# Understanding Industry Perspectives of Static Application Security Testing (SAST) Evaluation

YUAN LI, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
PEISEN YAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
KAN YU, Ant Group, China
CHENGPENG WANG, Hong Kong University of Science and Technology, China
YAOYANG YE, Zhejiang University, China
SONG LI, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
MENG LUO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
YEPANG LIU, Southern University of Science and Technology, China
KUI REN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China

The demand for automated security analysis techniques, specifically static application security testing (SAST), is steadily rising. Assessing the effectiveness of SAST tools is crucial for evaluating current techniques and inspiring future technical advancements. Regrettably, recent research suggests that existing benchmarks used for evaluation often fail to meet the industry's needs, significantly impeding the adoption of SASTs in real-world scenarios. This paper presents a qualitative study to bridge this gap. We investigate why industrial professionals utilize SAST benchmarks, identify barriers to their usage, and explore potential improvements for existing benchmarks. Specifically, we conducted in-depth, semi-structured interviews with twenty industrial professionals possessing diverse field experience and backgrounds in security and product development. As the first comprehensive investigation of SAST evaluation from an industrial perspective, our findings would break down the barriers between academia and industry, providing valuable inspiration for designing better benchmarks and promoting new advances in SAST evaluation.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Static Application Security Testing, Qualitative Study

**ACM Reference Format:**
Yuan Li, Peisen Yao, Kan Yu, Chengpeng Wang, Yaoyang Ye, Song Li, Meng Luo, Yepang Liu, and Kui Ren. 2025. Understanding Industry Perspectives of Static Application Security Testing (SAST) Evaluation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE134 (July 2025), 24 pages. https://doi.org/10.1145/3729404

Authors' Contact Information: Yuan Li, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, ly.liyuan@zju.edu.cn; Peisen Yao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China, pyaoaa@zju.edu.cn; Kan Yu, Ant Group, China, kan.yk@antgroup.com; Chengpeng Wang, Hong Kong University of Science and Technology, China, cwangch@cse.ust.hk; Yaoyang Ye, Zhejiang University, China, makise@zju.edu.cn; Song Li, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, songl@zju.edu.cn; Meng Luo, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, meng.luo@zju.edu.cn; Yepang Liu, Southern University of Science and Technology, China, liuyp1@sustech.edu.cn; Kui Ren, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China, kuiren@zju.edu.cn.


ACM 2994-970X/2025/7-ARTFSE134
https://doi.org/10.1145/3729404

## 1 Introduction

Static Application Security Testing (SAST) tools have been increasingly adopted by organizations to enhance the security of their software applications [26, 27, 49, 72]. These tools analyze the source code without executing it, allowing for the identification of potential vulnerabilities. They can be seamlessly integrated into various stages of the life cycle of software development, ranging from requirements engineering to fault identification, debugging, and fixing.

Comparative studies are crucial to ensuring progress, monitoring scientific advancements in this field, and developing effective tools. Benchmarking SASTs provides practitioners with the necessary information to make informed decisions when developing and deploying these tools. While there is a vast body of literature on evaluating SASTs [9, 37, 39, 43, 57–60], neither the industry nor the academic community has set evaluation standards for SASTs' technical capabilities. On the one hand, micro-benchmarks help assess the specific capabilities of SASTs but may not provide a holistic representation of the complexities and nuances found in realistic software. On the other hand, benchmarks based on real-world programs capture the complexities of actual applications. Still, they may lack the granularity to provide in-depth insights into tool performance or the quantifiability to represent the diversity of real-world applications.

Unfortunately, existing benchmarks for evaluating SASTs often fall short of meeting practitioners' needs [11]. However, significant discrepancies exist across many regularly used benchmarks, ranging from tens to tens of thousands of samples. Most existing benchmarks are merely stacked with vulnerability samples; the implementations may be inadequate, and the functional points assessed are unevenly distributed. As a result, the test results offered to practitioners are neither assured nor sensible [59]. Worse, while practitioners seek to identify the technical strengths and weaknesses of SASTs finely, the evaluation results of many existing benchmarks look pretty much like a "black box", providing only overall recall rate and false positive rate data.

Hence, there is an urgent need to bridge the gap between practitioners' expectations and existing benchmarks. The first step to better assist practitioners in evaluating SASTs is to gain a good knowledge of current practices. However, no systematic study has ever been conducted, and little empirical knowledge has ever been provided. Without such expertise, understanding and improving current SAST evaluation practices remain difficult and untargeted. Moreover, this inadequacy might hinder any attempt to improve the support of corresponding tools.

To tackle these issues from an industrial perspective, we present a qualitative study in collaboration with Ant Group, an international software vendor providing services for over 1 billion global users. It is active in database management systems, business analytics, cloud services, data transfer, and security solutions. As a large software vendor, Ant Group is strongly interested in the security of its software products and uses SASTs extensively. Through in-depth, semi-structured interviews with practitioners from Ant Group with diverse expertise in software development, security, and product development, our study seeks to understand practitioners' use of SAST benchmarks, identify associated barriers, and explore avenues for potential improvements. In our study, we address the following research questions:

- **RQ1**: *What are the reasons practitioners use SAST benchmarks and their concerns about SAST evaluation goals? Specifically, we intend to investigate under which evaluation scenario(s) benchmarks are required and practitioners' expectations of them.*
- **RQ2**: *What barriers hinder the adoption of existing SAST benchmarks? In particular, we investigate what types of information are difficult to acquire from SAST evaluation results, revealing the limitations of current benchmarks.*
- **RQ3**: *How can the effectiveness of SAST evaluation be enhanced? We strive to explore effective approaches for constructing benchmarks and help evaluate SASTs more comprehensively.*

To answer these questions, we identified 12 key findings about practitioners' expectations of SAST evaluation (§ 4.1), the challenges of utilizing SAST benchmarks (§ 4.2), and insights into future initiatives to improve SAST evaluation practices (§ 4.3). We also summarize our findings (§ 5.1) and draw several actionable implications of our study grouped according to practitioners of different roles (§ 5.2). In summary, this paper makes the following contributions:

- We present the first qualitative study of industry perspectives in understanding practitioners' motivations, challenges, and expectations for SAST evaluation.
- We comprehensively explore practitioners' information needs for evaluating SASTs and identify gaps in existing benchmarks.
- We provide valuable guidance for future research and benchmark construction regarding evaluating SASTs.

## 2 Background and Motivation

This section introduces the concept of Static Application Security Testing (SAST), highlights the challenges of evaluating SASTs, and motivates our work.

### 2.1 Evaluating SAST Tools

Static Application Security Testing (SAST) tools (e.g., Soot [5], FlowDroid [3], SVF [6], Clang Static Analyzer (CSA) [2], and CodeQL [75]) analyze the application's source code without executing it, enabling the early detection of potential security flaws, such as input validation issues and insecure cryptographic algorithms. SASTs identify these weaknesses during the development process and assist developers in addressing them before deploying the software in production. Besides, SASTs also prove valuable in debugging and fixing processes that follow the discovery of security breaches. As such, SASTs have been widely adopted across industries for critical tasks, such as securing proprietary business code, assessing open-source software supply chains, ensuring compliance, and obtaining essential industry certifications.

Given SASTs' crucial role, evaluating their effectiveness and capabilities is essential for developers and users to make informed decisions regarding tool development and adoption. Hence, a benchmarking system for creating and executing targeted test cases is vital for such evaluations. Currently, two common benchmark types are used, as shown in Table 1.

**Real-world benchmarks**, such as Defects4J [33], aim to capture the complexities and challenges in actual applications. These benchmarks utilize real-world software projects as their foundation, incorporating various syntactic features and vulnerability models. They provide diverse test cases and scenarios to assess the effectiveness and capabilities of SASTs. There are also benchmarks targeting specific classes of bugs, such as crypto API usages [9]. However, the diversity and dynamic nature of real-world software can make creating comprehensive and representative benchmarks challenging. Variations in coding styles, application architectures, and the presence of third-party libraries can introduce complexities that are difficult to capture in a benchmark environment.

**Micro-benchmarks**, such as OWASP [30] and Juliet Test Suite [17], consist of simple test programs designed to assess SASTs. They can be manually collected or automatically generated, having several advantages. For instance, they are usually smaller in size, making it feasible to inspect a tool's report with moderate effort manually and to provide controlled scenarios for evaluating the individual functionalities of SASTs. However, micro-benchmarks often lack the complexity and diversity found in real-world applications. This may limit their representativeness and applicability. Therefore, the evaluation results obtained from these benchmarks may not fully reflect the performance of SASTs in real-world scenarios.

