Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities

Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, *Member, IEEE,* Lin Chen, Wei Dong, and Yuming Zhou

***Abstract*—Static analysis (SA) tools can generate useful static warnings to reveal the problematic code snippets in a software system without dynamically executing the corresponding source code. In the literature, static warnings are of paramount importance because they can easily indicate specific types of software defects in the early stage of a software development process, which accordingly reduces the maintenance costs by a substantial margin. Unfortunately, due to the conservative approximations of such SA tools, a large number of false positive (FP for short) warnings (i.e., they do not indicate real bugs) are generated, making these tools less effective. During the past two decades, therefore, many false positive mitigation (FPM for short) approaches have been proposed so that more accurate and critical warnings can be delivered to developers. This paper offers a detailed survey of research achievements on the topic of FPM. Given the collected 130 surveyed papers, we conduct a comprehensive investigation from five different perspectives. First, we reveal the research trends of this field. Second, we classify the existing FPM approaches into five different types and then present the concrete research progress. Third, we analyze the evaluation system applied to examine the performance of the proposed approaches in terms of studied SA tools, evaluation scenarios, performance indicators, and collected datasets, respectively. Fourth, we summarize the four types of empirical studies relating to SA warnings to exploit the insightful findings that are helpful to reduce FP warnings. Finally, we sum up 10 challenges unresolved in the literature from the aspects of systematicness, effectiveness, completeness, and practicability and outline possible research opportunities based on three emerging techniques in the future.**

***Index Terms*—Static warnings, false positives, defects, static analysis tools, software quality assurance.**

Manuscript received 14 May 2022; revised 2 July 2023; accepted 29 October 2023; Date of publication 2 November 2023; date of current version 12 December 2023. This work was supported in part by the Natural Science Foundation of China under Grants 62172205, 62072194, 62172202, 62272221, and 62032019, in part by the National Key Research and Development Program of China under Grant 2022YFB4501903, and in part by the Natural Science Foundation of Jiangsu Province under Grant SBK2023022696. Recommended for acceptance by W. Visser. *(Corresponding author: Yuming Zhou.)*

Zhaoqiang Guo, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, and Yuming Zhou are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China (e-mail: gzq@smail.nju.edu.cn; shiranliu@smail.nju.edu.cn; xryu@smail.nju.edu.cn; DZ20330011@smail.nju.edu.cn; yangyibiao@nju.edu.cn; yanhuili@nju.edu.cn; lchen@nju.edu.cn; zhouyuming@nju.edu.cn).

Tingting Tan was with Beijing Bytedance Network Technology Company Ltd., Beijing 100086, China. She is now with Alibaba Group Holding Limited, Hangzhou, Zhejiang 310023, China. (e-mail: tantingting.ttt@taobao.com).

Wei Dong is with the College of Computer Science, National University of Defense Technology, Changsha, Hunan 211101, China (e-mail: wdong@nudt.edu.cn).

Digital Object Identifier 10.1109/TSE.2023.3329667

I. Introduction

STATIC analysis (SA) tools have been widely used in software quality assurance (SQA) activities to detect the potential problematic code snippets [99], [133], [136] of both commercial and open source software (OSS) systems. The reasons are as follows. First, plenty of software quality issues, such as coding defects [143], vulnerabilities [115], and code style violations [87], can be detected by SA tools. Therefore, various SQA resources (e.g., human costs and test suites) can be assigned more effectively to improve software quality based on the detection results of SA tools. Second, SA tools provide a simple and convenient way to detect quality issues in a target program without a process of dynamical execution. Instead, these tools retrieve a set of pre-defined common bug patterns that are summarized by software experts, and then report the information of all problematic code captured by the bug patterns. Notably, most SA tools are designed as flexible and lightweight tools (e.g., `FindBugs` [12], and `PMD` [41]), which can be used in the form of either independent command line tools or built-in components of some popular IDEs such as Eclipse and IntelliJ IDEA. As a result, developers could leverage SA tools to extract a set of warnings from the target software project and then manually review, understand, and fix them later [147].

Unfortunately, despite the usefulness of SA tools in maintaining software quality, there has always been a serious problem with them that greatly affects their user experience. Specifically, in order to not miss any actual defects, SA tools often are based on the abstraction and conservative approximations of the source code semantics [63], [90]. As a consequence, a large number of warnings are actually false positives (FP, i.e., the warnings that do not indicate the real bugs) [7], [22], [101]. For instance, Kremenek et al. [1] pointed out that the rate of FP warnings can easily reach 30-100%. In fact, even if the FP warnings may indicate some improper coding practices (e.g., a variable is defined but not used), they are unpopular with developers because they do not necessarily result in software defects [9]. Faced with such huge amount of FP warnings, the usefulness of SA tools is greatly limited. In terms of practical use, developers usually feel that the FP warnings are unimportant or inconsequential to the functionality of the target program [45]. However, they are obliged to waste much effort to eliminate FP warnings and identify the valuable warnings that can indicate

TABLE I
COMPARISON WITH PRIOR SURVEYS ON FPM STUDIES

Survey	Title	Time Span	#P	◈ Difference and ✚Novelty of Our Paper
[120]	A systematic literature review of actionable[1] alert identification techniques for automated static code analysis.	1998∼2009	21	◈ More FPM surveyed papers ✚ Analysis for studies on FPM issues. ✚ Comprehensive challenges and opportunities
[121]	Survey of approaches for handling static analysis alarms.	2002∼2016	79	◈ More FPM surveyed papers ✚ Summary for different SA tools ✚ Comparison on evaluation systems ✚ Analysis for studies on FPM issues ✚ Comprehensive challenges and opportunities
[166]	Survey of approaches for postprocessing of static analysis alarms. (Extended version of Ref [121])	2003∼2019	130	◈ New FPM papers published in 2020 and 2021 ✚ The novelties are same as that of [121]

real bugs. Under such circumstances, information overload [7] and high FP rate lead developers to lose patience. Finally, they tend to ignore SA warnings or even discard SA tools in practice [29].

During the past two decades, researchers have made significant efforts on study of FPM (i.e., false positive mitigation) to make SA tools more practical by mitigating as many FP warnings as possible for developers. To this end, on the one hand, researchers have proposed a large number of automatic FP warnings mitigation approaches based on different categories of techniques such as statistical analysis, machine learning, and/or data mining. Most of these approaches belong to post-processing technologies, which first evaluate the suspicious score of each warning and then decide whether it is a FP instance or not. On the other hand, some researchers have conducted many empirical studies to investigate the characteristics of SA warnings from multiple perspectives such as SA tools, developers, and warning instances. The insightful findings of these studies help researchers and developers understand SA warnings more comprehensively and provide guiding information on identifying and excluding FP warnings.

Due to the diversity of SA tools and the inconsistency of research objectives in different studies, it is not easy to grasp the current research status quo of FPM studies. Therefore, this paper aims to present a comprehensive survey on the progress, challenges, and opportunities of studies in mitigating FP warnings. To this end, we collect, review, and analyze the related papers published in the top conferences and journals since 2002. In summary, this paper mainly makes the following four contributions:

1) **Revealing the research trends of the surveyed papers.** We analyze the research trends from the perspective of publication years, venues, types of main contributions, and SA tools, respectively.
2) **Presenting the surveyed FPM approaches.** We divide existing FPM approaches into five categories (e.g., machine learning) based on the technical details of different approaches and then introduce each approach in turn.
3) **Analyzing the evaluation system used for FPM approaches.** We introduce the evaluation system for FPM approaches from the aspects of studied SA tools, experimental scenarios, performance indicators, and collected datasets, respectively.
4) **Summarizing the FPM empirical studies.** We conclude and present four types of empirical studies (e.g., investigating SA warnings) that are highly correlated with the topic of FPM, which are conducted to understand the characteristics, relevancy, or influences of warnings.
5) **Exposing the unsolved challenges and possible opportunities.** From the perspectives of systematicness, effectiveness, completeness, and practicability, we sum up 10 challenges that have not been resolved in existing studies. Meanwhile, we outline possible research opportunities in the future by considering three emerging techniques, e.g., natural language processing (NLP).

It should be pointed out that, as an important research topic, there have been three previous works (i.e., Heckman et al. [120] and Muske et al. [121], [166]) dedicated to the categorization and review of SA warning processing techniques in 2011 and 2019, respectively. In particular, Ref [166] is an extended version of Ref [121]. In comparison to their work, our research exhibits significant divergence in terms of the papers studied and the scope of analysis. Table I summarizes the previous surveys (i.e., [120], [121], [166]) and points out how their scope differs from the current survey. First, our survey investigates 130 related papers published between 2002 and 2021, which has a longer time span and more papers than Ref [120] (i.e., 21 papers between 1998 and 2009) and Ref [121] (i.e., 79 papers between 2002 and 2016). Many new papers (i.e., 55) published in recent years (after 2016) are included so that the state-of-the-art research focus can be tracked. Compared the extended version, there are still 19 new papers published after 2019 included in our survey. Note that, due to different inclusion and exclusion criteria, the surveyed papers we selected before 2019 are not exactly the same as in prior works (see Section III-B for details).

Second, this survey conducts a comprehensive comparison and analysis on the progress of FPM approaches. Specifically,

[1]Note that many papers utilize the term 'actionable' to refer to 'true positive' warnings. However, it is important to note that the exact meaning may vary across different research studies. In the following sections, we will use 'actionable' and 'unactionable' in accordance with the context of the respective works.

Heckman et al. [120] aimed to inform evidence-based selection of warning identification techniques. Compared with their work, this work not only introduces the proposed approaches and the evaluation system, but also investigates FPM issue related empirical studies. As for Muske et al.'s works [121], [166], they only surveyed and introduced a wide range of approaches for handling static analysis warnings. However, our work focuses more on FPM approaches as well as the investigation on the relevant empirical studies and evaluation system.

Third, this survey exposes significant challenges that have not been solved under current progress from different perspectives and outlines new opportunities to proposing more effective FPM approaches.

The rest of this article is organized as follows: Section II introduces the background. Section III presents the methodology used for our survey. Section IV analyzes the research trends. Section V presents the existing approaches. Section VI analyzes the evaluation systems. Section VII sums up the empirical studies. Section VIII concludes the challenges and opportunities. Section IX states the threats to validity of this study. Finally, Section X concludes the article.

II. Background

Before we start to conduct this survey, it is necessary to outline the background. We first briefly describe the importance and necessity of software quality assurance. Then, we introduce the workflow of applying static analysis tools. Finally, we present the false positive issue of static analysis warnings.

A. Software Quality Assurance (SQA)

Software quality measures the degree to which the software is consistent with explicitly and implicitly defined multidimensional requirements (e.g., reliability and security), which is an important attribute of a software product. A high-quality software is favored by users since it can work efficiently and continues to generate revenue for users. Unfortunately, software defects will be inevitably introduced by developers in the development process of almost all software products due to incorrect design or mistaken programming. The existence of defects can bring non-negligible threats to software quality and hence lead to severe consequences, including data loss, customer churn, and even personal casualty.

Software quality assurance (SQA), a software engineering (SE for short) practice throughout the entire software development and maintenance life cycle, is hence proposed to ensure the quality of a software product. Specifically, SQA aims to monitor the execution of software production tasks in a manner of independent review, provide developers and managers with information and data that reflect product quality, and replicate SE groups to produce high quality products. To maximize software quality, various SQA activities such as software testing [33], code review [66], or formal specification [127], can be carried out to reveal defects at different stages. For example, developers can conduct static program analysis to identify basic bad programming practices and some security vulnerabilities at the early stage. Meanwhile, they can also conduct developer testing before releasing a product.

According to the experience of software development, the earlier a defect is discovered, the less it will cost to fix it. Following Google's innovation factory [152], developers need increased costs by orders of magnitude to fix uncaught defects with the development progress. Therefore, detecting defects as early as possible is considered a critical task and hence it continues to attract the attention of researchers [101]. In both industry and academia, SA techniques are used for the early detection and elimination of defects [128], [139], [145]. Another strong argument for using SA techniques is that it can find issues that are not found by software testing. In testing, one will only find defects that are triggered by the test suite, i.e., the quality of the product is contingent on the quality of the test suite. Static analysis can theoretically explore all paths through the code, and provide a wider coverage. It can also sometimes uncover issues that may be hard to find using testing, such as memory leaks or security vulnerabilities.

B. Static Analysis Tools

Static analysis (SA) tool, a.k.a. static analyzer [60], [63], [73], [77], [80], [90], [94], [106], static bug-finding tool [13], [14], [41], [124], [133], or static code analysis tool [7], [29], [42], [50], [70], [78], [79], is developed to analyze code and detect potential bugs of software systems via static program analysis technologies such as abstract interpretation [22], dataflow analysis [29], model checking [63], or type checking [13]. To lower the costs of fixing defects and to improve the chances of fixing the defects correctly [32], these tools are often used as a complementary step of software reviews and testing activities in an early stage of software development cycle [7], [50], [125], [131]. In this stage, most of the bugs that have just been introduced into the code are the result of programming oversight (e.g., non-standard coding styles) by developers. In most cases, the defects caused by improper coding specifications have similar code patterns and are frequent, which actually can be easily detected by static program analysis and then removed by developers. Therefore, leveraging SA tools to detect bugs initially is of critical importance for ensuring the quality, security, understanding, and reliability level of a software system [7], [135].

In addition to their important role in detecting bugs, SA tools have gained popularity also because of their usability and convenience. First, SA tools can analyze a target program without actually executing the corresponding source code [129]. Instead, they check the target system based on the corresponding static resources such as source code or document. As a result, users can obtain bug detection results in a low-cost manner. Furthermore, the internal details of SA tools are opaque to the user. This means that users simply need to configure some detection conditions and then they can directly run SA tools to obtain the required warning information. Therefore, SA tools can be quickly applied to production practice without too much learning cost. Last but not least, SA tools often offer multiple ways (e.g., programming API, standalone cmd tool, or IDE built-in plugin) to use them, which further increases the scope of their

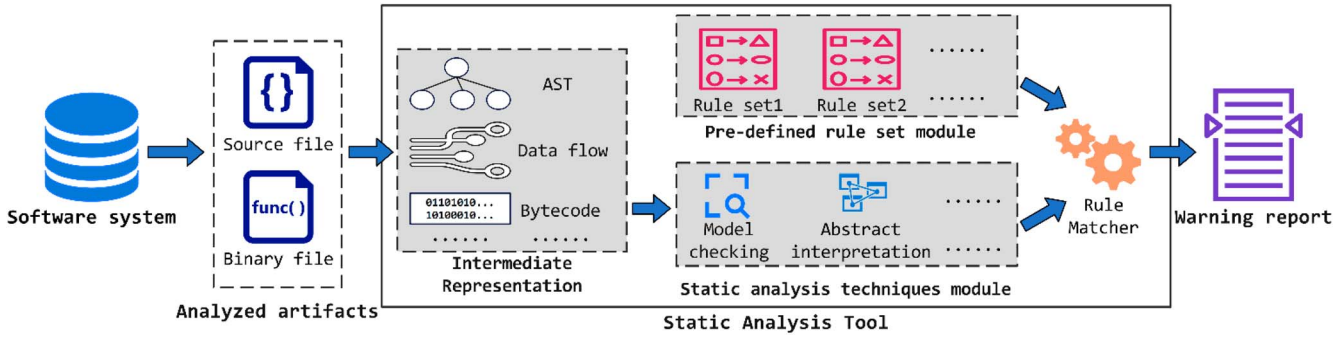

Fig. 1. The general workflow of applying a static analysis (SA) tool.

applications. Therefore, SA tools have been incorporated into the project build process for a growing number of organizations.

The key idea of detecting bugs by SA tools is to look for and report the code locations that violate the programming rules that are usually summarized by tool builders in advance. These rules abstract either good programming standards or frequent programming mistakes. From programming practice, there is a great possibility to introduce bugs directly or indirectly due to the violations of these reasonable and recommended programming practices [132], [134]. Notably, SA tools can only identify the bugs whose corresponding rules have been predefined. However, according to practical application requirements, users can add new custom rules for some SA tools to identify more specific defects. In addition, it is normal that the tool providers of different SA tools manually predefined a set of distinct bug rules to indicate the specific bugs or anomalies. For example, there are 9 different bug categories (e.g., Correctness) with a total of 424 patterns in the latest version of `FindBugs`[2] while 8 different bug categories with 310 patterns in `PMD`. Further, different SA tools may concern the same vital bug categories. The categories of Performance and Security, for instance, have been provided in both `FindBugs` and `PMD`.

Fig. 1 presents a general workflow of applying an SA tool to analyze a target software system. Given a software system, a SA tool first extracts all program entities (e.g., source files or executable binary files) that need to be analyzed. Next, each entity is transformed into multiple forms of intermediate representations (IRs) such as abstract syntax tree or data flow graph, which can clearly reveal the organizational structure and internal relationship of the software. Based on these program structures, a SA tool will leverage various static analysis techniques (e.g., model checking [63]) to check the IRs and match it with the predefined rule set to capture the potential bugs or warnings, Specifically, a SA tool will output a warning instance once it identifies a code snippet that violates any of the pre-defined rules. Finally, all identified warning instances will be reported to users together for later manual code review.

C. Static Analysis Warnings

A static analysis warning, a.k.a. alert [17], [27], [126], alarm [4], [5], finding [81], or violation [115], usually contains rich information to help developers locate the problematic code directly, understand each warning quickly, and fix the corresponding bugs correctly [147]. First, the basic information a warning usually points out is the line-level location of suspicious code, which can construct a link between the warning and its corresponding problematic code. Second, each warning in general has a companion illustrative documentation that describes the cause why the warning was reported, the consequence of the warning, and even the suggested fixes if possible. Based on the documentation, most of the reported issues can be solved with ease. Third, to facilitate the management and efficient processing of warning instances, additional information such as warning category, priority, and confidence are often provided. As a matter of fact, this information provides developers good suggestions on how to reasonably allocate limited code review resources to solve as many serious problems as possible with as little cost as possible.

Fig. 2 shows a real warning example identified by `FindBugs` on the open-source project Apache Ant[3], which demonstrates the usefulness of SA tools. For ease of presentation, we simplified the original source code. In the figure, the left shows the buggy (upper left) and fixed (bottom left) code snippets and the right shows the warning report generated by `FindBugs` that exhibits the detailed information for the found warning. On the buggy version, `FindBugs` detects a warning called NP_NULL_ON_SOME_PATH_EXCEPTION, which indicates that there is a possible null pointer dereference in the method doReportSystemProperties on its exception path (i.e., line 225). More detailed information can be acquired from the attribute of warning description. On the fixed version, the developer added an exception capture code before the buggy code lines to deal with the case that a null pointer exception is triggered.

D. False Positive Warnings Mitigation

Unfortunately, SA tools are often limited by a significant drawback: they frequently generate a large number of false positive (FP) warnings, which are warnings that do not indicate actual bugs [7], [22], [101]. This issue arises because these tools rely on abstractions and conservative approximations of the source code semantics [63], [90], thereby impacting the overall

[2]`FindBugs` has been superseded by `SpotBugs`.

[3]https://github.com/apache/ant/blame/master/src/main/org/apache/tools/ant/Diagnostics.java

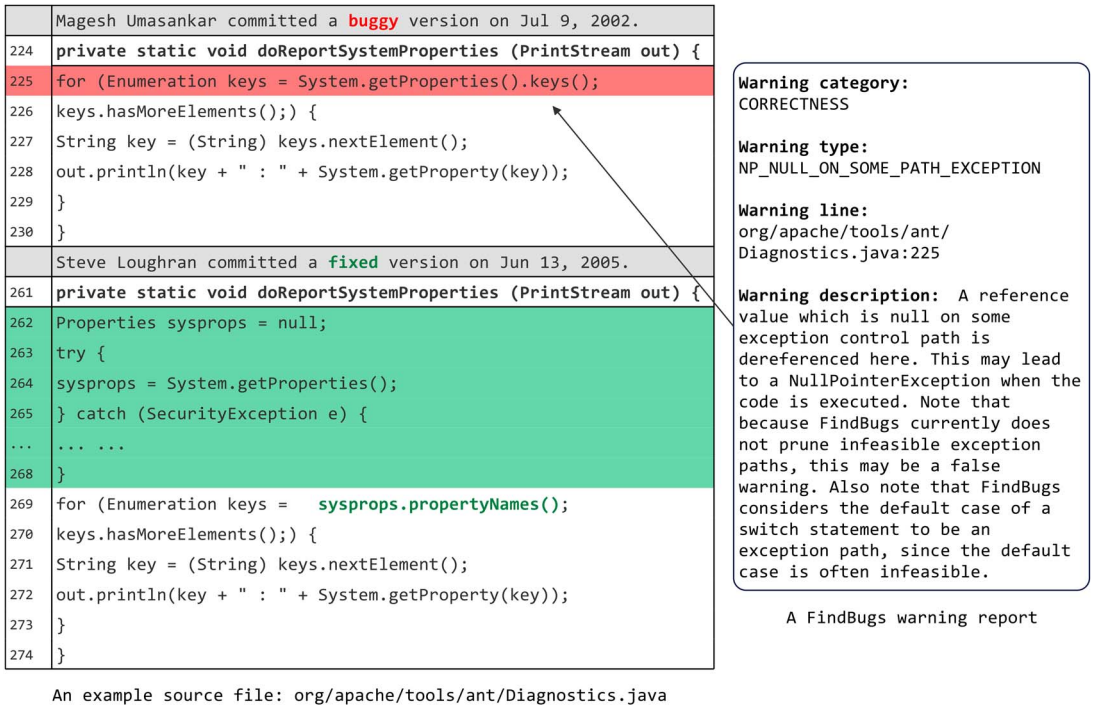

Fig. 2. A real example of SA warnings detected by FindBugs.

user experience. Notably, Kremenek et al. [1] highlighted that the rate of FP warnings can easily reach 30-100%.

