Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
June 2024, vol. 29, no. 1, pp. 98–111
https://doi.org/10.2478/acss-2024-0013
https://content.sciendo.com

Analysing the Analysers: An Investigation of Source Code Analysis Tools

Vikram Bhutani[1*], Farshad Ghassemi Toosi[2], Jim Buckley[3]
[1,2]*Department of Computer Science, Munster Technological University, Cork, Ireland*
[3]*Lero and CSIS, University of Limerick, Limerick, Ireland*

Abstract – Context: **The primary expectation from a software system revolves around its functionality. However, as the software development process advances, equal emphasis is placed on the quality of the software system for non-functional attributes like maintainability and performance. Tools are available to aid in this endeavour, assessing the quality of a software system from multiple perspectives.**

Objective: **This study aims to perform a comprehensive analysis of a particular set of source code analytical tools by examining diverse perspectives found in the literature and documentations. Given the vast array of programming languages available today, selecting appropriate source-code analytical tools presents a significant challenge. Therefore, this analysis aims to provide general insights to aid in selecting a more suitable analytical tool tailored to specific requirements.**

Method: **Seven prominent static analysis tools, namely SonarQube, Coverty, CodeSonar, Snyk Code, ESLint, Klocwork, and PMD, were chosen based on their prevalence in the literature and recognition in the software development community. To systematically categorise and organise their distinctive features and capabilities, a taxonomy was developed. This taxonomy covers crucial dimensions, including input support, technology employed, extensibility, user experience, rules, configurability, and supported languages.**

Results: **The comparative analysis highlights the distinctive strengths of each tool. SonarQube stands out as a comprehensive solution with a hybrid approach supporting static and dynamic code evaluations, accommodating multiple languages and integrating with popular Integrated Development Environments (IDEs). Coverity excels in identifying security vulnerabilities and defects, making it an excellent choice for security-focused development. CodeSonar prioritises code security and safety, offering a robust analysis. Snyk Code and ESLint, focusing on JavaScript, emphasise code quality and standards adherence. Klocwork is exceptional in defect detection and security analysis for C, C++, and Java. Lastly, PMD specialises in Java, emphasising code style and best practices.**

Keywords **– Artificial intelligence, code quality, object oriented paradigm, source code analysis tools.**

I. Introduction

Source code stands as a pivotal asset for software companies, and contemporary security-driven development methodologies advocate for code reviews leveraging static analysis tools [1]. Many of the defects that are present in a program are not visible to the compiler. Static code analysis is a way to find bugs and reduce the defects in a software application [2]. The projections indicate that software size by 2025 will surpass 1 trillion lines of code [3], underscoring the growing importance of source code analysis and manipulation in the future. Source code analysis can be performed using static, dynamic, or hybrid approaches (a combination of static and dynamic approaches) [4], [5].

The use of object-oriented design in software is seeing tremendous growth, mainly driven by factors such as improved maintainability, ease in development, reuse and understandability of software systems [6]–[9]. Figure 1 illustrates the software languages paradigm, emphasising the hierarchy of object-oriented languages applied to the imperative language paradigm.

The inferior quality code is vulnerable to security breaches, maintenance limitations, and potential performance issues [10]. Source code analysis offers invaluable information for architectural extraction, reverse engineering [11], and software reengineering of software tools and applications. It facilitates comprehension of programs, software optimisation, maintenance [11], [12], and reusability [13], [14]. Source code analysis tools, often referred to as static code analysis tools or code review tools, are used by developers and software teams to analyse source code for various purposes, which includes identification of potential bugs, security vulnerabilities, coding standards violations, and other issues.

From an executive point of view, Source Code Analysis (SCA) tools are divided into two groups listed below. This high-level categorisation support developers in their work by building a taxonomy that can categorise SCA tools and, thus, guide developers in SCA tool selection.

1) *Static Source Code Analysis*: Tools that analyse source code without executing it to find potential issues, such as bugs, security vulnerabilities, and coding standards violations.
2) *Dynamic Source Code Analysis*: Tools that analyse code during runtime or through dynamic testing to identify issues such as memory leaks, runtime exceptions, and performance bottlenecks.

* Corresponding author's e-mail: vikram.bhutani@intel.com
Article received 2024-03-09; accepted 2024-07-29

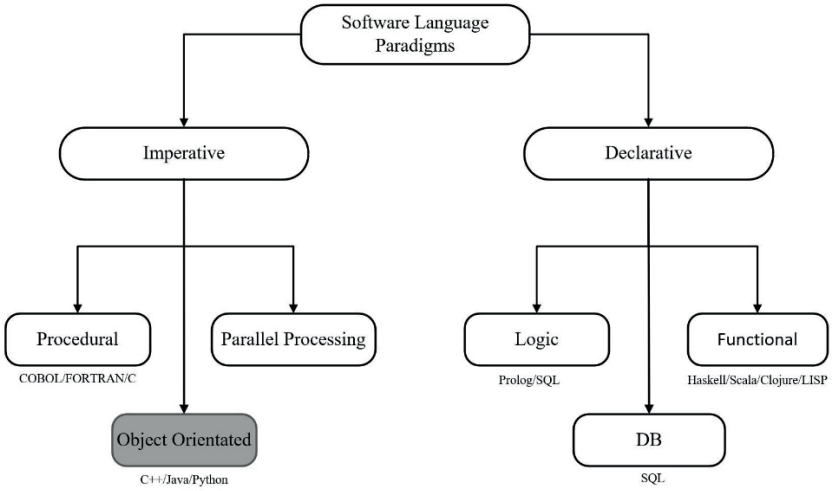

Fig. 1. Software language paradigm.

Static analysis should be contrasted with dynamic analysis, which concerns analysis of programs based on their execution, and includes, e.g., testing, performance monitoring, fault isolation, and debugging. Static analysis does not, in general, guarantee the absence of runtime errors, and while it can reduce the need for testing or even detect errors that in practice cannot be found by testing, it is not meant to replace testing. The source code can get rusty over time due to lack of regular maintenance actions [1], [15]–[17]. In software design, these rusts are known as code smells [16]. For example, having dead code as part of the comments is an example of code clutter. On the other hand, multiple nested loops or branching statements (a.k.a. bow-and-arrow pattern) are examples of bad software design, and a namespace with numerous classes has a high conceptual load, i.e., it takes a lot to comprehend what that namespace is trying to do. The same is true for a class with a long list of methods.

The analysis of source code has gained immense importance, particularly in the context of code reuse, translation, and AI-driven code generation [18]. For example, tools like Roslyn Syntax API [19], LINQ [20] use an AI driven mechanism and are employed to highlight JavaScript code smells, checking if the source code exhibits the mentioned issues. Initiatives have focused on improving quality through source code comments, considering comments as a primary source for system documentation. Consequently, comments play an important role in understanding source code for both development and maintenance purposes [21]. However, these approaches might prove inadequate to address code smells if there are issues with code style. Studies reveal that PMD [22] (Programming Mistake Detector) was used to verify the code flaws, unused variables, and other code-style standards. PMD tool has successfully reduced the workload of code reviewers by minimising such issues in the source code [23].

SCA tools stand as software utilities employed by developers to scrutinise and evaluate source code, detecting errors, security vulnerabilities, style inconsistencies, and other potential issues [16], [24]–[26]. They significantly contribute to upholding code quality and bolstering software dependability by automatically parsing through code and furnishing feedback on likely problems. Static tools primarily examine code statically, signifying their evaluation without code execution. They offer insights and feedback, empowering developers to rectify potential issues prior to deployment, thereby augmenting code quality, dependability, and security. These tools aid developers in spotting security threats, complexities, and other potential problems originating from the source code. However, with the proliferation of available tools, practitioners face challenges in selecting the most appropriate tool (or combination thereof) that aligns with their requirements [27]. This assumes that all developers can accurately discern a warning from a static analysis tool (SAT) as a security threat necessitating rectification.

II. Why Source Code Analysis Tools?

As software applications diversify and become more heterogeneous, static analysis encounters increased demands and challenges [13], [17], [28]. In the dynamic landscape of software development, source code analysis tools become indispensable assets catering to various motivations and needs. These tools aim to address diverse challenges faced by developers, teams, and organisations. The borderline is not clear; some checks done by compilers, such as type checking in a statically typed language, are closer to runtime properties than syntactic ones. For example, compilers [10], [29] do not generally detect issues such as improper resource management, illegal operations (division by zero, over- or underflow in arithmetic expressions, addressing arrays out of bounds, dereferencing of null pointers, freeing already deallocated memory. Dead code and data: Code and data that cannot be reached or are not used. Bad coding style causing signal logical errors or misspellings in the code, incomplete code, which includes the use of uninitialised variables, functions with unspecified return values (due to, e.g., missing return statements) and incomplete branching statements.

