A SYSTEMIC REVIEW OF STATIC MEMORY ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

This review aims to evaluate and compare various static analysis tools across multiple programming languages for memory management. The tools and techniques under scrutiny include pattern matching, symbolic execution, CppCheck, SharpChecker, FindBugs, CheckStyle, and Pylint. When examining the methods, pattern-matching, and symbolic execution, we identified implementations using pattern-matching and symbolic execution for each programming language. We focus on understanding the full scope of their capabilities and effectiveness in managing internal and external memory components such as RAM, SRAM, PROM, Cache, Optical Drive, etc. While static analysis tools do not directly analyze physical memory components, they are crucial in enhancing memory behavior. By detecting and addressing memory-related issues early in the development process, these tools contribute significantly to the overall quality of software systems. This review will thoroughly examine the strengths and weaknesses of each static analysis tool, aiding in selecting the most suitable tool or combination of tools for effective memory management across diverse programming environments.

1 INTRODUCTION

Software systems have become more complex as they must ensure their software's reliability, security, and management. Not upkeeping software systems may result in software defects. Those include (Emanuelsson & Nilsson, 2008) logical or functional errors, where the program sometimes computes incorrect values; runtime errors, where the program typically crashes; resource leaks, where the performance of the program degrades until the program freezes or crashes; and minuscule security vulnerabilities that malicious attackers can exploit to obtain control over computers.

Various methods are employed to identify software errors and defects, using tools compiled explicitly for certain programming languages or implementing a feature to assist. Tools specifically compiled for certain languages include CppCheck for C/C++ programs, FindBugs for Java programs, and many more. Features implemented to assist include pattern matching and specific functionalities within a given tool. Software developers use those tools and features to mitigate the risks mentioned earlier.

1.1 DEFINITION AND OBJECTIVES

Static analysis Rival & Yi (2020) is a technique aimed at discovering expressive properties of program code without executing it. It shouldn't be confused with dynamic analysis, (Ernst, 2003) which operates by running a program and observing its outputs or executions. Static analysis tools are (Lenarduzzi & Fabio, 2023) instruments that examine a program's source code without executing it to discover potential quality issues such as resource management, syntax errors, and runtime behavior. These tools benefit developers, as they help build high-quality software, reduce the risk of security breaches, and minimize debugging time. However, some techniques lack scalability, and analyzing large software programs may require more advanced computational tools to ensure accurate results, which current tools may not consistently deliver.

Our **objective** is to gather research on static memory analysis and identify the most efficient tools for memory management. This work comprehensively reviews static memory analysis techniques and tools to detect source code defects across programming languages, as presented in Table 1.

Using insights from academic research, we will develop a process to perform a complete memory analysis on a virtual machine, covering internal (e.g., RAM, DRAM, SRAM, ROM, PROM, EPROM, Cache) and external memory components (e.g., Optical Drive, Solid State Drives, and Virtual memory).

Table 1: Selected static analysis tools and corresponding programming languages

Tool/Technique	Developer Language	C/C++	C#	Java	Python
Mach7	C/C++	Yes	No	No	No
CppCheck	C/C++	Yes	No	No	No
Clang Static Analyzer	C/C++	Yes	No	No	No
Language-Independent Tool for C#	C#	No	Yes	No	No
SharpChecker	C#	No	Yes	No	No
Symbolic JPF	Java	No	No	Yes	No
FindBugs	Java	No	No	Yes	No
CheckStyle	Java	No	No	Yes	No
Python CHEF Engine	Python	No	No	No	Yes
Pylint	Python	No	No	No	Yes

This paper is structured as follows: Section 1 covers the background of software system defects and errors and introduces static and dynamic analysis, emphasizing the critical difference between them. Section 2 introduces various static analysis tools and their general features. Section 3 composes literature reviews of the innovative implementations of these tools conducted by researchers and developers to address successes and gaps in the existing tools. Section 4 compares and contrasts the approaches of each tool. Finally, section 5 concludes the paper by summarizing the findings, identifying lessons learned, and describing our future work in developing our automated process.

The main contributions of this paper are as follows:

- We reinforce existing research and tools, offering a clear understanding of the current outlook of static analysis.
- By reviewing the existing literature, we highlight the strengths and weaknesses of different static analysis tools and methodologies' effectiveness in memory management, as proper memory management can ensure efficiency, stability, security, data integrity, and scalability in a program.
- We provide a foundation for future studies by laying out what has already been explored, giving practitioners potential areas for further investigation and implementation.