Table 1. Representative benchmarks. ("Size" refers #case for micro and #project for real-world benchmarks.)

| | Benchmark | Language | Size | Release Time | Update Time |
|---|---|---|---|---|---|
| Micro | OWASP [30] | Java | 2,740 | 2015-04-16 | 2016-10-02 |
| | Juliet Test Suite [17] | C++, Java | 92,980 | 2010-12-01 | 2017-10-01 |
| | SecuriBench-Micro [56] | Java | 96 | 2005-08-01 | 2014-12-23 |
| | PointerBench [76] | Java | 34 | 2015-06-11 | 2016-04-28 |
| | PTABen [4] | C/C++ | 400+ | 2015-11-16 | 2025-02-25 |
| | DataraceBench [51] | C/C++, Fortan | 373 | 2017-05-26 | 2023-08-24 |
| | DroidBench [14] | Java | 190 | 2013-05-04 | 2023-04-17 |
| | ICCBench [83] | Java | 24 | 2015-04-27 | 2017-06-21 |
| | WebGoat [31] | Java | 70+ | 2016-02-06 | 2023-12-05 |
| Real-world | DaCapo [18] | Java | 8 | 2006-10-22 | 2023-11-08 |
| | Renaissance [68] | Java | 21 | 2019-05-06 | 2024-11-23 |
| | ManyBugs [48] | C | 9 | 2015-07-09 | 2019-09-04 |
| | Defects4J [33] | Java | 17 | 2015-04-17 | 2024-11-27 |
| | BugSwarm [81] | Java | 335 | 2019-02-17 | 2024-11-19 |
| | TaintBench [58] | Java | 39 | 2020-07-02 | 2020-07-02 |
| | BugsC++ [12] | C/C++ | 22 | 2021-04-28 | 2023-12-20 |
| | SecBench.js [16] | JavaScript | 19 | 2021-05-12 | 2024-10-04 |

### 2.2 Motivation

Despite the existing benchmarks for evaluating SASTs, there is a need to understand the limitations of current benchmarks and develop effective approaches for assessing their capabilities and limitations. Recent research [11] has highlighted the inadequacy of existing evaluation methods, and practitioners' awareness of benchmarks does not necessarily translate into confidence. Instead of trusting benchmarks, practitioners emphasize other subjective considerations, such as cost, corporate pressures, and peer recommendations, as well as the overall reputation of the tools. Furthermore, practitioners find it hard to obtain accurate and valuable results using existing benchmarks [59]. Diverse benchmark design standards can skew evaluation results, whereas coarse-grained metrics may not accurately represent the capabilities of SASTs.

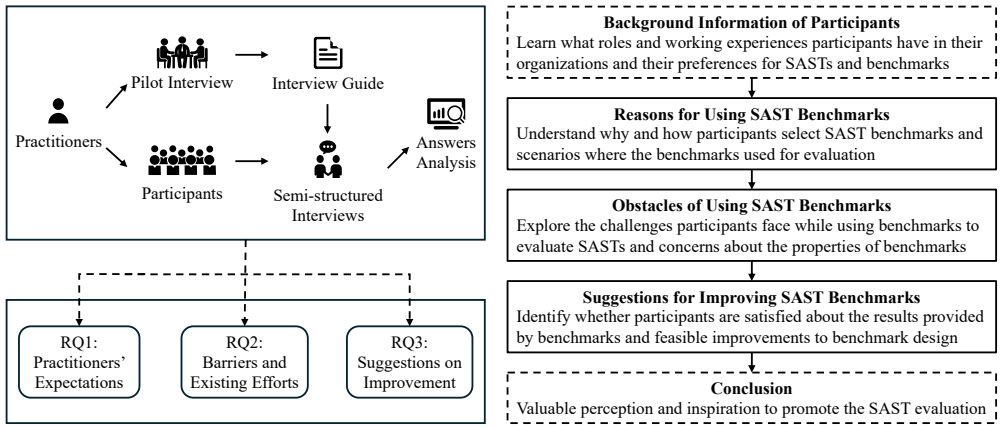
Hence, there is an urgent need to address the disparity between practitioners' expectations and the current evaluation process. This work aims to address this gap by understanding the expectations and challenges of practitioners during the evaluation process. Although several studies have introduced new benchmarks for SASTs [37, 43, 58, 60, 65], or used various benchmarks to assess existing SASTs [49, 55, 67, 89], there is still a lack of research that examines the expectations of the practitioners from benchmarks, the difficulties they encounter with existing benchmarks, what aspects could be improved, and why these improvements are necessary.

## 3 Study Methodology

We introduce our overall study methodology as shown in Figure 1. To investigate why practitioners utilize SAST benchmarks, identify the barriers they face in using them, and explore potential improvements for current benchmarks, we conducted semi-structured interviews with practitioners, which allowed for achieving the flexibility needed to get as much detailed information as possible [40]. Our research is conducted as shown in Figure 1(a).

### 3.1 Participants

The population we selected for the interviews included SAST developers, program managers, and security experts. In Ant Group, these three core roles closely collaborate in developing and




(a) Research methodology; (b) Interview guide.

Fig. 1. Overview of study methodology.

Table 2. Overview of interview participants.

| ID | Years | Position | Domain | Daily Tasks |
|---|---|---|---|---|
| P1 | 13 | SAST Developer | Application security | Code review, integration |
| P2 | 6 | SAST Developer | Data security | Testing, debugging |
| P3 | 7 | SAST Developer | Data security | Vulnerability analysis |
| P4 | 3 | SAST Developer | Quality assurance | Automated testing |
| P5 | 8 | SAST Developer | Endpoint security | Security audits |
| P6 | 5 | SAST Developer | Testing | Test-driven development |
| P7 | 7 | SAST Developer | Quality assurance | Code Quality checks |
| P8 | 8 | SAST Developer | Testing | Performance testing |
| P9 | 10 | SAST Developer | Application security | Code audits, maintenance |
| P10 | 11 | Program Manager | Endpoint security | Project planning |
| P11 | 15 | Program Manager | Security incident response | Incident management |
| P12 | 10 | Program Manager | Application security | Strategy development |
| P13 | 8 | Program Manager | Data security | Resource allocation |
| P14 | 7 | Security Expert | Application security | Threat modeling |
| P15 | 5 | Security Expert | Endpoint security | Risk assessment |
| P16 | 7 | Security Expert | Security development | Tool development |
| P17 | 10 | Security Expert | Security product | Product evaluation |
| P18 | 11 | Security Expert | Security incident response | Response planning |
| P19 | 7 | Security Expert | Data security | Data protection strategies |
| P20 | 4 | Security Expert | Application security | Security training |

evaluating SASTs. Hence, they could offer insightful opinions on the practice of SAST evaluation. To ensure a diverse representation of expertise, we relied on snowball sampling [35], i.e., emailed invitations to engineers and managers within our professional network and asked them to forward the invitation to their colleagues. We extended 25 official invites to potential participants using the direct and indirect methods described above.

After excluding candidate participants (2 excluded) without experience with SAST evaluation, we recruited 20 participants (3 refused) for our semi-structured interviews. To ensure a comprehensive perspective, we carefully considered various factors, including product backgrounds (e.g., development, marketing, evaluation), static analysis experience (e.g., usage, research, improvement),

security contexts (e.g., bug finding, quality assurance, security service, etc.), and familiarity with SAST benchmarks, ranging from knowing a few to frequently using them.

Table 2 provides statistics and background information about the participants. The first column lists the participants' IDs provided to maintain confidentiality. The second column shows their years of experience developing or evaluating SASTs. The last three columns provide information about their positions, domains, and daily tasks within Ant Group.

### 3.2 Interviews

**Pilot Interviews.** To design the interview guide for our study, we conducted pilot interviews with three practitioners from our professional network. Two hold Ph.D. degrees in computer science, while the other has decades of experience in the software analysis industry. The three practitioners are neither affiliated with Ant Group nor involved in the following formal interviews.

**Protocol.** To facilitate a thorough investigation of the participants' expectations, we conducted a series of in-person, semi-structured interviews using the interview guide developed through the pilot interviews. We conducted 30–40-minute interviews with 20 practitioners who accepted our invitation. These participants are referred to as P1–P20 throughout the rest of the paper.

We performed the interviews using a predefined set of questions as a guide, following the guide for a semi-structured interview [8, 40]. Meanwhile, we encouraged the participants to express their thoughts and insights on related topics freely. The interview guide, shown in Figure 1(b), consists of questions arranged in four segments, ordered by increasing depth as applicable:

- *Background Information of Participants.* We asked the participants what roles they play in their organization and how long they have worked on stuff related to the SAST evaluation.
- *Reasons for Using SAST Benchmarks.* We asked the participants general questions about how they select SAST benchmarks, instances where they have utilized them, and what they care about in the evaluation results.
- *Obstacles of Using SAST Benchmarks.* We asked the participants about the deficiencies when using SAST benchmarks, seeking to understand the barriers to their practical use.
- *Suggestions for Improving SAST Benchmarks.* We asked the participants to provide perspectives on improving the benchmarks' usefulness and their desires.

**Answers Analysis.** Following the procedure in previous work [11, 34], after the interviews, the answers were transcribed using an automated transcription service [64], and one of the co-authors anonymized the text. We chose reflexive thematic analysis [19] combined with inductive coding for our analytical approach [21] as it offered us the flexibility of capturing both latent and semantic meaning based on the complex interactions between the participants' perceptions and contexts. Our thematic codes focused on the roles of SAST benchmarks, barriers to using the benchmarks, and improvements to current practices. One of the co-authors looked through the transcripts and assigned thematic codes as they went. Once the set of codes was established, another co-author validated it. To assess the level of agreement, we compute Cohen's Kappa coefficient (0.76) for code sets between the two co-authors. The outcome indicates substantial agreement in the coding decision of the two co-authors. They address their disagreements to find a consensus. To reduce bias, the other co-authors evaluated, agreed on, and validated the final set of clear codes.