The key primary motivations for employing source code analysis tools are as follows.

1) *Enhancement of Code Quality* [25], [30]: Maintaining optimal code quality stands as a foundational pursuit in software development. Code analysis tools play a pivotal role in detecting and rectifying coding inconsistencies, identifying breaches of coding standards, and ensuring the readability of codebase and adherence to the best practices. Elevated code quality translates to reduced bugs, streamlined maintenance, and heightened collaboration among developers.
2) *Early Bug and Defect Identification* [30], [31]: Detecting bugs and defects at an early stage significantly mitigates potential costly and time-consuming setbacks during both development and production phases. Code analysis tools facilitate the early detection of coding errors, potential runtime exceptions, code divergence [32], and logic flaws, empowering developers to address these issues proactively before they escalate into critical problems.
3) *Security Fortification* [34]–[36]: In the face of escalating cyber threats, fortifying software applications against security vulnerabilities remains a paramount concern. Code analysis tools equipped with security-focused features are indispensable. They identify vulnerabilities such as SQL injection, cross-site scripting, and other threats, fortifying applications against potential security breaches.
4) *Compliance and Standards Alignment* [10], [25], [26]: Adherence to specific coding standards, regulations, or best practices is imperative across various industries and projects, especially in safety-critical and highly regulated domains. Source code analysis tools ensure code compliance with industry-specific standards, be it for medical devices, automotive systems, or financial software.
5) *Sustaining and Scalability* [10], [17]: As software projects expand, maintaining and scaling codebases become increasingly challenging. Code quality lapses or non-adherence to best practices can impede scalability and maintenance. These tools are indispensable for identifying and refurbishing complex or outdated code, thereby enhancing scalability and long-term maintainability.
6) *Facilitation of Code Review and Collaboration* [25], [37]: Collaboration stands as a cornerstone in software development. Code review tools foster collaboration and knowledge exchange among team members. They offer features for reviewing and annotating code changes, augmenting code quality through peer review and knowledge dissemination.
7) *Real-Time Feedback Mechanisms* [37], [38]: Developers benefit immensely from instantaneous feedback during the coding process. Such prompt feedback aids in early issue identification and adherence to coding standards. Tools that seamlessly integrate into development environments, providing real-time feedback, empower developers to write superior code and make necessary adjustments as they progress.
8) *Integration into Continuous Integration/Continuous Deployment (CI/CD)* [39]–[41]: In the epoch of CI/CD, the automation and integration of code analysis into the pipeline are pivotal. This integration ensures that only high-quality, secure code is deployed into production environments. Code analysis tools must seamlessly integrate into CI/CD workflows, automatically scrutinising code and furnishing reports to uphold code quality and security throughout the development life cycle.
9) *Need for Code Analysis for a Machine Translated Code:* Over time, the emergence of numerous programming languages has facilitated tailored program creation across diverse domains. However, diversity poses integration challenges when combining programs from different linguistic frameworks. Consequently, software migration, the shifting of software to new platforms, remains a common industry practice. Yet, translating programs between languages can be costly, time-intensive, and resource-demanding. Program translation involves transposing a program from one language to another, aiming to create an equivalent, syntactically correct code that computes similarly to the original [13], [14]. However, achieving this across a wide array of source and target languages demands substantial effort from both tooling and human perspectives. Researchers have tried to develop cost-effective techniques for automatic program translation to simplify software migration complexities [26], [42], [43]. Despite extensive research, creating a translator remains a costly endeavour in software engineering. Full automation in programming language translation has proven elusive, with previous attempts typically capping at a 90 % success rate [44]. The critical aspect of software translation lies in the quality of the resulting software system, measured by functionality and adherence to quality metrics. Hence, the importance of a reliable, compatible, and comprehensive SCA tool is acknowledged to ensure the integrity and quality of the translated software system. By demonstrating the feasibility and effectiveness of our approach through experiments and evaluations, we hope to contribute to the advancement of the existing static analysis tools in software engineering and inspire further research in this area.

III. Related Work

In this section, we introduce previous research efforts about the comparison of static analysis tools. As the Internet and Web applications burgeoned in the late 1990s and early 2000s, security became paramount. Commercial static code analysis tools like Coverity [45] and Klocwork [46] emerged, focusing on high-assurance software, particularly in safety critical industries. To address vulnerabilities such as SQL injection and cross-site scripting, security-focused tools such as Fortify and OWASP [1], [10] have gained prominence in software development.

Oyetoyan et al. [47] evaluated six static analysis tools, namely, Findbug, Find Security Bugs, SonarQube, JLint,

LAPSE+, and one anonymous commercial tool using the Juliet Test Suite. The results showed variations in the detection capabilities of the tools studied. Gomez et al. and Zistner et al. [46], [48] explored the evaluation of features in five static analysis tools, specifically cantering on the software metric of buffer overflow. In contrast, Hooper [49] employed 30 static analysis tools, encompassing various security and performance software metrics. Ashouri [50] evaluated the bug detection capabilities of three static analysis tools, namely YASCA, CAT.NET, and Findbug, based on the software assurance reference dataset (SARD). The finding showed that the tools were effective and could effectively be applied for bug detection. Manzoor et al. [51] conducted a comparative study on three static analysis tools, namely, Check Thread, Racer, Check Thread, and RELAY to find concurrency bugs in open-source benchmark programs, provided by NASA and IBM. Thung et al. [52] and Nanda et al. [53] assessed the effectiveness of static analysis tools by studying the source code of industrial programs. The findings showed that the tools were effective and could be used in detecting defects in the source code of programs. Schultz et al. [54] analysed five ASAT applied to a variety of Java programs.

The 2010s ushered in the popularity of multifaceted tools like SonarQube [55], offering comprehensive code analysis across multiple programming languages. Concurrently, modern Integrated Development Environments (IDEs) integrated built-in code analysis tools, providing real-time feedback during the coding process – an exemplar being ESLint for JavaScript. A prominent trend emerged in integrating these tools into the CI/CD pipeline, automating code analysis, report generation, and feedback provision at various development stages.

Due to a vast variety of SCA tools, gap exits to perform the comparative study of such tools based on the key functionality, metric used in evaluating such tools such as code design, performance and quality as illustrated under Section III-A.

Figure 2 illustrates the literature review employed in the study in more detail. It illustrates our process of identifying and examining existing SCA tools, their usage models, adoption in software industry, efficiency, and comparative studies conducted in this research area. Overall, we shortlisted 21 articles after performing all processes as shown in Fig. 2.