Over the past two decades, significant efforts [7], [13], [22], [24], [31], [63] have been devoted to mitigating false positive (FP) warnings. Various approaches have been developed, which can be broadly categorized into five categories.

(1) Statistical probability approaches: These approaches analyze the structure and characteristics of the target code to predict the likelihood that a warning is a false positive. Warnings are ranked based on their likelihood and presented to developers accordingly.
(2) Static program analysis approaches: These approaches employ more accurate static analysis (SA) operations, such as bound model checking [22], to reduce the number of FP warnings produced by SA tools.
(3) Dynamic program testing approaches: These approaches combine SA tools with testing tools to mitigate FP warnings. Test cases are executed to identify real bugs and distinguish them from false positives.
(4) Machine learning approaches: These approaches extract warning features from source code or warning reports and build prediction models using training datasets. These models estimate the probability of a test warning being a false positive instance.
(5) Clustering approaches: These approaches group similar warnings together, based on the observation that they are either all true defects or all false positives. Developers are then advised to review a dominant warning within each group, reducing the overall review effort.

The above FP mitigation approaches have made significant contributions to the literature. Further categorization and detailed progress in each approach are discussed in Section V.

III. Methodology

This section sheds light on the methodology of our survey on studies in mitigating FP warnings of static analysis tools. First, we list all research questions the survey would like to answer. Second, we introduce the bibliography retrieval and selection process. Third, we describe the data items need to be extracted for each research question.

A. Research Questions

The goal of this survey is to provide details and evidence for the status quo of false positive static warning mitigation studies by retrieving, categorizing, and reporting the available FPM papers. To this end, we would like to propose and answer the following research questions (RQs):

RQ1: What are the research trends in the FPM papers?
RQ1.1: What is the distribution of publication years?
RQ1.2: What is the distribution of publication venues?
RQ1.3: What is the distribution of types of contributions?
RQ1.4: What is the distribution of interested SA tools?
RQ2: Which FPM approaches have been proposed?
RQ2.1: Which types can the FPM approaches be divided into?
RQ2.2: How is each type of FPM approach implemented?
RQ2.3: What lessons can be learned from the above progress?
RQ3: How the current FPM approaches are evaluated?
RQ3.1: What SA tools are considered as the research objects?
RQ3.2: What evaluation scenarios are leveraged?
RQ3.3: Which performance indicators are selected?
RQ3.4: Which benchmarks are collected and applied?
RQ3.5: What lessons can be learned from evaluation system?

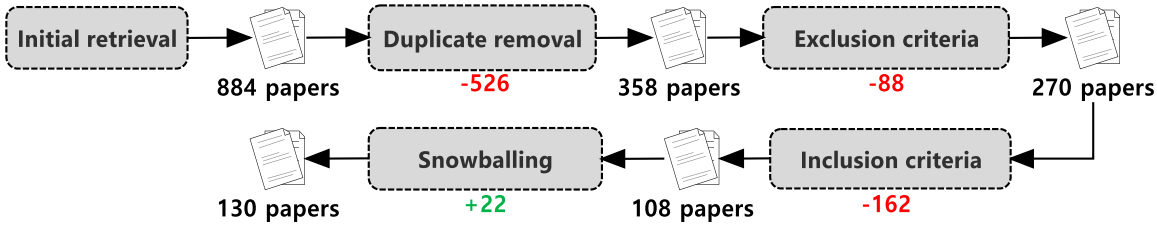

Fig. 3. The process of bibliography retrieval and selection.

RQ4: Which empirical studies on FPM are conducted?
RQ4.1: Which types can empirical studies be divided into?
RQ4.2: How is each type of empirical studies conducted?
RQ5: What are the unsolved challenges and potential opportunities for FPM?
RQ5.1: Which challenges remain unsolved in the literature?
RQ5.2: What are potential opportunities for future studies?

The relationships among each research question are as follows. RQ1 is a demographic question that provides an overview of the studied FPM papers. RQ2-RQ4 state the complete research progress of the studied FPM papers, which is one keynote of the survey. Specifically, based on the different categories of main contributions, RQ2 and RQ4 introduce the taxonomy and analysis for the novel FPM approaches and the FPM issue studies, respectively. Note that, RQ3 makes comparisons for the proposed FPM approaches from the perspective of evaluation system. Finally, based on the progress stated by above-mentioned RQ2-RQ4, RQ5 summarizes the challenges left unsolved in the existing works and the potential opportunities to designing more FPM approaches in future work, which is another keynote of the survey.

B. Bibliography Retrieval and Selection

We follow an explicit bibliography retrieval process (similar to [151]) to collect the relevant papers so that the process can be reproduced. To this end, four factors are carefully considered: retrieval databases, keywords, time interval, and inclusion & exclusion criteria, respectively.

1) Retrieval Databases, Strings, and Time Interval: **Databases.** To retrieve surveyed papers as comprehensively as possible, we selected the following four popularly used online libraries as our retrieval databases:

- ACM digital library[4]
- IEEE Xplore digital library[5]
- Web of Science[6]
- DBLP Computer Science Bibliography[7]

Strings. As described in Section II, there are some aliases for SA tools and warnings. Therefore, we select 30 ($30 = 6 \times 1 \times 1 \times 5$) combinations of four groups of terms as the retrieval keywords so that more relevant papers can be retrieved. Specifically, the four groups of terms are 1) "*eliminate*", "*reduce*", "*rank*", "*prioritize*", "*mitigate*", and "*classify*", 2) "*false*", 3) "*static*", 4) "*warning*", "*alert*", "*alarm*", "*finding*", and "*violation*". For example, the term "*reduce false static warning*" is one candidate of the selected retrieval strings.

Time interval. The survey aims to review the works in the past two decades. Therefore, the papers were published between the year 2003 and 2022 are considered. Although the previous works [120], [121], [166] have investigated similar themes, for the consistency of the research introduction, some important papers involved in previous surveys will be also included. Nevertheless, note that the newly published papers will be introduced in more detail.

2) Inclusion and Exclusion Criteria: Note that, a large number of papers with uneven quality can be searched under the given retrieval conditions. Therefore, the following criteria are designed to select the qualified papers and filter out the irrelevant ones.

1) Inclusion criteria:
✓ Studies must be written in English.
✓ Studies must be related to mitigating false positive SA warnings.
✓ Studies whose research objective is to improve some specific (open-source or commercial) SA tools instead of only introducing new SA tools or refining some kinds of SA methodology (e.g., taint analysis).

2) Exclusion criteria:
✗ Books, book chapters, or technical reports (most of which have been published as articles) are dropped.
✗ If a conference paper has an extended journal version, only the journal version is included.
✗ The length of paper less than 4 pages.

3) Retrieval and Selection Process: The process of bibliography retrieval and selection is conducted as follows (shown in Fig. 3):

1) We started an initial retrieval based on the advance function of the four databases to collect all surveyed papers whose meta data (e.g., title and abstract) contain the retrieval strings. Finally, we collected 884 papers in total.
2) A common phenomenon is that a same paper is included in different online databases, which leads to a huge amount in our preliminary retrieval results. Therefore, we manually removed all duplicated papers appeared in multiple databases. At the end of this step, 358 papers were retained.
3) We applied the exclusion criteria to eliminate the irrelevant or inappropriate papers. We conducted a manually review process to exclude them from the candidate set of papers. As a result, 88 papers were excluded.
4) We applied the inclusion criteria to select the required papers. Specifically, we read the abstract of each paper

[4]https://www.acm.org/
[5]https://ieeexplore.ieee.org/Xplore/home.jsp
[6]https://www.webofscience.com/wos/alldb/basic-search
[7]https://dblp.uni-trier.de/

TABLE II
DATA COLLECTION FOR EACH RESEARCH QUESTION

RQs	Extracted Data Items	Explanation
RQ1	Publication year/venue	Basic information of each primary FPM paper (title, authors)
RQ1	Contribution	The category of main contribution in each primary paper (e.g., algorithm or empirical study)
RQ1	Static analysis tool	The analyzed SA tools of each primary paper (e.g., `FindBugs`, `PMD`, etc.)
RQ2	Approach category	The classification for the proposed FPM approaches in the surveyed papers
RQ2	Theory	The heuristics or rationales for each category of approaches
RQ3	Static analysis tool	Which SA tools are studied in their work
RQ3	Performance indicator	What indicators are selected to evaluate the performance (e.g., Accuracy)
RQ3	Evaluation scenario	How the scenarios are designed to conduct experiments (e.g., cross-validation)
RQ3	Collected dataset	Which dataset are collected to conduct experiments
RQ4	Study category	The classification for the FPM issue studies in the surveyed papers
RQ4	Finding	The summarized useful findings in the surveyed papers
RQ5	Challenge	Unsolved challenges in the surveyed papers
RQ5	Opportunity	Potential opportunities in future studies

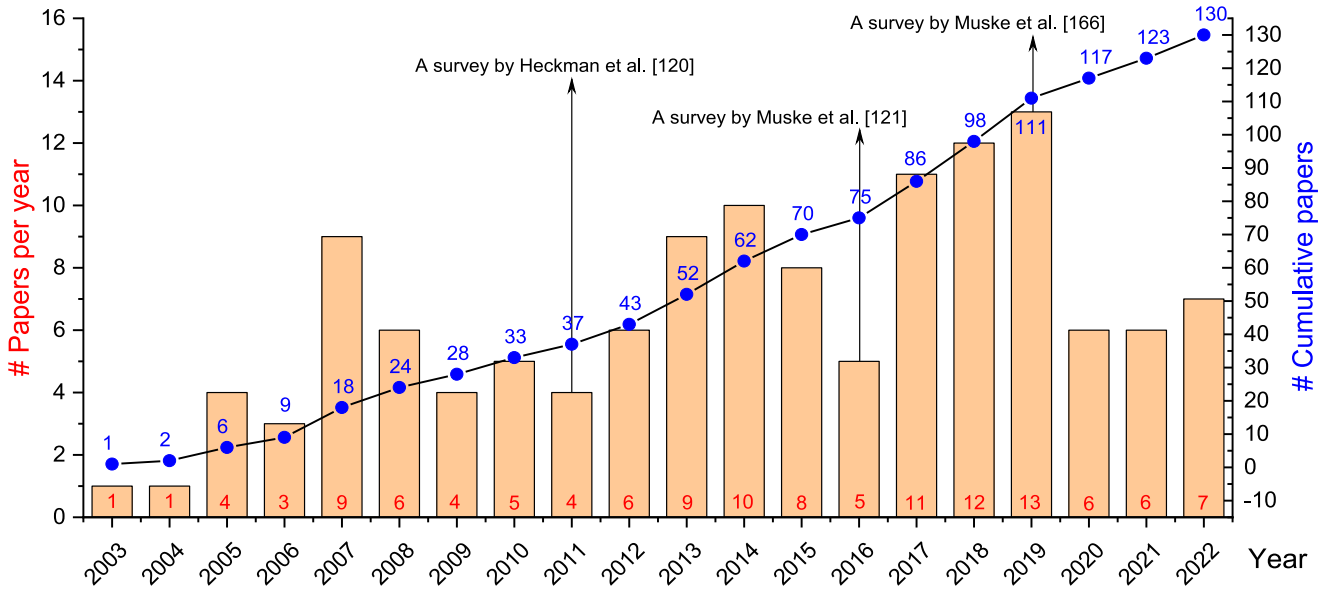

Fig. 4. Distributions of studied FPM papers in terms of publication years.

initially to judge whether the paper fits the research theme of our survey. After this step, 108 studied papers met our inclusion criteria and were thus considered.

5) The final step is snowballing and manual inclusion of selected papers, which helps to ensure we are gathering other related works that were not included in our main retrieval process. More specifically, we manually checked the references of the selected papers to avoid missing some relevant papers whose titles do not contain the pre-selected keywords. According to our previous expertise in this research area, we manually added 22 related works that were missed by our systematic process. As a result, we finally gather 130 relevant papers.

C. Data Items Extraction

After retrieving the surveyed papers to be analyzed, we extract and record the essential data items, which can assist us to answer the research questions addressed by the study. Table II reports the detailed alignment relations between the extracted data and the research questions. The first column is the abbreviation for each research question. The second and third columns denotes the data items as well as the corresponding explanation. To ensure the quality of data extraction, the whole process is divided into two stages: data extraction and data revision. At the previous stage, the first two authors are responsible for the data extraction. At the next stage, the remaining authors are responsible for the revision.

IV. RQ1: WHAT ARE THE RESEARCH TRENDS IN THE FPM PAPERS?

In this research question, we analyze the basic demographic information of the 130 studied FPM papers in terms of publication years, publication venues, main types of contributions, and studied static analysis tools, respectively. For ease of other researchers, we make the list of studied FPM papers publicly available [122].

A. RQ1.1: What Is the Distribution of Publication Years?

Fig. 4 reports the publication years of studied FPM papers between 2003 and 2022. Specifically, the histogram marked by

TABLE III
DISTRIBUTION OF STUDIED SURVEYED PAPERS IN TERMS OF JOURNALS (J) AND CONFERENCES (C)

Cat.	Acronym	Publication Venue (Bold: Main Keyword)	Num	References
J	TSE	IEEE Transactions on **Software Engineering**	2	[115], [178]
	EMSE	Empirical **Software Engineering**	2	[111], [118]
	JSS	Journal of Systems and Software	2	[95], [113]
	Others	-	12	[11], [19], [20], [26], [30], [40], [68], [91], [109], [112], [117], [172]
C	ICSE	International Conference on **Software Engineering**	11	[10], [24], [32], [34], [46], [73], [76], [107], [119], [173], [174]
	ESEC/FSE	ACM SIGSOFT Symposium on the Foundation of **Software Engineering**/European **Software Engineering** Conference	5	[2], [13], [64], [80], [89]
	SANER	International Conference on Software Analysis, Evolution, and Reengineering	5	[66], [72], [74], [92], [98]
	ICST	IEEE International Conference on Software Testing, Verification and Validation	5	[27], [28], [35], [100], [101]
	COMPSAC	International Computer Software and Applications Conference	4	[6], [33], [41], [82]
	MSR	Mining Software Repositories	4	[14], [53], [83], [97]
	APSEC	Asia-Pacific **Software Engineering** Conference	4	[39], [51], [60], [69]
	ISSRE	International Symposium on **Software** Reliability **Engineering**	4	[49], [55], [65], [71]
	SCAM	IEEE International Working Conference on Source Code Analysis and Manipulation	6	[7], [48], [110], [168], [176], [177]
	ASE	International Conference on Automated **Software Engineering**	3	[22], [31], [87]
	SAS	International Static Analysis Symposium	3	[1], [3], [4]
	ESEM	International Symposium on Empirical **Software Engineering** and Measurement	3	[16], [21], [93]
	APLAS	Asian Symposium on Programming Languages and Systems	3	[5], [59], [103]
	ICSME	International Conference on Software Maintenance and Evolution	2	[56], [57]
	ISSTA	International Symposium on Software Testing and Analysis	2	[47], [88]
	LADC	Latin-American Symposium on Dependable Computing	2	[70], [105]
	PASTE	ACMSIGPLAN-SIGSOFT Workshop on Program Analysis for **Software** Tools and **Engineering**	3	[9], [12], [167]
	WCRE	Working Conference on Reverse Engineering	2	[37], [44]
	SAFECOMP	International Conference on Computer Safety, Reliability, and Security	2	[18], [96]
	Others	-	39	[8], [15], [23], [25], [29], [36], [38], [42], [43], [45], [50], [52], [54], [58], [61], [62], [63], [67], [75], [77], [78], [79], [81], [84], [85], [86], [90], [94], [99], [102], [104], [106], [108], [114], [116], [169], [170], [171], [175]
Total	-		118	

red number reports the individual number of published papers per year and line chart marked by blue number shows the cumulative number of published papers. According to the publication times (i.e., 2011 [120] and 2016 [121]) of the prior two surveys, the surveyed papers in the last two decades can be divided into three different research stages (i.e., 2003-2011, 2012-2016, 2017-now). In each stage, the number of published papers each year shows the characteristics of rising first and then falling (i.e., ∩-Curve). In addition, the years of 2007, 2014, and 2019 are the corresponding peak years for the three stages, respectively. On the whole, the number of papers per year in recent years (since 2012) is higher than that in previous years. In terms of the line chart, a total of 46 new FPM papers were published in the recent five years (since 2017), which accounts for about 40% of all surveyed papers. This indicates that the study on the mitigation for false positive static warning has received more attention in recent years.

Overall, the number of FPM papers published each year exhibits a rising trend in fluctuation. Additionally, there is a possible correlation between the survey type of works and the promotion of research attention.

B. *RQ1.2: What Is the Distribution of Publication Venues?*

Table III lists the distribution of studied papers in terms of publication venues. Specifically, the first column reports the category each publication venue belongs to. In particular, the term "J" and "C" refer to journals and conference, respectively. The second and third columns report the acronym and full name of each publication venue, respectively. The last two columns report the number and the corresponding references of the studied papers published in each publication venue. For each category, we rank publication venues in descending order from the most to the least numbers. Note that only the publication venues

TABLE IV
THE DEFINITION AND DISTRIBUTION OF MAIN CONTRIBUTIONS IN PRIMARY STUDIES

Cat.	Type	Definition	Number (Ratio)
Novel FPM approach	Methodology	The study proposes a novel approach a.k.a. framework, algorithm, solution to mitigating false positive warnings of the concrete static analysis tool(s). The study describes a detailed methodology with complete approach evaluation experiments, which is usually introduced in a full length or regular paper.	76 (64.4%)
	Tool	The study implements and publishes a new tool or prototype targeting FPM issue, which is usually introduced in a short paper (less than 5 pages).	9 (7.6%)
FPM issue study	Empirical study	The study collects primary data and performs a quantitative and quality analysis on the data to explore interesting findings.	30 (25.5%)
	User study	The study conducts a survey to investigate the attitudes of different users or developers towards FPM issues.	3 (2.5%)

TABLE V
THE STATISTICS FOR ALL STUDIED SA TOOLS BASED ON THE FREQUENCY OF EACH SA TOOL

Classification for SA Tools	Num. of SA Tools	Total Freq. of Each Group of SA Tools
Popular SA tools	12	123
Ordinary SA tools	52	75
Anonymous SA tools	9	9
Total	73	207

that have published not less than two papers are explicitly displayed and the remaining papers are summarized in the rows of "Others".

From Table III, the following three observations can be made. First, the studied FPM papers were published more in conference publications than in journal publications. More specifically, the number of conference papers account for 86% (i.e., 102) of the total number, while only 14% (i.e., 16) of papers published in journals. This may be partly due to the longer and more rigorous review process of journals, which prompts more researchers to publish their studies in conference publications for good timeliness. Second, there are a large number of publication venues (68 in total) suitable for the studied FPM studies. Specifically, the number of suitable journals is 14, which consists of two top (i.e., frequently selected) ones (i.e., JSS and EMSE) and 12 publications that were selected only once. Among all journal papers, the majority of papers (i.e., 12/16) are published in the individual journals. Meanwhile, there are 54 suitable conferences, which are made up of 19 top ones (e.g., ICSE, ESEC/FSE, SANER, ICST, etc.) and 35 individual publications. More than half of the conference papers (i.e., 67/102) were published in the top ones. Note that, ICSE are the most popular conferences that include the highest number of related papers (i.e., 9). Third, although these studied FPM papers are scattered among different publications, Software Engineering is the main keyword (bold font in Table I) of them regarding to the domain of their corresponding publication venues. As can be seen, the keyword exists among eight different top publications. In addition, other keywords such as Software Testing/Analysis/Evolution also occurs multiple times, which indicate the correlations between them and the studied FPM papers.

In summary, the surveyed studies are mainly concentrated around Software Engineering conferences, and hence researchers are recommended to publish their future FPM studies in top conference publications such as ICSE and ESEC/FSE for more attention in the literature.

C. RQ1.3: What Is the Distribution of Types of Contributions?

Following the recent survey [151], we manually identify the main contribution of each studied FPM study and classify them into different categories. Table IV reports the contribution definition and the corresponding distribution (includes the number and percentage of the surveyed papers) of each category. As can be seen, the contributions of the surveyed papers can be summarized by four types: *Methodology*, *Tool*, *Empirical study*, and *User study*. The first two contribution types belong to the category of Novel FPM approach, which accounts for 72.0% of the total number. Specifically, 76 surveyed papers introduced detailed methodology for their novel FPM approaches, and 9 papers only briefly describe their implemented FPM tools. Meanwhile, the main contribution of 28.0% surveyed papers belongs to the category of FPM issue study, consisting of the 31 empirical study papers and 3 user study papers, respectively.