Furthermore, we had to iterate through the steps of thematic analysis (familiarization, coding, identifying potential themes, refining to finalize the themes). Finally, we identified 37 initial codes and later grouped them into ten key themes, all generated naturally from the data using our inductive approach. We discuss data availability in our study in § 7. In the following sections, we answer our research questions by linking the questions to the interview parts and discussing our findings in detail.

### 3.3 Scope and Limitations

Our study was primarily confined to practitioners from Ant Group. The use of snowball sampling and the exclusive participation of specific roles (SAST developers, program managers, and security experts) might have influenced the diversity of perspectives obtained. This approach, while facilitating deep insights from a focused group, may limit the breadth of viewpoints.

However, our methodology aligns with previous studies that concentrated on individual companies [25, 34, 80] and similar sample sizes [11, 34, 44]. Moreover, we want to emphasize that we aim not to achieve statistical generalizability but to provide salient insights into practitioners' experiences, challenges, and suggestions for evaluating SASTs. Future work could extend these findings by examining organizations of varying sizes and roles to capture broader needs. We present a more detailed discussion of threats to validity in § 5.3.

## 4 Study Results

We begin by describing the benefits of SAST benchmarks to practitioners (§ 4.1). We then focus on their obstacles to using existing benchmarks (§ 4.2) and suggestions for improving them (§ 4.3).

### 4.1 Roles of SAST Benchmarks

Participants' reasons for using SAST benchmarks vary according to their responsibilities in the company and the common circumstances they encounter. When evaluating SASTs, they have distinct expectations and prioritize different goals (**RQ1**).

**SAST Developers**. The SAST developers (9/20) rely on benchmarks for various reasons. Benchmarks establish a standardized and reproducible framework for evaluating proposed analysis algorithms. They enable the developers to systematically compare new approaches against the existing ones, allowing for the quantification of improvements. Specifically, benchmarks support both functional and non-functional requirement testing.

First, all the SAST developers (9/9) explicitly mentioned that benchmarks help test the functionalities of SASTs. By measuring the outputs of SASTs on a benchmark suite, the developers can gain confidence that the tools satisfy expected behaviors. For instance, the developers may discover that the tool fails to detect particular vulnerabilities or exhibits low precision for specific dataflow patterns, prompting them to refine the algorithms. "*While optimizing a static taint analyzer, I used OWASP extensively. For example, I improved the memory modeling module to pass the test programs that use Java containers, such as ArrayList, LinkedList, and HashSet) to propagate tainted values.*" (P6)

Second, some SAST developers (4/9) also talked about benchmarks aiding in efforts to improve the code coverage of SASTs. By evaluating SASTs on different benchmark programs, the developers can identify areas where the tools may have limited code coverage. This information helps focus engineering efforts on targeted improvement areas. "*Our team will collect existing test cases from various sources, such as the Juliet Test Suite, benchmarks from other research papers, and regression tests from open-source SASTs. After adapting them accordingly, I will add them to the regression test suite of the SAST tool I developed.*" (P2)

> **Finding 1** (F1): SAST developers utilize benchmarks to test the functionalities of static analyzers and increase the analyzers' code coverage.

**Program Managers**. SAST benchmarks serve multiple crucial purposes for the program managers (4/20) responsible for managing and coordinating security efforts within their departments.

One significant benefit of benchmarks is that they provide tangible evidence of a tool's capabilities, which is important for *marketing initiatives* (3/4). By having benchmarks that showcase

the performance of a security tool, program managers can attract potential customers by demonstrating the tool's effectiveness in various scenarios. For example, a manager emphasized the value of demonstrating a tool's benchmark performance to potential customers, stating, "*Our customers indeed care about the performance of SASTs on various benchmarks. Demonstrating a tool's benchmark performance can be influential in attracting potential customers.*" (P10)

Besides, benchmarks play a vital role in *ensuring compliance* (2/4), e.g., meeting specific industry standards. Security vendors must often demonstrate that their tools "*meet specific certifications or compliance requirements*" (P12). Vendors who evaluate their tools against well-recognized benchmarks can demonstrate to their clients their compliance and readiness for certification. This action builds confidence in the tool's capabilities and meets the expectations of customers who prioritize adhering to industry standards. Some notable industry standards include MISRA-C, AUTOSAR, CERT, and the CWE Top 25.

**Finding 2** (F2): SAST benchmarks are critical for program managers in marketing initiatives and ensuring compliance.

**Security Experts**. The security experts (7/20) utilize SASTs to examine proprietary codebases, open-source code (e.g., for software supply chain security), and customer code. They are one of the roles in Ant Group that are most concerned with ensuring trustworthy SAST evaluation.

One of the main reasons most security experts (5/7) use benchmarks is to enable objective comparisons between SASTs regarding precision, recall, and scalability. These comparisons help the security team with technology selection and product planning. For example, some experts mentioned the importance of having a comprehensive benchmark suite: "*We have many requirements for developing our SASTs and buying third-party ones. In addition to the claims made by the developers or salespersons, we need a benchmark suite to examine the capabilities of the SASTs comprehensively. However, the industry's lack of authoritative and effective evaluation methods is a 'pain point' in application security.*" (P14, P17)

Another goal of benchmarking is to determine where a SAST tool meets the specific needs of security experts (4/7). For example, they may need to customize SASTs, which entails various aspects, such as customizing sources, sinks, sanitizers, and reports, representing the import, outbreak, processing, and log of taint data, respectively. Benchmarks enable experts to evaluate the customizability. Hence, they desire to examine the flexibility of extending the capabilities of SASTs beyond their out-of-the-box functionalities.

**Finding 3** (F3): Security experts rely on benchmarks to provide fair comparisons across SASTs' strengths and weaknesses and assess their customizability.

### 4.2 Barriers in Using the Benchmarks

We identified several concerns participants raised regarding using SAST benchmarks (**RQ2**). First, the participants mentioned several challenges when running and analyzing each test program within the benchmark suites. Additionally, they are concerned about the properties and guarantees of the benchmark suites as a whole.

**Comprehensive Results Diagnosis**. Most participants (17/20) found it challenging to diagnose the results reported by SAST tools when using benchmarks comprehensively. The results necessitate auditing by practitioners with security and development capabilities, such as SAST developers and security experts. While two-sided test cases within several benchmark suites can help with such challenges, the participants acknowledged a lack of comprehensive diagnostic information in existing benchmarks to identify the reasons for positive and negative cases. Here, positive cases

are actual bugs placed into benchmark programs, whereas negative cases are safe cases that may be misidentified as false positives.

*Fine-grained Characteristic Labels.* Some participants (11/20) expected benchmarks to reveal individual analysis capabilities by incorporating fine-grained characteristic labels for each test case. While existing benchmarks include various language features, they may not discriminate the analyzers' capability at a fine-grained level, and they may not provide detailed documentation or guidance for users. For example, one SAST developer commented, "*When I'd like to test the tool's capacity to analyze collections, the fine-grained cases evaluating iterators, maps, and more complex collection copying and passing methods would be favorable for me over ordinary cases involving simple collection deposit or retrieve.*" (P2) Coarse-grained test cases result in SAST benchmarks not being well utilized in industrial practices.

*Comprehensive Fault Interpretation.* Several participants (9/20) stressed the importance of fault interpretation in performing a comprehensive diagnosis. The most frequently mentioned difficulty when using benchmarks for diagnosis is the lack of a comprehensive bug trace, a series of program locations that indicate the possible occurrence of a bug. Without this information, it is difficult to determine if a SAST tool accurately detects the underlying bug. A SAST developer lamented, "*Several benchmarks only annotate the crash point but do not provide the detailed bug trace leading to the crash. Even if our tool identifies the crash point (e.g., the sinks), we have no confidence that it reports the bug based on the correct root cause.*" (P1) Additionally, participants noted the lack of proof-of-concept (PoC) information, which could help users validate whether the reported vulnerability can be triggered. Without this support, practitioners struggle to assess the severity and impact of the bugs. A security expert complained, "*The lack of PoC support makes it difficult for us to assess the severity and impact of identified bugs, not to mention reproduce the bug.*" (P16)

*Granularity of Abstraction.* Participants also raised concerns about the granularity of abstraction used by static analysis tools. These tools rely on various abstractions and levels of analysis sensitivity to strike a balance between precision and efficiency [50]. Several participants (9/20) expected that benchmarks could provide a deeper understanding of the extent to which the analyzer achieves the *X-sensitivity*. For example, a security expert mentioned the importance of covering paths of different hardness: "*Although existing benchmarks, such as Juliet Test Suites, contain many infeasible paths, their path constraints are simple and do not push the limit of a path-sensitive analyzer, in terms of the number of the path conditions, the types of the constraints, and the hardness of solving those constraints.*" (P15) Besides, context sensitivity can be instantiated into different forms, such as object context [41], calling context [42], and thread context [54]. However, existing benchmarks rarely specify the "sizes" or complexity of the calling contexts they cover.

**Finding 4** (F4): Participants are concerned about the lack of diagnostic information about SAST evaluation results, indicated by fine-grained characteristic labels, comprehensive fault interpretation, and granularity of abstraction.

**Revealing Real-World Difficulties**. Many participants (15/20) expected a comprehensive benchmark suite to effectively reflect the intricacies of the SAST tools regarding real-world issues.