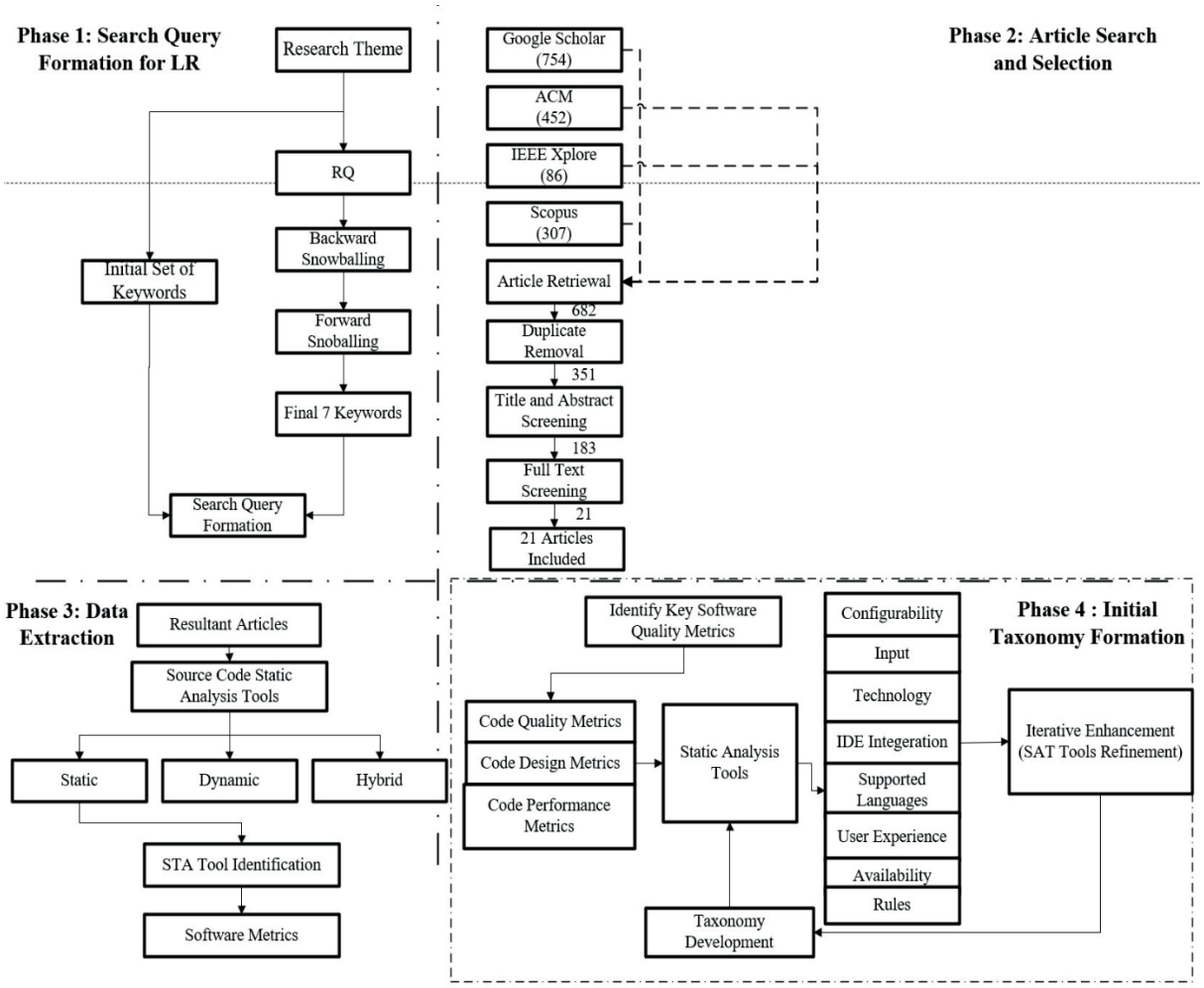

Fig. 2. Literature review of SCA tools.

The literature review (LR) process consisted of the following phases:

1) *Phase 1*: Defining the research question (RQ) and identifying relevant keywords related to SCA tools and evaluation. This phase involved formulating a clearly selecting appropriate keywords to ensure the comprehensive retrieval of relevant articles.
2) *Phase 2*: Extracting relevant articles from leading research repositories. In this phase, a search was conducted in prominent research repositories and databases using the identified keywords. The goal was to retrieve a broad range of articles that addressed the research question.
3) *Phase 3*: Shortlisting articles for further data extraction. In this phase, the retrieved articles were screened based on predefined inclusion and exclusion criteria. Articles that met the criteria were selected for further analysis, while irrelevant articles were excluded.
4) *Phase 4*: Mapping the data extracted from Phase 3 and forming an initial taxonomy for source code static analysis tools. In this phase, the selected articles were carefully analysed, and relevant data were extracted. The extracted data were then organised and categorised to form an initial taxonomy that captured the different SCA analysis tools identified in the literature.

Novak et al. [56] categorised SCA tools into distinct clusters: *Technology*, *Rules*, *Supported Languages*, and *Extensibility*. Figure 3 delineates the taxonomy of SCA tools and the wide range of features used for software quality measurement. Each tool encompasses subcategories associated with static code analysis, such as supported languages, technology dependencies, rules, extensibility, user experience, and configurability.

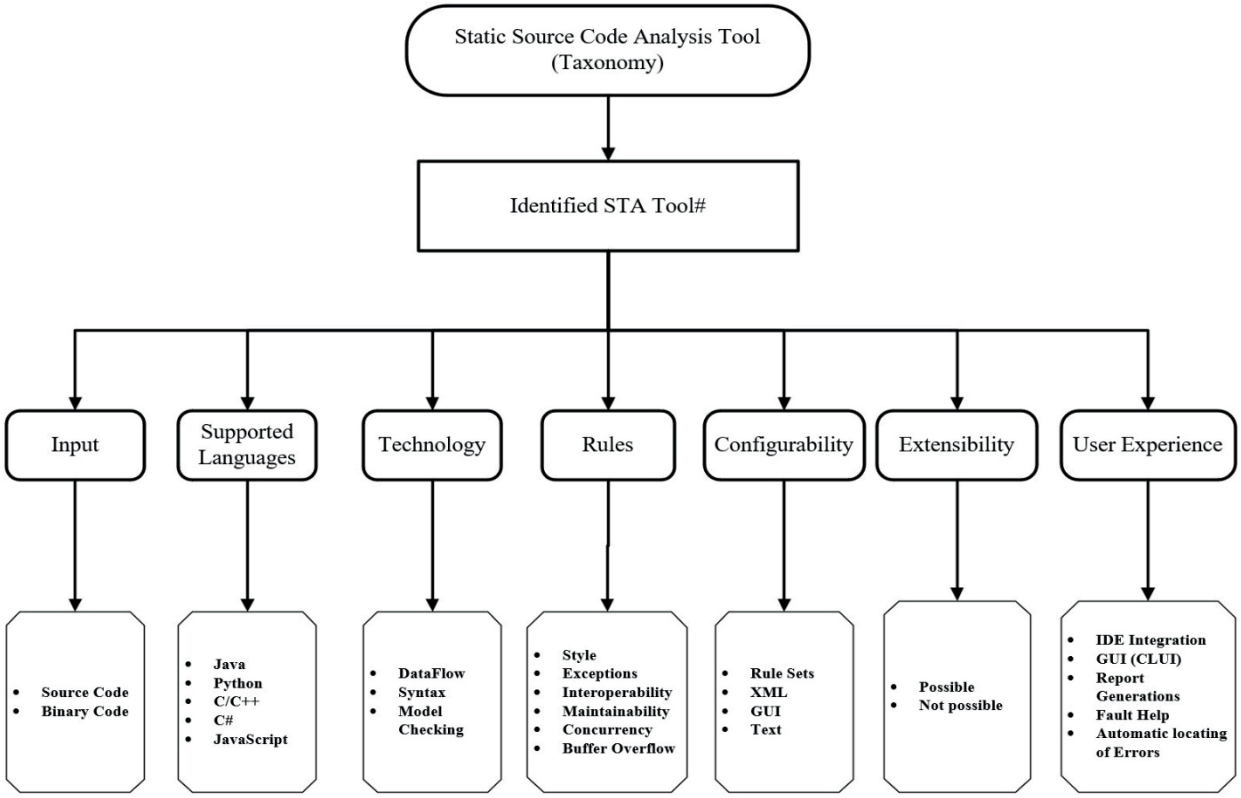

Fig. 3. SCA tool taxonomy.

This methodical approach facilitates a comprehensive understanding of the capabilities and functionalities offered by various SCA tools, aiding researchers, practitioners, and decision-makers seeking nuanced insights within the realm of software quality assurance.

A. Key Software Metrics

Evaluating a software system requires a set of metrics. Software metrics are a critical and essential part of all engineering disciplines, particularly in software development [32], [57], [58], providing fundamental insights into the development process to assess maintainability, reliability, and even development progress. There are also metrics that are product driven, such as size of code (LOC), code complexity, failure rate, design features, complexity, performance, level of quality. Software metrics offer reproducible indicators useful for estimating quality, performance, management, and cost within a project. They bring benefits such as the possibility of analysing the data to understand, improve, and predict future behaviours to take corrective actions on time.

There are many metrics that can determine source code quality, maintainability, and performance [1], [17], [26]. In general, source code metrics are mainly categorised into code quality, code design metrics, and performance-based metrics [26]. However, more granular metrics [10], [26] exist, as shown below:

1) *Code Quality Metrics*:
   a) Magic Numbers in Mathematics;
   b) Magic Numbers Used in Index;
   c) Magic Numbers Used in Conditions;
   d) Ladder if Statements;
   e) Fragmented Conditions;
   f) Hungarian Names for Variable;
   g) Lots of Local Variables in Methods;
   h) Methods Not Using All Parameters;
   i) Multiple Return Statements;
   j) Long Parameter Lists;
   k) Large Class;
   l) Lines of Code;
   m) Control Flags bad use;
   n) Code-to-Comment Ratio.