D. RQ1.4: What Is the Distribution of Interested SA Tools?

As is known, the warning instances are generated by various static analysis tools. Table V summarizes the statistics for all studied SA tools based on the frequency of each SA tool. Table VI

TABLE VI
TOP 10 POPULARLY STUDIED SA TOOLS

Cat.	Freq.	Name	Description	Home Page Link
O	49	FindBugs	The tool examines classes or jar files and compares bytecode with a set of defect patterns to find possible problems. It supports Java.	http://findbugs.sourceforge.net/
O	18	PMD	The tool finds common programming bugs like unused variables, empty catch blocks, and unnecessary object creation. It supports Java, JavaScript.	https://pmd.github.io/
C	11	TECA	The full name is TCS Embedded Code Analyzer. It helps developers deliver zero-defect C/C++ code by streamlining defect validation workflows.	https://www.tcs.com/tcs-embedded-code-analyzer
C	8	Coverity	The tool finds and fixes defects in Java, C/C++, C#, etc. open-source project for free. It tests every line of code and potential execution path. The root cause of each defect can be clearly explained, making it easy to fix bugs. In addition, it can be integrated with GitHub and Travis CI.	https://scan.coverity.com/
O	7	JLint	The tool checks inconsistencies and synchronization problems by data flow analysis and a lock graph. It supports Java.	http://jlint.sourceforge.net/
O	7	Checkstyle	The tool helps programmers write Java code that adheres to a coding standard.	https://checkstyle.sourceforge.io/
C	6	Astrée	The tool is a sound C/C++ static code analyzer that proves the absence of run-time errors and invalid con-current behavior in safety-critical software.	https://www.absint.com/astree/index.htm
O/C	5	Cppcheck	The tool provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs. It is designed to be able to analyze codes even if it has non-standard syntax.	http://cppcheck.net/
O/C	5	SonarQube	The tool is an automatic code review tool to detect bugs, vulnerabilities, and code smells. It can integrate with existing workflow to enable continuous code inspection across the project branches and pull requests.	https://www.sonarqube.org/
C	4	Fortify	The tool builds secure software fast. It finds security issues early with the most accurate results and fix at the speed of DevOps and offers strong language support (C/C++ and Java) and IDE integrations that identify flaws in real time.	https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer
C	4	Polyspace	The tool identifies run-time errors, concurrency issues, security vulnerabilities, and other defects in C/C++ embedded software by analyzing software control flow, data flow, and interprocedural behavior.	https://www.mathworks.com/products/polyspace-bug-finder.html
O	4	CSA	The full name is Clang Static Analyzer. It is built on top of Clang and LLVM.	https://clang-analyzer.llvm.org/

reports the top 10 popularly studied SA tools (i.e., tools that were studied in at least four papers), including the tool category ("O": open source or "C": commercial), the studied number, the tool name, the brief description, and its corresponding homepage.

From Table V, a total of 73 different SA tools was investigated 207 times in the 130 studied papers. Note that, all SA tools can be divided into three groups: Popular SA tools (that occurred in at least 4 studied papers), Ordinary SA tools (that occurred in 1∼3 studied papers), and Anonymous SA tools (whose tool names were hidden). Although the 12 popular SA tools only account for 16% of the total number of SA tools, they were investigated 123 times in the studied papers, which accounts for the 59% the total frequency number. According to Table VI, FindBugs and PMD are the most popular SA tools, which has been studied in 49 and 18 studied papers, respectively. In addition to the popular SA tools, 61 other SA tools were also studied, whose list can be acquired by online resources [122] due to space constraints. For example, Liang et al [31] automatically constructed an effective training set to prioritize warnings detected by an open-source SA tool called Lint4j. Another example, Boogerd et al. [7] selected Codesurfer, an open-source SA tool for C/C++, as their research object.

Fig. 5 reports the distributions for the supported programming languages of SA tools appearing in the perspective of the number of SA tools (73 in total) and the frequency number of SA tools (207 in total), respectively. As can be seen, the studied SA tools mainly focus on eight programming languages. Note that, the "Multiple" refers to the SA tools that support multiple languages and the "Anonymous" denotes the SA tools whose names were hidden. Java and C/C++ are the most frequently supported languages by the studied SA tools from multiple perspectives. In particular, the number of SA tools supporting C/C++ language is the largest, accounting for 32.9%, while the SA tools that support Java language have been studied most frequently (36.7%). Meanwhile, the percentages of other supported languages of SA tools are less than one third of the total.

In summary, there is a great imbalance for the studies of SA tools in the surveyed papers. Specifically, SA tools for the languages of Java (e.g., FindBugs) and C/C++ (e.g., TECA) are highly valued by the academic community.

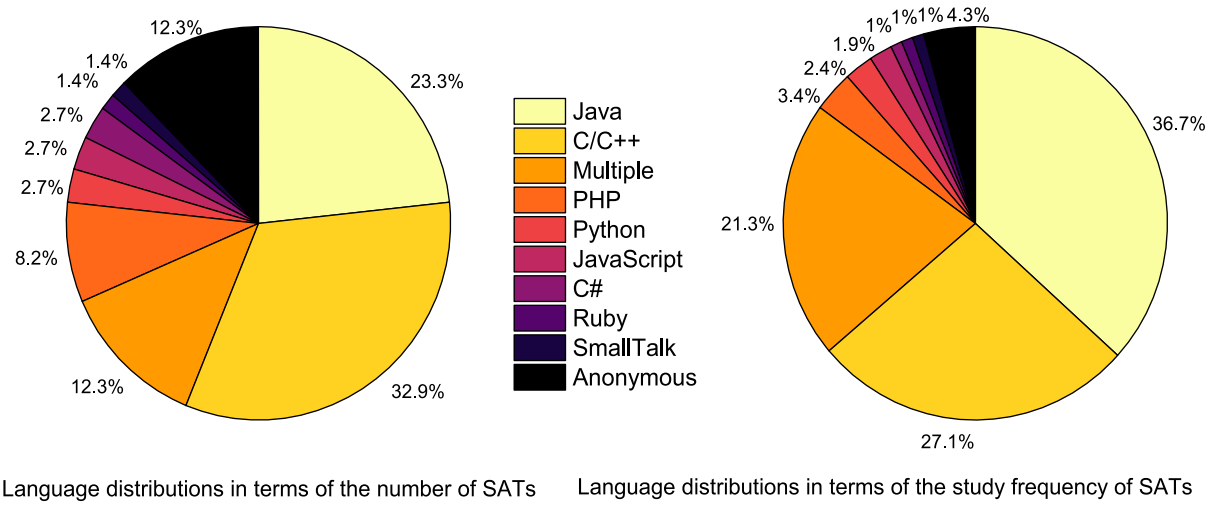

Fig. 5. The distributions for the supported programming languages of SA tools in the studied studies.

Summary of results for RQ1:

1) The number of FPM papers published each year exhibited a rising trend in fluctuation. Additionally, the survey type of works may have a positive effect on promoting of the research attention.
2) Existing FPM papers were mainly concentrated in the conference publications with a keyword related to Software Engineering, and hence the researchers are recommended to publish their future FPM studies in top conference publications such as ICSE and ESEC/FSE for more attention in the literature.
3) Most of the surveyed papers made two types of main contributions: *Methodology* and *Empirical study*. Meanwhile, the remaining few papers provided the contributions of *Tool* and *User study*.
4) A total of 77 distinct SA tools were studied in the surveyed papers, of which 12 SA tools were studied most frequently. Furthermore, the SA tools supporting Java and C/C++ were studied most extensively.

V. RQ2: Which FPM Approaches Have Been Proposed?

This RQ quantifies the distributions of different types of FPM approaches based on different technical details, and states the research progress of each type of approaches.

A. RQ2.1: Which Types Can the FPM Approaches be Divided Into?

1) The Types of FPM Approaches: Based on the rationale and technical details of the studied papers, the existing FPM approaches can be divided into the following five categories (shown in Fig. 6):

1) **Statistical probability.** Statistical probability techniques are the first proposed type that are used to determine the likelihood that a warning is a FP by extracting and analyzing the structures (e.g., execution likelihood [7]), or characteristics (e.g., lifetime of warnings [13]) of a target program. For each warning, the kind of approaches first assign it a suspicious score that measures the probability of indicating a true bug. Then, they rank all warnings from the most to the least suspicious scores and send the ranked list to developers. The purpose of such approaches (or models) is to ensure more TP warnings at the top of the ranked list so that developers do not need to check many FP warnings.
2) **Static program analysis.** As is known, SA tools are built on various static analysis techniques in an over-approximative manner. In this type of approach, more accurate static analysis operations (e.g., bound model checking [22] and abstract interpretation [63]) are performed to further reduce the FP warnings produced by SA tools. Generally, the application languages of static program analysis-based approaches are specific due the grammatical differences between different programming languages.
3) **Dynamic program testing.** Dynamic program testing refers to the approaches that combine the SA tool and the dynamic program testing tool to mitigate FP warnings. Since dynamic program testing tools need to run test cases to find bugs, they are usually precise and hence can be considered as an important supplement for SA tools. This type of approaches usually consists of static phase and dynamic phase: static phase detects potential warnings while the dynamic phase runs test cases to confirm or reject these potential warnings.
4) **Machine learning.** Machine learning (ML) has already been the most popular technique to solve various tasks in the field of software engineering, which belongs to a kind of data mining technique and is also a derived type of statistical probability. Generally, machine learning has a mature methodology. To identify and eliminate FP warnings, researchers first extracted a set of features, a.k.a. factors [24], or attributes [31] (e.g., warning name) from source code or warning reports and then built prediction models to predict the probability of a warning being a FP. In the literature, researchers mainly focused

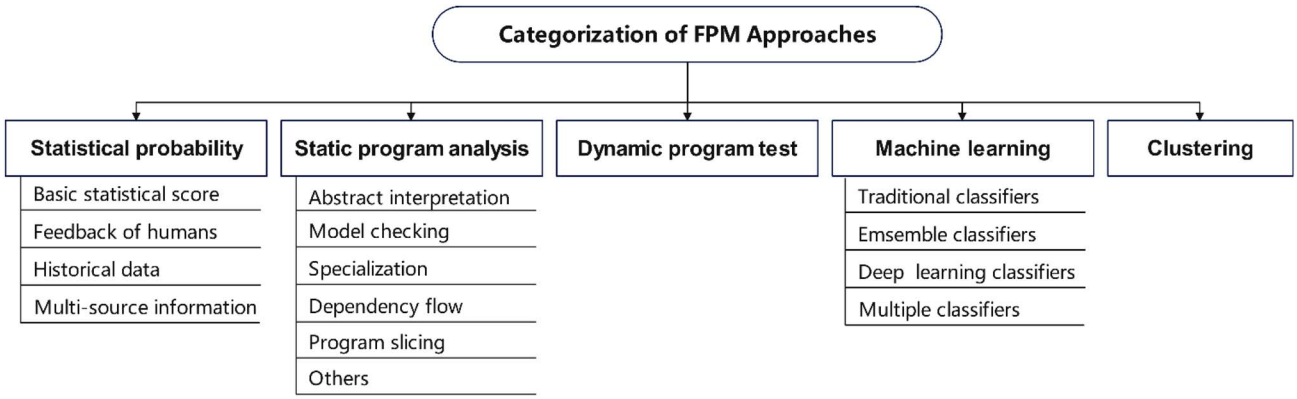

Fig. 6. Categorization of FPM approaches.

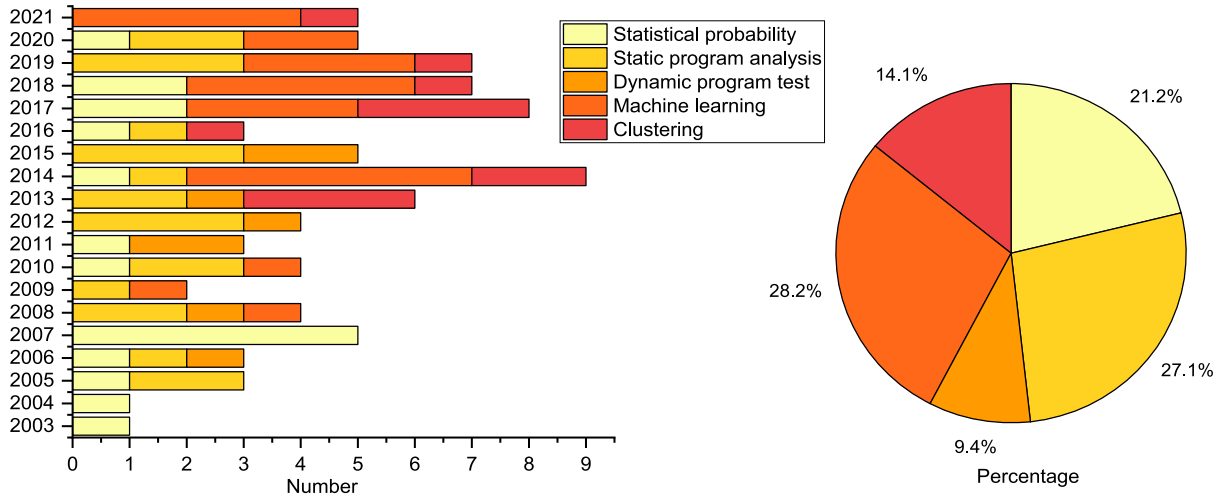

Fig. 7. Distribution of types of FPM approaches in the surveyed papers.

on the extraction of discriminative features and the selection of efficient classifiers for modeling.

5) **Clustering.** Clustering technique is another important kind of data mining method that is used by researchers to reduce the number of FP warnings for SA tools. The intuition behind clustering is that similar warnings (i.e., they are either all true defects or all false positives) can be clustered into the same group. Developers only need to review one dominant warning to know the classification of all other warnings in the same group. Thus, a huge effort can be saved in the process of warning reviewing.

2) The Distribution of Types of Approaches: Fig. 7 reports the distribution overview for five types of FPM approaches. According to the distribution of years, 18 approaches belong to the type of statistical probability, which most were proposed early (i.e., 2003) in the literature. Static program analysis and machine learning are another two primary types that have always been paid attention to by researchers over the years. Dynamic program testing is popular type in previous years (before 2015). Clustering is the latest type proposed in recent years (i.e., 2013). In terms of a single year, 2014 is the most active year, a total of 8 different approaches were proposed in 2014. Meanwhile, there is only one paper in 2003, 2004, and 2016, respectively. Considering the percentage for each type, the primary types of most of FPM approaches are statistical probability, static program analysis, and machine learning, which account for 21.2%, 27.1% and 28.2%, respectively. The remaining two types of approaches account for less than one third of the total. Specifically, the percentages of dynamic program testing, and clustering are 9.4% and 14.1%, respectively.

B. RQ2.2: How Is Each Type of FPM Approach Was Implemented?

This sub-question reviews and summarizes the implementation details for each type of FPM approach.

1) Statistical Probability: A total of 18 FPM approaches ([1], [2], [3], [7], [10*], [11], [13], [14*], [15], [32], [35], [52], [72*], [80], [81], [89], [94], [108]) are based on statistical probability. Note that, the references with a star indicate that the approach was introduced in a short paper. In the literature, statistical probability can be calculated by collecting information from the basic statistical score, feedback of humans, historical data or multiple resources.

Basic statistical score. Kremenek et al. [1] proposed *Z-Ranking* to rank warnings by leveraging z-test statistical analysis, which is to measure the statistical difference between the populations of true bugs and false positives. *Z-Ranking* can find 3-7 times more real bugs than randomized ranking within the top 10% of inspected warnings. Kremenek et al. [2] pointed out that warning reports were highly correlated by code locality and they presented a general probabilistic technique called *Feedback-Rank* for effective warning ranking by exploiting the correlations among the warning reports. *Feedback-rank* shows a factor of 2-8 improvement than randomized ranking. Jung et al. [3] leveraged Bayesian statistical analysis to reduce the inevitable false warnings from a domain-unaware static analyzer `Airac`. Their approach can filter 74.83% of false warnings from Linux kernel sources. Boogerd et al. [7] put forward that the execution likelihood of warning-related code, i.e., the probability that the target code will be executed at least once in an arbitrary run, can be regarded as a contextual measure of warning severity and proposed *ELAN* (Execution Likelihood ANalysis). Wang et al. [94] proposed a semantic based three-phase (i.e., pre-process phase, defect propagation analysis phase, and defect prioritization phase) approach to rank the reported warnings. Their approach evaluated the importance of defect reports by their fatality and ranked defects by their affection to critical functions.

Feedback of humans. Shen et al. [35] proposed a two-stage warning ranking model *EFindBugs*. At the first stage, they used `FindBugs` to detect all warnings existing in the sample project (i.e., JDK). Based on these warnings, they manually classified the true defects and FPs and then assigned an initial approximate defect likelihood for each bug pattern and bug kind. At the second stage, *EFindBugs* updated the defect likelihoods adaptively according to the user's designation. Within the top 60% warnings, the ranking performance (precision and recall) of *EfindBugs* is proven to be better than that of `FindBugs`. Wei et al. [80] leveraged app user reviews to prioritize Android static analysis warnings detected by `Lint`. Their approach *OASIS* extracted the intrinsic links between the `Lint` warnings and the user complaints by combining program analysis and natural language processing techniques. In their study, they considered the strength of these links as the indicator for warnings prioritization. Overview, *OASIS* can achieve over 50% precision@30 and over 44% precision@50 for four Android apps.

Historical data. Kim et al. [13] found that different warning categories had very different lifetimes and proposed a program-specific warning prioritization approach (*HWP*) by exploiting the warning fix experience from software change history. The basic intuition is that the important warnings will be eliminated by fix-changes very soon while the unimportant ones may be not removed for a long time. *HWP* will scan the software change history and assign higher weights for the warning categories that are fixed frequently and early in the past. As a result, HWP can improve warning precision to 17%, 25%, and 67% for three projects: Columba, Lucene, and Scarab.

Multi-source information. Heckman [11] presented *ARM* (i.e., Adaptive Ranking Model) whose probability is calculated based on the feedbacks from developers, historical data from previous releases about ranking factors (i.e., warning type accuracy and code locality), and homogeneity of populations of generated warnings. Their study showed that ARM can provide users with 81% of true warnings after inspecting only top 20% warnings. Xypolytos et al. [81] proposed a conceptual, performance-based warnings prioritizing framework. The framework can combine and rank warnings by cross-validating the performance statistics of warnings from multiple SA tools.

2) Static Program Analysis: A total of 23 FPM approaches ([4], [5], [8], [22], [23], [26], [29], [30], [39*], [40], [41], [43], [49], [59], [63], [64], [65], [71], [96], [103], [104], [109], [110]) are based on static program analysis techniques, which can be divided into abstract interpretation, model checking, dependency flow, specialization, program slicing, and others.

Abstract interpretation. Post et al. [22] leveraged an abstract interpretation tool to generate the potential warnings, and then reduced the FP number by applying a source code bounded model checker (i.e., CBMC). They first fed CBMC with warnings to automatically obtain more information that helped classify warnings. Their study on 77 warnings showed that more than 23% of warnings can be reduced. Once a warning cannot be classified successfully, they manually added more information such as invariants, input constraints, and non-formal requirements and rerun CBMC to obtain the final warning classification. Based on the abstract interpretation, Kim et al. [30] leveraged SMT solvers to filter the FPs of buffer overflow. Cousot et al. [8] argued that the separation of the abstraction into a combination of cooperative abstract domains of `Astree` makes it more extensible, which is an essential feature to deal with FPs. Their approach can reduce about 68% false warnings on average based on the results in three open-source projects.

Model checking. Muske et al. [49] pointed out that model checking was a time-consuming process as it needed to verify all redundant warning related assertions. They leveraged three techniques to speed up the checking process and eliminate FPs. They first used *PI-RAIT* (i.e., Property Independent Redundant Assertion Identification Technique) to group assertions and select a leader assertion for each group, and used a similar *PD-RAIT* (i.e., Property Dependent Redundant Assertion Identification Technique) to group assertions according to the characteristic of a run-time property. Then, they used *NVIT* (i.e., Non-Verifiable Assertion Identification Technique) to identify and skip useless assertion verifications. The approach was useful in reducing the number of assertions being verified by 53%, and false positives elimination time by 60%. Valdiviezo et al. [59] leveraged program analysis and model checking techniques to detect memory leak warnings more precisely. Their approach achieved a precision of 84.5%. Muske et al. [65] designed a new model checking technique to verify the assertion corresponding to each warning so that the true bugs or false positives can be identified. Given the state space explosion of model checking, they proposed the concept of CNV (i.e., complete-range non-deterministic values) variable and used it to identify and avoid the redundant model checking calls during the elimination of FPs. Their approach can avoid 49.49% of the total model checking calls, and it reduced the false positives elimination time by 39.28%. Rungta et al. [23] generated a small sequence of

program locations that represent points of interest in checking the feasibility of a particular warning. Afterwards, they used a meta heuristic to automatically control scheduling decisions in a model checker to guide the program along the input sequence to test the feasibility of the warning.

Specialization. Chen et al. [43] proposed to use thread specialization to prune the FPs of static data-race warnings. They first transformed a program to a simplified version based on the number of fixed threads found by thread specialization. Then, the range of addresses accessed by each thread can be inferred more accurately. Their approach pruned false positives by an average of 89.2% and reduced dynamic instrumentation by an average of 63.4% in seven benchmarks. Nguyen et al. [104] proposed an FPM approach that combined static analysis and deductive verification. In particular, their approach was designed to identify whether the reported warnings violated the SEI CERT C Coding Standard. To this end, they used ACSL (i.e., ANSI/ISO C Specification Language) to describe behavioral properties of the program at positions suggested by Rosecheckers and then used deductive verification to prove these properties. Their study showed that 20% of warnings of `Rosecheckers` can be dealt with automatically and 90% false positives of them are reduced. Mangal et al. [64] formulated the problem of user-guided program analysis in terms of solving a combination of hard rules and soft rules: hard rules capture soundness while soft rules capture degrees of approximations and preferences of users. Note that, their analysis process was guided by user feedback.