*Intended Unsound Trade-offs.* One of the most challenging aspects of SAST benchmark design is capturing the trade-offs that static analysis tools often make to remain scalable and precise. For example, some participants (8/20) noted that many industry-strength tools might intentionally sacrifice soundness to achieve better scalability and precision [36, 38, 47, 63]. These situations give rise to what we refer to as *"the intended unsound trade-offs"*. Unfortunately, these limitations are often undocumented and vary between tools, making it difficult for practitioners to evaluate their effectiveness in real-world applications. One security expert mentioned, "*The complexity of large*

*projects or systems often forces static analysis tools to make unsound trade-offs. We may misinterpret the results and make incorrect decisions without clearly understanding these limitations.*" (P18)

*Diverse Deployment Input.* Several participants (6/20) doubted whether static analysis tools could handle diverse input scopes when deployed in real-world scenarios. In practical applications, these tools often encounter challenges associated with different build systems, compilers, and dependencies. The complexity of these elements can significantly impact the performance and accuracy of static analysis tools. One program manager stated, "*Some of our partners have expressed the need to evaluate the capability of deployment contexts, which is widely considered in large-scale software development. However, the existing SAST test suites provided few insights into the number of relevant cases.*" (P13). This highlights the necessity for benchmarks that account for the complexities of various environments, ensuring that tools are robust and versatile enough to handle the intricacies of real-world software development processes.

*Realistic Business Logic.* An often-overlooked aspect is how to reveal the difficulty of dealing with business logic in bug detection. Business logic is a critical element of software that indicates how a project operates, processes data, and outputs results. It influences the specification (e.g., sources, sinks, and propagation rules) of the security bug (for both new and known bug types). Business logic bugs are often highly context-specific and may involve complex control flow patterns based on specific business requirements. Unfortunately, existing benchmarks may not capture the complexity and nuances of these real-world scenarios. Some participants (9/20) desired a comprehensive benchmark, including tests that evaluate how well tools understand and analyze business logic, as this can significantly impact the accuracy and relevance of the analysis results. By integrating business logic considerations, benchmarks can provide stakeholders with more relevant information for decision-making in real-world deployments.

**Finding 5** (F5): Participants find that SAST benchmarks struggle to expose the difficulties that tools encounter in real-world settings, such as intended unsound trade-offs, diverse deployment input, and realistic business logic.

**Flexible Benchmark Customization**. Some participants (16/20) recognized that many existing benchmark suites provide limited support for customization based on different contextual factors. They desire flexibility to customize and extend SAST benchmarks.

*User-friendly Tool Support.* Half of the participants (10/20) prefer to use their custom benchmarks or adapt existing benchmarks to perform specific security purposes, similar to regression testing selection [71], as most of the existing benchmarks are initially designed for research-first validation purposes rather than industrial practices. However, they lamented the lack of user-friendly tools for easily customizing and expanding benchmarks for specific evaluation scenarios and code patterns. For example, performing incremental analysis on large-scale projects has become common in industry. However, few user-oriented interfaces or plug-ins exist to customize benchmarks with a "one-click" approach. A security expert mentioned, "*A viable method is to publish benchmarks with mutation templates that facilitate creating variants of the test cases. For example, I may want to change the taint propagation paths by 'injecting' the code snippets I found intriguing.*" (P17)

*Extensive Community Collaboration.* Several participants (9/20) expressed a desire for a community-driven approach to assist in extending existing benchmarks. They stressed the insufficiency of collaboration among diverse practitioner roles. For example, static analysis researchers often lack incentives to maintain benchmark suites for SAST tools in the long term, as it requires time and effort for them to keep benchmarks updated, which can divert energy from publishing research papers. A security expert stated, "*Researchers are typically more incentivized to publish new findings than construct new test programs, leading to potential stagnation in benchmark evolution.*" (P18)

Table 3. Examples of intended unsound trade-offs. ("Tool P" is one of the SASTs developed in Ant Group.)

| Unsound Choices | CSA | Infer | Tool-P |
|---|---|---|---|
| loop unroll times | ✓ | ✗ | ✓ |
| Call depth | ✓ | ✓ | ✓ |
| Callback level | ✗ | ✓ | ✗ |
| Iteration rounds | ✗ | ✗ | ✓ |

Additionally, practitioners in companies with proprietary codebases may be reluctant to contribute to existing benchmarks, considering intellectual property protection and competitive advantage. As expressed by a SAST developer, "*Industries are hesitant to share proprietary code for benchmarking purposes, which limits the availability of diverse datasets critical for accurate SAST tool evaluations.*" (P9) Worse, participants mentioned that even open-source developers, the main strength of benchmark construction and contribution, rarely offer documents for adding new cases within their repositories. There is currently a lack of joint community participation from practitioners such as developers and researchers.

**Finding 6** (F6): Participants attribute the inadequacy of SAST benchmark customization and contribution to the lack of user-friendly tools and extensive community collaboration.

### 4.3 Improving the Current Practices

Our main goal in this work is to improve the evaluation benchmarks for SASTs by incorporating participants' preferences regarding benchmark design. Based on the input from participants, we have identified several intriguing proposals for enhancing the evaluation process (**RQ3**).

*4.3.1 Analysis Capabilities.* Participants highlighted several important capabilities of SAST tools that are rarely assessed or are not well-addressed by current benchmarks.

**The Intended Unsoundness Problem**. The intended unsoundness accounting for scalability and precision is prevalent in industry-strength analyzers. In Table 3, we list a few sources of unsoundness mentioned by some SAST developers. These participants (6/20) demand benchmarks capable of revealing them. "*I would like to have benchmarks that can reveal such trade-offs so that I can understand them and trace their sources.*" (P15) For example, to reveal the limitation of loop unrolling, we may bury the sink of a bug trace in the $n$-th iteration of a loop.

Notably, there are many bounded or under-approximation analysis techniques [22] that *go beyond the syntax restrictions*. For example, Tool P performs pointer analysis "*within a fixed time and memory budget*" (P1). When terminated early, it provides an underapproximate result, meaning that the computed points-to relations may not be sound but are often sufficient for finding bugs. More specifically, Tool P employs priority-driven pointer analysis, prioritizing the analysis of methods more likely to generate and propagate taint. Such heuristics can be unintuitive to SAST users but are important for evaluating SASTs in-depth. As a security expert said, "*Many 'tricky' trade-offs in SASTs are rarely mentioned in the existing literature. I would like to have benchmarks that can reveal such trade-offs so that I can understand and may adjust the heuristics.*" (P19)

**Finding 7** (F7): Intended unsound trade-offs in SASTs can come from different sources of under-approximation, such as the intuitive loop unroll times and the "tricky" iteration bounds in pointer analysis. SAST benchmarks should consider both to identify root causes.

**Deployment Robustness through the Lens of Input Acquisition**. The robustness of acquiring the necessary input for analysis is important for deploying static analysis tools. To elaborate,

various analyzers, including Infer, CodeQL, Coverity, and Tool P, rely on capturing the build process to construct a "compilation database" [1] to facilitate their analysis. This component is fundamental to many industry-strength analyzers. Participants in our study expect benchmarks that can assess the capability of SASTs to handle different contexts, such as build systems, compilers, and related dependencies, when acquiring the compilation database.

- *Build Systems*: Programs can rely on various build systems, such as GNU Make, Bazel, Ant, Maven, etc. To truly reflect how well SASTs handle the difficulties of building programs, benchmarks that emulate popular build systems should be included (8/20). A SAST developer mentioned, "*I've already tried and succeeded in analyzing a 'Maven-built' project using Tool P, but the practicality of constructing the same project with alternative build configurations, dependencies, and build scripts without completing the analysis is still unknown.*" (P4)
- *Compilers*: SASTs often operate over different Intermediate Representation (IR) [24] generated by compilers or fuzzy parsers. The version and the settings of the IR generation tools affect the IR and the applicability of SASTs (3/20). According to a SAST developer, "*It is possible that a program can only be compiled with Clang version 12.0.0 or higher, while the analyzer Tool P can only take LLVM IR version 3.6. However, it is hard for me to find benchmarks that comprehensively assess the compatibility and effectiveness of SASTs across various compilation contexts.*" (P5)
- *Dependencies and Configurations*: When using SASTs, practitioners may experience challenges with third-party dependencies, configurations, and environment setup [61, 91]. It is recommended that these scenarios be included to assess how well SASTs handle complex environments in real-world settings (5/20). As a developer commented, "*Our tools are designed to be more robust in handling diverse dependency scenarios to ensure accurate analysis in complex projects; thus, it is necessary to find benchmarks evaluating it.*" (P7)

Indeed, previous research [44, 82] has emphasized the importance of integrating SASTs into developers' workflows. However, a noticeable gap exists in constructing SAST benchmarks that systematically evaluate these aspects.

**Finding 8** (F8): SAST benchmarks need to evaluate the tools' capabilities to acquire the input for the analysis, such as handling various build systems, compilers, and dependencies, reflecting the flexibility of workflow integration.

**The Evaluation of Incremental Analysis**. Several participants (5/20) stressed the importance of evaluating incremental program analysis in realistic scenarios, particularly in large-scale companies where the complexity and volume of code can pose frequent challenges. The existing work on incremental analysis has made considerable strides; however, there are notable limitations in the datasets used for these studies. Specifically, these datasets often feature either small granularity [52, 53] changes, such as single-line additions, modifications, or deletions, or large granularity [13] changes, which might encompass entire project versions. This dichotomy makes it difficult for practitioners to capture the diverse range of code changes in real-world scenarios, thus limiting the applicability and robustness of the evaluation results.