Code quality metrics may be viewed from Code Design point of view [1], [10], [17]:

2) *Code Design Metrics*:
   a) Deeply Nested Loops/ Deeply Nested IF block;
   b) Out Parameters;
   c) Long List of Switches;
   d) Empty Interfaces;
   e) Sealed Class and Protected Members;
   f) Abstract Types with Constructors;
   g) Static Members on Generic Types;
   h) Object Obsession;
   i) Refused Bequest Metric.

From the perspective of code performance, metrics can be expressed as follows [1], [10], [26]:

3) *Code Performance Metrics*:
   a) Avoid Excessive Local Variables;
   b) Prefer Literals over Evaluation;
   c) Don't Use an Object Array in params;
   d) Avoid Volatile Declarations;
   e) Value Types Should Override Equals and GetHash-Code;
   f) Avoid Using Empty Strings to Find Zero Length Strings;
   g) Prefer "Jagged" Arrays over Multidimensional Arrays;
   h) Don't Return Array from a Property;
   i) Don't Use Thread.Abort or Thread.Suspend;
   j) Avoid Boxing.

In addition to the above, Chidamber et al. introduced explicit metrics that are designed for OO (Object Oriented) paradigm. These metrics are based upon measurement theory and are informed by the insights of experienced object-oriented software developers [59]. These OO design metrics used are stated below:

4) *Object Oriented Software Paradigm Metrics*:
   a) Weighted Methods per Class (WMC);
   b) Depth of Inheritance Tree (DIT);
   c) Number of Children (NOC);
   d) Response for a Class (RFC);
   e) Lack of Cohesion in Methods (LCOM);
   f) Lack of Cohesion in Methods (LCOM).

Furthermore, metrics associated with code forensics [60] significantly impact the quality of source code. As developers advance in their careers, they inherently develop different coding styles, each person cultivating unique preferences for naming variables and functions. In addition, individual familiarity with a programming language profoundly influences coding practices. Some developers are inclined towards specific language features or data structures, naturally integrating them more frequently than others. Essentially, developers often imprint a recognizable 'signature' in their authored code. However, quantifying such metrics requires historical data, which is beyond the scope of this paper.

B. Taxonomy Development of Source Code Analysis Tools

In the conclusive phase of our literature review, our efforts culminated in the development and presentation of a comprehensive taxonomy. The taxonomy was designed to categorise SCA tools based on their discerning features, shaped by the final articles selected during the thorough data extraction process. This ensured the creation of a robust and relevant classification framework.

The taxonomy was crafted carefully to encompass critical dimensions that define the distinctiveness of each SCA tool. Key categories within the taxonomy include the following:

- *Input Support*: Distinguishing between tools operating on byte code or source code, reflecting the diversity in analysis approaches.
- *Technology Used*: Categorisation based on the underlying technology, including syntax-based analysis and other advanced methodologies employed by each tool.
- *Static or Dynamic*: Support to analyse the given source code performing statics checks or also capabilities to perform relevant analysis during run time of the source code.
- *Extensibility*: Evaluating the extensibility of tools, shedding light on their adaptability and capacity for integration with external components.
- *User Experience*: Assessing user experience factors such as IDE integration, aiming to provide insights into the accessibility and usability of the tools.
- *Rules*: Analysing the rule sets employed by each tool, delineating their capabilities in identifying and enforcing coding standards.
- *Configurability*: Evaluating the extent to which each tool allows for configuration, providing insights into the adaptability of the tool to specific project requirements.
- *Supported Languages*: Categorising the tools based on the programming languages they support, a crucial factor for developers working in diverse language environments.

The creation and presentation of this taxonomy significantly enhance the significance of the literature review. It not only synthesises diverse findings but also establishes a systematic approach to comprehending the complexities of SCA tools. This expanded and structured taxonomy now incorporates additional dimensions, serving as a valuable resource for researchers, practitioners, and decision-makers seeking nuanced insights into the distinctive features of various tools within the domain of software quality assurance.

IV. Identification of Static Analysis Tools

By the term "static analysis", we mean automatic methods to reason about run-time properties of program code without actually executing it. Properties that we consider include those which lead to premature termination or ill-defined results of the program, but preclude, for instance, purely syntactic properties such as syntax errors or simple type errors. Figure 4 displays the selected state-of-the-art (SOTA) source code analysis tools and the metrics utilised within them. As numerous studies exhibit heterogeneity, our endeavour was to propose a generic methodology applied across leading Automated Static Analysis Tools (ASAT) catering to the most prevalent programming languages [63]. Based on the literature reviewed and the shortlisted articles, we identified SCA tools based on relevance and usage in the software industry. There are many static code analysers that work in different ways. Some static code analysers operate on the source code, while others check the intermediate code and the libraries created. Another difference is the fact that different static analysers operate in different programming languages, such as C, Python, PHP, C++, C# or Java.

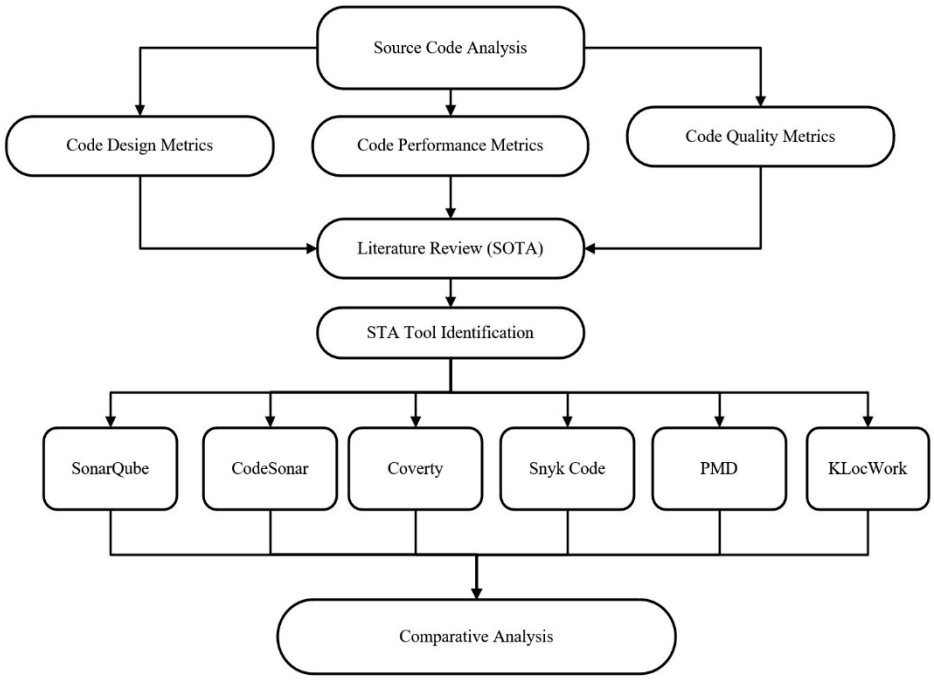

Fig. 4. Identification methodology of static analysis tools.

This study adopts a systematic approach to identify and assess software quality metrics and Static Analysis Tools (SAT). The methodology encompasses key steps:

- *Identification of Software Quality Metrics*: Identifying and cataloguing diverse software quality metrics. This rigorous process involves reviewing existing literature, industry standards, and software engineering best practices, and establishing a solid metric foundation for software quality evaluation.
- *Compilation of Software Metrics*: Following the identification of software quality metrics, a comprehensive list is compiled that covers code complexity, security vulnerabilities, code duplication, and adherence to coding standards. This compendium serves as a reference for assessing SAT capabilities.
- *Identification of Static Analysis Tools*: A spectrum of widely used SATs for code analysis is identified based on their prevalent use in software development and industry standards [29]. No single software fault-detection technique can address all fault-detection concerns [62]. This phase involves a detailed review of industry reports, peer-reviewed articles, and surveys to curate a relevant list of tools aligned with contemporary software development practices. SATs keep evolving with time as software grows and gets more complex. For example, Emanuelson et al. [29] used three SOTA tools (Coverty, KlocWork and PolySpace). It was not trivial to identify SOTA tools today from nearly over 60 tools available. ChaptGPT [66] is an emerging AI based tool that can be used for SCA, although research is under exploration in this area.

- *Development of Taxonomy and Comparative Analysis*: During the fourth phase of the literature review, a taxonomy encompassing diverse features embedded in the SATs is formulated. This taxonomy becomes the framework for conducting structured evaluations of each tool capabilities. Tools are categorised according to their support, features, and analysis metrics.

