key factors contributing to this increased memory overhead. *I. Persistence of intermediate results*: IPA relies on reusing existing results to avoid redundant computations, necessitating the storage of extensive intermediate results, which are not required in exhaustive analysis. *II. Change tracking and propagation*: IPA tools need to maintain additional data structures to track and propagate changes, which also increases memory usage.

However, excessive memory overhead impairs the usability of IPA tools, especially in resource-limited environments. Figure 3b shows $ME > 2.0$ in a small subset of program changes, potentially making the tools impractical or inefficient in real-world use. Our sampling study reveals these high-memory-overhead cases strongly overlap with IPA performance degradation, primarily occurring in scenarios with extensive code changes or complex dependency changes.

Finding 7: The IPA tools exhibit consistent higher memory overhead compared to their exhaustive analysis counterparts, Reviser and JavaDL showing average *ME* of 1.41 and 1.55, respectively. In specific scenarios involving extensive code changes or complex dependency changes, *ME* can exceed 2.0, doubling memory overhead and potentially limiting tool usability in resource-limited environments.

3) RQ2.3 Correctness: This research question examines correctness (consistency between incremental and exhaustive analysis results) by comparing their outputs on identical inputs, where incremental results are obtained by updating exhaustive results from the previous commit. We focus on two sub-metrics: the proportion of inconsistent results and the maximum error rate of such inconsistencies. For each commit, we collect incremental (R_i) and exhaustive (R_f) results, then calculate the error rate as $ER = |R_i \oplus R_f|/|R_f|$. Smaller *ER* indicates greater consistency; $ER > 0$ signifies inconsistency. Finally, we compute the proportion of changes with $ER > 0$.

Table III shows these sub-metrics for both IPA tools across all benchmark changes: Reviser has a 2.76% inconsistency rate and 3.27% maximum error rate, while JavaDL has 1.35% and 1.62% respectively. This indicates IPA tools mostly consistent with exhaustive analysis but have rare inconsistencies. We found that these inconsistencies occur mainly in scenarios that involve complex dependency changes. Complex dependency changes may lead to errors in an IPA tool during processing, affecting the consistency of the results.

TABLE III: IPA correctness in terms of two sub-metrics. The Inconsistency column represents the proportions of inconsistent results between incremental analysis and exhaustive analysis. The Max ER column indicates the maximum of error rate in inconsistent analysis results over all program changes.

IPA Tool	Inconsistency	Max ER
Reviser	2.76%	3.27%
JavaDL	1.35%	1.62%

To illustrate this inconsistency, consider the example code: `A a = AFactory.newA()`, where `AFactory` and `A` belong to third-party library `libA`. In the change, the example code is unchanged, but `libA` updates: `newA()`'s return type shifts from `A1` to `A2` (both `A` subclasses). Before the change, the analysis result `pts(a)` is {`new A1`}; post-change, the expected result is {`new A2`}. If the IPA tool mishandles library changes, it may treat the example code as unaffected, leaving the result as {`new A1`} and thus causing inconsistency.

Inconsistent results can mislead developers, potentially causing them to overlook real issues or chase false positives. Therefore, ensuring consistency between incremental and exhaustive analysis results is crucial for the reliability of IPA tools in real-world applications.

Finding 8: The results show that the two IPA tools maintain consistency for the vast majority of program changes in the benchmarks. However, partial inconsistencies occur in a minority of program changes, particularly those involving complex dependency changes.

D. Discussion

This section discusses the lessons learned from the evaluation, aiming to guide future IPA research and development.

First, from the literature analysis and all the evaluation results, we can see that change handling is at the core of IPA. IPA tools must be able to handle input program changes quickly, accurately, and locate and propagate changes efficiently during the analysis process. However, the complexity and diversity of change handling pose challenges for IPA tools when dealing with different types of program changes.