Dependency flow. Liang et al. [41] pinpointed that the defect patterns of SA tool were not precise enough and the defect detection processes of the corresponding defect pattern were also not suitable enough. They proposed the *EDPSN* (Expressive Defect Pattern Specification Notation) to define conditional method calls and the guiding information for the defect detection and warning prioritization process of resource leaks. Based on the warnings in Eclipse, JBoss, and Weka, EDPSN can achieve a precision of 96% and a recall of 74%. Muske et al. [103] used non-impacting control dependencies (NCDs) to reduce the number of warnings. Note that, an NCD of a warning is a transitive control dependency of the warning's program point, that does not affect whether the warning is an error. Their approach reduced the number of warnings by up to 23.57%, 29.77%, and 36.09% for 16 open source C applications, 11 industry C applications, and 5 industry COBOL applications, respectively. Livshits et al. [26] proposed *MERLIN* to infer explicit information flow specifications from program code. *MERLIN* can reduce 15% of false CAT.NET warnings reported in 10 programs. Giet et al. [96] presented a novel abstract domain developed for Astree to detect finite state machines and their state variables, and allows to disambiguate the different states and transitions by partitioning.

Program slicing. Rival et al. [4] proposed to compute an approximation of a set of traces specified by an initial and a (set of) final state(s) to help classifying warnings as true errors or false warnings. Later, Rival et al. [5] leveraged semantic slice (i.e., the computation of precise abstract invariants for a set of erroneous traces) to construct a useful characterization of a possible error context.

Others. Baca [29] identified security relevant warnings through code taint analysis. First, they determined the data entry points (i.e., any type of user data) in the source code. Second, they performed a data flow analysis to uncover the propagation path of the user data and marked the lines affected by the user data as tainted. Finally, only the tainted source code related warnings are considered as potential security vulnerabilities. Their approach cut down the total amount of reported warnings to only one fifth. Joshi et al. [40] argued that limiting SA tools to only report interleaved bugs can reduce FPs for static concurrency analysis. Therefore, they formalized interleaved bugs as a differential analysis between the original program and its sequential version and provided various techniques for finding them. Gao et al. [109] proposed *BovInspector* to validate static buffer overflow warnings and providing suggestions for automatic repair of true buffer overflow warnings for C programs. Their study showed that *BovInspector* can automatically validate on average 60% of total warnings reported by SA tools. Muske et al. [71] leveraged the collaboration between the user and the client analysis helps the SA tools to weeding out the false positives. The approach was proven to be able to reduce 42% of manual effort on average. Chimdyalwar et al. [63] combined program slicing, ICE (i.e., Iterative Context Extension), and LABMC (Loop Abstraction for Bounded Model Checking) to eliminate FPs. Their approach was evaluated in the warning sets generated by two sound SA tools (Polyspace and TCS ECA). Their approach can identify more than 70% of the SA warnings on the applications as false positives.

3) Dynamic Program Testing: A total of 8 FPM approaches ([6], [19], [33], [34], [38], [47], [67], [68]) are based on dynamic program testing. Aggarwal et al. [6] integrated static analysis and dynamic analysis techniques in a complimentary manner to detecting buffer overflow vulnerabilities with great precision and coverage. Csallner et al. [19] designed a three-step approach *DSD-Crasher* to find bugs more accurately. Firstly, they conducted dynamic invariant detection to capture the intended execution behavior of the program. Secondly, they analyzed the program within the restricted input domain to explore many paths statically. Finally, to confirm results are feasible, they generate test cases that focus on reproducing the predictions of the static analysis. Li et al. [33] proposed to combine static analysis and automatic dynamic testing to filter out FP warnings. Firstly, they leveraged static checkers to identify a list of warnings. Next, they ran tests generated by tester to cover the erroneous program code that are detected by static checkers. Once a set of warnings triggered failed tests, they will be sent to developers for fixing. The preventer leveraged the static checker and test cases to check any new changes to the target program, which were able to detect similar warnings right away and stop it from reoccurring by informing the developer the issues. Their approach had a precision of over 70% in experimental projects Ge et al. [34] proposed a defect-detection tool DyTa that leveraged dynamic symbolic execution to guide static verification results.

Chebaro et al. [38] pointed out that, because real programs usually are large scale, dynamic test generation may time out before confirming some warnings as real bugs or rejecting some others as false positives. Therefore, they proposed to leverage program slicing to reduce the size of source code before test generation. Their experiments showed that it easier to analyze detected bugs and remaining warnings by simplifying the target program. Li et al. [47] developed a dynamic technique that automatically validates and categorizes the warnings related to memory leaks. Their approach can analyze each warning containing leak allocation site and leak path information, generate test cases that override the leak path, and track the objects created by the leak allocation site. After that, all warnings will be classified into four distinct categories. With their approach, the developer's manual validation effort is usually much smaller than the original warning collection because only MAY-LEAK warnings need to be inspected by developers. Salvi et al. [67] pointed out that, a warning can be proven to a true positive when the corresponding test vector reproducing the error has been found. Based on this idea, they presented an approach to couple model-based testing with static analysis to find test vectors of warnings. Sozer et al. [68] designed a complementary approach. Specifically, they utilized warnings that are generated by static analysis tools are to generating runtime verification specifications automatically. Meanwhile, they used runtime verification results to generate filters for static analysis tools to eliminate FP automatically.

4) Machine Learning: A total of 24 FPM approaches ([24], [27], [31], [53], [54], [58], [60*], [61], [75*], [76], [78], [84], [85], [86], [90], [101], [105], [106], [112], [113], [114], [116], [117], [118]) are based on machine learning. In the following, each approach is introduced according to their used classifiers.

Traditional classifiers. Ruthruff et al. [24] proposed a four-stage feature screening methodology with analysis of deviance to eliminate the features with a low predictive power and to build *Logistic Regression* models in a cost-effective way. Their approach achieved over 70% accuracy in both predicting false warnings and actionable warning. Liang et al. [31] presented a *K-nearest Neighbors* approach for constructing an effective training set for any given project so that a sound learner can be trained. By mining generic-bug-fix-history, they collected 22 input features and labeled each warning with the help of "generic-bug-related lines". With the help of our constructed training set, their approach achieved a precision of 100% for the top 22 warnings of Lucene, 20 of ANT, and 6 of Spring. Alikhashashneh et al. [84] built a three-classification model based on *Random Forest* classifier trained by code metrics (e.g., complexity and coupling). In particular, they extracted 21 function-level code features and leveraged SMOTE technique to handle the unbalanced training data. They found that the complexity degree and coupling between the functions in source code were reported to have a significant influence on the correctness of SA tools. The overall results of their approach exhibited F1-scores range from 83% to 98%. Cheirdari et al. [75] proposed to leverage *SGD* classifier (implemented by Weka) to increase accuracy of FPM approaches for warnings generated to depict system security posture by identifying specific features, algorithms, and processes to label the training set. Cheirdari et al. [85] proposed *SAPI* (i.e., Software Assurance Personal Identifier), which considered the personal identifiers (e.g., author name) as an additional significant feature. Their study showed that there was a significant correlation between the personal identifier and the likelihood of true or false positives. Cheirdari et al.'s approaches can achieve an F1 value of over 95% on experimental data.

Yoon et al. [60] parsed the AST (i.e., Abstract Syntax Tree) of source code to represent the approximated structural characteristics (i.e., the executable paths of the warnings) and selected *SVM* (i.e., Support Vector Machine) as the classifier to learn the patterns of both true bugs and FP warnings. Their approach can remove 37.33% of false warnings while only removing 3.16% of true bugs. Heo et al. [76] held that the program components (e.g., loops and library calls) that produce false warnings are alike, predictable, and sharing some common properties and used the one-class *SVM* to capture the common characteristics of the harmless and precision-decreasing program components. Based on the "Golden" feature set [93], Yang et al. [117] designed an incremental AI tool implemented by *SVM* to distinguish FP warnings. Later, Yang et al. [118] pointed out that the "intrinsic dimensionality" of warning feature set was low, indicating that it was not necessary to build a sophisticated deep learning model. Their study results concluded that leveraging *linear-SVM* model to identify actionable warnings was simple yet effective. One more important implication their study conveyed is that it is important to match the complexity of the solutions and their corresponding problems. Note that their approaches can identify over 90% of actionable warnings.

Ensemble classifiers. Yuksel [61] et al. employed a trust-based classifier fusion method, which is an enhanced voting ensemble that considers the trustworthiness of the involved classifiers. Note that the approach is evaluated on the real warning instances from digital TV software system. Ribeiro et al. [90] developed *Kiskadee* to continuously conduct static analysis on software packages and prioritize the generated warnings. Specifically, they built an AdaBoost classifier based on the base learner DT (Decision Tree). Note that, *Kiskadee* does not leverage project-specific intrinsic features (e.g., code metrics) from the analyzed project when building the ranking model and it can be used directly without training new different models. *kiskadee* can hits 5.2 times less false positives before each bug than when using a randomly sorted warning list. Ribeiro et al. [106] proposed to train a stronger ensemble learning model using AdaBoost algorithm based on a set of decision trees. Notably, they incorporated multiple specialist SA tools to produce a reasonably accurate model, which also could increase the coverage of static analysis so as to minimize as many true warnings as possible that were missed by any individual tool. The approach can achieve 80% classification accuracy.

Deep learning classifiers. Koc et al. [78] aimed to discover and utilize the relations between program structures and FP warnings. Their approach consists of two steps: code preprocessing and learning. The purpose of the former step is to get appropriate reduced code snippets associated with warnings automatically. The latter step is to train and apply the recurrent neural network model (*LSTM*, Long Short-Term Memory)

based on the reduced code snippets to filter out the FP warnings. The approach can achieve an accuracy over 85% on average. Lee et al. [101] proposed an automated *CNN* (Convolutional Neural Network) model to identify FP in the context of CI (i.e., Continuous Integration). In particular, instead of summarizing a fixed set of warning features, their *CNN* model can automatically learn the lexical patterns related to FP warnings from historical dataset. Therefore, any effort on feature engineering or extraction is not required. Their approach can effectively identify false warnings with the average precision of 79.72%.

Multiple classifiers. Many researchers examined more than one classifier in their studies. Heckman et al. [27] proposed a four-stage process for building warning classification models, including warning features collection, feature subset evaluation, model construction, and model selection. They found that, although some common features (e.g., file name and file creation revision) were useful in building a best prediction model, 11 additional features were project-specific. In addition, different ML classifiers should be used when predicting the labels of warnings in different projects. Hanam et al. [53] predicted actionable warnings by mining the underlying warning patterns. They first generated a backwards program slice for each warning statement and then extracted a set of characteristics from the statements touched by the slicer as code patterns. Next, these patterns as well as the true warning labels were regarded as prior knowledge to train a ML model. Their study confirmed that there existed some common static analysis warning patterns indeed. Their approach was able to identify 57 of 211 actionable warnings when a developer inspects the top 5% of all warnings.

Medeiros et al. [54] presented hybrid method that first leveraged taint analysis to flag candidate vulnerabilities and then built ML models to identify FP vulnerabilities in WEB application. Tripp et al. [58] leveraged an interactive solution to build ML models based on user feedback on a small set of warnings. Their solution is user centric, i.e., it can execute the different classification policies expressed by users, ranging from strong bias toward elimination of FPs to strong bias toward preservation of true warnings. Flynn et al. [86] developed a model by fusing features derived from multiple SA tools, code metrics, and archived audit determinations. Notably, according to the user-specified confidence levels, their model can classify warnings into three categories: expected-true positive, expected-false-positive, and indeterminate. Their work gained an understanding of correlations between different SA tools. The above approaches can achieve an accuracy of nearly 90%. Pereira et al. [105] leveraged the information of output from multiple SA tools to build regression-based ML classifiers for the purpose of both reducing the FP rate and retaining the ability to defect detection. The difference is, their approach was evaluated in a PHP project to classify two critical warning types: SQL Injection (SQLi) and Cross-Site Scripting (XSS), which showed the proposed approach outperformed the traditional 1-out-of-N (1ooN) heuristic. Zhang et al. [112] designed a set of novel fine-grained variable-level features (e.g., AS_C denotes related variable assigned by a constant value) to build a prediction model. They selected 13 base classifiers built in Weka and then introduced two ensemble learning classifiers (Majority Voting and RF) to achieve more accurate classifications. Note that their proposed models were merely trained for predicting the warnings of null pointer dereference (NPD). Zhang et al. [113] presented a two-stage feature ranking-matching transfer learning approach based on a set of novel fine-grained features at variable-level to increase the performance of cross-project defect identification. Flynn et al. [114] considered static analysis test suites (i.e., repositories of "benchmark" programs that are purpose built to test coverage and precision of SA tools) as a novel source of training data. They found that pre-training classifiers on test suite data is helpful to jumpstart SA warning classification in data-limited contexts. Their approach obtained high precision (90.2%) and recall (88.2%) for a large number of code flaw types on a hold-out test set.

5) Clustering: A total of 12 FPM approaches ([44], [48], [51], [55*], [56*], [73], [74*], [77], [79], [88], [107], [115]) are based on clustering. Fry et al. [44] presented a parametric clustering approach that leveraged both syntactic and structural information available in warning reports. To ensure a low maintenance cost and high accuracy, the optimizer goal of the approach is to maximize the size of clusters and the similarity of warnings in a same cluster. The approach can reduce the number of individually inspected warnings by 21.33% at a level of 90% accuracy. Based on the assumption that two warnings are correlated if occurrence of one warning causes another warning to occur, Zhang et al. [51] proposed diagnosis-oriented warning correlations. Their approach had the effect of reducing 33.1% of warning identification. Lee et al. [77] proposed a sound octagon-based method to cluster warnings by leveraging sound dependencies among all warning candidates. Their approach first tracked all warnings to slice the static analysis result and appointed the warning candidate that was possible to kill other warnings as the dominant one. Next, the warnings that can be killed by the same dominant warning were clustered. This approach can reduce 45% of warnings in their study. Yang et al. [107] proposed *Priv* to identify preferred fix locations for the detected vulnerability warnings and group them based on the common fix locations. Then, Priv identifies actionable vulnerability warnings by removing SA tool-specific FPs. Finally, *Priv* provides customized fix suggestions for vulnerability warnings.

Muske et al. proposed a series of clustering-based approaches. Specifically, Muske et al. [48] proposed to identify redundant warnings to reduce manual review efforts. They first partitioned the reported warnings and assigned a leader warning for each partition. Once a leader warning was identified as a FP, developers do not need to continue the following warnings of the same partition. Then, they leveraged the similarity of the modification points of variables referred to their expressions to group the leader warnings. Clustering is commonly used technique to scale SA tools to large systems by breaking a system into multiple clusters. They found 60% of warnings are redundant in the review context and skipping their review would lead to a reduction of 50-60% in manual review efforts. Muske et al. [55] identified and reported four types of suitable information (e.g., warning point, and code slice) about the warnings impacted by the shared variables, which implemented the inter-cluster communication. Muske et al. [56] eliminated the

TABLE VII
THE COMPARISON AMONG DIFFERENT TYPES OF FPM APPROACHES

Category	Pros	Cons
Statistical probability	- Can be used to identify patterns and trends in data. - Can be used to calculate the probability of a warning being a false positive.	- Can be limited by the quality and quantity data available. - Can be difficult to interpret to non-technical users. - Can produce wrong answers if the underlying assumptions are incorrect.
Static program analysis	- Can capture root cause of false positive. - Can be used to identify code that violates best practices and coding standards. - Can produce high confidence results.	- Can be limited by the scale, complexity of the code being analyzed and the selection of SA tools. - Can produce wrong answers if the analysis method is not thorough enough.
Dynamic program testing	- Can identify issues that may not be caught by static analysis. - Can provide more accurate results. - Can be used to test code to identify issue in real-world scenarios.	- Can be time-consuming and resource-intensive especially for complex code. - Can be difficult to reproduce issues in a controlled environment. - Can produce wrong answers if the testing is not thorough enough.
Machine learning	- Can learn from data and improve over time. - Can be used to identify patterns and trends in data. - Can identify patterns and trends that may not be apparent to humans.	- Can be limited by the quality and quantity data available. - Can be difficult to interpret to non-technical users. - Can produce wrong answers if the underlying features are incorrect.
Clustering	- Can group similar warnings together, making it easier to identify false positives. - Can be used to identify patterns and trends in data. - Can be used to prioritize warnings based on their likelihood of being false positives.	- Can be limited by the quality and quantity data available. - Can be difficult to interpret to non-technical users. - Can produce wrong answers if the clustering algorithm is not accurate enough.

redundancy by grouping the inter-cluster common warnings such that review of a grouped warning under given constraint guarantees same review judgment for the warnings in other clusters. Their approach can decrease 60% of the manual efforts required to review the common warnings. Muske et al. [88] proposed a repositioning way, aiming to reduce the warning count and to report warnings as close as possible to their cause points. The technique would reposition a new warning by grouping several related original warnings and was able to maintain the traceability links between the repositioned warning and its original counterparts. Because more similar warnings are merged together in the process of repositioning, the number of warnings is expected to be reduced further. Their results indicated that the proposed approach can reduce the warnings count by up to 20% over the SOTA clustering-based approaches.

Reynolds et al. [79] presented a general process to eliminate FP warnings by identifying the corresponding code patterns. Because code patterns often span over a small portion of program locations rather than the entire code, they proposed to reduce FP warning related source code so that the essence code of the warning remained. All reduced codes were then standardized as several FP patterns consisting of four attributes (e.g., FP warning message). In total, their study summed up 14 core false positive patterns. Liu et al. [115] pointed out that, the warnings can be regarded as true positives if they were recurrently removed by developers through code changes. They referred to these frequent warnings as common warning patterns and conducted a large empirical study to investigate the recurrences of `FindBugs` warnings and their fixing patterns. Specifically, they designed a *CNN* model to extract pattern features and then leveraged X-means technique to mine the common code patterns and the corresponding fix patterns of warnings.

C. RQ2.3: What Lessons Can be Learned From the Above Progress?

This sub-question aims to conduct a comprehensive comparison of each category of FPM approaches, leading to the identification of essential insights. In Table VII, we provide a summarized comparison of the different types of FPM approaches, outlining their respective advantages and limitations. To facilitate understanding from a technical standpoint, we classify the surveyed FPM approaches into two categories: data-oriented and tool-oriented technologies. Furthermore, we present valuable prospects for the future advancement of FPM techniques.

1) Data-Oriented Technology: **Advantages.** The architectures of statistical probability, machine learning, and clustering approaches are constructed by analyzing the distribution of existing warning data, which exhibit strong data dependencies. These data-oriented technologies offer several key advantages. First, these approaches are designed and applied independently of specific SA tools. As long as suitable data instances can be gathered, these methods can be easily employed to mitigate any SA tools. Second, these methods possess the capability of automatic model self-updating. As new data is continuously input into the model, its performance gradually improves, moving closer to the desired outcome. Third, the principles behind these methods are simple and easy to understand, allowing for continuous improvement and optimization. This is evident from the

significant number of research papers focusing on these approaches. For instance, existing machine learning approaches aim to explore the optimal selection of warning features, learning algorithms, and data preprocessing methods. As machine learning technology matures, the support provided by numerous established frameworks (e.g., Weka or Scikit-Learn) greatly simplifies the modeling process.

Limitations. One major limitation is the dependence on the quality and quantity of available data. Models constructed using larger training datasets with minimal label biases and outliers tend to demonstrate superior performance. Conversely, limited or biased data can hinder their effectiveness. Another limitation arises from the potential for incorrect outcomes when the underlying assumptions are flawed. For instance, in the work of Wang et al. [93], certain features were erroneously defined, resulting in information leakage and overestimated results from the ML method. Lastly, these data-oriented technologies often lack explanatory capabilities. Relying solely on black box prediction scores does not provide intuitive reasons, making it challenging for developers to confidently apply them in real-world scenarios.

2) Tool-Oriented Technology: **Advantages.** Both static program analysis and dynamic program testing approaches are part of tool-oriented technology, specifically designed for specific static analysis or testing tools. These approaches offer several significant advantages. First, they often yield more accurate prediction results. Static program analysis deals directly with the source code or its intermediate representations, enabling high-confidence results based on a detailed understanding of the concrete code logic. Meanwhile, dynamic program testing simulates the target program's execution under real-world scenarios, allowing accurate predictions of warning authenticity based on instances of failed tests. Second, these approaches provide enhanced interpretability of the output results. In the case of static program analysis, it is relatively easy to understand the reasons for the removal of FP warnings by examining specific techniques employed by the analysis. For dynamic program testing approaches, warnings that pass the test can be interpreted as instances of false positives.

Limitations. One major drawback of tool-oriented technology is the high application cost in terms of analysis time and hardware resources. This limitation becomes apparent in two ways. First, complex program structures can significantly increase the constraint-solving space, leading to slow static analysis processes. Second, executing test cases can be time-consuming, particularly when dealing with large test suites. It is important to note that static program analysis approaches have an additional limitation. They are typically customized and optimized for specific types of warnings, which restricts their applicability in various scenarios. Different types of FP warnings, such as those caused by incomplete program analysis, imprecise data flow analysis, or incorrect modeling of program behavior, require different mitigation strategies. Similarly, dynamic program testing has its limitations. If the test cases are not designed properly or are incomplete, the effectiveness of removing false positive instances will be significantly weakened.

3) Research Prospects: The following three research prospects are summarized:

Technology combination. Mitigating FP warnings does not have a one-size-fits-all approach. Data-oriented technology is usually tool-agnostic and cost-effective, while tool-oriented technology excels in providing explanatory information. Combining multiple techniques can often yield better results than using a single technique alone. However, it is crucial to minimize the negative impact of each technique's limitations while leveraging their respective strengths.

Participant collaboration. Although existing FPM approaches show promising prediction results under experimental scenarios, their performance in real-world scenarios, especially for data-oriented technology, is rarely explored. Collaboration between developers and researchers is essential in this context. Developers can provide feedback on FP warnings, helping to improve the accuracy of FPM approaches. This collaboration can lead to better mitigation strategies and more effective SA tools.