To truly reflect a tool's analysis capabilities of the multifaceted nature of code development and maintenance in practice, it is crucial to build datasets encompassing *various granularities* of code changes. This approach would better align the incremental analysis methods with the actual needs and practices observed in the industry. As a SAST developer highlighted, "*For example, in real-time analysis during the coding process in an integrated development environment (IDE), it may be necessary to perform incremental checks after modifications involving several lines, functions, classes, files, packages, or sub-projects.*" (P5)

**Finding 9** (F9): Effectively evaluating incremental analysis capability requires SAST benchmarks that reflect more diverse code change granularities, aligning with industry needs.

*4.3.2 Benchmark Construction.* During our study, the participants proposed several suggestions for constructing new benchmarks.

**Quantifying Semantic Diversity**. Quantitative diversity is important for comparing and improving SAST benchmarks. Furthermore, several participants (6/20) call for going beyond simple syntactic metrics and considering the complexity of the semantic analysis: While existing benchmarks often rely on basic metrics like the lines of code and the functionalities of the programs [18], these metrics alone may not accurately reflect the difficulty of the static analysis techniques. For example, context sensitivity is a common axis for improving analysis precision. One participant mentioned covering programs with different scales of calling contexts. "*For interprocedural analysis, we want to evaluate the SAST tool's capability to analyze code across multiple functions. Even if a program is large, its number of calling contexts can be small. Thus, we cannot evaluate the tool's capability to handle large calling contexts.*" (P7)

Unfortunately, while syntax features such as lines of code can be measured precisely and faithfully, assessing programs' "semantic properties" is more challenging. For instance, determining the number of calling contexts in programs requires constructing a call graph. If we use existing pointer analyses to approximate the call graph, the results may be biased due to the limits of the analysis (precision, recall, etc.). Hence, if the "a posteriori" static analysis is not a good choice, a possible direction is to build test programs via the "a priori" approach, i.e., controlling the semantic features when generating the programs. For example, a SAST developer commented: "*Consider a taint-style bug where the sink hides behind the n-th iteration of a loop. We can control n when building the loop. However, analyzing the exact number of iterations for an unseen loop is stunningly challenging.*" (P1)

**Finding 10** (F10): To realize realistic diversity, SAST benchmarks can involve semantic complexity indicators besides simple syntactic ones, accomplished via the "a priori" approach.

**Reducing Large, Real-world Programs**. Participants expressed concerns about the ability of benchmarks to reflect the complexities and challenges encountered in real-world scenarios accurately. To this end, a common practice is to collect benchmarks from real-world failures. However, real programs can be large, and bug traces can be complex, making it challenging to automatically extract well-defined, easy-to-reproduce test cases.

Therefore, some participants (8/20) emphasized it is essential to construct *small test programs* that exhibit *identical* root causes as their large-scale counterparts. For example, PTABen [4] has offered complex test cases simplified from real programs to examine pointer analysis thoroughly. By providing such programs, practitioners can quickly assess whether SASTs handle real-world challenges. One SAST developer stated, "*Similar to test case reduction, I would like to have some minimal proof-of-concept programs to test my algorithms or third-party SASTs quickly.*" (P9)

Unfortunately, constructing such programs poses a significant challenge. One potential solution is to utilize delta debugging [86] to reduce real-world programs. However, there are two challenges to address. First, checking whether a reduced program exhibits identical root causes is challenging, and the decision may not be based on the default output of SASTs. Hence, we need to guide the reduction with more informative outputs from SASTs (e.g., the "necessary program points" in bug traces). Second, since the size of real-world programs can be huge, we need to adapt existing test case reduction algorithms [28, 62, 70, 87] for this specific context. For example, "*We may use static analyzers to perform some forms of program 'program debloating,' which helps reduce the input of the delta debuggers.*" (P16)

**Finding 11** (F11): To reflect real-world complexities, it is advisable to create reduced test programs with identical root causes as their larger-scale counterparts. Potential solutions like improved delta debugging help guide the reduction process.

**Customization Support**. In addition to looking for "better off-the-shelf benchmarks," participants also desire flexibility to customize benchmarks based on contextual factors, such as offering optional subsets, creating variants, and incentivizing contributions.

- *Subsets Selection*: Many participants (12/20) highlighted requirements for selecting subsets of benchmarks based on different contexts. They suggested that benchmarks offer tools/interfaces for selecting subsets that only include particular features that they really want. For example, one security expert shared an interesting finding: "*When utilizing OWASP to test my analyzer, I may want only to run some cases relevant to my focus, e.g., container-induced taint flows. However, the customization support has not been supplied well.*" (P18)
- *Variants Creation*: Some participants (5/20) mentioned the need to create "variants/mutants" of existing benchmarks. One of the motivations for mutating existing benchmarks is to address the evolving nature of software platforms. For example, taint analyses must adapt to the frequent updates of the Android operating system when evaluating apps designed for older versions. This demands the creation of benchmarks that can reliably test new tools and their improvements over time, and reusing existing benchmarks such as DroidBench [14] (e.g., by mutating them) can reduce the manual efforts in creating new ones from scratch.
- *Contribution Incentive*: Several participants (9/20) desired feasible approaches to incentivizing benchmark contribution by encouraging community collaboration among practitioners in various roles. When it comes to benchmark contribution, they find that there is little interaction between existing academic efforts (e.g., papers/benchmarks related to static analysis) and industrial practices (e.g., SASTs evaluation), and similar circumstances happen among different industries. Participants suggest that we launch contribution platforms with incentives and promote stakeholders to establish partnerships with others to maintain available benchmarks. Collaboration within the community can promote knowledge exchange, drive continuous improvement, and ensure that the benchmarks remain relevant and up-to-date with evolving industry practices. For example, one security expert said, "*If I encounter certain troubles about the benchmarks, I will log on to the related forums, platforms, or conference sites and attempt to obtain solutions from practitioners sharing their experiences.*" (P20)

**Finding 12** (F12): Participants call for more options to customize SAST benchmarks, emphasize selecting subsets, creating variants/mutants, and incentives for community contribution.

## 5 Discussions

In this section, we summarize our findings (§ 5.1), delve into actionable implications for practitioners (§ 5.2), present the threats to validity (§ 5.3), and discuss the ethical considerations (§ 5.4).

### 5.1 Summary of Findings

Our findings reveal salient aspects of practitioners' motivations, challenges, and expectations for SAST evaluation. Addressing them in our software engineering community is particularly important because we often use the benchmarks to evaluate and improve the artifacts we create.

**Role-specific Motivations**. Practitioners expressed different motivations for using SAST benchmarks regarding their various roles. While existing research presents the study of developers or tool users [11, 25, 27, 44, 84], it has not addressed motivations based on the specific roles of practitioners.

Table 4. Summary of Benchmark Limitations.

| Limitation | Finding | Existing Related Efforts |
|---|---|---|
| Diagnostic Gaps | Fine-grained labels | [4, 12, 14, 16, 30, 31, 51, 56, 68, 76] |
| | Fault traces | [17, 30, 58] |
| Revealing Real-world Complexity | Intended unsound trade-offs | [20, 23, 46, 46, 79, 85, 88] |
| | Deployment complexities | [12, 16, 33, 48, 68] |
| Insufficient Customizability | Customization tools | [4, 12, 16, 18, 33, 51, 68] |
| | Community collaboration | [4, 16, 81] |

- **RM1** *(SAST Developers)*: Many practitioners prioritize the capacity of benchmarks to evaluate the capabilities of SASTs finely, particularly SAST developers. They claim that they largely utilize benchmarks to verify functionality (e.g., taint propagation accuracy) and further investigate how to increase the tools' code coverage (F1). It has proved that benchmarks help developers continuously produce effective tools [10, 55], improve techniques for detecting vulnerabilities [12, 16], and enhance the soundness of tools [20].
- **RM2** *(Program Managers)*: Several practitioners, particularly program managers, regarded benchmarks as helpful in ensuring SASTs comply with domain-specific needs or compliance (e.g., CERT C/C++ standards [74]). They also emphasized that the outstanding benchmarking performance of tools contributes to marketing (F2).
- **RM3** *(Security Experts)*: Most practitioners believed that benchmarks could provide objective evaluations and comparisons across various SASTs. Furthermore, we observe that practitioners, particularly security experts, expressed the positive effect of benchmarks on evaluating the customizability of SASTs, promoting extension beyond out-of-the-box functionalities (F3). It is critical that benchmarks can evaluate the *actual* performance of tools [66] and fairly compare them [67, 73, 89], providing experts with recommendations for selecting tools [55] and determining the best tool for individual work [49].

**Benchmark Limitations**. The findings analyze several limitations in designing and constructing SAST benchmarks, which have been highly concerning to practitioners. Table 4 summarizes relevant findings in our study and representatives of existing efforts.