Existing IPA works have made some progress in change handling, but a unified change definition and handling framework has not yet been established. This leads to different IPA tools potentially adopting different strategies and methods to handle changes, resulting in inconsistencies in results and performance differences. At a high level, changes can be roughly divided into three levels: text changes, syntactic changes, and semantic changes. Different types of changes at different levels may require different handling strategies and methods. Therefore, future research can further explore how to standardize change definitions and improve the change

handling capabilities of incremental analysis tools to improve their performance and correctness in various scenarios.

Lesson 4: One of the cores of an IPA tool lies in change handling, which requires rapid and accurate processing of input program changes alongside efficient change propagation during analysis. Existing IPA works lack a unified change definition and handling framework. Standardizing change definitions and improving change handling frameworks for IPA tools will be critical to improving their performance and correctness in diverse real-world scenarios.

Another research direction closely related to change handling is the restart mechanism of IPA. As mentioned earlier, IPA tools need to quickly locate and propagate changes when processing input changes. However, in some cases, the IPA tools may encounter complex changes that cannot be efficiently handled, necessitating a restart mechanism to perform an exhaustive analysis again. In such cases, the IPA tools need to quickly determine whether a restart is necessary.

Some IPA works have explored restart mechanisms. JavaDL [26] proposes restarting for merge commits and major move/rename commits. Zhao *et al*. [40] uses an analysis time threshold to trigger automatic switching to exhaustive analysis when exceeded. While these efforts have advanced IPA restart mechanisms and improved tool usability, there are still challenges and room for improvement. Future research should focus on optimizing restart mechanisms by dynamically detecting unmanageable changes, integrating context-aware thresholds or machine learning models to minimize unnecessary exhaustive analyses while ensuring correctness.

Lesson 5: The restart mechanism in IPA tools, a critical complement to change handling, addresses scenarios where complex program changes outpace the tool's incremental processing capabilities, necessitating a fallback to exhaustive analysis. Future research can further explore how to optimize the restart mechanism of IPA to enhance its efficiency and correctness in handling complex program changes.

Finally, we discuss the issue of memory overhead. As shown in the previous section, incremental analysis generally incurs higher memory overhead than exhaustive analysis. The core reason for this issue lies in the need for incremental analysis to maintain extensive intermediate results and additional state information to track and propagate program changes. Memory overhead is a critical consideration for the practical application of IPA tools, especially in resource-constrained environments. Additionally, high memory overhead can trigger frequent garbage collection, leading to stuttering or performance jitter during analysis. In particular, existing IPA tools can consume more than twice the memory of exhaustive analysis when dealing with large-scale program changes, which further highlights the urgency of memory optimization.

Future research can approach this issue from multiple aspects: first, designing lightweight state representation methods, such as hash-based change fingerprint compression techniques; second, constructing adaptive memory management strategies to dynamically adjust cache granularity based on available resources; third, exploring collaborative mechanisms between IPA and garbage collection. These optimizations will not only enhance tool applicability in resource-constrained scenarios but will also provide more reliable analysis support for practical industrial-scale software development.

Lesson 6: IPA tools inherently incur higher memory overhead than exhaustive analysis. This poses critical challenges in resource-constrained environments. To address this, future research can address this issue through lightweight state representations, adaptive memory management, and integrated state pruning.

E. Implications

The implications of this work cover multiple parties involved with IPA. For *IPA researchers*, more efforts are needed to address the issues of memory overhead and correctness, and we discussed the possible future research directions in Section IV-D. For *IPA developers*, it would be better to provide an infrastructure capable of enabling end-to-end IPA on real-world program changes; otherwise, it is challenging to genuinely evaluate the effectiveness of their implementations. For *IPA users*, our work enables them to better understand the real-world performance of existing IPA tools, facilitating informed tool selection tailored to their needs.

F. Threats to Validity

External Threats. The primary external threat involves bias in the collection of real-world program changes. To mitigate this threat, the benchmark construction pipeline we designed maximizes automation to reduce such bias. Limiting selection to Maven-based projects may also introduce bias, but it does not hinder the generalizability of our findings, as the build tool used does not affect the program changes themselves. We mitigate this threat by including a wide range of program changes from various projects, ensuring diversity in the types of changes. Another external threat is that we only evaluate two IPA tools. However, substantial engineering efforts are required to adapt most existing IPA tools for benchmarking against real-world program changes, which is beyond the scope of our work. To address this threat, we provide a unified evaluation framework for various IPA tools.