V. Main Static Source Code Analysis Tools

In this section, we delve deeply into the widely employed SATs used in source code analysis, drawing from insights gleaned through a literature review. These tools are identified based on the wide variety of their use in software industry, languages supported, and metric used in evaluating source code analysis.

A. SonarQube

SonarQube [55] is an open-source platform with the primary goal of assessing the quality of software projects. It is widely adopted by over 400 000 software organisations worldwide [63]. Going beyond simple Linting, SonarQube offers a comprehensive code analysis that includes aspects of code quality, security, and maintainability [1], [17], [29]. SonarQube supports multiple programming languages, including Java, C/C++, JavaScript, Python, and more. From a scalability perspective, SonarQube is suitable for both small projects and large enterprises, providing centralised management of code quality across an entire codebase. It is frequently integrated into continuous integration pipelines and utilised for reporting and tracking code quality over time. SonarQube offers an extensive set of rules for code quality and security [29], [64], calculating metrics such as code complexity, duplications, and maintainability. The platform includes features for security scanning, identifying vulnerabilities, and ensuring adherence to security standards (e.g., OWASP Top Ten). SonarQube generates reports on duplicate code (code clone), lax coding standards, unit tests, code coverage, code complexity, and comments.

In contrary, SonarQube has shown weaknesses in finding bugs [65]. Vassalo et al. [70] conducted a study on 119 OSS projects and found that SonarQube ineffectiveness on prioritising issues was largely because developers mainly fixed major issues, which were supposed to highly impact developers' productivity. This was also supported by Marcillo et al. who claimed that not all violations were fixed using SonarQube [63].

B. Coverty

Coverity [64] is a static analysis solution that enables the early identification of software issues in the development life cycle by analysing source code. It specifically targets: a) software quality and security issues, and b) violations of common coding standards [24], [46]. As a commercial static code analysis tool, Coverity is designed to uncover defects and vulnerabilities in code [1]. It places a strong emphasis on identifying complex software flaws, security issues, and potential runtime errors. Supporting C, C++, and Java, Coverity is well suited for applications in industries where safety and reliability are paramount. Coverity was perceived both as easy to install and use, and no modifications to the existing development environment were needed. The tool performs in-depth analysis, employing techniques such as symbolic execution and abstract interpretation to identify complex issues that might lead to defects and security vulnerabilities. Coverity is seamlessly integrated into the development workflow and CI pipelines, ensuring code quality at each stage. Its scalability makes it a preferred choice for large projects and organisations, particularly those involved in the development of high-assurance and safety software. Coverity also provides options for customising rules and analysis based on project specific needs.

An empirical study with 20 use cases in Java by Robert Feldt et al. [68] showed that Coverty was not that successful as compared to Jtest and JLint tools in detecting all bugs related to weak-reality (two stage access), Bblocking-critical section, Orphaned-thread data race, and atomicity violation. Study by Lenarduzzi et al. [69] also suggested that Covert had a high number of false positives. Coverity Prevent and Klocwork K7 is that the products are not suitable for detecting arithmetic defects, such as over- and underflows or illegal operations like division by zero [29].

C. CodeSonar

CodeSonar [70] is a commercial static analysis tool designed to identify and correct software vulnerabilities and bugs within complex codebases. Developed by GrammaTech, a company specialising in software analysis tools and cybersecurity solutions, CodeSonar stands out for its unique capability to analyse both source and binary code. This dual approach provides comprehensive information on potential issues and vulnerabilities within software systems. CodeSonar performs a detailed static analysis of code, pinpointing a diverse range of issues such as bugs, security vulnerabilities, and potential flaws in software architecture. The advantages include support for multiple programming languages, including C, C++, Java, and Ada, making it a versatile choice for various software projects. Using advanced techniques such as abstract interpretation and data flow analysis, CodeSonar excels in detecting complex software flaws, potential security vulnerabilities, and runtime errors. Its scalability accommodates both small- and largescale projects, offering detailed analysis even for expansive codebases. CodeSonar seamlessly integrates into the software development workflow and provides customisation options, allowing users to define rules and adjust the analysis based on specific project needs.

The CodeSonar SCA tool also generates detailed reports with visualisations, facilitating a clear understanding of identified issues for developers and teams. CodeSonar is particularly renowned for its focus on security analysis, which assists in the identification and elimination of potential security vulnerabilities in software. Compatible with various operating systems, it proves versatile for different development environments.

D. ESLint

ESLint [71] stands out as a widely utilised static code analysis tool, primarily tailored for JavaScript. With the inclusion of plugins, it can seamlessly extend its support to TypeScript. Operating as a Linter, ESLint concentrates on enforcing coding style and pinpointing potential issues and errors within JavaScript codebases. The core function of ESLint is to conduct Linting, meticulously analysing JavaScript code to identify errors, enforce coding style conventions, and flag potential issues. This meticulous process ensures strict adherence to the coding standards, contributing to enhanced code quality. ESLint excels in maintaining consistent code formatting throughout a project, assisting developers in adhering to best practices. ESLint distinguishes itself through its high flexibility and configurability, empowering developers to craft custom Linting rules and modify existing ones to align with project-specific requirements. Configuration for ESLint can be articulated using files like eslintrc.js or within package.json. Beyond core JavaScript Linting, ESLint's advantages include a robust ecosystem of plugins that provide additional functionalities. This ecosystem facilitates support for various JavaScript frameworks and libraries, significantly extending its capabilities. ESLint seamlessly integrates with a variety of Integrated Development Environments (IDEs), such as Visual Studio Code, Sublime Text, Atom, and others. This integration ensures that developers receive real-time feedback and linting assistance as they actively write code. Furthermore, ESLint offers an extensive set of built-in rules for common issues. Developers also have the flexibility to create custom rules, enabling the enforcement of their coding standards or project-specific practices. Often incorporated into Continuous Integration (CI) pipelines, ESLint automatically checks code as part of the build process. This practice guarantees consistent quality and adherence to code standards across the entire project. ESLint provides deep insights into semantic analysis but lacks fully detecting overflow and illegal operations (by zero) [10], [29].

E. Snyk

Snyk [72] is a security-focused static code analysis tool offered by Snyk, a company recognised for its commitment to help developers and organisations improve the security of their software. Specifically crafted to pinpoint and mitigate security vulnerabilities within the source code of various programming languages, SnykCode excels in its security-focused static code analysis. The tool is adept at identifying diverse security vulnerabilities, encompassing known security flaws, potential exploits, and weaknesses that could be exploited by attackers. Supporting multiple programming languages, including popular ones like JavaScript, Java, Python, Ruby, and more, SnykCode accommodates a wide array of codebases. Integration with CI/CD pipelines is a notable feature, enabling developers to automate security checks seamlessly throughout the development process. SnykCode stands out by providing real time feedback to developers as they code, promptly identifying potential security issues during the coding process.

Moreover, Snyk offers customisation options that allow for the definition of rules or configurations tailored to the specific needs of a project. It goes beyond detection by providing comprehensive guidance and recommendations for addressing identified vulnerabilities, suggesting effective ways to tackle the security issues found in the code.

Snyk is a tool designed to identify and fix vulnerabilities and license violations in open-source dependencies and containers. Its design is primarily focused on detecting security related vulnerabilities in source code, which are lacking in actual code analysis and detecting functional bugs such as arithmetic or overflow that can affect significantly impact code quality.

F. Klocwork

Klocwork [46] stands out as a powerful tool for identifying and correcting coding issues, defects, and security vulnerabilities within software applications. It helps development teams in ensuring both code quality and security by providing comprehensive static code analysis. Klocwork conducts static analysis on source code, identifying potential defects and issues without the need for code execution. The tool excels at recognising a broad spectrum of potential defects, including memory leaks, null pointer dereferences, race conditions, and other programming errors. Notably, it specialises in uncovering security vulnerabilities within the code, highlighting potential flaws that could be exploited by attackers. Klocwork advantages include the support for multiple programming languages, including C, C++, Java, and C#, among others, rendering it versatile for diverse development environments. The tool seamlessly integrates with popular IDEs and CI pipelines, facilitating smooth inclusion in the development workflow.

For added flexibility, Klocwork offers customisation options, enabling the definition of rules and policies tailored to the specific needs and coding standards of a project. Furthermore, it generates detailed reports that provide insights and visualisations, assisting developers in comprehending and addressing the identified issues.