- **BL1** *(Diagnostic Gaps)*: Nearly all practitioners attribute insufficient benchmark evaluation to a lack of diagnostic information about results (F4). First, they usually struggle with missing the fine-grained labels (e.g., data flow types and path sensitivity levels) of benchmarks. Real-world benchmarks tend to ignore information about labels or just list the list of libraries or projects the case arises from (e.g., Renaissance [68], BugsC++ [12], and SecBench.js [16]). Micro-benchmarks realize it to varied extents: benchmarks such as OWASP [30], WebGoat [31], and Juliet Test Suite [17] label each case with the vulnerability, while benchmarks such as SecuriBench-Micro [56], PointerBench [76], PTABen [4], DataraceBench [51], and DroidBench [14], have made attempts by providing characteristic labels for the test cases; however, their type of labels, usually relevant to specific targeted problems (e.g., analysis sensitivities, and data race types), vary greatly. Practitioners noted that these benchmarks still lack standardization, resulting in significant design variation. This inconsistency makes it challenging to comprehensively and objectively reflect the "full picture" of SASTs. Second, to supplement fault traces, although existing efforts have generated patches and reports for several benchmarks (e.g., OWASP [30], Juliet Test Suite [17], and TaintBench [58]), they may not reveal the root causes of false positives and negatives and lack sufficient evidence for accurate diagnoses (e.g., the exact capability required for fixing the FP/FN).

- **BL2** *(Revealing Real-world Complexity)*: We find that most practitioners struggle to expose the issues that tools face in real-world settings via existing benchmarks (F5). First, our findings expose a blind point within SAST evaluation: while there have been several efforts to find soundness bugs in various analyzers [23, 46, 46, 79, 85, 88] (including design and implementation flaws), there is a lack of study on systematically evaluating and understanding the intended unsound trade-offs. Existing research finds that benchmarks could help practitioners uncover and document potential unsound properties that affect the efficacy of SASTs [20]. However, few benchmarks have been designed to evaluate the capability. Second, we find that practitioners always use SASTs in various environments, prompting them to consider factors that affect the ability to obtain the "inputs" for SASTs. Existing benchmarks considering deployment issues mainly lie in three aspects. Some benchmarks are initially created for deployment in specific environments (e.g., Renaissance [68] and ManyBugs [48]), and others have considered presetting frameworks or command-line interfaces for case execution and tool evaluation based on environment setup (e.g., Defects4J [33] and BugsC++ [12]). Some even offer exploit scripts with test oracles to replicate various deployment circumstances (e.g., SecBench.js [16]). However, there is a dearth of research on deployment robustness measurement, and existing benchmarks cannot analyze complicated business logic or allow evaluation of different deployment challenges.
- **BL3** *(Insufficient Customizability)*: We observe that practitioners are concerned about benchmark customization for SAST evaluation. Most practitioners desired flexibility in customizing benchmarks to evaluate SASTs under diverse contextual factors and further seek potential avenues (F6). Fortunately, several existing benchmarks have designed different approaches for their customization, such as "parameterized" commands (e.g., *test-harness.sh* script in DataraceBench [51]), local code modification (e.g., class *Callback* in Dacapo [18]), specific selection options (e.g., versions/projects in Defects4J [33], cases in BugsC++ [12], and modules in SecBench.js [16]), high-level tools (e.g., interfaces in PTABen [4] and plug-ins in Renaissance [68]). Their contribution should be recognized, as many other benchmarks are static and provide little customization. However, as the development of SASTs paces, practitioners have increasingly specific and professional requirements for the dynamic customization of benchmarks, such as mutation test programs. Furthermore, several benchmarks also provide avenues for the community of relevant fields to contribute test programs or evaluation options (e.g., PTABen[4], BugSwarm [81], and SecBench.js [16]), but collaboration has not been highly motivated.

### 5.2 Actionable Implications

As summarized in Table 5, we draw several key actionable implications categorized by researchers and benchmark builders based on our findings.

**Researchers**. The findings reveal several problems and research opportunities for evaluating and testing SAST tools, which would be worth further exploration.

- **R1** *(Identifying Intended Unsoundness)*: Beyond testing for soundness bugs resulting from design and implementation flaws, our study suggests that it is promising to automatically identify the "intended unsoundness", that is, to integrate test cases with controlled elements, such as bounded loop unrolling and under-approximated pointer analysis (F7). Further studies may explore various sources, such as combining documents, issue trackers, and dynamic analysis, to identify the undocumented or misclaimed intended unsoundness designs.
- **R2** *(Measuring Deployment Robustness)*: To measure the deployment robustness, we should focus the evaluation study on the robustness of input acquisition, such as different build systems, compiler versions, third-party dependencies, configuration settings, and other environmental setup complexities that users may encounter (F8).

Table 5. Summary of actionable implications.

| Roles | Implication | Potential Inspiration |
|---|---|---|
| Researchers | Identifying Intended Unsoundness | Integrate test cases with controlled elements:<br>(1) bounded loop unrolling;<br>(2) under-approximated pointer analysis. |
| | Measuring Deployment Robustness | Collect the robustness of input acquisition by including:<br>(1) build systems and compiler versions;<br>(2) dependencies and configuration settings;<br>(3) other environmental setup complexities. |
| | Reducing Real Programs | Develop small test programs that reflect<br>real-world challenges by improving:<br>(1) delta-debugging techniques;<br>(2) program debloating techniques. |
| | Evaluating Incremental Analysis | Collect programs with code changes of various granularities:<br>(1) small: single-line additions, modifications, or deletions;<br>(2) large: entire files, projects, or repositories. |
| Benchmark Builders | Enhancing Quantitative Diversity | Ensure benchmarks quantify both syntactic<br>and semantic diversity by covering:<br>(1) features of the language, frameworks, patterns;<br>(2) controlled features using a prior" approach. |
| | Prioritizing Customization | Provide benchmarks that allow flexibility to<br>customize for specific purposes by offering:<br>(1) specific features: language constructs, dataflow patterns;<br>(2) tool/interface: plug-ins, case-generating templates. |
| | Academia-Industry Collaboration | Foster academia-industry collaboration by incenting:<br>(1) providing community honors;<br>(2) issuing management privileges;<br>(3) organizing workshops and competitions. |
| | Industry-specific Collaboration | Foster industry-specific collaboration by incenting:<br>(1) industry-specific awards or funds;<br>(2) simplified contribution mechanisms. |

- **R3** *(Reducing Real Programs)*. There is a need for small test programs that accurately reflect the challenges in practical settings (F11). However, constructing such programs is non-trivial, which could be mitigated by improving existing delta debugging or program debloating techniques [28, 45, 62, 87], e.g., by utilizing more output information and combining semantic-based program trimming [29] methods.
- **R4** *(Evaluating Incremental Analysis)*: Nowadays, it is crucial to evaluate incremental analysis capability, as there are frequent code modifications in the software ecosystems of giant corporations such as Ant Group, with numerous pull requests submitted daily. Future studies must be sensitive to the deficiencies of benchmarks in evaluation incremental program analysis [78, 90]. To evaluate the incremental analysis capability of SASTs, they need to collect programs with code changes of various granularities (F9).

**Benchmark Builders**. The findings provide promising recommendations for improving SAST benchmarks; we now examine several proposed solutions, believing them worth further discussion.

- **B1** *(Enhancing Quantitative Diversity)*: First, benchmark builders should ensure and justify that their benchmarks comprehensively cover program features (F4). They may provide an explicit list of the covered language features, framework features, and architectural patterns commonly used in real-world applications. Second, our study suggests guaranteeing quantitative semantic diversity of the benchmarks (F10). While enforcing such properties is challenging, building test programs with controlled features (using the "a priori" approach) can help users examine and *compare* test data quality through the quantified diversity.

- **B2** *(Prioritizing Customization)*: Considering evaluation needs and benchmark features, benchmark builders could also develop specific tools (e.g., plug-ins and case-generating templates) for subset selection and variant creation to meet their objectives about tools (F12). For example, suppose some developers merely focus on evaluating certain language constructs or dataflow patterns. In that case, publishing benchmarks with related selection/mutation templates could be more efficient and effective in meeting these security objectives.
- **B3** *(Academia-Industry Collaboration)*: To bridge the gap between research and real-world needs, practitioners should actively participate in collaborative efforts between academia and industry by engaging in discussions and sharing challenges, insights, and benchmarks (F12). We could encourage contributing to up-to-date benchmark customization in different ways, such as providing community honors, issuing management privileges, and organizing workshops and competitions [7]. For example, the verification community organized the SV-COMP (Competition on Software Verification) to facilitate the proliferation of different algorithms, implementations, and benchmarks. This model's success is spreading to other venues, such as the Test-COMP. However, it appears that such competitions have limited appeal in the SAST community. We hypothesize that, in part, this could be addressed by defining problems with an appropriate scope (manageable yet impactful) to attract more community participation.
- **B4** *(Industry-specific Collaboration)*: To guarantee that SAST benchmarks meet varied industrial needs, the community should promote collaboration across regulated industries and open-source projects (F12). This can be accomplished by providing focused incentives to contributors, such as industry-specific recognition (e.g., "Active Contributor" awards) or funds for collaborative projects. Partnerships with healthcare companies, for example, may focus on HIPAA compliance benchmarks, whereas finance collaborations may prioritize secure transaction management. Furthermore, simplified contribution mechanisms—such as anonymized code snippets or reduced test cases—can allow enterprises with proprietary codebases to join while maintaining secrecy. By widening collaboration, benchmarks can better reflect the particular issues of many sectors, enabling broader applicability and relevance across industries.

### 5.3 Threats to Validity

**External**. First, our study is confined to Ant Group, which may limit the generalizability of our findings. While Ant Group comprises thousands of practitioners working across diverse domains with a variety of SAST tools, its organizational practices and tool preferences may not reflect those of teams operating under different workflows or constraints. Second, our study focuses on industry perspectives, specifically program managers, security experts, and SAST developers. However, they may not cover the entire spectrum of practitioners, such as static analysis researchers.