Internal Threats. Our internal threat is that the sampling analysis of the results of the large-scale evaluation might not fully reflect the complete distribution. We address this by uniformly sampling the results, and each result sampled was independently inspected by two authors, and any discrepancy was discussed until a consensus was reached.

G. Limitations and Future Work

In this work, we only evaluated specific IPA tools for Java, which may limit the generalizability of our findings. First, we only evaluate IPA tools for Java. This is because Java has a rich ecosystem of program analysis research and tools. While IPA

works also exist for other languages (*e.g.*, C/C++), we plan to extend our evaluation pipeline to support IPA tools for other languages in the future. Second, we restricted the evaluation to two IPA tools, due to the limited availability of IPA tools that can be directly applied to real-world program changes. Notably, our evaluation framework can be extended to support more IPA tools in the future. Third, we only consider three metrics in this work. However, there are other metrics that can be used to evaluate IPA tools. We plan to extend the evaluation framework to support more metrics in the future.

Additionally, the constructed benchmarks are tool-agnostic, so their metadata only contains tool-independent information. However, users may be interested in how well these benchmarks cover specific tasks. Providing task-specific metadata for every task is not only challenging but also contradicts the tool-agnostic design. In the future, we plan to introduce a metadata analysis infrastructure, enabling users to generate task-specific metadata tailored to their needs.

Finally, the relationships between program changes and IPA evaluation metrics also merit in-depth exploration in future research. Clarifying the relationships can not only provide a clear direction for the performance optimization of IPA tools, but also help developers select IPA solutions suitable for specific change scenarios, further improving analysis efficiency and resource utilization in practical development processes, and filling the gap in current research regarding the quantitative analysis of metric-influencing factors.

V. Related Work

A. Incremental Program Analysis

Incremental program analysis has emerged as a promising technique in static program analysis, driven by its ability to timely update code insights in response to program changes. Researchers have developed various incrementalization approaches that can be broadly categorized into incrementalizing individual analyses and incremental analysis frameworks.

Incrementalizing individual analyses. The former focuses on adapting a specific program analysis (such as data flow analysis) for incremental updates. The works of Pacak *et al*. [16] and Zwaan *et al*. [17] focus on automated incremental type checkers for real-time IDE feedback, while Lu *et al*. [23] and Liu *et al*. [3], [24] have proposed incremental pointer analysis algorithms based on CFL reachability and context sensitivity. Zhao *et al*. [1] construct call graphs incrementally based on the CHA-style algorithm. All of these works incrementalize a specific static analysis algorithm, which typically requires significant modifications to the original analysis implementation using algorithms such as DRed [41] to precisely reset invalid results and propagate changes. Generally, this could achieve high-performance incrementalization, but it requires more work to ensure correctness and consistency.

Incremental analysis frameworks. In contrast, there are incremental analysis frameworks that provide a generic mechanism for automatic incrementalization. Reviser [4] incrementalizes data flow analyses within IDE/IFDS framework. More recently, a paradigm shift has emerged with the advent of incremental analysis engines, which enables the incrementalization without modifying existing analysis implementations. Notable frameworks include IncA [2], iQL [42], and more [26], [40]. The core of these incremental analysis frameworks is to encapsulate the complexity of change management and dependency tracking, allowing developers to implement analyses using familiar paradigms without grappling with incrementalization details.

Beyond program analysis, incremental computation has been applied in diverse domains, including databases [43], logic programming [44], [45], compilers [46], build systems [47], model checking [48], SAT solving [49], to name a few. Liu [50] gave a high-level overview of the work on incremental computation and highlighted incrementalization as the essence underlying all of them.