2 METHODOLOGIES

2.1 WHAT MAKES A GOOD STATIC ANALYSIS TOOL?

As previously mentioned, the primary purpose of a static analysis tool is to discover runtime errors, resource leaks, and security vulnerabilities without executing the code. However, finding a tool that identifies all defects without limitations is challenging. One common issue is the prevalence of false positives (i.e., reporting non-issues) and false negatives (i.e., missing actual problems). Managing and detecting these is crucial, but many tools do not explicitly label them as functionalities. Instead, they address these through configurable settings, algorithm improvements from user feedback, and data-flow or path-sensitive analysis integration that allows a user to determine the feasibility of a path and optimize resource usage.

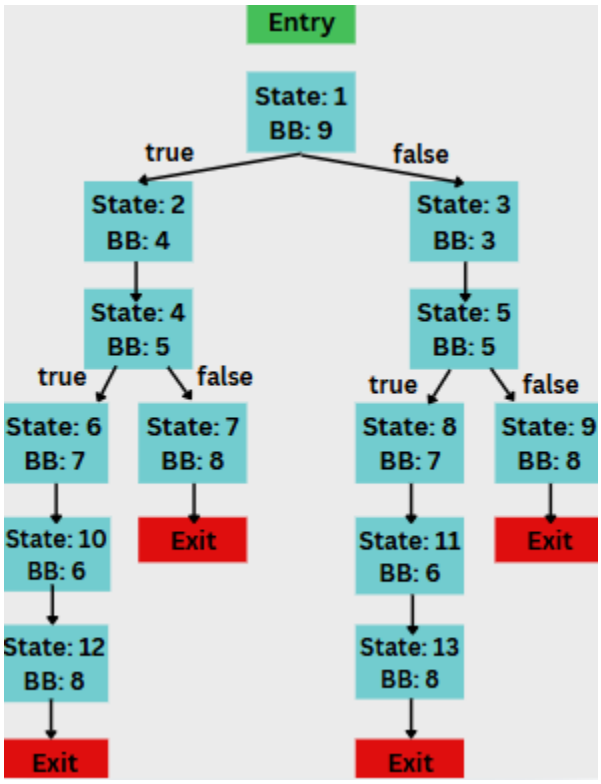

Figure 1: Symbolic Execution Tree referenced from RTEHunter.

2.2 Static Analysis Tools

2.2.1 Pattern Matching

Pattern matching is a technique that (Ferrara, 2023) for checking if an expression conforms to a particular pattern by examining a series of case expressions. Static analysis tools (Wen, 2024) utilize pattern matching alongside techniques such as symbolic execution and separation logic to identify issues such as **poor use of language constructs** and **violation of coding guidelines**. Identifying those issues helps (par, 2024) prevent defects such as resource leaks, performance and security issues, and API misuse. Tools that implement pattern matching, like CppCheck, have been observed by developers in its earlier versions (Gulabovska & Porkolab, 2019) to report relatively low false positives and identified that well-written regular expressions are more effortless to predict. An implementation that we reviewed to use pattern-matching was **Mach7**.

2.2.2 CppCheck

CppCheck (Moerman, 2018) is an open-source tool similar to C++test, which is capable of analyzing C and C++ code with simple control flow analysis. While CppCheck cannot filter certain defects through command-line flags, it offers useful features like sorting defect reports by severity and analyzing entire code bases using a compilation database that can manually remove the need to provide paths to all source files.

2.2.3 Symbolic Execution

Symbolic execution (Păsăreanu & Visser, 2009) is a notable static analysis technique that uses symbolic values to represent program inputs rather than initialized data, allowing for the manipulation of program expressions symbolically. It is considered highly powerful as it (Gulabovska & Porkolab, 2019) leverages program structure, type systems, and data-flow information while following function calls. Symbolic execution can handle complex constructs such as recursive data structures,

arrays, preconditions, and multithreading. However, when applied to programs with loops or recursion, it can result in an unbounded number of execution paths, leading to resource-intensive processes. Those results occur because loops and recursion can generate infinite possible states. As a result, symbolic execution may not terminate due to the explosion of configurations. [1]

Additionally, the analysis state is tracked using an exploded graph, which can grow exponentially in size due to program control branches (i.e., conditions). To mitigate the resource cost, constraints or heuristics are introduced to limit exploration depth or prioritize paths, ensuring efficient execution.