**Internal**. Another threat to the validity of this study is how we conducted interviews. We requested participants respond to the questions using their expressions and offer as much pertinent information as necessary to address each question. Each interview ended with an open-ended question to encourage participants to share any additional information they wanted to provide on the topic. Besides, the face-to-face and remote text interviews had to be conducted differently. Despite this, there was still value in the results obtained from our remote participants; they could still give valuable insights from their previous experiences. Only four of the interviews fell into this category, limiting the impact of this threat.

**Construct**. A key threat stems from the interpretation of qualitative labels during thematic analysis. The terms in our study are defined based on participants' subjective descriptions, which may not fully align with standardized definitions in existing literature. While we mitigated this by cross-validating codes between authors, nuances in role-specific terms could lead to oversimplification of

complex constructs. Additionally, our interview guide focused on predefined research questions (RQ1–RQ3), potentially omitting other emerging themes that practitioners might be concerned about. Future work could supplement our study with comprehensive methodologies like surveys with Likert-scale items to enhance qualitative findings with quantitative metrics.

### 5.4 Ethical Considerations

Collecting sufficiently valuable responses from our interviews with practitioners was critical to this study. As a result, we followed previous studies published in recent years at top software engineering and computer security conferences (e.g., [11], [34]) that used similar participant recruitment and focused on a variety of industry concerns. We also followed standard ethical guidelines while doing qualitative software engineering research [77]. We assure that no personally identifying information is gathered and that responses are anonymized.

## 6 Related Work

**Study of SAST Benchmarks.** To date, there have been relatively few empirical studies analyzing SAST benchmarks. Zhu and Rubio-González [91] conducted a study on the reproducibility issues of Java defect datasets and proposed solutions that include automated dependency caching and artifact isolation to increase reproducibility. Hirsch and Hofer [39] performed a literature survey on benchmark suites for debugging. Miltenberger et al. [59] found that existing benchmarks often lack clear assumptions and threat models, leading to misleading evaluation results. They proposed a specification language that allows benchmark authors to specify security assumptions and a tool to generate exploit code based on the specifications. Previous studies primarily addressed the technical limitations of benchmarks, including issues of reproducibility and the absence of explicit assumptions, while giving insufficient attention to practitioner-centric factors that affect benchmark adoption, such as the necessity for diagnostic information and customization. Our work complements the existing literature by detailing practitioners' perceptions of SAST benchmarks, including their expectations, dissatisfaction, and suggestions for improving them.

**Study of Developers' Perspectives on SASTs**. Understanding developers' motivations and challenges in adopting SASTs is essential for improving usability and integration. Johnson et al. [44] identified high false positive rates and poor workflow integration as primary obstacles, undermining developers' trust and efficiency. Christakis and Bird [25] further emphasized the need for actionable warnings and contextual explanations, highlighting the importance of aligning static analysis tools with debugging practices. Expanding on these findings, Do et al. [27] conducted a user-centered study, revealing that compliance requirements and peer recommendations often drive adoption despite perceptions of tool inefficiency. More recently, Ami et al. [11] explored industry perspectives on static analysis for security testing, identifying concerns over false negatives and insufficient diagnostic information—issues that align with our findings on the need for fine-grained benchmark evaluations. Additionally, Witschey et al. [84] quantified security tool adoption, demonstrating that usability and seamless integration are critical for sustained use. These studies underscore the necessity of designing SASTs and benchmarks that reduce noise, provide actionable insights, and integrate effectively into development pipelines. Our study extends the research by focusing on benchmarks that evaluate SASTs, ensuring they address the diverse requirements of practitioners across roles and industries.

**Constructing SAST Benchmarks.** Several benchmark suites have been developed to evaluate SASTs, such as DaCapo [18], ICC-Bench [32], Defects4J [33], and OWASP [30]. These benchmark suites provide diverse test cases and scenarios to assess the effectiveness and capabilities of SASTs. Hao et al. [37] presented a security benchmark suite for testing static code analysis tools with test

cases derived from real-world applications. Luo et al. [58] focused on real-world benchmarking Android taint analyses, generating a complex benchmark suite. However, these benchmarks either rely on synthetic micro-benchmarks that lack real-world complexity or consider code complexity but overlook deployment challenges. This paper complements previous work by identifying gaps in capturing real-world trade-offs and challenges, such as diverse deployment contexts and business logic complexity. Additionally, most previous research focused on generating benchmarks automatically instead of offering dynamic customization or involving open-source contributions. For example, Witschey et al. [84] created a tool to generate XSS and SQL injection test cases for PHP. Bhandari et al. [15] retrieved CVE entries from open-source software and correlated the corresponding fixes. Our work uncovers solutions like flexible customization and community-driven collaboration, which could serve as a guide for benchmark design and usage.

**Comparing SASTs via Benchmarks**. Comparative studies are a crucial component of the SASTs evaluation. Pauck et al. [67] proposed a comparative framework for Android taint analysis tools, comparing six tools to assess whether the tools met expected outcomes in terms of functionality and accuracy. Qiu et al. [69] conducted a large-scale comparison and evaluation of three Android static analyzers. Zhang et al. [89] updated the versions of tools and benchmarks, adding a set of real-world applications to compare the tools in real-world scenarios. By building synthetic and real-world benchmarks, Li et al. [49] comprehensively evaluated and compared seven free or open-source SAST tools from different perspectives, such as effectiveness, consistency, and performance. Additionally, Liu et al. [55] mapped the tools' scanning rules to CWE, analyzing the coverage and granularity of the rules to provide a comprehensive multidimensional evaluation. Industry-strength SASTs often expose many configurable options. Recently, several efforts have evaluated the impact of the configurations. Mordahl [60], Mordahl and Wei [61] evaluated configurations in two Android taint analysis tools and found that configurations significantly impact the tools' performance, accuracy, and correctness. These comparative evaluations measured the performance of SAST tools but did not explore the practitioners' concerns and preferences regarding SAST benchmarks, which our work highlights in a role-specific manner.

## 7 Conclusion

SAST benchmarks play a pivotal role in evaluating the effectiveness of existing tools and guiding the development of new static analysis techniques. We present a qualitative study investigating the gap between current SAST benchmarks and industrial practitioners' demands. Our findings provide insights into practitioners' perceptions, uncover long-standing deficiencies in current practice, and reveal several directions for future exploration.

**Data Availability**: Following the prior work [34], we release the codebook used for analyzing interview answers at https://tinyurl.com/sast-eval. Due to Ant Group's data privacy policy, we cannot provide a detailed replication package containing other interview data and intermediate information. However, we emphasize that our study's primary contribution arises from the qualitative insights derived through in-depth interviews and rigorous analysis.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful feedback. This work is supported by the National Key R&D Program of China (Grant No. 2023YFB3106000), the National Natural Science Foundation of China (Grant No. 62302434, U2341212, 62302442), Ant Group, and the Leading Innovative and Entrepreneur Team Introduction Program of Hangzhou (Grant No. TD2020001). Peisen Yao is the corresponding author.

## References


[1] 2025. Compilation base of Infer. https://fbinfer.com/docs/analyzing-apps-or-projects/.
[2] 2025. CSA. https://clang-analyzer.llvm.org/.
[3] 2025. FlowDroid. https://github.com/secure-software-engineering/FlowDroid.
[4] 2025. PTABen. https://github.com/SVF-tools/Test-Suite.
[5] 2025. Soot. https://github.com/soot-oss/soot.
[6] 2025. SVF. https://github.com/SVF-tools/SVF.
[7] 2025. xAST Benchmark. https://github.com/alipay/ant-application-security-testing-benchmark.
[8] William C Adams. 2015. Conducting semi-structured interviews. *Handbook of practical program evaluation* (2015), 492–505. https://doi.org/10.1002/9781119171386.ch19
[9] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A comprehensive benchmark on Java cryptographic API misuses. In *2019 IEEE Cybersecurity Development*. 49–61. https://doi.org/10.1109/SecDev.2019.00017
[10] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Yao. 2023. Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks. *IEEE Trans. Software Eng.* 49, 2 (2023), 485–497. https://doi.org/10.1109/TSE.2022.3154717
[11] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2024. "False negative-that one is going to kill you."-Understanding Industry Perspectives of Static Analysis based Security Testing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 19–19. https://doi.org/10.1109/SP54263.2024.00019
[12] Gabin An, Minhyuk Kwon, Kyunghwa Choi, Jooyong Yi, and Shin Yoo. 2023. BUGSC++: A Highly Usable Real World Defect Benchmark for C/C++. In *38th IEEE/ACM International Conference on Automated Software Engineering*. 2034–2037. https://doi.org/10.1109/ASE56229.2023.00208
[13] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering*. 288–298. https://doi.org/10.1145/2568225.2568243
[14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 259–269. https://doi.org/10.1145/2666356.2594299
[15] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39. https://doi.org/10.1145/3475960.3475985
[16] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *45th IEEE/ACM International Conference on Software Engineering*. 1059–1070. https://doi.org/10.1109/ICSE48619.2023.00096
[17] Paul E Black and Paul E Black. 2018. *Juliet 1.3 test suite: Changes from 1.2.* US Department of Commerce, National Institute of Standards and Technology.
[18] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 169–190. https://doi.org/10.1145/1167473.1167488
[19] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. Analysing Data: The Editor's Work. In *Qualitative HCI Research: Going Behind the Scenes*. 51–60. https://doi.org/10.1007/978-3-031-02217-3_5
[20] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1263–1280.
[21] Virginia Braun and Victoria Clarke. 2021. Thematic analysis: A practical guide. (2021). https://doi.org/10.1177/1035719X211058251
[22] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A correctness and incorrectness program logic. *J. ACM* 70, 2 (2023), 1–45. https://doi.org/10.1145/3582267
[23] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *33rd ACM/IEEE International Conference on Automated Software Engineering*. 768–778. https://doi.org/10.1145/3238147.3240464
[24] Fred Chow. 2013. Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers. *Queue* 11, 10 (2013), 30–37. https://doi.org/10.1145/2542661.2544374
[25] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *31st IEEE/ACM international conference on automated software engineering*. 332–343. https://doi.org/10.1145/2970276.