Gomes et al. [46] found that Klocwork K7 seemed to produce a higher rate of false positives but remained approximately accurate in other metrics. The comparative study [29] found that Kloc was unable to detect all defects in C++ test cases regressed during the study.

G. PMD

PMD [73] stands out as an open-source static analysis tool primarily utilised to identify problems in Java code. Its role involves scrutinising the Java source code to identify potential problems, coding errors, and code style violations. PMD, being an open-source static analysis tool tailored for Java, proves invaluable for Java developers aiming to uphold code quality and adhere to coding standards. Specifically designed for Java code analysis, PMD excels at identifying common coding mistakes, potential performance issues, and enforcing the coding style. The tool adeptly identifies a wide array of potential issues, including unused variables, empty catch blocks, unnecessary object creation, and other code patterns that might lead to bugs or maintainability issues. PMD employs

predefined rules that span various aspects of coding standards, best practices, and potential code issues. PMD seamlessly integrates into popular DEs) such as Eclipse and IntelliJ IDEA, along with build tools like Apache Maven. This integration ensures smooth analysis within the development workflow. Additionally, PMD offers customisation options that allow users to define rules based on project-specific requirements and enforce specific coding standards or practices.

The tool generates detailed reports outlining identified issues and violations, providing developers with valuable insights into potential problems within the code. PMD and FindBugs [74] have a high correlation in warnings. Furthermore, PMD is a mature open-source software tool that can run without compilation, making it convenient for exploring a large number of open-source projects. PMD has limitations in validating issues like synchronisation mechanisms and threading concepts. PMD cannot be used alone and must be combined with effective coding and testing practices such as using coding standards, inspections, and code review processes [23].

VI. Results and Conclusion

The demand for static code analysis tools is increasing as the complexity and size of the software continue to expand. Numerous instances demonstrate that static analysis has uncovered critical defects and vulnerabilities that would have been extremely difficult to detect with standard testing methods. One notable example is the Scan Project, which identified 700 defects. Comparative analysis of existing source code analysis tools indicates that SonarQube excels in defect detection and offers a wider range of software metrics compared to Coverity, CodeSonar, PMD, SnykCode, and Klocwork.

The evaluation of all identified SATs reveals a diverse spectrum of functionalities, addressing various aspects of software quality assurance. We have detailed how these tools differ in terms of language support, key metrics utilised, and user experience features such as IDE integration.

A. Comparison of Source Code Analysis Tools

Table I shows the comparative analysis of each source code analysis tool we identified as part of the methodology section.

Figure 5 shows the distribution of all static analysis tools that we compared as part of this study. Figure 5 shows that SonarQube supports a maximum number of languages and being a hybrid tool means support of static and dynamic analysis of source code makes it more versatile than other tools. On the other hand, PMD and ESlint have support for least number of languages as they support only C++ or Java source code, respectively.

TABLE I
Comparison of Source Code Analysis Tools

SCAT	Languages supported	Type	Software metrics	IDE integration
SonarQube	33	Hybrid	Code defects, Maintainability, Reliability, others	Eclipse, IntelliJ, VS Code, others
Coverity	14	Static	Code security, others	Eclipse, Visual Studio
CodeSonar	4	Static	Code quality, security	Eclipse, Visual Studio
Snyk Code	16	Static	Security vulnerabilities	VS Code, IntelliJ, others
ESLint	2	Static	Code quality, standards	Most JavaScript IDEs
Klocwork	7	Static	Code defects, others	Eclipse, Visual Studio, others
PMD	15	Static	Code quality, others	Eclipse, IntelliJ, others

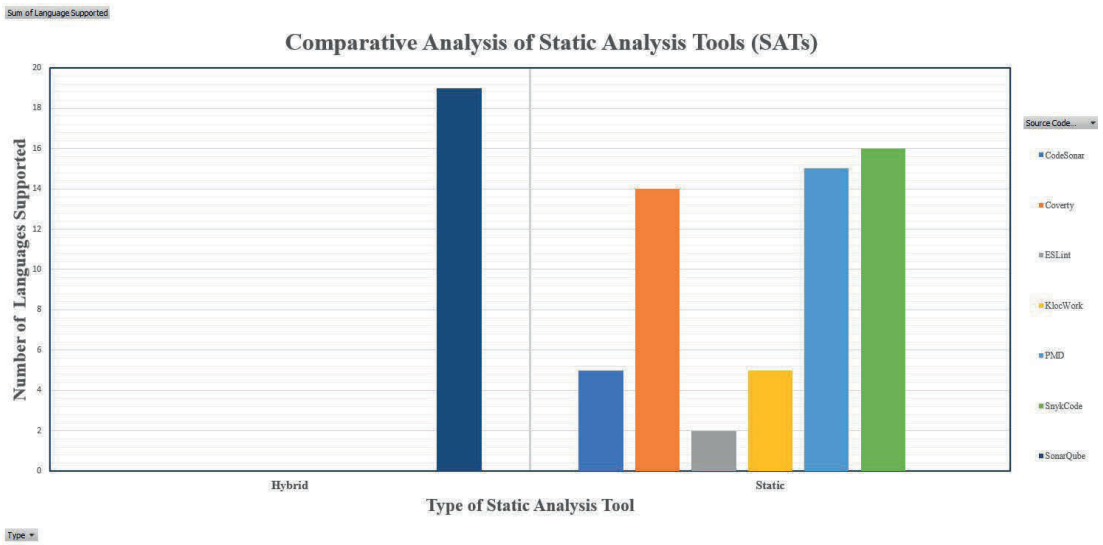

Fig. 5. Comparative Analysis of SATs

All SATs are also evaluated based on their effectiveness using standard evaluation metrics, such as the number of warnings, false positives, and coverage. The tools compared in Table I exhibit variations in precision and recall results, as identified in the literature. These results also highlight significant differences in the vulnerability detection capabilities of the tools. To sum up, the comparison of SATs reveals a diverse landscape where each tool has unique strengths.

SonarQube stands out as a comprehensive solution with a hybrid approach, supporting multiple languages and seamlessly integrating with popular IDEs. Coverity excels in identifying security vulnerabilities and defects, making it a robust choice for security-focused development. CodeSonar prioritises code security and safety, providing extensive language support. Snyk tool is particularly valuable for open-source security, targeting JavaScript developers. ESLint emphasises JavaScript code quality and standards adherence. Klocwork is notable for defect detection and security analysis in C, C++, and Java. Lastly, PMD specialises in Java, promoting code style and best practices.

Based on the taxonomy discussed in the previous section, developers need to decide which tool is suitable taking into requirements such as language support, software model, extensibility, etc. It should be noted that the selection of the most suitable tool depends on project-specific requirements, language preferences, and emphasis on aspects such as security, code quality, or open-source vulnerabilities. Developers might find tools like SonarQube and Coverity useful due to their versatility and robust security features. Their specialisation in security-focused development makes them strong choices for protecting software systems. ESLint and Snyk Code are ideal for JavaScript-centric projects. Ultimately, choosing the right tool involves careful consideration of project needs and language compatibility.

It is imperative to acknowledge that utilising tools with a high false positive rate is deemed unacceptable, particularly when dealing with large-scale projects and introducing the tool late in the development process. Even if different tools target the same defect categories, such as memory leaks or out-of-array bounds issues, the specific defects identified within a given category by one tool may significantly differ from those identified by another tool.

Based on the results, it is concluded that there are no currently released (commercially) static analysis tools that can detect all defects defined by the presented classification. However, a combination of certain tools can address the problem presented. In addition, using the same classification in all studies leads to a better understanding of the differences between the tools. Further research involves performing experiments on the presented tools to compare their advantages and disadvantages in SCA, as well as expanding the scope of studies and tools for SCA and their classification according to the supported programming languages and the defects they detect. The use of tools for SCA and continuous work on improving and maintaining the quality of source code provides higher-quality software solutions and represents a step forward in software development.

In conclusion, selecting a static analysis tool depends on the specific requirements of the users, language preferences, and the development environment. Software developers also need to fine-tune tools and techniques to induce a variety of bugs to improve performance in real-time software testing. Each tool contributes uniquely to detecting and resolving code issues, improving overall code quality, and advocating secure coding practices within software development projects.

VII. Future Work

We plan to extend the current research in source code quality as part of the future work in the domain of software source code analysis and security.