B. Empirical Study on Static Analysis

Static program analysis has a long history of theoretical and practical advances, with empirical studies focusing primarily on evaluating bug detection capabilities and developer adoption. Lipp *et al*. [51] highlighted limitations in C code analyzers, while Tomassi *et al*. [52] categorized the strengths and weaknesses of Java null pointer exception detectors. Habib *et al*. [53] and Qiu *et al*. [54] explored bug detection on Defects4J and Android apps respectively, identifying roots of inaccuracy. All these works examined only traditional exhaustive analysis tools, whereas our work is the first empirical study of incremental analysis on real-world program changes.

Parallel research has investigated the interactions between developers and analysis tools. Johnson *et al*. [55] cited false positives and warning presentation as adoption barriers, while Christakis *et al*. [56] and Do *et al*. [57] uncovered practitioner preferences and decision-making factors of analysis tools. Our work helps developers understand the real-world performance of existing IPA techniques in practical software development and provides guidance to them in selecting IPA tools.

VI. Conclusion

This paper investigates the empirical performance of IPA tools in real-world scenarios. We conducted a comprehensive literature analysis in the field of IPA for Java, and identified two key limitations in existing IPA evaluation. Motivated by the literature analysis, we designed an automated benchmark construction pipeline and an incremental evaluation framework, constructed large-scale benchmarks of real-world program changes, and then conducted systematic evaluation using the benchmarks. In general, we present eight key findings and six lessons from the literature analysis and the empirical evaluation, which can help researchers and developers better understand the current state-of-the-art of IPA tools and guide future IPA research and development.

Acknowledgment

This work is supported in part by the National Key Research and Development Program of China (2024YFB2505604) and the National Natural Science Foundation of China (No.62232008, 62172200).

References

[1] Z. Zhao, X. Wang, Z. Xu, Z. Tang, Y. Li, and P. Di, "Incremental Call Graph Construction in Industrial Practice," in *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2023, pp. 471–482. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP58684.2023.00048

[2] T. Szabó, S. Erdweg, and G. Bergmann, "Incremental Whole-Program Analysis in Datalog with Lattices," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3453483.3454026

[3] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking Incremental and Parallel Pointer Analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, pp. 6:1–6:31, 2019. [Online]. Available: http://doi.org/10.1145/3293606

[4] S. Arzt and E. Bodden, "Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 288–298. [Online]. Available: https://doi.org/10.1145/2568225.2568243

[5] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective Context-sensitivity Guided by Impact Pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 475–484. [Online]. Available: https://doi.org/10.1145/2594291.2594318

[6] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A Principled Approach to Selective Context Sensitivity for Pointer Analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 2, pp. 10:1–10:40, 2020. [Online]. Available: https://doi.org/10.1145/3381915

[7] D. He, J. Lu, Y. Gao, and J. Xue, "Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 2, pp. 719–742, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3162236

[8] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 389–404. [Online]. Available: https://doi.org/10.1145/3037697.3037744

[9] M. Méndez-Lojo, A. Mathew, and K. Pingali, "Parallel Inclusion-based Points-to Analysis," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2010, pp. 428–443. [Online]. Available: https://doi.org/10.1145/1869459.1869495

[10] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[11] B. Stein, B.-Y. E. Chang, and M. Sridharan, "Demanded Abstract Interpretation," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021, pp. 282–295. [Online]. Available: https://doi.org/10.1145/3453483.3454044

[12] P. W. O'Hearn, "Continuous Reasoning: Scaling the Impact of Formal Methods," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 2018, pp. 13–25. [Online]. Available: https://doi.org/10.1145/3209108.3209109

[13] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from Building Static Analysis Tools at Google," *Communications of the ACM (CACM)*, vol. 61, no. 4, pp. 58–66, 2018. [Online]. Available: https://doi.org/10.1145/3188720

[14] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling Static Analyses at Facebook," *Communications of the ACM (CACM)*, vol. 62, no. 8, pp. 62–70, 2019. [Online]. Available: https://doi.org/10.1145/3338112

[15] E. Kuci, S. Erdweg, O. Bracevac, A. Bejleri, and M. Mezini, "A Co-contextual Type Checker for Featherweight Java," in *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP)*, vol. 74. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017, pp. 18:1–18:26. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