2970347
[26] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112
[27] Lisa Nguyen Quang Do, James Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* 48, 3 (2020), 835–847. https://doi.org/10.1109/TSE.2020.3004525
[28] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Stephen N. Freund and Eran Yahav (Eds.). 1017–1032. https://doi.org/10.1145/3453483.3454092
[29] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-directed program trimming. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 174–185. https://doi.org/10.1145/3106237.3106249
[30] OWASP Foundation. [n.d.]. OWASP Benchmark. https://owasp.org/www-project-benchmark/.
[31] OWASP Foundation. [n.d.]. OWASP WebGoat Benchmark. https://owasp.org/www-project-webgoat/.
[32] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2013. Highly precise taint analysis for android applications. (2013).
[33] Gregory Gay and René Just. 2020. Defects4J as a challenge case for the search-based software engineering community. In *Search-Based Software Engineering: 12th International Symposium*. 255–261. https://doi.org/10.1007/978-3-030-59762-7_19
[34] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *46th IEEE/ACM International Conference on Software Engineering*. 187:1–187:13. https://doi.org/10.1145/3597503.3639581
[35] Leo A Goodman. 1961. Snowball sampling. *The annals of mathematical statistics* (1961), 148–170.
[36] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *Proc. ACM Program. Lang.* 3, POPL (2019), 57:1–57:29. https://doi.org/10.1145/3290370
[37] Gaojian Hao, Feng Li, Wei Huo, Qing Sun, Wei Wang, Xinhua Li, and Wei Zou. 2019. Constructing benchmarks for supporting explainable evaluations of static application security testing tools. In *2019 International Symposium on Theoretical Aspects of Software Engineering*. 65–72. https://doi.org/10.1109/TASE.2019.00-18
[38] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *39th International Conference on Software Engineering*. 519–529. https://doi.org/10.1109/ICSE.2017.54
[39] Thomas Hirsch and Birgit Hofer. 2022. A systematic literature review on benchmarks for evaluating debugging approaches. *Journal of Systems and Software* 192 (2022), 111423. https://doi.org/10.1016/j.jss.2022.111423
[40] Siw Elisabeth Hove and Bente Anda. 2005. Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research. In *11th IEEE International Symposium on Software Metrics*. 23. https://doi.org/10.1109/METRICS.2005.24
[41] 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 (2020), 179:1–179:30. https://doi.org/10.5281/zenodo.4040341
[42] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.5281/zenodo.5652640
[43] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. 2022. BugBuilder: An Automated Approach to Building Bug Repository. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1443–1463. https://doi.org/10.1109/TSE.2022.3177713
[44] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *35th International Conference on Software Engineering*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613
[45] Myeongsoo Kim, Santosh Pande, and Alessandro Orso. 2024. Improving Program Debloating with 1-DU Chain Minimality. In *IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 384–385. https://doi.org/10.1145/3639478.3643518
[46] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 239–250. https://doi.org/10.1145/3293882.3330553
[47] Yoonseok Ko and Hakjoo Oh. 2023. Learning to Boost Disjunctive Static Bug-Finders. In *45th IEEE/ACM International Conference on Software Engineering*. 1097–1109. https://doi.org/10.1109/ICSE48619.2023.00099
[48] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Software*

*Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[49] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 921–933. https://doi.org/10.1145/3611643.3616262

[50] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[51] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1145/3126908.3126958

[52] Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 359–373. https://doi.org/10.1145/3296979.3192390

[53] Bozhen Liu and Jeff Huang. 2022. SHARP: fast incremental context-sensitive pointer analysis for Java. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–28. https://doi.org/10.1145/3527332

[54] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When threads meet events: efficient and precise static race detection with origins. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 725–739. https://doi.org/10.1145/3453483.3454073

[55] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. 2023. A Comprehensive Study on Quality Assurance Tools for Java. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 285–297. https://doi.org/10.1145/3597926.3598056

[56] Benjamin Livshits. 2005. Stanford securibench. *Online: http://suif.stanford.edu/livshits/securibench* (2005).

[57] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[58] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering* 27 (2022), 1–41. https://doi.org/10.1007/s10664-021-10013-5

[59] Marc Miltenberger, Steven Arzt, Philipp Holzinger, and Julius Näumann. 2023. Benchmarking the Benchmarks. In *2023 ACM Asia Conference on Computer and Communications Security*. 387–400. https://doi.org/10.1145/3579856.3582830

[60] Austin Mordahl. 2023. Automatic Testing and Benchmarking for Configurable Static Analysis Tools. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1532–1536. https://doi.org/10.1145/3597926.3605232

[61] Austin Mordahl and Shiyi Wei. 2021. The impact of tool configuration spaces on the evaluation of configurable taint analysis for android. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 466–477. https://doi.org/10.1145/3460319.3464823

[62] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *45th IEEE/ACM International Conference on Software Engineering*. 550–562. https://doi.org/10.1109/ICSE48619.2023.00056

[63] Peter W. O'Hearn. [n.d.]. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL ([n. d.]). https://doi.org/10.1145/3371078

[64] Otter.ai. 2025. Otter.ai - Voice Meeting Notes & Real-time Transcription. https://otter.ai/.

[65] Reza M Parizi, Kai Qian, Hossain Shahriar, Fan Wu, and Lixin Tao. 2018. Benchmark requirements for assessing software security vulnerability testing tools. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 825–826. https://doi.org/10.1109/COMPSAC.2018.00139

[66] Ivan Pashchenko. 2017. FOSS version differentiation as a benchmark for static analysis security testing tools. In *11th Joint Meeting on Foundations of Software Engineering*. 1056–1058. https://doi.org/10.1145/3106237.3121276

[67] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 331–341. https://doi.org/10.1145/3236024.3236029

[68] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–47. https://doi.org/10.1145/3314221.3314637

[69] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–186. https://doi.org/10.1145/3213846.3213873

[70] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jan Vitek, Haibo

Lin, and Frank Tip (Eds.). 335–346. https://doi.org/10.1145/2254064.2254104
[71] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Software Eng.* 22, 8 (1996), 529–551. https://doi.org/10.1109/32.536955
[72] 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 (2018), 58–66. https://doi.org/10.1145/3188720
[73] Michael Schlichtig, Anna-Katharina Wickert, Stefan Krüger, Eric Bodden, and Mira Mezini. 2022. CamBench - Cryptographic API Misuse Detection Tool Benchmark Suite. *CoRR* abs/2204.06447 (2022).
[74] Robert C Seacord. 2013. *Secure Coding in C and C++*. Addison-Wesley.
[75] Semmle. 2025. CodeQL: Variant Analysis Engine For Product Security. https://semmle.com/codeql.
[76] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming*, Vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22
[77] Per Erik Strandberg. 2019. Ethical Interviews in Software Engineering. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–11. https://doi.org/10.1109/ESEM.2019.8870192
[78] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509
[79] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 81–93. https://doi.org/10.1145/3368826.3377927
[80] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 51. https://doi.org/10.1145/2393596.2393656
[81] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *41st International Conference on Software Engineering*. 339–349. https://doi.org/10.1109/ICSE.2019.00048
[82] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering*. 38–49. https://doi.org/10.1109/SANER.2018.8330195
[83] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.* 21, 3 (2018), 14:1–14:32. https://doi.org/10.1145/3183575
[84] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying developers' adoption of security tools. In *10th Joint Meeting on Foundations of Software Engineering*. 260–271. https://doi.org/10.1145/2786805.2786816
[85] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. 2013. Effective dynamic detection of alias analysis errors. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 279–289. https://doi.org/10.1145/2491411.2491439
[86] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* (1999). https://doi.org/10.1145/318774.318946
[87] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. 2014. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*. 160–170. https://doi.org/10.1145/2610384.2610392
[88] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 763–773. https://doi.org/10.1145/3338906.3338932
[89] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2022. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Trans. Software Eng.* 48, 10 (2022), 4014–4040. https://doi.org/10.1109/TSE.2021.3109563
[90] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *23rd International Symposium on Principles and Practice of Declarative Programming*. 1–16. https://doi.org/10.1145/3479394.3479415
[91] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the Reproducibility of Software Defect Datasets. In *45th IEEE/ACM International Conference on Software Engineering*. 2324–2335. https://doi.org/10.1109/ICSE48619.2023.00195

Received 2024-09-13; accepted 2025-04-01