Code Forensics: Code forensics is pivotal for software companies in detecting potential plagiarism issues that might compromise Intellectual Property security. Tools like Moss (Measure of Software Similarity) and JPlag are specifically tailored to uncover code similarities, aiming to pinpoint instances of plagiarism. Despite its significance, this field remains relatively under-explored, warranting the establishment of specific metrics to efficiently detect such issues.

Machine Learning-Based STA: Emerging techniques like ChatGPT harness machine learning algorithms to bolster bug detection, vulnerability identification, and code analysis. For instance, machine learning models, like warning prioritisation models based on optimal feature sets, showcase the potential of leveraging artificial intelligence within Static Analysis Tools for enhanced performance and accuracy [75]. Code forensics is an active area of research, machine learning approaches such as kNN (k-Nearest Neighbour) can improve the accuracy if a neural network is fed the cleaned data. Our future work includes making use of such technologies which can help in achieving better result.

Safeguarding with Binary Code Scanners: Binary Code Scanners operate at the binary code level, post-compilation, enabling the identification of vulnerabilities within compiled and imported binary libraries. These scanners scrutinise machine code, constructing a language-agnostic representation of program behaviour, control, data flow, call trees, and external function calls. Subsequently, automated vulnerability scanners traverse this model to uncover potential vulnerabilities, fortifying software security at a deeper level.

Acknowledgement

The research has been funded with the financial support of the Science Foundation Ireland grant 13/RC/2094_P2 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero – the Science Foundation Ireland Research Centre for Software (www.lero.ie).

REFERENCES

[1] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities – does experience matter?" in *2009 International Conference on Availability, Reliability and Security*, Fukuoka, Japan, Mar. 2009, pp. 804–810. https://doi.org/10.1109/ARES.2009.163

[2] A. G. Bardas *et al.*, "Static code analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[3] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings. 26th International Conference on Software Engineering*, Edinburgh, UK, Jun. 2004, pp. 273–281. https://doi.org/10.1109/ICSE.2004.1317449

[4] T. B. C. Arias, P. Avgeriou, and P. America, "Analyzing the actual execution of a large software-intensive system for determining dependencies," in *2008 15th Working Conference on Reverse Engineering*, Antwerp, Belgium, Oct. 2008, pp. 49–58. https://doi.org/10.1109/WCRE.2008.11

[5] F. Angerer, "Variability-aware change impact analysis of multi-language product lines," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2014, pp. 903–906. https://doi.org/10.1145/2642937.2653472

[6] G. Booch, "Object-oriented development," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 211–221, Feb. 1986. https://doi.org/10.1109/TSE.1986.6312937

[7] D. De Champeaux, A. Anderson, and E. Feldhousen, "Case study of object-oriented software development," *ACM SIGPLAN Notices*, vol. 27, no. 10, pp. 377–391, Oct. 1992. https://doi.org/10.1145/141937.141967

[8] K. Lieberherr and C. Xiao, "Object-oriented software evolution," *IEEE Transactions on Software Engineering*, vol. 19, no. 4, pp. 313–343, Apr. 1993. https://doi.org/10.1109/32.223802

[9] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*. Springer Nature, 2023.

[10] M. Mantere, I. Uusitalo, and J. Roning, "Comparison of static code analysis tools," in *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, Athens, Greece, Jun. 2009, pp. 15–22. https://doi.org/10.1109/SECURWARE.2009.10

[11] R. Lammel, M. Leinberger, T. Schmorleiz, and A. Varanovich, "Comparison of feature implementations across languages, technologies, and styles," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Antwerp, Belgium, Feb. 2014, pp. 333–337. https://doi.org/10.1109/CSMR-WCRE.2014.6747188

[12] N. F. Schneidewind, "The state of software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 303–310, Mar. 1987. https://doi.org/10.1109/TSE.1987.233161

[13] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "Ware: A tool for the reverse engineering of web applications," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, 2002, pp. 241–250.

[14] L. Coyle, M. Hinchey, B. Nuseibeh, and J. L. Fiadeiro, "Guest editors' introduction: Evolving critical systems," *Computer*, vol. 43, no. 05, pp. 28–33, May 2010. https://doi.org/10.1109/MC.2010.139

[15] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *2009 3rd international symposium on empirical software engineering and measurement*, Lake Buena Vista, FL, USA, Oct. 2009, pp. 390–400. https://doi.org/10.1109/ESEM.2009.5314231

[16] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, Nov. 2018, Art. no. 273. https://doi.org/10.3390/info9110273

[17] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering (FOSE'07)*, Minneapolis, MN, USA, May 2007, pp. 104–119. https://doi.org/10.1109/FOSE.2007.27

[18] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, "Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches," *AI*, vol. 2, no. 1, pp. 1–16, Jan. 2021. https://doi.org/10.3390/ai2010001

[19] N. Harrison, *Code Generation with Roslyn*. Springer, 2017.

[20] F. Nagel, G. M. Bierman, and S. D. Viglas, "Code generation for efficient query processing in managed runtimes," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 12, pp. 1095–1106, Aug. 2014. https://doi.org/10.14778/2732977.2732984

[21] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*, San Francisco, CA, USA, May 2013, pp. 83–92. https://doi.org/10.1109/ICPC.2013.6613836

[22] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *2008 32nd annual IEEE software engineering workshop*, Kassandra, Greece, Oct. 2008, pp. 169–174. https://doi.org/10.1109/SEW.2008.17

[23] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, Raleigh, NC, USA, Oct. 2017, pp. 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[24] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, Jan. 2002. https://doi.org/10.1046/j.1365-2575.2002.00117.x

[25] M. Harman, "Why source code analysis and manipulation will always be important," in *2010 10Th IEEE working conference on source code analysis and manipulation*, Timisoara, Romania, Sep. 2010, pp. 7–19. https://doi.org/10.1109/SCAM.2010.28

[26] S. Mukherjee, *Source Code Analytics with Roslyn and JavaScript Data Visualization*. Springer, 2016.

[27] T. W. Thomas, H. Lipford, B. Chu, J. Smith, and E. Murphy-Hill, "What questions remain? An examination of how developers understand an interactive static analysis tool," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, Jun. 2016.

[28] A. R. Yazdanshenas and L. Moonen, "Crossing the boundaries while analyzing heterogeneous component-based software systems," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Williamsburg, VA, USA, Sep. 2011, pp. 193–202. https://doi.org/10.1109/ICSM.2011.6080786

[29] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, Jul. 2008. https://doi.org/10.1016/j.entcs.2008.06.039

[30] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, Jul.–Aug.2006. https://doi.org/10.1109/MS.2006.114

[31] N. E. Fenton and M. Neil, "Software metrics: roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, May 2000, pp. 357–370. https://doi.org/10.1145/336512.336588

[32] F. G. Toosi, J. Buckley, and A. R. Sai, "Source-code divergence diagnosis using constraints and cryptography," in *Proceedings of the 13th European Conference on Software Architecture, ECSA '19*, vol. 2, New York, NY, USA, Sep. 2019, pp. 205–208. https://doi.org/10.1145/3344948.3344983

[33] Z. Zhioua, S. Short, and Y. Roudier, "Static code analysis for software security verification: Problems and approaches," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, Vasteras, Sweden, Jul. 2014, pp. 102–109. https://doi.org/10.1109/COMPSACW.2014.22

[34] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*, Sep. 2012, pp. 7–10. https://doi.org/10.1145/2372225.2372230

[35] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, Oct. 2008, pp. 31–38. https://doi.org/10.1145/1456362.1456370

[36] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information systems journal*, vol. 12, no. 1, pp. 43–60, 2002.