Incremental mitigation. Existing approaches often lack an incremental mitigation path, resulting in the inability to effectively inherit previous research findings, particularly for tool-oriented technology. It is important to note that FP warnings can never be completely eliminated, but ongoing research can help reduce their occurrence. As software systems become more complex, new types of FP warnings may emerge, necessitating new incremental mitigation solutions that build upon previous research insights.

Summary of results for RQ2:

1) According to the different rationales of existing studies, the 85 FPM approaches can be classified into the following five categories: statistical probability, static program analysis, dynamic program testing, machine learning, and clustering, respectively.
2) Researchers pay more attention to three types of approaches: statistical probability, static program analysis, and machine learning, which accounts for 76.5% (65/85) of the total approaches. Meanwhile, approaches based on dynamic program testing have disappeared in recent years (since 2016).
3) Existing FPM approaches apply either data-oriented or tool-oriented technology, showing complementary strengths and weaknesses. Three prospects, technology combination, participant collaboration, and incremental mitigation, can be summarized.

VI. RQ3: How the Current FPM Approaches Is Evaluated?

In this section, the main progress of the evaluation system for FPM approaches are introduced. Evaluation system is very critical to evaluate the effectiveness of any proposed novel approaches, which typically consists of three factors: studied SA tools, experimental scenarios, and performance indicators. The details can be found in [122].

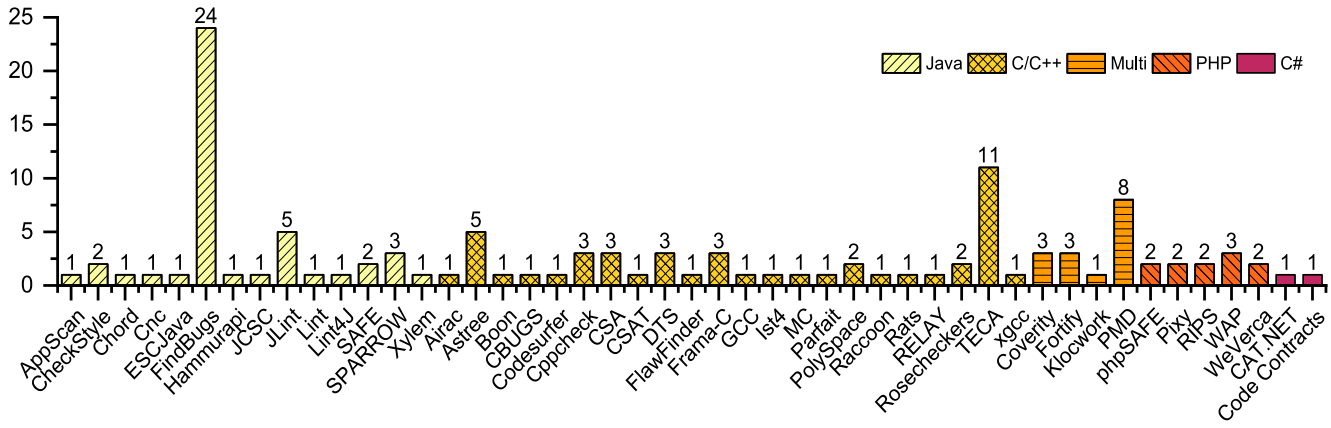

Fig. 8. Distribution of studied SA tools in the studied FPM approaches.

TABLE VIII
COMPARISON ON DIFFERENT EXPERIMENTAL SCENARIOS

Cate.	Description	Advantage	Ref. Value	Number (Ratio)
Experiment	The studies designed detailed evaluation process methodology to test the performance of the proposed approaches in terms of multiple indicators.	Report fully evaluated performance	High	71 (77.2%)
Case study	The studies conducted several cases or investigate some examples to illustrate the usefulness of their approaches.	Intuitive and easy to understand	Medium	14 (15.2%)
None	There was no evaluation section for these studies.	-	Low	7 (7.6%)

A. RQ3.1: What SA Tools Are Considered as the Research Objects?

Fig. 8 reports the distribution of studied SA tools. In order to locate a specific SA tool quickly from the figure, we arrange them in alphabetical order. From Fig. 8, the following three observations can be made. First, the studied SA tools are widely distributed, and a total of 47 different SA tools were considered as the research objects of existing FPM approaches. This means that the issue of false positive warnings exists in a large number of SA tools. Second, the frequency number of the studied SA tools was extremely imbalanced. Only five SA tools of which were studied multiple times (i.e., not less than 5), especially for `FindBugs` (i.e., 24). Meanwhile, about 55.3% (i.e., 26/47) SA tools were studied only once, which indicates that existing FPM approaches proposed for these SA tools may not be suitable for others. Third, these SA tools are mainly used to analyze five different programming languages (e.g., `FindBugs` for Java and DTS for C/C++). Java and C/C++ were the most popular languages and other languages received little attention.

B. RQ3.2: What Evaluation Scenarios Are Leveraged?

Evaluation scenario refers to the organizational structure and whole workflow of each evaluation process, which determines the effectiveness and value of an approach. Among the studied FPM approaches, the evaluation scenarios can be summarized by following three categories (shown in Table VIII):

Experiment. This category of scenario conducts a detailed evaluation methodology, which consists of large-scale experimental data, performance indicators and experimental setting (e.g., hold out and cross validation). Under this scenario, the performance of the proposed approaches can be fully evaluated and proved. Therefore, the reference value of these approaches is relatively high and they are recommended to follow. Specifically, a total of 71 primary FPM approaches were evaluated by Experiment, which accounts for 77.2% of the total number.

Case study. This category of scenario only analyzes an approach based on several illustrative cases or examples, which is intuitive and easy to understand the rationale and procedure of the specified approach. However, there is a potential disadvantage that the generalization of the approaches is unknown due to they were not verified in large practical projects. Therefore, the reference value of these approaches is considered as medium in this paper. Specifically, 15.2% studied approaches (i.e., 14) belong to this category.

None. Although some researchers have proposed novel approaches or tools, they have not evaluated them by any one of above two common scenarios. Therefore, it is not clear whether these approaches in this category are valuable. In this paper, the reference value of them is considered as low. Specifically, the scenarios of only 7 studied approaches were *None*, which accounts for 7.6% of the total.

C. RQ3.3: Which Performance Indicators Are Selected?

Performance indicators play a crucial role in assessing the effectiveness and efficiency of novel approaches. In the context of FPM, most approaches either generate a ranked list of

warning candidates or classify them into distinct categories. To evaluate existing FPM approaches, seven commonly used performance indicators have been identified and summarized in Table VIII. These indicators serve as valuable tools for evaluating and comparing the performance of different approaches.

1) Classification Indicators: From the perspective of a binary classification, the classification result of each warning instance can be divided into four cases: TP (a true warning is correctly classified as a positive instance), TN (a false warning is correctly classified as a negative instance), FP (a false warning is incorrectly classified as a positive instance), and FN (a true warning is incorrectly classified as a negative instance). In the literature, most performance indicators are calculated based on TP, TN, FP, and FN directly or indirectly.

Recall measures the proportion of correctly classified true warnings by a prediction model out of the total number of true warnings. The goal of a prediction model is to identify true warnings as many as possible. Therefore, the higher is the Recall value, the better is the prediction model. Note that, recall can be more useful when missing a true positive report is unacceptable.

Precision measures the proportion of correctly classified true warnings out of all true warnings classified by a prediction model. In particular, precision should be considered seriously when the cost of reviewing false positive reports is unacceptable. Note that, a prediction model can be less useful when a developer must inspect many false warnings to find true warnings.

Accuracy measures the proportion of warning instances correctly classified by a prediction model. Note that, this is a good indicator of effectiveness when the test data has a balanced class distribution (i.e., true and false). However, such an indicator is of little value when evaluating an approach on a class imbalanced data set.

F1-score denotes the harmonic mean of precision and recall. As can be seen, this is a comprehensive indicator that takes into account both precision and recall, which is usually used to evaluate the overall performance of a prediction model.

AUC measures the two-dimensional area under the Receiver Operator Characteristic (ROC) curve. *AUC*, the area under ROC, provides an aggregate and overall evaluation of performance across all possible classification thresholds [118]. Note that, this indicator is widely applied in class imbalanced data sets.

2) Ranking Indicators: Under a ranking, however, an approach will rank warnings in descending order from the most to the least suspicious scores. Developers can inspect each warning in turn based on the list of recommendations.

Effort measures the number of warnings one developer must inspect to find a true warning. As can be seen, developers would like to stop inspecting when they could not find any true warning in the top recommended list. Therefore, the lower is the effort value, the better is the prediction model. Similar to *Count*, this indicator is often used in an empirical study, especially where the oracle is the behavior of the developers and no ground truth is known.

FDRC (Fault Detection Rate Curve) describes the percentage of all true warnings found within the first x% warnings of the ranking list. From the curve, various fault detection rates can be easily acquired by selecting different thresholds. Once a threshold is specified, the fault detection rate can be regarded as the precision indicator.

PR (Performance Ratio) measures the performance ratio of the proposed model against a random model in terms of the average of the cumulative number of false positive warnings found before reaching each true positive. $S(m)$ denotes the sum of FP_j, which is the cumulative number of FP warnings found before reaching the j^{th} TP warning. N_{tp} denotes the total number of true positive warnings. FP_{avg} denotes the average of the cumulative number of false positive warnings found before reaching each true positive warning.

D. RQ3.4: Which Datasets Are Collected and Applied?

A dataset is a set of data instances manually or heuristically extracted from software projects. In particular, in our context, an instance refers to a concrete SA warning generated by an SA tool and its corresponding label (i.e., whether a warning indicates a real defect). The effectiveness of a novel technique needs to be verified based on a reliable dataset so the quality of a dataset directly determines the correctness of evaluation results. In the literature, according to reliability and universality, various datasets can be roughly classified into two categories: benchmark datasets and research-specific datasets.

1) Benchmark Datasets: Benchmark can be regarded as a procedure, problem, or test that can be used to make fair comparisons among various systems or approaches, which also provides an experimental basis for evaluating software engineering theories in an objective and repeatable manner [21]. Notably, collaboration within a research community can be promoted greatly when successful benchmarks (with properties of accessibility, affordability, charity, relevance, solvability, portability, and scalability) are provided [179].

Table X reports four publicly available benchmarks used in multiple studies for evaluation of FP mitigation approaches. Heckman et al. [21] contributed the FAULTBENCH benchmark to software anomaly detection community, which not only includes a suite of subject programs and warning oracles, but also provides repeatable procedures for evaluation of FP mitigation (both prioritization and classification) techniques. Specifically, their benchmark consists of three components: six open-source Java subject programs, an analysis procedure, and four evaluation metrics. Note that, the classification of each warning was only labelled by one single author, which may be introduce a subjective bias to FAULTBENCH. United States National Security Agency (NSA) created a synthetic C/C++ test suite Juliet [180], consisting of 61,387 test cases with 118 different software flaw categories. In particular, each test case is a code section with an instance of a specific flaw (to capture true positives) and an additional section with a correct, fixed instance of that previous flawed section (to capture false positives). Note that, it also includes a user guide with instructions on how to assess and label static analysis tool warnings generated over its test cases [90]. Based on 12 Java projects with totally 60 different revisions, Wang et al. [93] created a golden feature set consisting of 23 common features that can be used for building

effective actionable warning identification models. They constructed the ground truth by leveraging the heuristic rule that a warning was treated as actionable if it disappeared in a later revision [21], [31], [53]. The OWASP (i.e., Open Web Application Security Project) benchmark project [181] is also a Java test suite designed to evaluate accuracy, coverage, and speed of automated software vulnerability detection tools. It is a fully runnable open-source web application that contains thousands of exploitable test cases. In particular, each test case is mapped to specific CWEs, which can be analyzed by any type of Application Security Testing (AST) tool, including SAST, DAST (like OWASP ZAP), and IAST tools [181].

2) Research Specific Datasets: Note that, although there are benchmarks in the literature, they are rarely used in later studies. Unlike benchmark datasets, many researchers collected research-specific projects in their individual studies. Note that, the original papers had not explicitly stated the concrete information about their used datasets or they just conducted some small-scale case studies. Therefore, most of the evaluation capability are limited to a certain extent. For example, Koc et al. [100] created a real-world dataset consisting of 14 Java programs with 400 vulnerability warnings reported by FindSecBugs. Shen et al. [35] collected a dataset consisting of 3 Java projects with 174 warnings generated by FindBugs. Additionally, the classification (i.e., true or false positive) of each warning in these projects was manually labelled by the corresponding authors.

E. RQ3.5: What Lessons Can be Learned From Evaluation System?

1) SA Tools: **The generalizability of existing research findings is constrained by the proprietary nature of SA tools.** This limitation arises from both language differences and technological disparities among these tools. Language differences pose a challenge as different programming languages exhibit variations in syntax, semantics, and programming paradigms. Consequently, it becomes arduous to directly transfer analysis methods and algorithms from one language to another. For instance, SA tools tailored for Java may heavily rely on APIs, libraries, and language structures specific to Java, which might either not exist or have different implementations in other languages. Moreover, SA techniques themselves differ in their underlying principles and implementations. For example, FindBugs utilizes pattern matching and data flow analysis methods to conduct static scanning of code, enabling the identification of common programming errors, potential logical issues, and violations of coding style. Conversely, TECA adopts an abstract syntax tree (AST)-based approach to capture abstract information from code and applies techniques such as symbolic execution, path sensitivity analysis, and data flow analysis to uncover potential issues. These factors hinder the direct transferability of analysis methods and algorithms across different programming languages and SA tools.

To enhance the generalizability of existing research findings, it is recommended to explore the development of a generic research framework that can accommodate multiple SA tools and programming languages. This framework would involve several key components, including the definition of intermediate representations or unified abstract models specific to different tools, as well as standardized evaluation methods and metrics. By establishing such a framework, it would become easier to compare and conduct transferability studies among different SA tools. Furthermore, the literature has been actively exploring collaborations across different disciplines to foster the integration of knowledge and techniques. This interdisciplinary approach aims to advance transferability studies between various SA tools. For instance, the application of natural language processing and machine learning (or deep learning) techniques to SA holds the potential to provide general methods and models that can be applied across different tools and programming languages. By developing a generic research framework and fostering interdisciplinary collaborations, the generalizability of existing research findings in SA can be significantly enhanced. This would enable easier comparison, facilitate transferability studies, and promote more effective analysis across different SA tools and languages.

2) Evaluation Scenarios: **Researchers should aim for a balanced approach by incorporating both experimental evaluations and case studies in their work.** This combination allows for a comprehensive understanding of the proposed approaches, their performance, and their practicality in real-world scenarios. Notably, a significant majority of the studies (77%) included a well-designed evaluation process methodology, highlighting the importance of conducting thorough evaluations to assess the performance of the proposed approaches. For comprehensive evaluations, it is crucial to have both quantitative and qualitative indicators in place to measure the effectiveness and impact of the methods being studied. However, it is concerning that a non-negligible proportion of papers (23%) either presented only case studies or lacked an evaluation section altogether. This raises potential limitations in terms of the rigor and validation of these approaches. A thorough evaluation is essential to ensure the reliability and effectiveness of proposed approaches, and its absence could raise concerns about the credibility of the findings. By emphasizing the incorporation of both experimental evaluations and case studies, researchers can strengthen the validity of their work and provide more robust evidence for the effectiveness of their proposed solutions.

3) Performance Indicators: **The prevailing perspective in the literature is to consider FPM as a classification task.** This is primarily because classification tasks provide users with a clearer basis for selecting review warnings compared to ranking tasks. Table IX demonstrates that Precision is the most commonly utilized classification indicator, appearing in 22 papers. Following Precision, Recall is selected to evaluate 20 FPM approaches. These two indicators are complementary, as they tend to focus on different aspects of an approach. It is worth noting that recent studies have also incorporated AUC as a comprehensive classification indicator, which enables the evaluation of performance across various classification thresholds. As for ranking performance, FDCR has been recommended in multiple works.

The selection of performance indicators varies due to different application scenarios. Simplified indicators are often

TABLE IX
SUMMARY OF POPULAR PERFORMANCE INDICATORS

Name	Name	Definition	References	Freq.
Classification	Precision	$\frac{TP}{TP+FP}$	[13], [21], [24], [27], [33], [35], [41], [45], [53], [57], [75], [78], [85], [90], [100], [101], [105], [106], [112], [172], [175], [178]	22
	Recall	$\frac{TP}{TP+FN}$	[21], [27], [33], [35], [41], [45], [53], [57], [75], [78], [85], [100], [101], [105], [106], [112], [118], [172], [175], [178]	20
	F1-score	$\frac{2\times Precision\times Recall}{Precision+Recall}$	[27], [35], [53], [75], [84], [85], [101], [105], [106], [112], [172], [173]	12
	Accuracy	$\frac{TP+TN}{TP+FP+TN+FN}$	[21], [45], [75], [78], [85], [86], [101], [100], [106], [172]	10
	AUC	$\frac{1}{2}\sum_{i=1}^{n}(x_{i+1}-x_i)(y_{i+1}-y_i)$	[112], [118], [173], [178]	4
Ranking	FDRC	$\frac{\#\ actionable\ wanrings}{\#\ inspected\ warnings}$	[1], [10], [21], [31], [37], [53], [80], [105]	8
	PR	$S(m)=\sum_{j=1}^{Ntp}FP_j$, $FP_{avg}=\frac{S(m)}{Ntp}$, $PR=\frac{FP_{avg}(random)}{FP_{avg}(m)}$	[2], [90], [106]	3
	Effort	# Inspected warnings	[1], [37]	2

TABLE X
SUMMARY OF BENCHMARKS USED IN MULTIPLE STUDIES

Year	Authors	Name	Lang.	Description	Used by
2008	Heckman et al. [21]	FAULTBENCH	Java	A benchmark consisting of three components: six open-source Java programs, an analysis procedure, and four evaluation metrics.	[27], [45], [53]
2012	Boland et al. [180]	Juliet	C++	A publicly available test suite consisting of a collection of source code snippets with specific flaws injected in known locations, which facilitates the assessment of static analysis tools.	[79], [106], [143]
2017	OWASP Foundation [181]	OWASP	Java	A benchmark designed to evaluate the accuracy, coverage, and speed of automated software vulnerability detection tools.	[78], [100]
2018	Wang et al. [93]	"Golden" feature set	Java	A golden feature set consisting of 23 common features that can be used for building effective actionable warning identification models.	[117], [118], [173], [178]

favored as they are suitable for more general scenarios. For instance, the widely used complementary pair of precision and recall is a popular choice as a classification indicator, while AUC is employed in specific studies. Similarly, FDCR stands out as the preferred ranking indicator. However, establishing a unified standard for indicator selection is challenging due to the interdependence between indicators and the diverse preferences of researchers. For instance, the F1-score combines precision and recall, but it is not always selected or abandoned alongside these individual indicators simultaneously.

4) Datasets: An important lesson learned is that both existing benchmark datasets and research-specific datasets have limitations in terms of quantity and quality.

The quantity dimension. Existing benchmark datasets often do not cover all possible scenarios and edge cases, which can restrict the generalizability of approaches verified on these datasets. When a dataset is small-scale or biased, it can lead to overfitting and limited applicability to real-world problems. Creating a comprehensive benchmark dataset is a time-consuming and expensive process that requires manual analysis of a large number of code samples to determine TPs and FPs. Consequently, there is a pressing need for tools that can automate the collection of large amounts of warning information.

The quality dimension. Warning dataset labels are subject to dynamic changes and evolution. In other words, labels collected at a certain point in time may become outdated as new information emerges. For example, a warning initially marked as an FP may later be recognized as a true warning due to corresponding defects found in subsequent versions. Consequently, researchers often have to rely on outdated datasets, which may not accurately reflect the current state of research in the literature. Therefore, it is crucial to continuously update datasets to ensure that each

approach can be evaluated under more reliable scenarios with high-quality warning labels. This can be achieved by actively seeking out and incorporating labeling methods, addressing bias and fairness concerns, and regularly re-evaluating and updating existing datasets to reflect changes in the real world.

Summary of results for RQ3:

1) Researchers mainly focused on proposing novel approaches to mitigate FP warnings in SA tools for the languages of Java and C/C++. In particular, `FindBugs` and `TECA` were the most popularly studied SA tools.
2) Most of the studied approaches were evaluated under the scenario of *Experiment* and *Case Study*, which account for 75.3% and 16.5% of the total number, respectively.
3) *Count*, *Recall* and *Precision* were the three most commonly used performance indicators to evaluate performance of FPM approaches, which were selected 30, 21, and 19 times, respectively.
4) There are four publicly accessible benchmark datasets in the literature, three of which consist of projects written in Java language and the rest is based on C++ language. Note that, they are not widely used in the later studies.

VII. RQ4: Which Types of Empirical Studies Are Conducted?

This RQ summarizes the studies that aims to investigate the FPM issue from various perspectives. These studies are very important because they can either compare and verify the conclusions of existing FPM approaches or provide novel insights on how to reduce more FP warnings.

A. RQ4.1: Which Types Can Empirical Studies be Divided Into?

1) The Types of Empirical Studies: According to the research target and content of the studied papers, the existing studies can be divided into the following four categories:

1) **Investigating SA tools.** As the body to generate warnings, the characteristics of SA tools and their ability to detect potential software defects are important for developers, maintainers, and software quality managers. Researchers have conducted many studies to understand the values of SA tools, apply them in an effective way and improve their usefulness in practice.
2) **Investigating warnings.** This type of studies regarded the warnings as research object, which is meaningful to propose novel FPM approaches. Specifically, they analyzed the reasons why FP warnings were introduced, found the potential values of FP warnings, and summarized the characteristics of warnings.
3) **Building benchmark dataset.** A dataset is an indispensable factor in software engineering experiments, which is very important for evaluating the performances of any proposed FPM approaches. Some studies collected publicly accessible benchmark datasets to verify the effectiveness of existing and novel approaches.
4) **Evaluating FPM approaches.** As answered in RQ2, a large number of FPM approaches have been proposed in the literature. However, these approaches were usually evaluated in different experimental settings so that the advantages and disadvantages of different approaches were unclear. In order to inform the selection of the proper FPM approaches, some of existing FPM approaches have been evaluated and compared in empirical studies.