[16] A. Pacak, S. Erdweg, and T. Szabó, "A Systematic Approach to Deriving Incremental Type Checkers," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 127:1–127:28, 2020. [Online]. Available: https://doi.org/10.1145/3428195

[17] A. Zwaan, H. van Antwerpen, and E. Visser, "Incremental Type-Checking for Free: Using Scope Graphs to Derive Incremental Type-Checkers," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 140:424–140:448, 2022. [Online]. Available: https://doi.org/10.1145/3563303

[18] A. Nougrahiya and V. K. Nandivada, "IncIDFA: An Efficient and Generic Algorithm for Incremental Iterative Dataflow Analysis," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 617–648, 2025. [Online]. Available: https://doi.org/10.1145/3720436

[19] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 504–515. [Online]. Available: https://doi.org/10.1145/1993498.1993558

[20] S. Makhdoom, M. A. Khan, and J. H. Siddiqui, "Incremental Symbolic Execution for Automated Test Suite Maintenance," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 271–276. [Online]. Available: https://doi.org/10.1145/2642937.2642961

[21] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental Symbolic Execution of Concurrent Software," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 531–542. [Online]. Available: https://doi.org/10.1145/2970276.2970332

[22] Y. Lin, S. Zhang, and J. Zhao, "Incremental Call Graph Reanalysis for AspectJ Software," in *Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2009, pp. 306–315. [Online]. Available: https://doi.org/10.1109/ICSM.2009.5306311

[23] Y. Lu, L. Shang, X. Xie, and J. Xue, "An Incremental Points-to Analysis with CFL-Reachability," in *Proceedings of the 22nd International Conference on Compiler Construction (CC)*. Springer, 2013, pp. 61–81. [Online]. Available: https://doi.org/10.1007/978-3-642-37051-9_4

[24] B. Liu and J. Huang, "SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 88:1–88:28, 2022. [Online]. Available: https://doi.org/10.1145/3527332

[25] T. Szabó, S. Erdweg, and M. Voelter, "IncA: A DSL for the Definition of Incremental Program Analyses," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 320–331. [Online]. Available: https://doi.org/10.1145/2970276.2970298

[26] A. Dura, C. Reichenbach, and E. Söderberg, "JavaDL: Automatically Incrementalizing Java Bug Pattern Detection," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 165:1–165:31, 2021. [Online]. Available: https://doi.org/10.1145/3485542

[27] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006, pp. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[28] J. Krainz and M. Philippsen, "Diff Graphs for a Fast Incremental Pointer Analysis," in *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ACM, 2017, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3098572.3098578

[29] A. L. Souter and L. L. Pollock, "Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2001, p. 682. [Online]. Available: https://doi.org/10.1109/ICSM.2001.972787

[30] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proceedings of the 30th International Conference*

on Software Engineering (ICSE). ACM, 2008, pp. 291–300. [Online]. Available: https://doi.org/10.1145/1368088.1368128

[31] T. Szabó, E. Kuci, M. Bijman, M. Mezini, and S. Erdweg, "Incremental Overload Resolution in Object-Oriented Programming Languages," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 2018, pp. 27–33. [Online]. Available: https://doi.org/10.1145/3236454.3236485

[32] S. Erdweg, T. Szabó, and A. Pacak, "Concise, Type-Safe, and Efficient Structural Diffing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021, pp. 406–419. [Online]. Available: https://doi.org/10.1145/3453483.3454052

[33] C. Zhou, Y. Fang, J. Wang, and C. Wang, "An Incremental Algorithm for Algebraic Program Analysis," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 65:1934–65:1961, 2025. [Online]. Available: https://doi.org/10.1145/3704901

[34] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing Lattice-based Program Analyses in Datalog," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 139:1–139:29, 2018. [Online]. Available: https://doi.org/10.1145/3276509

[35] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and Scalable Security Analysis of Web Applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2013, pp. 210–225. [Online]. Available: https://doi.org/10.1007/978-3-642-37057-1_15