[37] E. L. Vargas, J. Hejderup, M. Kechagia, M. Bruntink, and G. Gousios, "Enabling real-time feedback in software engineering," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, Sydney, NSW, Australia, May 2018, pp. 21–24. https://doi.org/10.1145/3183399.3183416

[38] W. Maalej and D. Pagano, "On the socialness of software," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, Dec. 2011, pp. 864–871. https://doi.org/10.1109/DASC.2011.146

[39] E. Soares, G. Sizilio, J. Santos, D. A. da Costa, and U. Kulesza, "The effects of continuous integration on software development: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 3, Mar. 2022, Art. no. 78. https://doi.org/10.1007/s10664-021-10114-1

[40] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, Mar. 2017. https://doi.org/10.1109/ACCESS.2017.2685629
[41] S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," in *2018 Moratuwa Engineering Research Conference (MERCon)*, Moratuwa, Sri Lanka, May–Jun. 2018, pp. 156–161. https://doi.org/10.1109/MERCon.2018.8421965
[42] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, Nov. 2015, pp. 574–584. https://doi.org/10.1109/ASE.2015.36
[43] D. A. Plaisted, "Source-to-source translation and software engineering," *Journal of Software Engineering and Applications*, vol. 6, no. 4A, pp. 30–40, Apr. 2013. https://doi.org/10.4236/jsea.2013.64A005
[44] M. Harsu, "Identifying object-oriented features from procedural software," *Nordic Journal of Computing*, vol. 7, no. 2, pp. 126–142, 2000.
[45] S. Corporation, "Coverty: Static analysis tool," Synopsys Documentation Portal. [Online]. Available: https://www.synopsys.com/software-integrity/static-analysis-tools-sast/coverity.html
[46] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto,* Portugal, 2009. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ce3c584c906eea668954f6a1a0ddbb295c6ec5a2
[47] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. Soares Cruzes, "Myths and facts about static application security testing tools: An action research at telenor digital," in *Agile Processes in Software Engineering and Extreme Programming. XP 2018. Lecture Notes in Business Information Processing,* J. Garbajosa, X. Wang, and A. Aguiar, Eds., vol 314. Springer, Cham. https://doi.org/10.1007/978-3-319-91602-6_6
[48] M. Zitser, "Securing software: An evaluation of static source code analyzers," Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.
[49] T. Hofer, "Evaluating static source code analysis tools," Tech. Rep., Ecole Polytechnique Fédérale de Lausanne, 2010. [Online]. Available: https://core.ac.uk/download/pdf/147963417.pdf
[50] M. Ashouri, "Practical dynamic taint tracking for exploiting input sanitization error in java applications," in *Information Security and Privacy: 24th Australasian Conference, ACISP 2019,* Christchurch, New Zealand, Jul. 2019, pp. 494–513. https://doi.org/10.1007/978-3-030-21548-4_27
[51] N. Manzoor, H. Munir, and M. Moayyed, "Comparison of static analysis tools for finding concurrency bugs," in *2012 IEEE 23rd international symposium on software reliability engineering workshops*, Dallas, TX, USA, Nov. 2012, pp. 129–133. https://doi.org/10.1109/ISSREW.2012.28
[52] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? An empirical study of false negatives in static bug finding tools," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2012, pp. 50–59. https://doi.org/10.1145/2351676.2351685
[53] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, vol. 2, May 2010, pp. 99–108. https://doi.org/10.1145/1810295.1810310
[54] E. E. Schultz Jr, D. S. Brown, and T. A. Longstaff, "Responding to computer security incidents: Guidelines for incident handling," Tech. Rep., Lawrence Livermore National Lab., CA (USA), 1990.
[55] A. S. S. C. A. T. K. SonarQube, "Code Quality Tool & Secure Analysis with SonarQube." https://www.sonarsource.com/products/sonarqube/
[56] J. Novak, A. Krajnc, and R. Zontar, "Taxonomy of static code analysis tools," in *The 33rd International Convention MIPRO*, 2010, pp. 418–422. [Online]. Available: https://www.researchgate.net/publication/251940397_Taxonomy_of_static_code_analysis_tools
[57] E. E. Mills, *Software Metrics*. Software Engineering Institute, 1988.
[58] L. Rosenberg, T. Hammer, and J. Shaw, "Software metrics and reliability," in *9th international symposium on software reliability engineering*, Nov. 1998.
[59] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994. https://doi.org/10.1109/32.295895
[60] MacDonell, S.G., Buckingham, D., Gray, A.R. and Sallis, P.J., 2002. Software forensics: extending authorship analysis techniques to computer programs. Journal of Law and Information Science, 13, pp.34-69. https://openrepository.aut.ac.nz/server/api/core/bitstreams/963770a7-cd78-4105-8385-99c6b0c4f64e/content
[61] "PYPL PopularitY of Programming Language index." [Online]. Available: https://pypl.github.io/PYPL.html
[62] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, Apr. 2006. https://doi.org/10.1109/TSE.2006.38
[63] D. Guaman, P. Sarmiento, L. Barba-Guaman, P. Cabrera, and L. Enciso, "SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis," in *7th International Workshop on Computer Science and Engineering, WCSE*, Jan. 2017, pp. 171–175.
[64] J. García-Munoz, M. García-Valls, and J. Escribano-Barreno, "Improved metrics handling in SonarQube for software quality monitoring," in *Distributed Computing and Artificial Intelligence, 13th International Conference*, S. Omatu *et al*., Eds., Springer Cham, Jun. 2016, pp. 463–470. https://doi.org/10.1007/978-3-319-40162-1_50
[65] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube rules inducing bugs?" in *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*, London, ON, Canada, Feb. 2020, pp. 501–511. https://doi.org/10.1109/SANER48275.2020.9054821
[66] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, "Continuous code quality: are we (really) doing that?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sep. 2018, pp. 790–795. https://doi.org/10.1145/3238147.3240729
[67] D. Marcilio, R. Bonifacio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? A closer look at realistic usage of SonarQube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, Montreal, Canada, May 2019, pp. 209–219. https://doi.org/10.1109/ICPC.2019.00040
[68] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for java concurrency bugs," in *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, 2010, pp. 18–19.
[69] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, Apr. 2023, Art. no. 111575. https://doi.org/10.1016/j.jss.2022.111575
[70] R. P. Jetley, P. L. Jones, and P. Anderson, "Static analysis of medical device software using CodeSonar," in *Proceedings of the 2008 workshop on Static analysis*, Jun. 2008, pp. 22–29. https://doi.org/10.1145/1394504.1394507
[71] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Osaka, Japan, Mar. 2016, pp. 470–481. https://doi.org/10.1109/SANER.2016.105
[72] Snyk, "Snyk analysis tool for code security & code quality scanning," 2023. [Online]. Available: https://snyk.io/product/snyk-code/
[73] S. A. Licorish and M. Wagner, "Combining GIN and PMD for code improvements," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Jul. 2022, pp. 790–793. https://doi.org/10.1145/3520304.3528772
[74] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, Trento Italy, Jul. 2010, pp. 241–252. https://doi.org/10.1145/1831708.1831738
[75] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv preprint arXiv:2110.09610*, 2021.

Vikram Bhutani is a Principal Engineer at Intel Corporation™ and PhD scholar at MTU, Cork, Ireland. Vikram holds as M.Sc. degree in Artificial Intelligence (AI) from the University of Limerick, Ireland.
Vikram embarked on his journey with Intel in 2002, amassing an impressive 22-year tenure as a professional in the area of Pre-Si RTL validation/verification. Throughout his extensive career, Vik has been an integral force behind numerous CPU client projects in Intel, showcasing his expertise and leadership in key validation areas, most recently on AI in PC projects. As an IEEE Senior Member and a Fellow of IETE (India) and IEI (India), Vik has authored multiple publications in the areas of machine learning, x86 IA core validation, power and performance, and circuit testability.
E-mail: vikram.bhutani@intel.com
ORCID iD: https://orcid.org/0000-0002-3565-2581

Farshad Ghassemi Toosi earned his PhD in Computer Science from the University of Limerick in Ireland in 2017. His Doctoral research focused on data visualisation, particularly in the domain of Graph Drawing, and the application of Genetic Algorithms in this field. Following the completion of his PhD, he served as a PostDoctorate for approximately two years, concentrating on Software Engineering with a specific emphasis on Source Code Manipulation and Feature Location. Subsequently, he assumed the role of a full-time Lecturer at the Department of Computer Science of Munster Technological University in Cork starting from 2019. Farshad Ghassemi Toosi is an esteemed academic member of Lero, the SFI Research Centre for Software in Ireland.
E-mail: farshad.toosi@mtu.ie
ORCID iD: https://orcid.org/0000-0002-1105-4819

Jim Buckley received the PhD degree in Computer Science from the University of Limerick, in 2002. He is a Professor at the Computer Science and Information Systems Department, University of Limerick, Ireland, and is a co-principal investigator with Lero, the Irish Research Centre for Software. He was awarded the (Lero) Director's prize for Research Excellence in 2020 and his main research interests focus on supporting software developers who are tasked with maintaining and evolving software systems. Thus, specific areas of interests include feature location, software comprehension, and architectural analysis of such systems.
Email: Jim.Buckley@ul.ie
ORCID iD: https://orcid.org/0000-0001-6928-6746