2) The Distribution of Each Type: Fig. 9 reports the distribution overview for four types of FPM studies. In summary, the proportion of FPM studies is lower than that of FPM approaches. According to the distribution of years, SA tools and warnings are the most popular research themes over the last two decades. Meanwhile, several studies were conducted to build benchmark dataset and evaluate FPM approaches intermittently. In terms of a single year, 2007 and 2019 were the most active year, a total of 5 different studies were conducted in each of the two years. Meanwhile, there is no any study in five years (e.g., 2006). Considering the percentage for each type, the primary types of most of FPM studies are investigation on SA tools and warnings, which account for 48.5% and 30.3%, respectively. The remaining two types account for less than one third of the total.

B. RQ4.2: How Each Type of FPM Empirical Studies Conducted?

1) Investigating SA Tools: A total of 16 studies ([12], [16], [18], [20], [28], [36], [46], [57], [83], [87], [91], [92], [97], [98], [102], [111]) were conducted to investigate SA tools. More specifically, these studies can be summarized by the following three aspects:

Cognition of SA tools. Hovemeyer et al. [12] examined several techniques to identify more high-impact null pointer bugs by `FindBugs`. Souyris et al. [18] presented an industrial point of view on how Astrée can analyze real-size industrial programs and be extremely precise. They showed how abstract interpretation based static analysis will contribute to the safety of avionics programs and how a user from industry can achieve the false warning reduction process. Emanuelsson et al. [20] investigated the supporting technology of three SA tools `Polyspace`, `Coverity`, and `Klocwork`. Wedyan et al. [28] studied the effectiveness of SA tools for fault detection and refactoring prediction. They found that the studied SA tools were more effective in identifying refactoring modifications. Almost all the coding concerns were FPs that do not relate to any fault or refactoring modification. Vetro et al.'s study [36] demonstrated that few warnings of `FindBugs` are related to real defects with high precision. Therefore, to discover defects as early as possible, developers need to prioritize them so that the information overload can be reduced. Rahman et al. [57] made a comparison on performance between SA tools and statistical defect prediction (SDP) models. They found that SDP models performed better than `PMD` while worse than `FindBugs` and SDP models

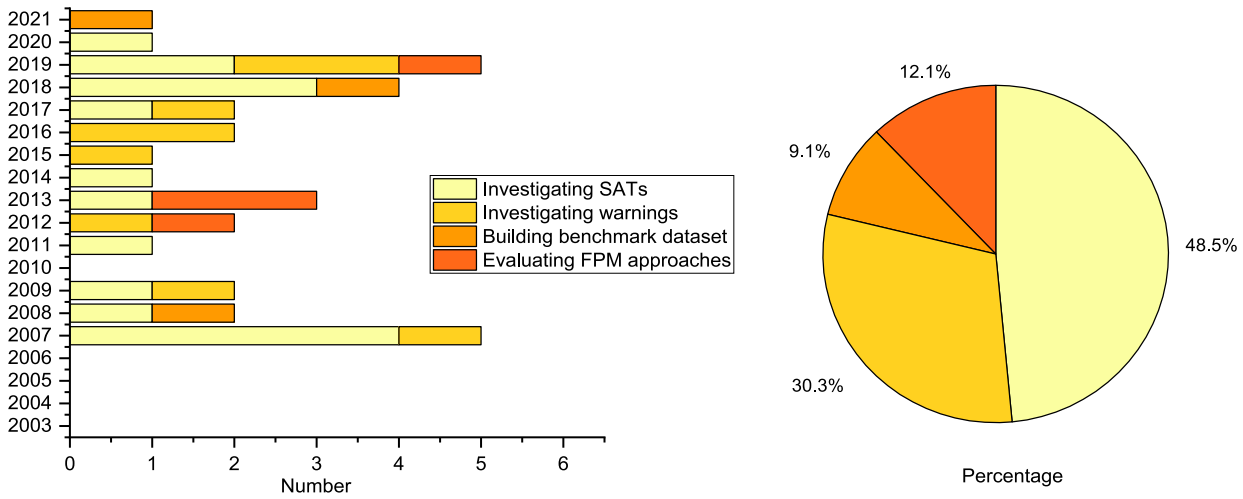

Fig. 9. Distribution of types of FPM studies in the surveyed papers.

could be used to improve the performance of the both SA tools. Habib et al. [87] studied how many bugs SA tools can find in Defect4J dataset. Their findings can be summarized by follows: 1) SA tools can find a non-negligible amount of all bugs. 2) Different SA tools were mostly complementary to each other. 3) SA tools missed the large majority of the studied bugs due to the domain-specific problems that do not match any existing bug pattern. Vassallo et al. [111] found that developers usually did not switch to a new warning configuration in different development contexts.

Adaption of SA tools. Zampetti et al. [83] studied the usage of SA tools within continuous integration (CI) pipelines. Their study indicated that although SA tools could be useful in reporting likely defect, performance, concurrency, and security issues, many of them were FPs. In addition, SA tools did not provide more advanced analyses except avoiding some simple licensing problems. Therefore, in their viewpoint, developers should include more sophisticated tools in CI pipelines. Marcilio et al. [102] conducted a closer look at realistic usage of `SonarQube`. They conjectured that just a subset of the warning rules reveals real design and coding flaws, and this might artificially increase the technical debt of the systems. Imtiaz et al. [98] conducted a study on the usage of `Coverity` to investigate how developers acted on the warnings reported by SA tools. They found that around 36.7% `Coverity` warnings were actionable. Furthermore, they suggested that the warnings indicating a critical issue should be prioritized because they often required immediate addressing by developers. Vassallo et al. [92] explored the adoption of SA tools in practice and they found that developers usually use SA tools in different development contexts.

Improvement of SA tools. Johnson et al. [46] investigated 20 developers to explore why developers did not widely use SA tools and how current tools could be improved. They found that developers were not satisfied with these tools due to the large number of FPs and a high developer overload. Meanwhile, to increase the usage of current SA tools, they should be improved mainly from the design of features on quick fix, warning notification, and manipulation. Sadowski et al. [91] introduced important lessons from building SA tools at Google. Specifically, static analysis authors should focus on the feedback from developers and careful developer workflow integration is key for SA tool adoption. Imtiaz et al. [97] investigated the challenges for developers in understanding and responding to warnings by analyzing 280 questions on Stack Overflow. They highlighted that there were gaps and mismatches between the cognition of developers and the presentation form of warning information. Therefore, they suggested that easy-to-use features and project-specific customization should be integrated by tool builders so that such differences can be reduced. Layman et al. [16] towered to reduce fix time of defect by understanding developer behavior for the design of SA tools. They suggested that SA tools should present fault information that is relevant to the primary programming task with accurate and precise descriptions. The fault severity and the specific timing of fault notification should be customizable.

2) Investigating SA Warnings: A total of 10 studies ([9], [25], [42], [62], [66], [69], [70], [82], [95], [99]) were conducted to investigate warnings. More specifically, these studies can be summarized as the following three aspects:

Reasons of FP warnings. Ayewah et al. [9] conducted experiments to evaluate the accuracy and value of FindBugs warnings. Specifically, they discussed the kinds of warnings and the classification of warnings into FPs, trivial bugs and serious bugs. They pointed out that the tool only looked for issues about unusual code or bad coding styles that usually could not affect the functionality of the program. Therefore, the tool often reported true but trivial warnings. Nadeem et al. [42] found that a large number of FP were reported when analyzing code without leveraging contextual information such as the deployment structure of a project, configuration files, and dependent libraries. Therefore, they advised to incorporate contextual information into SA tools to reduce the number of FPs.

Values of FP warnings. Dimastrogiovanni et al. [70] investigated the hypothesis that there might be a link between FPs and potential security problems. Their results showed that, a FP warning usually indicated a fragility of the program, which was possible to turn into a real defect in different degrees. Aloraini et al. [95] investigated the different types of warnings and their

evolution over time to determine if one type of warning is more likely to have FPs than others. The results showed that most of the tools produce on IVR (Input Validation and Representation), API (API abuse), and CQ (Indicator of Poor Code Quality) warnings. Meanwhile, SA tools could be used to measure the quality of a product and the potential risks without manually reviewing the warnings.

Characteristics of warnings. Yan et al. [82] revisited the correlations between warnings and software defects. Their result showed that only actionable warnings were proper to be considered as early indicators of defects while there was no significant correlation between unactionable warnings and defects. Imtiaz et al. [99] investigated the characteristics of warnings that were most likely to be actionable. They found that, although large number of warnings were associated with control flow issues, developers often considered they were less important. Furthermore, they found that security-related warnings were significantly more likely to be triaged than non-security ones. Ayewah et al. [25] asked users to review warnings using a checklist. They found that warnings can be reviewed in a consistent way. Burhandenny et al. [69] studied the change pattern of warnings over project releases. The results indicated that although many warnings may be reported, most of them are likely to be worthy in improving the code quality, and it is ineffective to reduce the warnings by eliminating such unimportant warnings. Araujo et al. [62] surveyed the correspondence between mutations and warning and found that a correspondence exists when considering specific mutation operators such that warnings may be prioritized based on their correspondence level with mutations. Panichella et al. [66] investigated whether static warnings were taken seriously during code review and what type of warnings were more likely to be removed. They found that 6%-22% of warnings were removed during code review. In addition, developers tended to focus on certain types of warnings (e.g., regular expression category). Meanwhile, removing certain types of warnings before submitting a patch is helpful to reduce the burden during the review process.

3) Building Benchmark Dataset: A total of 3 studies ([21], [93], [119]) were conducted to build benchmark datasets. Heckman et al. [21] contributed the FAULTBENCH benchmark to software anomaly detection community, which not only includes a suite of subject programs and warning oracles, but also provides repeatable procedures for evaluation of FPM approaches. Specifically, their benchmark consists of three components: six open-source Java subject programs, an analysis procedure, and four evaluation metrics. Note that, the classification of each warning was only labelled by one single author, which may be introduce a subjective bias to FAULTBENCH. Wang et al. [93] created a golden feature set consisting of 23 common features that can be used for building effective actionable warning identification models. They constructed the ground truth by leveraging the heuristic rule that a warning was treated as actionable if it disappeared in a later revision. Zheng et al. [119] provided a dataset built for AI-based vulnerability detection approaches. They also used the differential analysis [93] based approach to label issues reported by SA tools.

4) Evaluating FPM Approaches: A total of 4 studies ([37], [45], [50], [100]) were conducted to investigate SA tools. Allier et al. [37] made a comparison among 6 different warning ranking algorithms. They made the following observations: 1) AWARE [42] and FeedBackRank [2] algorithms were significantly better than other 4 ranking approaches; 2) the ranking algorithms that worked on individual warnings instead of on rules were recommended; and 3) it is easier to implement ad-hoc algorithms than statistical algorithms because of less information were required. Therefore, they suggested ad-hoc (e.g., AWARE) algorithms should be considered first. Heckman et al. [45] compared 6 actionable warning identification techniques based on FAULTBENCH [42]. Their results showed that, in terms of accuracy, SAAI [27] was the best overall technique, as it achieved the highest value on 6 of the 9 treatments. Furthermore, ATL-D [14], ATL-R [14], and LRM [24] achieved the highest values on 3, 3, and 2 of 9 treatments. However, they also pointed out that the results were not conclusive, which still needed more comparison on larger projects. Yuksel et al. [50] conducted an industrial case study to evaluate 34 machine learning based warning classification algorithms. They extracted 10 warning characteristics as attributes for building each warning classification algorithm. According to their results, some classifiers achieved a precision of 0.9, indicating that it was viable to use machine learning techniques for automated warning classification. Koc et al. [100] conducted a comparative empirical assessment of four families of machine learning based warning triage approaches. Their results indicated that recurrent neural networks (RNN) approach performed better than the others in classifying FP warnings. Meanwhile, the performance of RNN approach could be improved further with more precise data preparation. In addition, different data preparation techniques should be used when the application scenario was changed.

Summary of results for RQ4:

1) The FPM studies can be divided into four different aspects: investigating SA tools, investigating warnings, building benchmark datasets, and evaluating FPM approaches, respectively.
2) The majority of the empirical studies were conducted to investigating SA tools and warnings, which accounts for 78.8% (26/33) of the total approaches. However, few studies provided publicly accessible datasets or compared the existing APM approaches, which needs to be enhanced in the future.

VIII. RQ5: What Are the Unsolved Challenges and Potential Opportunities for FPM?

A. RQ5.1: Which Challenges Remain Unsolved in the Studied Studies?

Although many studies have proposed various FPM approaches and deeply investigated FP warnings, there exist several pending challenges to the prospect of the accurate and cost-efficient approaches that can really relieve the application

burden for developers. Specifically, they can be summarized in the perspectives of systematicness, effectiveness, completeness, and practicability, respectively.

1) Systematicness: Systematicness refers to whether related studies are carried out according to the same criteria, and whether there are progressive correlations among the different approaches. Systematic researches can help speed up the research process. Based on the investigated papers, we found that the research of static analysis warnings mitigation approaches is chaotic and lacks a coherent research process. Specifically, in terms of systematicness, there are three main challenges in the current FPM studies.

Ambiguous concept definition. Generally, false positive warnings refer to the instances that do not indicate real software defects while true positive refer the instances that reveal real software defects that should be fixed. Note that, the definition of true positive can bring confusion in understanding. If it is defined as any code that contains defects, the warnings that identify bad practice or places where a bug that could be easily be introduced cannot be covered. Alternatively, defining true positive as any warning identified by SA tools would erroneously contain the situations where a developer writes code that performs as intended and does not wish to modify. Thus, a concept similar to true positive has been introduced and used by many researchers, called actionable warnings. For example, Ruthruff et al. [24] pointed out that some warnings were not always acted on by developers even if they indicate true defects because the defects are considered as "trivial" with no impact on the user. Similarly, Hanam et al. [53] also only considered the warnings that developers would act on as actionable warnings. As can be seen, the difference is that actionable warnings provide clear instructions to respond to threats effectively. Such ambiguity in concept definition and usage will result in inconsistent or even opposite conclusions when comparing different studies. Therefore, researchers should clearly point out the concept definition in future studies.

Imbalanced research objects. The diversity of programming languages and corresponding Static Analysis (SA) tools has resulted in a scattered landscape of research objects. According to the statistics obtained in RQ3.1, current studies predominantly concentrate on two programming languages, namely Java and C/C++, and three SA tools, namely FindBugs, TECA, and PMD. This disproportionate distribution of research efforts has led to an imbalance in the progress made across different research objects. In contrast, other research objects have experienced relatively slower progress, and some may even lack any relevant research altogether. This discrepancy can be attributed to the availability of a plethora of SA tools (more than 10) for the two dominant programming languages. Additionally, the cumulative effect plays a significant role, as previously popular SA tools tend to attract more attention from subsequent researchers. Based on our statistical results, we highlight the need for future research to focus on languages such as PHP and C#, as well as SA tools like PolySpace. Researchers can use the level of research enthusiasm surrounding a particular SA tool as an indicator of its value. This information can help guide their decision-making process, allowing them to determine whether to continue using a specific tool or explore alternative options in the future.

Lack of comparative experiments. In the majority of studies focusing on providing novel approaches, there is a common limitation: they only conduct evaluation experiments for their own approaches without comparing them with existing baselines or previous approaches. This absence of comparative experiments leads to two significant limitations: isolation and confliction [93]. On one hand, different studies evaluate the effectiveness of their approaches using distinct datasets, experimental scenarios, and performance indicators. Consequently, it becomes uncertain whether a newly proposed approach indeed possesses noticeable advantages in identifying false positive warnings. On the other hand, conflicting conclusions arise from different studies due to variations in experimental details. This discrepancy can potentially confuse researchers in subsequent studies. While it is true that the uniqueness of different SA tools and their corresponding research targets cannot be ignored (as different SA tools target different types of violations and specialize in diverse properties), if possible, it is still advisable to compare newly proposed approaches with previous ones under reasonable experimental settings to demonstrate their superiority. To accomplish this, the establishment of a comprehensive methodology suitable for comparing multiple approaches, including the utilization of a universal dataset and evaluation system, is essential for future research. By addressing these limitations and implementing a robust comparative framework, researchers can enhance the reliability and validity of their findings, and foster a more cohesive and coherent understanding of the effectiveness of different approaches in the field of Static Analysis.

2) Effectiveness: Effectiveness refers to whether a false positive mitigation approach can really work and hence reduce the effort of developers. In the literature, the effectiveness of an approach can be evaluated by an appropriate evaluation system. In terms of effectiveness, there are three main challenges in the current FPM studies.

Uncertain generalization. Generalization measures how well an approach performs on unknown projects, which will be influenced by the quality of experimental datasets. According to Section VI-D and Section VII.B.3, there is a lack of large public benchmark datasets in the literature. Since the existing benchmarks do not meet the requirements or other reasons, the existing studies usually collected distinct research-specific projects as the validation dataset to evaluate the performance of their approaches. These datasets often consist of few software projects and contain a small number of instances, so they cannot provide a good guarantee for the generalization of the newly proposed approaches. Therefore, researchers are suggested to collect large scale public benchmark datasets for effectiveness evaluation. In addition, some wider corpus of labeled benchmarks (e.g., Semgrep[8], and WebGoat[9]) have not be used in the literature even if these can also be used to estimate the false positive rate of a tool.

Incomplete evaluation scenarios. For some FPM approaches (e.g., [6], [7]), a detailed experiment was not conducted to verify

[8]https://github.com/returntocorp/semgrep
[9]https://owasp.org/www-project-webgoat/

the usefulness of the corresponding approaches. Researchers will not be able to confirm whether the approach is really effective on real projects. Given the incomplete evaluation scenarios (*Case Study* and *None*), detailed scenario *Experiment* should be conducted as far as possible in future researches.

Insufficient performance indicators. As indicated in Table IX, a large number of papers have only utilized a limited subset of performance indicators during their evaluation process. Specifically, for classification indicators, Precision and Recall were frequently employed, while other metrics such as AUC were overlooked in many studies. Similarly, most ranking indicators were only utilized in a small portion of the research. In some cases (e.g., [79]), no indicators were used at all, and only a few examples were presented to demonstrate effectiveness. To be fair, it is reasonable for certain works (e.g., [2], [10]) to use only a few indicators that are relevant to their specific scenarios. However, it is important to recognize that employing a broader range of indicators can lead to a more comprehensive evaluation. The absence of certain indicators can hinder subsequent researchers' understanding of previous approaches and may raise doubts about their actual performance. Therefore, it is crucial for future studies to consider and calculate a sufficient number of performance indicators. In particular, it is necessary to utilize additional indicators to validate conclusions from previous papers that did not consider important metrics. By doing so, researchers can ensure a more thorough evaluation and provide a clearer understanding of the performance of different approaches in the field of Static Analysis.

3) Completeness: Completeness refers to whether all related fields or possible solutions are fully exploited to boost the mitigation for false positive warnings in the literature. Expanding multiple research avenues is necessary because it can lead to new discoveries and heuristics. In terms of completeness, there are two main challenges in the current FP warning mitigation studies.

Lack of just-in-time studies. In actual software development activities, a software project can produce many revisions over time. Note that, most of the code are the same in the adjacent revisions, and only a small portion of the code are newly submitted into the project. By only checking the changed code each time, developers can avoid receiving a large number of duplicate unimportant warnings that have been reported in previous revisions. In the literature, the researches that focus on the newly submitted codes are called as just-in-time studies. There have been several studies [141], [146] in this area, but more attention is needed. Therefore, more just-in-time researches are recommended in future studies.

Incomprehension on internal of SA tools. The current approaches are mostly post-processing techniques, which aimed to classify or rank the static analysis warnings generated by various tools. They usually do not care about the implementation details inside the tools. Note that, a large number of FP warnings are generated because the analysis process of these tools is based on conservative approximations. It is possible to reduce the false positives if the internal details can be investigated, comprehended and then adjusted properly. Therefore, researchers are recommended to mitigate false positives by investigating the internal of SA tools, as the work of [150].

4) Practicability: Practicability refers to whether current research achievements can be easily and quickly applied into practice. Notably, the important value of academic research is that it can provide convenience to developer in practice. In terms of practicability, there are two main challenges in the current FP warning mitigation studies.

Low explainability. Current false positive mitigation approaches only output classification or ranking results for the warnings. We notice that, in addition to these results, developers are more concerned whether these approaches can provide the corresponding interpretations so that they can understand the causes and hence make corresponding mitigation actions more easily. For example, Vassallo et al. [111] pointed out that the advantages of SA tools were overshadowed by the low understandability of their generated warnings and the lack of automated fix suggestions. Therefore, more studies towards to the explainability of static analysis warnings would be expected in future studies.

Not replicable studies. Although a large number of studies have been conducted in the literature, the research in this area has been hindered to some extent by the fact that many of them have not published the experimental data and code involved in their research. Therefore, many existing results or conclusions cannot be replicated by later researchers. It should be noticed that, reproducibility is very important in any field of research, because only replicable studies are truly useful for real-world practices. Therefore, in future studies, researchers are suggested to publish their experimental data and source code so that the research achievements can be really applied to the SQA activities in the industry.