[36] B. Liu and J. Huang, "D4: Fast Concurrency Debugging with Parallel Differential Analysis," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 359–373. [Online]. Available: https://doi.org/10.1145/3192366.3192390

[37] S. Krishna, A. Lal, A. Pavlogiannis, and O. Tuppe, "On-the-Fly Static Analysis via Dynamic Bidirected Dyck Reachability," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 1239–1268, 2024. [Online]. Available: https://doi.org/10.1145/3632884

[38] Y. Li, K. Satya, and Q. Zhang, "Efficient Algorithms for Dynamic Bidirected Dyck-Reachability," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 62:1–62:29, 2022. [Online]. Available: https://doi.org/10.1145/3498724

[39] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1995, pp. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[40] D. Zhao, P. Subotic, M. Raghothaman, and B. Scholz, "Towards Elastic Incrementalization for Datalog," in *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2021, pp. 1–16. [Online]. Available: https://doi.org/10.1145/3479394.3479415

[41] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining Views Incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 1993, pp. 157–166. [Online]. Available: https://doi.org/10.1145/170035.170066

[42] T. Szabó, "Incrementalizing Production CodeQL Analyses," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2023, pp. 1716–1726. [Online]. Available: https://doi.org/10.1145/3611643.3613860

[43] A. Balmin, Y. Papakonstantinou, and V. Vianu, "Incremental Validation of XML Documents," *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 4, pp. 710–751, 2004. [Online]. Available: https://doi.org/10.1145/1042046.1042050

[44] D. Saha and C. R. Ramakrishnan, "A Local Algorithm for Incremental Evaluation of Tabled Logic Programs," in *Proceedings of the 22nd International Conference on Logic Programming (ICLP)*. Springer, 2006, pp. 56–71. [Online]. Available: https://doi.org/10.1007/11799573_7

[45] L. Ryzhyk and M. Budiu, "Differential Datalog," in *Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0)*. CEUR-WS.org, 2019, pp. 56–67. [Online]. Available: http://ceur-ws.org/Vol-2368/paper6.pdf

[46] T. Reps, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1982, pp. 169–176. [Online]. Available: https://doi.org/10.1145/582153.582172

[47] G. Konat, S. Erdweg, and E. Visser, "Scalable Incremental Building with Dynamic Task Dependencies," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 76–86. [Online]. Available: https://doi.org/10.1145/3238147.3238196

[48] O. Sokolsky and S. A. Smolka, "Incremental Model Checking in the Modal Mu-Calculus," in *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*. Springer, 1994, pp. 351–363. [Online]. Available: https://doi.org/10.1007/3-540-58179-0_67

[49] Y. Kilani, M. Bsoul, A. Alsarhan, and A. Al-Khasawneh, "A Survey of the Satisfiability-Problems Solving Algorithms," *International Journal of Advanced Intelligence Paradigms (IJAIP)*, vol. 5, no. 3, pp. 233–256, 2013. [Online]. Available: https://doi.org/10.1504/IJAIP.2013.056447

[50] Y. A. Liu, "Incremental Computation: What Is the Essence?" in *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM, 2024, pp. 39–52. [Online]. Available: https://doi.org/10.1145/3635800.3637447

[51] S. Lipp, S. Banescu, and A. Pretschner, "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2022, pp. 544–555. [Online]. Available: https://doi.org/10.1145/3533767.3534380

[52] D. A. Tomassi and C. Rubio-González, "On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 292–303. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678535

[53] A. Habib and M. Pradel, "How Many of All Bugs Do We Find? A Study of Static Bug Detectors," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 317–328. [Online]. Available: https://doi.org/10.1145/3238147.3238213

[54] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 176–186. [Online]. Available: https://doi.org/10.1145/3213846.3213873

[55] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606613

[56] M. Christakis and C. Bird, "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)*. ACM, 2016, pp. 332–343. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[57] L. N. Q. Do, J. R. Wright, and K. Ali, "Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 3, pp. 835–847, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3004525