B. RQ5.2: What Are Potential Opportunities for Future Studies?

In addition to the above challenges that should be paid more attention to, more accurate and understandable FPM approaches can be further explored with the rise of new technologies. In the following, we discuss possible opportunities in the future FPM studies.

In the future, it is crucial for researchers to pay closer attention to ML-based techniques. Specifically, there are promising opportunities in the areas of natural language processing (NLP), deep learning (DL), and big data analysis and parallel computing (BDA-PC). The exploration of ML-based techniques is motivated by three key reasons. First, previous studies [60], [76] have demonstrated the high effectiveness of ML-based approaches in the field of FPM. It is important to note that the quality of training data plays a significant role in achieving satisfactory results. Therefore, a key aspect of building excellent ML-based models lies in acquiring good training data and extracting relevant features, which can be further enhanced through the application of NLP techniques. Second, ML-based approaches offer notable advantages in terms of flexibility, particularly in their ability to continuously update themselves with newly obtained training data. This continuous learning process

enables the models to improve their robustness by incorporating massive amounts of new data, without requiring manual modifications to the model structure. To fully leverage this potential, BDA-PC is suggested as an additional opportunity for effectively processing and analyzing vast amounts of data. Third, recent years have witnessed significant advancements in ML-based approaches, particularly in DL-based techniques, which have gained substantial popularity. Traditional approaches are increasingly benefiting from and integrating with ML-based technologies. For instance, DL-based large language models (LLMs) such as ChatGPT [186] have demonstrated remarkable capabilities in handling SE tasks, comparable to human performance. Inspired by these advancements, DL techniques are highly recommended for further exploration in the context of FPM tasks.

1) Leveraging NLP Techniques to Generate Semantically Rich Program Representations: **Background.** Natural language processing (NLP) techniques were popularly used to solve the tasks of text representation [148], program comprehension [142], information retrieval [163], and software maintenance [162] due to apparent advantages on analyzing and mining text data written in unstructured natural language. In the field of SQA, NLP techniques are often used as a preprocessing step. Specifically, they are first leveraged to transform program source codes or related text data (e.g., code comments [165]) into formal abstract representations. Based on the representations, approaches were then presented to solve various concrete SQA issues such as vulnerabilities detection [162], bug localization [163], and duplicated bug reports identification [164]. Recently, to retain the rich syntax and semantic information of text data, researchers proposed a large number of distributed NLP models [144], [148], [153], [158] called text embedding to encode each natural language text into a corresponding real value vector representation. According to different embedding granularity, researchers can obtain the vector representation of a single word, a phrase, a sentence, or a document. In addition to encoding ordinary text, many programs related elements, such as code changes [153], AST path [156], and graph node [154], can also be represented in an embedding way to facilitate the source code analysis tasks.

Opportunity. To identify FP warnings accurately, researchers usually need to mine the characteristics of source codes that are linked with each FP warning instance. Current studies often extracted human summarized trivial attributes, such as execution likelihood [7], code churn [24], and cyclomatic complexity [84] as source code characteristics. The ability of these characteristics to distinguish FP warnings is insufficient because there is no significant correlation between a trivial characteristic and FP warnings.

In this scenario, the utilization of embedding-based NLP models presents a compelling opportunity to overcome the limitations associated with human-derived summarization of characteristics. By embedding each source code snippet into a vector representation, such as code2vec [158], the resulting representation encompasses valuable semantic information. This technique has garnered significant attention in the field of software engineering (SE), with code2vec embedding proving advantageous for various SE tasks, including bad smell detection [182], duplicated code detection [183], software system comparison [184], and source code classification [185].

Intuitively, the integration of advanced embedding information provides valuable advantages in establishing meaningful connections between source codes and the classification of warnings. Researchers can opt to utilize these advanced embedding characteristics either independently or in conjunction with other pertinent attributes to construct prediction models or conduct extensive program analysis. Given the substantial effectiveness of embedding-based NLP techniques in accurately representing program elements [158], [172], we wholeheartedly urge researchers to explore the potential of these techniques for effectively mitigating FP warnings and enhancing their mitigation strategies.

2) Employing DL Techniques to Build Distinguished Warning Prediction Models: **Background.** Deep learning (DL) refers a set of deep neural network-based machine learning algorithms, which emphasizes the depth of the model structure and highlights the importance of automatic feature learning. Although Yang et al.'s work [118] indicated that a DL-based model is not superior than simple SVM model, a recent study [173] revealed that there is a serious data leakage issue with the "Golden" feature dataset used by Yang et al. This finding means that the conclusion provided by Yang et al. is unreliable and the ability of DL-based models needs to be further investigated.

Notably, DL-based models have gained tremendous success in handling various SE tasks especially for software quality assurance tasks. For example, Wen et al. [149] designed a recurrent neural network (RNN) model to encode features from sequence data automatically to perform defect prediction. Ren et al. [165] utilized a CNN model to identify self-admitted technical debts by capturing lexical gaps in text data. To understand software vulnerabilities, Zekany et al. [159] leveraged deep sequence learning models to statically analyze runtime behavior by learning sophisticated relationships between sequences of inputs. In addition, several studies relating DL-based static analyses [160], [161] also showed that DL-based models usually performed better than the other types of models.

Opportunity. The advantages of using deep learning models for modeling are threefold. First, DL based models can automatically learn meaningful lexical patterns from source code, which will reduce large amounts of human effort spent to manually summarize limited warning features. Second, compared with traditional machine learning algorithms, DL based models can generally output more accurate prediction results due to exquisite structural design. Last but not least, DL based models can provide some interpretable hints for the prediction result. For example, in the task of self-admitted technical debt (a special type of comment) identification, Ren et al. [165] proposed a CNN model to classify these comments and leveraged the CNN model to extract patterns (i.e., representative keywords) as an interpretation for the its high performance. Besides, recent DL based large language models such as GPT-4 have excellent contextual information combination ability and personified information output ability, which can effectively output richer and easier to understand problem explanation messages based on the static inspection of problem output, and solve the problem that

users see the report but do not know how to modify it. However, there is also a limitation (i.e., huge resource and time cost) that need to be noted when applying DL based models. As can be seen in Section IV, a large number of FPM approaches used machine learning algorithms to build prediction models directly (Section IV-E) or indirectly. Note that, most approaches leveraged traditional ML algorithms (e.g., LR [24], and KNN [31]) rather than deep learning algorithms to build prediction models. In the literature, there are only few explorations on building FP warnings prediction models based on deep learning [115], [119]. For example, Lee et al. [101] trained a CNN model to learn and detect the specific lexical patterns in codes. Similarly, Liu et al. [115] also leveraged a CNN model to mine the common warning patterns and their corresponding fix patterns. However, the great potential of DL-based models has not been fully exploited yet because some important model parameters need to be carefully analyzed and tuned. Additionally, with the rapid development of DL techniques, many novel DL models have been leveraged for SE tasks [138], [151], but these models have not been applied to the FPM task. Therefore, researchers are recommended to develop more advanced DL-based approaches to perform more accurate FP warning prediction.

3) Applying BDA-PC Techniques to Conduct More Sophisticated Program Code Analysis: **Background.** In the era of big data, there is a large number of external open-source software and defect data available on the web (e.g., Github) that can be used for program analysis and model learning. For example, Microsoft Corporation leveraged metadata associated with the historical commits in Github[10] to create a custom classification model (called Code Defect AI) to predict potential bugs. With the help of massive data, the constructed model showed a robust prediction performance. However, the disadvantage of leveraging a large amount of data is that it consumes a large amount of computing resources, resulting in slow modeling time. Correspondingly, big data analysis and parallel computing (BDA-PC) techniques are proposed to make it possible to use huge amounts of data for analysis. There have been exciting achievements on program static analysis based on big data and parallel computing techniques, into which there could be some important assistances for extending FPM approaches [123], [129], [130], [137], [140], [156], [157].

Opportunity. As is known, the conservative approximations analysis of source code semantics is the most critical factor that leads to a high FP rate for SA tools [63], [90]. Therefore, a possible research opportunity is to conduct more sophisticated static program analysis procedures to filter out FP warnings. For example, researchers can conduct sophisticated interprocedural analysis based on large-scale program data and then mine exact bug patterns associated with various types of warnings. Obviously, the analysis process for large-scale modern software is highly computation- and memory-intensive, leading to poor efficiency and scalability. To solve above issues, Gu et al. [123] introduced an efficient distributed and scalable solution for sophisticated static analysis. Zuo et al. [140] built an efficient interprocedural static analysis engine in the cloud to tackle the scalability problem. By drawing on these existing solutions, it is possible to reduce FP warnings under sophisticated static program analysis. Meanwhile, other complex static analysis such as abstract interpretation and constraint-based analysis can also be explored. On the whole, big data analysis can improve the accuracy the static analysis while parallel computing technique can accelerate the whole analysis process. Therefore, introducing big data analysis and parallel computing into future FPM approaches is imperative in a way.

[10]https://www.microsoft.com/en-us/ai/ai-lab-code-defect

Summary of results for RQ5:

1) There were 10 different unsolved challenges for the existing FPM researches, which were distributed in the aspects of systematicness, effectiveness, completeness, and practicability, respectively.
2) With the development of new techniques, researchers have the opportunities to build novel FPM approaches based on natural language processing (NLP), deep learning (DL), and big data analysis and parallel computing (BDA-PC), respectively.

IX. Threats to Validity

Although we do our best to ensure the accuracy and completeness of the survey, there may still be two inevitable threats (i.e., bibliography selection and data analysis) to validity just as any other surveys (e.g., [138], [151]).

A. Bibliography Selection Validity

Bibliography selection validity refers to whether all relevant papers have been included in this survey. A survey will be considered as incomplete if it does not guarantee the inclusion of all the relevant papers in the field. To mitigate this threat, we took the following three measures: 1) we started the retrieval process on four popular databases (e.g., IEEE Xplore digital library), which contain most of the major papers in the field of computer science. 2) We selected the combinations ($30 = 6 \times 1 \times 1 \times 5$) of four groups of terms as the retrieval keywords to mitigate missed papers introduced by various aliases of static analysis warnings. 3) We conducted a forward snowballing and manual inclusion process of selected papers to ensure we can gather other related papers that were not included in previous retrieval process. All the above effort has also been applied in other surveys (e.g., [151]), which can ensure that the number of missed surveyed papers as small as possible.

B. Data Analysis Validity

Data analysis validity refers to whether the selected papers have been classified and introduced in an objective way. A bias in the process of misclassification of papers or a subjective interpretation of the extracted contents can reduce the quality of a survey. To mitigate this threat, multiple authors were involved in the survey writing process. More specifically, each author firstly reviewed the selected papers and made an initial

classification independently. After that, we created a discussion group to communicate each authors' classification results and confirm the final results. If we were unable to reach an agreement on classification, we would invite another researcher to join in and resolve differences. Next, the first author set about writing the survey and all other authors carefully reviewed the relevant descriptions. All improper descriptions have been refactored as far as possible. After multiple authors' collaboration, the data analysis validity to this study was considered to be mitigated to a large extent.

X. Conclusion

This survey investigates the progresses and achievements on the issue of FPM for static analysis warnings. Given the collected 130 surveyed papers, the following key insights can be concluded. First, according to the rationale of each approach, the current FPM approaches can be divided into five categories: statistical probability, static program analysis, dynamic program testing, machine learning, and clustering. Second, the used evaluation systems from the perspective of studied SA tools, evaluation scenarios, and performance indicators, and datasets show a huge difference among the surveyed papers. Third, the existing empirical studies related to SA warnings illustrated in terms of four categories: investigating SA tools, warnings, building benchmark datasets, and evaluation for FPM approaches. Finally, from the standpoint of systematicness, effectiveness, completeness, and practicability, 10 different challenges unresolved in existing studies can be summarized and the NLP, DL and BDA-PC are three recommended techniques for future opportunities.

Based on the current research status, we believe that SA warnings will continue receiving attention in the upcoming years. As an immediate future, we plan on centralizing our effort on how to mitigate FP warnings in an easy, understandable, and practical way, which can facilitate the transformation from academic achievements into practical technology.

Acknowledgment

We are very grateful to anonymous reviewers and the editor for their very insightful comments and very helpful suggestions, which dynamically improve the quality of our manuscript.

References

[1] T. Kremenek and D. R. Engler, "Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Proc. 10th Int. Static Anal. Symp. (SAS)*, 2003, pp. 295–315.

[2] T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler, "Correlation exploitation in error ranking," in *Proc. 12th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2004, pp. 83–93.

[3] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis," in *Proc. 12th Int. Static Anal. Symp. (SAS)*, 2005, pp. 203–217.

[4] X. Rival, "Understanding the origin of alarms in Astrée," in *Proc. 12th Int. Static Anal. Symp. (SAS)*, 2005, pp. 303–319.

[5] X. Rival, "Abstract dependences for alarm diagnosis," in *Proc. 3rd Asian Program. Lang. Syst. Symp. (APLAS)*, 2005, pp. 347–363.

[6] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, 2006, pp. 343–350.

[7] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation (SCAM)*, 2006, pp. 149–160.

[8] P. Cousot et al., "Combination of abstractions in the Astrée static analyzer," in *Proc. 11th Asian Comput. Sci. Conf. (ASIAN)*, 2006, pp. 272–300.

[9] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2007, pp. 1–8.

[10] S. S. Heckman, "Adaptive probabilistic model for ranking code-based static analysis alerts," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE Companion)*, 2007, pp. 89–90.

[11] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *ACM Crossroads*, vol. 14, no. 1, pp. 1–11, 2007.

[12] D. Hovemeyer and W. W. Pugh, "Finding more null pointer bugs, but not too many," in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2007, pp. 9–14.

[13] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2007, pp. 45–54.

[14] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Proc. 4h Int. Workshop Mining Softw. Repositories (MSR)*, 2007, p. 27.

[15] D. Kong, Q. Zheng, C. Chen, J. Shuai, and M. Zhu, "ISA: A source code static vulnerability detection system based on data fusion," in *Proc. 2nf Int. Conf. Scalable Inf. Syst. (Infoscale)*, 2007, p. 55.

[16] L. Layman, L. A. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2007, pp. 176–185.

[17] M. Sherriff, S. S. Heckman, J. M. Lake, and L. A. Williams, "Using groupings of static analysis alerts to identify files likely to contain field failures," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2007, pp. 565–568.

[18] D. Delmas and J. Souyris, "Astrée: From research to industry," in *Proc. 14th Int. Static Anal. Symp. (SAS)*, 2007, pp. 437–451.

[19] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-crasher: A hybrid analysis tool for bug finding," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, pp. 8:1–8:37, 2008.

[20] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electron. Notes Theor. Comput. Sci.*, vol. 217, pp. 5–21, Jul. 2008.

[21] S. Smith Heckman and L. A. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proc. 2nd Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2008, pp. 41–50.

[22] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2008, pp. 188–197.

[23] N. Rungta and E. G. Mercer, "A meta heuristic for effectively detecting concurrency errors," in *Proc. 4th Int. Haifa Verification Conf. (HVC)*, 2008, pp. 23–37.

[24] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. G. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proc. 30th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 341–350.

[25] N. Ayewah and W. Pugh, "Using checklists to review static analysis warnings," in *Proc. 2nd Int. Workshop Defects Large Softw. Syst. (DEFECTS)*, 2009, pp. 11–15.

[26] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, 2009, pp. 75–86.

[27] S. S. Heckman and L. A. Williams, "A model building process for identifying actionable static analysis alerts," in *Proc. 2nd Int. Conf. Softw. Testing Verification Validation (ICST)*, 2009, pp. 161–170.

[28] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring Prediction," in *Proc. 2nd Int. Conf. Softw. Testing Verification Validation (ICST)*, 2009, pp. 141–150.

[29] D. Baca, "Identifying security relevant warnings from static code analysis tools through code tainting," in *Proc. 5th Int. Conf. Availability, Rel. Secur. (ARES)*, 2010, pp. 386–390.

[30] Y. Kim, J. Lee, H. Han, and K.-M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Inf. Softw. Technol.*, vol. 52, no. 2, pp. 210–219, 2010.

[31] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *Proc. 25th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2010, pp. 93–102.

[32] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, 2010, vol. 2, pp. 99–108.

[33] J. J. Li, J. D. Palframan, and J. Landwehr, "SoftWare IMmunization (SWIM) - A combination of static analysis and automatic testing," in *Proc. 35th Annu. IEEE Int. Comput. Softw. Appl. Conf. (COMPSAC)*, 2011, pp. 656–661.

[34] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "DyTa: Dynamic symbolic execution guided with static verification results," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, 2011, pp. 992–994.

[35] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective error ranking for FindBugs," in *Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation (ICST)*, 2011, pp. 299–308.

[36] A. Vetro, M. Morisio, and M. Torchiano, "An empirical validation of FindBugs issues related to defects," in *Proc. 15th Annu. Conf. Eval. Assessment Softw. Eng. (EASE)*, 2011, pp. 144–153.

[37] S. Allier, N. Anquetil, A. C. Hora, and S. Ducasse, "A framework to compare alert ranking algorithms," in *Proc. 19th Work. Conf. Reverse Eng. (WCRE)*, 2012, pp. 277–285.

[38] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand, "Program slicing enhances a verification technique combining static and dynamic analysis," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2012, pp. 1284–1291.

[39] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh, "Precise analysis of large industry code," in *Proc. 19th Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2012, pp. 306–309.

[40] S. Joshi, S. K. Lahiri, and A. Lal, "Underspecified harnesses and interleaved bugs," in *Proc. 39th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2012, pp. 19–30.

[41] G. Liang, Q. Wu, Q. Wang, and H. Mei, "An effective defect detection and warning prioritization approach for resource leaks," in *Proc. 36th Annu. IEEE Comput. Softw. Appl. Conf. (COMPSAC)*, 2012, pp. 119–128.

[42] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: A case study," in *Proc. 50th Annu. Southeast Regional Conf. (SRC)*, 2012, pp. 359–360.

[43] C. Chen, K. Lu, X. Wang, X. Zhou, and L. Fang, "Pruning false positives of static data-race detection via thread specialization," in *Proc. Adv. 10th Int. Symp. Parallel Process. Technol. (APPT)*, 2013, pp. 77–90.

[44] Z. P. Fry and W. Weimer, "Clustering static analysis defect reports to reduce maintenance costs," in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, 2013, pp. 282–291.

[45] S. Smith Heckman and L. A. Williams, "A comparative evaluation of static analysis actionable alert identification techniques," in *Proc. 9th Int. Conf. Predictive Models Softw. Eng. (PROMISE)*, 2013, pp. 4:1–4:10.

[46] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 672–681.

[47] M. Li, Y. Chen, L. Wang, and G. Xu, "Dynamically validating static memory leak warnings," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, 2013, pp. 112–122.

[48] T. B. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Proc. 13th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2013, pp. 106–115.

[49] T. Muske, A. Datar, M. Khanzode, and K. Madhukar, "Efficient elimination of false positives using bounded model checking," in *Proc. 5th Int. Conf. Adv. Syst. Testing Validation Lifecycle (VALID)*, 2013, pp. 13–20.

[50] U. Yuksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, 2013, pp. 532–535.

[51] D. Zhang, D. Jin, Y. Gong, and H. Zhang, "Diagnosis-oriented alarm correlations," in *Proc. 20th Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2013, pp. 172–179.

[52] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *Proc. 6th Int. Workshop Empirical Softw. Eng. Pract. (IWESEP)*, 2014, pp. 37–42.

[53] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proc. 11th Work. Conf. Mining Softw. Repositories (MSR)*, 2014, pp. 152–161.

[54] I. Medeiros, N. Ferreira Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proc. 23rd Int. World Wide Web Conf. (WWW)*, 2014, pp. 63–74.

[55] T. Muske, "Supporting reviewing of warnings in presence of shared variables: Need and effectiveness," in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSRE Workshops)*, 2014, pp. 104–107.

[56] T. Muske, "Improving review of clustered-code analysis warnings," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2014, pp. 569–572.

[57] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu, "Comparing static bug finders and statistical prediction," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 424–434.

[58] O. Tripp, S. Guarnieri, M. Pistoia, and A. Y. Aravkin, "ALETHEIA: Improving the usability of static security analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Scottsdale (CCS)*, 2014, pp. 762–774.

[59] M. Valdiviezo, C. Cifuentes, and P. Krishnan, "A method for scalable and precise bug finding using program analysis and model checking," in *Proc. 12th Asian Symp. Program. Lang. Syst. (APLAS)*, 2014, pp. 196–215.

[60] J. Yoon, M. Jin, and Y. Jung, "Reducing false alarms from an industrial-strength static analyzer by SVM," in *Proc. 21st Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2014, pp. 3–6.

[61] U. Yuksel, H. Sözer, and M. Sensoy, "Trust-based fusion of classifiers for static code analysis," in *Proc. 17th Int. Conf. Inf. Fusion (FUSION)*, 2014, pp. 1–6.

[62] C. Antônio de Araújo, M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, "Investigating the correspondence between mutations and static warnings," in *Proc. 29th Brazilian Symp. Softw. Eng. (SBES)*, 2015, pp. 1–10.

[63] B. Chimdyalwar, P. Darke, A. Chavda, S. Vaghani, and A. Chauhan, "Eliminating static analysis false positives using loop abstraction and bounded model checking," in *Proc. 20th Int. Symp. Formal Methods (FM)*, 2015, pp. 573–576.

[64] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proc. 10th Joint Meeting Found. Softw. Eng. (FSE)*, 2015, pp. 462–473.

[65] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *Proc. 26th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2015, pp. 270–280.

[66] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Proc. 22nd IEEE Int. Conf. Softw. Anal., Evol., Reengineering (SANER)*, 2015, pp. 161–170.

[67] S. Salvi, D. Kästner, C. Ferdinand, and T. Bienmüller, "Exploiting synergies between static analysis and model-based testing," in *Proc. 11th Eur. Dependable Comput. Conf. (EDCC)*, 2015, pp. 13–24.

[68] H. Sözer, "Integrated static code analysis and runtime verification," *Softw. Pract. Exp.*, vol. 45, no. 10, pp. 1359–1373, 2015.

[69] A. E. Burhandenny, H. Aman, and M. Kawahara, "Examination of coding violations focusing on their change patterns over releases," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf. (APSEC)*, 2016, pp. 121–128.

[70] C. Dimastrogiovanni and N. Laranjeiro, "Towards understanding the value of false positives in static code analysis," in *Proc. 7th Latin-Amer. Symp. Dependable Comput. (LADC)*, 2016, pp. 119–122.

[71] T. Muske and U. P. Khedker, "Cause points analysis for effective handling of alarms," in *Proc. 27th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2016, pp. 173–184.

[72] J.-P. Ostberg and S. Wagner, "At ease with your warnings: The principles of the salutogenesis model applied to automatic static analysis," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reengineering (SANER)*, 2016, pp. 629–633.

[73] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of JavaScript web applications in the wild," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 61–70.

[74] T. Buckers et al., "UAV: Warnings from multiple automated static analysis tools at a glance," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2017, pp. 472–476.

[75] F. Cheirdari and G. Karabatis, "On the verification of software vulnerabilities during static code analysis using data mining techniques," in *Proc. On Move Meaningful Int. Syst. (OTM)*, 2017, pp. 99–106.

[76] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proc. 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 519–529.

[77] W. Lee, W. Lee, D. Kang, K. Heo, H. Oh, and K. Yi, "Sound non-statistical clustering of static analysis Alarms," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 16:1–16:35, 2017.

[78] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proc. 1st ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang. (MAPL@PLDI)*, 2017, pp. 35–42.

[79] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill, "Identifying and documenting false positive patterns generated by static code analysis tools," in *Proc. 4th IEEE/ACM Int. Workshop Softw. Eng. Res. Ind. Pract. (SER&IP@ICSE)*, 2017, pp. 55–61.

[80] L. Wei, Y. Liu, and S.-C. Cheung, "OASIS: Prioritizing static analysis warnings for Android apps based on app user reviews," in *Proc. 11th Joint Meeting Found. Softw. Eng. (FSE)*, 2017, pp. 672–682.

[81] A. Xypolytos, H. Xu, B. Vieira, and A. M. T. Ali-Eldin, "A framework for combining and ranking static analysis tool findings based on tool performance statistics," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS)*, 2017, pp. 595–596.

[82] M. Yan, X. Zhang, L. Xu, H. Hu, S. Sun, and X. Xia, "Revisiting the correlation between alerts and software defects: A case study on MyFaces, Camel, and CXF," in *Proc. 41st IEEE Annu. Computer Softw. Appl. Conf. (COMPSAC)*, 2017, pp. 103–108.

[83] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Proc. 14th Int. Conf. Mining Softw. Repositories (MSR)*, 2017, pp. 334–344.

[84] E. A. Alikhashashneh, R. R. Raje, and J. H. Hill, "Using machine learning techniques to classify and predict static code analysis tool warnings," in *Proc. 15th IEEE/ACS Int. Conf. Comput. Syst. Appl. (AICCSA)*, 2018, pp. 1–8.

[85] F. Cheirdari and G. Karabatis, "Analyzing false positive source code vulnerabilities using static analysis tools," in *Proc. IEEE Int. Conf. Big Data (BigData)*, 2018, pp. 4782–4788.

[86] L. Flynn et al., "Prioritizing alerts from multiple static analysis tools, using classification models," in *Proc. 1st Int. Workshop Softw. Qual. Dependencies (SQUADE@ICSE)*, 2018, pp. 13–20.

[87] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 317–328.

[88] T. Muske, R. Talluri, and A. Serebrenik, "Repositioning of static analysis alarms," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2018, pp. 187–197.

[89] L.-P. Querel and P. C. Rigby, "WarningsGuru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (FSE)*, 2018, pp. 892–895.

[90] A. Ribeiro, P. Meirelles, N. Lago, and F. Kon, "Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories," in *Proc. 14th IFIP WG 2.13 Int. Conf. Open Source Syst., Enterprise Softw. Solutions (OSS)*, 2018, pp. 90–101.

[91] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[92] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *Proc. 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2018, pp. 38–49.

[93] J. Wang, S. Wang, and Q. Wang, "Is there a 'golden' feature set for static warning identification? An experimental evaluation," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2018, pp. 17:1–17:10.

[94] H. Wang, M. Zhou, X. Cheng, G. Chen, and M. Gu, "Which defect should be fixed first? Semantic prioritization of static analysis report," in *Proc. 8th Int. Conf. Softw. Anal., Testing, Evol. (SATE)*, 2018, pp. 3–19.

[95] B. Aloraini, M. Nagappan, D. M. Germán, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *J. Syst. Softw.*, vol. 158, pp. 1–25, 2019.

[96] J. Giet, L. Mauborgne, D. Kästner, and C. Ferdinand, "Towards zero alarms in sound static analysis of finite state machines," in *Proc. 38th Int. Conf. Comput. Saf., Rel., Secur. (SAFECOMP)*, 2019, pp. 3–18.

[97] N. Imtiaz, A. Rahman, E. Farhana, and L. A. Williams, "Challenges with responding to static analysis tool alerts," in *Proc. 16th Int. Conf. Mining Softw. Repositories (MSR)*, 2019, pp. 245–249.

[98] N. Imtiaz, B. Murphy, and L. A. Williams, "How do developers act on static analysis alerts? An empirical study of Coverity usage," in *Proc. 30th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2019, pp. 323–333.

[99] N. Imtiaz and L. A. Williams, "A synopsis of static analysis alerts on open source software," in *Proc. 6th Annu. Symp. Hot Topics Sci. Secur. (HotSoS)*, 2019, pp. 12:1–12:3.

[100] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a Java static analysis tool," in *Proc. 12th IEEE Conf. Softw. Testing, Validation Verification (ICST)*, 2019, pp. 288–299.

[101] S. Lee, S. Hong, J. Yi, T. Kim, C.-J. Kim, and S. Yoo, "Classifying false positive static checker alarms in continuous integration using convolutional neural networks," in *Proc. 12th IEEE Conf. Softw. Testing, Validation Verification (ICST)*, 2019, pp. 391–401.

[102] D. Marcilio, R. Bonifácio, E. Monteiro, E. D. Canedo, W. Pinheiro Luz, and G. Pinto, "Are static analysis violations really fixed? A closer look at realistic usage of SonarQube," in *Proc. 27th Int. Conf. Program Comprehension (ICPC)*, 2019, pp. 209–219.

[103] T. Muske, R. Talluri, and A. Serebrenik, "Reducing static analysis alarms based on non-impacting control dependencies," in *Proc. 17th Asian Symp. Program. Lang. Syst. (APLAS)*, 2019, pp. 115–135.

[104] T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita, and I. Yamada, "Reducing false positives of static analysis for SEI CERT C coding standard," in *Proc. Joint 7th Int. Workshop Conducting Empirical Stud. Ind. 6th Int. Workshop Softw. Eng. Res. Ind. Pract. (CESSER-IP@ICSE)*, 2019, pp. 41–48.

[105] J. D. Pereira, J. R. Campos, and M. Vieira, "An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools," in *Proc. 9th Latin-Amer. Symp. Dependable Comput. (LADC)*, 2019, pp. 1–10.

[106] A. Ribeiro, P. Meirelles, N. Lago, and F. Kon, "Ranking warnings from multiple source code static analyzers via ensemble learning," in *Proc. 15th Int. Symp. Open Collaboration (OpenSym)*, 2019, pp. 5:1–5:10.

[107] J. Yang, L. Tan, J. Peyton, and K. A. Duer, "Towards better utilizing static application security testing," in *Proc. 41st Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, 2019, pp. 51–60.

[108] L. C. Júnior, A. Belgamo, V. Rafael Lobo de Mendonça, and A. M. R. Vincenzi, "WarningsFIX: A recommendation system for prioritizing warnings generated by automated static analyzers," in *Proc. 19th Brazilian Symp. Softw. Qual. (SBQS)*, 2020, p. 26.

[109] F. Gao, Y. Wang, L. Wang, Z. Yang, and X. Li, "Automatic buffer overflow warning validation," *J. Comput. Sci. Technol.*, vol. 35, no. 6, pp. 1406–1427, 2020.

[110] T. Muske and A. Serebrenik, "Techniques for efficient automated elimination of false positives," in *Proc. 20th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2020, pp. 259–263.

[111] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Softw. Eng.*, vol. 25, no. 2, pp. 1419–1457, 2020.

[112] Y. Zhang, Y. Xing, Y. Gong, D. Jin, H. Li, and F. Liu, "A variable-level automated defect identification model based on machine learning," *Soft Comput.*, vol. 24, no. 2, pp. 1045–1061, 2020.

[113] Y. Zhang, D. Jin, Y. Xing, and Y. Gong, "Automated defect identification via path analysis-based features with transfer learning," *J. Syst. Softw.*, vol. 166, Aug. 2020, Art. no. 110585.

[114] L. Flynn, W. Snavely, and Z. Kurtz, "Test suites as a source of training data for static analysis alert classifiers," in *Proc. 2nd IEEE/ACM Int. Conf. Automat. Softw. Test (AST@ICSE)*, 2021, pp. 100–108.

[115] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for FindBugs violations," *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 165–188, Jan. 2021.

[116] M. G. Siavvas, I. Kalouptsoglou, D. Tsoukalas, and D. D. Kehagias, "A self-adaptive approach for assessing the criticality of Security-related static analysis alerts," in *Proc. 21st Int. Conf. Comput. Sci. Appl. (ICCSA)*, 2021, pp. 289–305.

[117] X. Yang, Z. Yu, J. Wang, and T. Menzies, "Understanding static code warnings: An incremental AI approach," *Expert Syst. Appl.*, vol. 167, Apr. 2021, Art. no. 114134.

[118] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empirical Softw. Eng.*, vol. 26, no. 3, 2021, Art. no. 56.
[119] Y. Zheng et al., "D2A: A dataset built for AI-based vulnerability detection methods using differential analysis," in *Proc. 43rd IEEE/ACM Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, 2021, pp. 111–120.
[120] S. S. Heckman and L. A. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.
[121] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *Proc. 16th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2016, pp. 157–166.
[122] "TSE-survey-FPM." GitHub. Accessed: Jul. 1, 2023. [Online]. Available: https://github.com/Naplues/TSE-Survey-FPM
[123] R. Gu et al., "Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 867–883, Apr. 2021.
[124] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Proc. 15th Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2004, pp. 245–256.
[125] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 580–586.
[126] S. E. Heckman and L. Williams, "Automated adaptive ranking and filtering of static analysis alerts," in *Proc. 17th Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2006, pp. 1–2.
[127] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs," in *Proc. Companion 21th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2006, pp. 673–674.
[128] J. Zheng, L. A. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.
[129] Z. Zuo et al., "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proc. 14th EuroSys Conf. (EuroSys)*, 2019, pp. 38:1–38:17.
[130] Z. Zuo et al., "Chianina: An evolving graph system for flow- and context-sensitive analyses of million lines of C code," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation (PLDI)*, 2021, pp. 914–929.
[131] R. Plösch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *Proc. 32nd Annu. IEEE Softw. Eng. Workshop (SEW)*, 2008, pp. 169–174.
[132] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep./Oct. 2008.
[133] A. Bessey et al., "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.
[134] N. Ayewah and W. Pugh, "The google FindBugs fixit," in *Proc. 19th Int. Symp. Softw. Testing Anal. (ISSTA)*, 2010, pp. 241–252.
[135] R. Kumar and A. V. Nori, "The economics of static analysis tools," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (FSE)*, 2010, pp. 707–710.
[136] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 931–940.
[137] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017, pp. 389–404.
[138] C. Watson, N. Cooper, D. Nader-Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," 2020, *arXiv:*2009.06520.
[139] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reengineering (SANER)*, 2016, pp. 470–481.
[140] Z. Zuo et al., "BigSpa: An efficient interprocedural static analysis engine in the cloud," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2019, pp. 771–780.
[141] L. Nguyen et al., "Just-in-time static analysis," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2017, pp. 307–317.
[142] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2Vec: Value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, pp. 233:1–233:27, 2020.
[143] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: Fixing semantic bugs with fix patterns of static analysis violations," in *Proc. 26th IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, 2019, pp. 456–467.
[144] B. Theeten, F. Vandeputte, and T. Van Cutsem, "Import2vec learning embeddings for software libraries," in *Proc. 16th Int. Conf. Mining Softw. Repositories (MSR)*, 2019, pp. 18–28.
[145] A. Trautsch, S. Herbold, and J. Grabowski, "A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open-source projects," *Empirical Softw. Eng.*, vol. 25, no. 6, pp. 5137–5192, 2020.
[146] A. Trautsch, S. Herbold, and J. Grabowski, "Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2020, pp. 127–138.
[147] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, "SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings," *J. Syst. Softw.*, vol. 168, pp. 1–20, Oct. 2020.
[148] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed representations of code changes," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 518–529.
[149] M. Wen, R. Wu, and S.-C. Cheung, "How well do change sequences predict defects? Sequence learning from software changes," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1155–1175, Nov. 2020.
[150] J. Wang, Y. Huang, S. Wang, and Q. Wang, "Find bugs in static bug finders," in *Proc. 30th Int. Conf. Program Comprehension (ICPC)*, 2021, pp. 516–527.
[151] Y. Yang, X. Xia, D. Lo, and J. C. Grundy, "A survey on deep learning for software engineering," 2020, *arXiv:*2011.14597.
[152] P. Copeland, "Google's innovation factory: Testing, culture, and infrastructure," in *Proc. 3rd Int. Conf. Softw. Testing, Verification Validation (ICST)*, 2010, pp. 11–14.
[153] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2Vec: Learning distributed representations of code changes," *SN Comput. Sci.*, vol. 2, no. 3, 2021, Art. no. 150.
[154] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, "Struc2vec: Learning node representations from structural identity," in *Proc. KDD*, 2017, pp. 385–394.
[155] K. Shi, Y. Lu, J. Chang, and Z. Wei, "PathPair2Vec: An AST path pair-based code representation method for defect prediction," *J. Comput. Lang.*, vol. 59, Aug. 2020, Art. no. 100979.
[156] B. Amen and G. Antoniou, "A theoretical study of anomaly detection in big data distributed static and stream analytics," in *Proc. 20th IEEE Int. Conf. High Perform. Comput. Commun., 16th IEEE Int. Conf. Smart City, 4th IEEE Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, 2018, pp. 1177–1182.
[157] J. Zhu, Q. Li, and S. Ying, "Failure analysis of static analysis software module based on big data tendency prediction," *Complex*, vol. 2021, pp. 1–12, Mar. 2021.
[158] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, pp. 40:1–40:29, Jan. 2019.
[159] S. Zekany, D. Rings, N. Harada, M. A. Laurenzano, L. Tang, and J. Mars, "CrystalBall: Statically analyzing runtime behavior via deep sequence learning," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, pp. 24:1–24:12.
[160] L. Liu and B. Wang, "Automatic malware detection using deep learning based on static analysis," in *Proc. 3rd Int. Conf. Pioneering Comput. Scientists, Eng. Educators (ICPCSEE)*, 2017, pp. 500–507.
[161] H. Darabian et al., "Detecting cryptomining malware: A deep learning approach for static and dynamic analysis," *J. Grid Comput.*, vol. 18, no. 2, pp. 293–303, 2020.
[162] I. Medeiros, N. Neves, and M. Correia, "Statically detecting vulnerabilities by processing programming languages as natural languages," 2019, arXiv:1910.06826.
[163] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, 2012, pp. 14–24.

[164] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, 2007, pp. 499–510.
[165] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, pp. 1–45, 2019.
[166] T. Muske and A. Serebrenik, "Survey of approaches for postprocessing of static analysis alarms," *ACM Comput. Surv.*, vol. 55, no. 3, Feb. 2022, Art. no. 48.
[167] D. Hovemeyer, J. Spacco, and W. W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2005, pp. 13–19.
[168] Z. D. Luo, L. Hillis, R. Das, and Y. Qi, "Effective static analysis to find concurrency bugs in Java," in *Proc. 10th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2010, pp. 135–144.
[169] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - Test and measurement of static code analyzers," in *Proc. 1st IEEE/ACM Int. Workshop Complex Faults Failures Large Softw. Syst. (COUFLESS)*, 2015, pp. 14–20.
[170] T. Chappell, C. Cifuentes, P. Krishnan, and S. Geva, "Machine learning for finding bugs: An initial report," in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Qual. Eval. (MaLTeSQuE@SANER)*, 2017, pp. 21–26.
[171] A. Algaith, P. J. C. Nunes, J. Fonseca, I. Gashi, and M. Vieira, "Finding SQL injection and cross site scripting vulnerabilities with diverse static analysis tools," in *Proc. 14th Eur. Dependable Comput. Conf. (EDCC)*, 2018, pp. 57–64.
[172] P. Hegedüs and R. Ferenc, "Static code analysis alarms filtering reloaded: A new real-world dataset and its ML-based utilization," *IEEE Access*, vol. 10, pp. 55090–55101, 2022.
[173] H. J. Kang, K. Loong Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 698–709.
[174] H. Kim, M. Raghothaman, and K. Heo, "Learning probabilistic models for static analysis alarms," in *Proc. 44th IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1282–1293.
[175] T. T. Vu and H. D. Vo, "Using multiple code representations to prioritize static analysis warnings," in *Proc. 14th Int. Conf. Knowl. Syst. Eng. (KSE)*, 2022, pp. 1–6.
[176] T. Muske and A. Serebrenik, "Classification and ranking of delta static analysis alarms," in *Proc. 22nd IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2022, pp. 197–207.
[177] N. Mansoor, T. Muske, A. Serebrenik, and B. Sharif, "An empirical assessment on merging and repositioning of static analysis alarms," in *Proc. 22nd IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, 2022, pp. 219–229.
[178] R. Yedida, H. J. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies, "How to find actionable static analysis warnings: A case study with FindBugs," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2856–2872, Apr. 2023.
[179] S. E. Sim, S. M. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *Proc. 25th Int. Conf. Softw. Eng. (ICSE)*, 2003, pp. 74–83.
[180] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct. 2012.
[181] "Open web application security project." OWASP. Accessed: Nov. 14, 2023. [Online]. Available: https://www.owasp.org/index.php/Benchmark
[182] I. Pigazzini, "Automatic detection of architectural bad smells through semantic representation of code," in *Proc. ECSA (Companion)*, 2019, pp. 59–62.
[183] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of Code2vec token embeddings," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2019, pp. 1–12.
[184] S. Karakatic, A. Milosevic, and T. Hericko, "Software system comparison with semantic source code embeddings," *Empirical Softw. Eng.*, vol. 27, no. 3, 2022, Art. no. 70.
[185] Z. Ding, H. Li, W. Shang, and T.-H. P. Chen, "Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks," *Empirical Softw. Eng.*, vol. 27, no. 3, 2022, Art. no. 63.
[186] T. Teubner et al., "Welcome to the era of ChatGPT," *Bus. Inf. Syst. Eng.*, vol. 65, no. 2, pp. 95–101, 2023.

Zhaoqiang Guo received the Ph.D. degree from the Department of Computer Science and Technology at Nanjing University. His current research direction is in software quality assurance, especially on code review, software defect prediction, and AI4SE.

Tingting Tan received the B.S. degree in software engineering from Nanjing University of Science and Technology. Her current research direction is in software quality assurance.

Shiran Liu received the Ph.D. degree from the Department of Computer Science and Technology at Nanjing University. His current research direction is in empirical software engineering.

Xutong Liu is working toward the Ph.D. degree with the Department of Computer Science and Technology at Nanjing University. Her current research direction is in software quality assurance.

Wei Lai is working toward the Ph.D. degree with the Department of Computer Science and Technology at Nanjing University. His current research direction is in automatic program repair.

Yibiao Yang received the Ph.D. degree from Nanjing University, in September 2016. He is currently an Associate Research Professor with the Department of Computer Science and Technology at Nanjing University. His main research interests include automated testing and analysis technologies for software and systems.

Yanhui Li (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Southeast University. He is currently an Assistant Professor with the Department of Computer Science and Technology, Nanjing University. His main research interests include AI testing and debugging, empirical software engineering, software analysis, knowledge engineering, and formal methods.

Lin Chen received the Ph.D. degree in computer science from Southeast University, in 2009. He is currently an Associate Professor with the Department of Computer Science and Technology, Nanjing University. His research interests include software analysis and software maintenance.

Wei Dong received the B.S. and Ph.D. degrees in computer science from the National University of Defense Technology, Changsha, China, in 1997 and 2002, respectively. He was a Lecturer from 2002 to 2004, and an Associate Professor from 2004 to 2010, with the College of Computer Science, National University of Defense Technology, where he has been a Professor in software analysis and verification since 2010. He has authored or co-authored more than 70 articles and two textbooks. He has served on more than 20 program committees and has served as the Program Co-Chair of several conferences and workshops. His research interests include program analysis and verification, program synthesis, and runtime verification. He is a member of the China Computer Federation.

Yuming Zhou received the Ph.D. degree in computer science from Southeast University, China, in 2003. He is a Professor with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University, China. His research interests focus on software quality assurance in software engineering, especially on software testing, defect prediction/detection, and program analysis.