Tools that we observed to implement symbolic execution were the **Clang Static Analyzer (CSA)**, the **Language-Independent Tool for Symbolic Execution in C#**, **Symbolic JPF**, and the **Python CHEF Engine using Symbolic Execution**.

2.2.4 SHARPCHECKER

SharpChecker is a static analysis tool (Koshelev & Belevanstev, 2017) that can detect 30 different types of bugs through parsing, data-flow analysis, and (the most significant) context and path-sensitive intraprocedural analysis. SharpChecker employs the **Roslyn** infrastructure to work with C# build system files and compile a program's source code. SharpChecker only uses the part of **Roslyn** that is responsible for parsing a project and solution files to determine a set of files for building a given program or library and to set an adequate environment.

2.2.5 FINDBUGS

FindBugs (Lenarduzzi & Fabio, 2023) is a static analysis tool mainly used to evaluate Java byte code. However, the tool's capability goes beyond that, as it can also highlight the exact position of an issue if the tool is provided the source code. The analysis is performed by detecting bug patterns that, according to FindBugs (Shen, 2011), can arise due to complex language features, misunderstood API features, misunderstood invariants when code undergoes modification during maintenance, and garden variety mistakes. Then, these bug patterns are categorized into nine categories: **bad practice**, **correctness**, **experimental**, **internalization**, **malicious code vulnerability**, **multi-threaded**, **performance**, **security**, and **dodgy code**. Per their observations, Shen et al. categorize three main ranking options for FindBugs' error reports as sorting reports in alphabetical order (e.g., bug patterns, packages), sorting reports in severity (i.e., assigning each error report a priority value of low, medium, or high), and sorting reports through user designation (i.e., allows users to manually designate each bug report according to categories (e.g., needing further study, not a bug, etc.).

Despite the number of ranking methods in FindBugs, users still face the risks of false positives and high inspection costs when dealing with large-scale software systems. Thus, FindBugs' developers (Shen, 2011) have two main goals for FindBugs:

1. The developers want to achieve a low false positive rate, particularly in correctness categories, which are primarily used to identify genuine errors in software systems.
2. Error reports of the same bug pattern(s) or bug kind(s) should ideally have the same designation since it would facilitate the management of error reports in groups.

FindBugs has been succeeded by *SpotBugs*. SpotBugs continued the legacy of FindBugs by providing advanced bug detection capabilities, with improvement and support for newer versions of Java.

[1] A symbolic execution tree (Păsăreanu & Visser, 2009) represents the potential execution paths a program can follow during symbolic execution. Each tree node corresponds to a program state, and transitions between nodes represent program actions or changes in state, as seen in Fig. 1.

2.2.6 CHECKSTYLE

CheckStyle (Yeboah & Popoola, 2023) is an open-source tool for evaluating Java code quality. Developers can use CheckStyle via a command-line tool or integrate it with Ant (i.e., Java library and command-line tool). CheckStyle processes code based on a configured set of checks, supporting standard configurations like ***Google Java Style*** and ***Sun Java Style*** while allowing personalized configurations as well. The checks are classified into 14 categories: annotations, block checks, class design, coding, headers, imports, Javadoc comments, metrics, miscellaneous, modifiers, naming conventions, regexp, size violations, and whitespace. Check violations (Lenarduzzi & Fabio, 2023) are grouped under two severity levels: **error** (actual problems) and **rule** (potential issues requiring verification).

2.2.7 PYLINT

Pylint (Gulabovska & Porkolab, 2019) is a static analysis tool for Python, capable of detecting logical errors, producing warnings regarding specific coding standards, offering details about code complexity, and suggesting refactoring. It leverages abstract syntax trees (AST) to manipulate source code and provides a range of code analysis and transformation pathways. [2]

2.3 DATASET

The first part of the project consists of a preliminary analysis of the chosen static analysis tools to determine which tool or tools is most efficient in managing the memory of a virtual machine. The dataset comprises open-source projects written in Java, C/C++, C#, and Python. However, the evaluations of each tool will be derived from the findings and results of existing implementations, providing insights into the capabilities and limitations of these tools across various programming languages.

3 LITERATURE REVIEW

In this section, we summarize the implementations of each tool and explore the similarities and differences of how one tool is typically applied compared to another.

3.1 MACH7

Introduced by Solodkyy (2013), they presented a functional-style pattern matching for C++ built as an ISO C++11 library, called ***Mach7***. They intended their solution to support the introduction of new patterns, ensure type safety, and provide a unified syntax for hierarchical data types. They performed several independent studies of their pattern-matching solution to test its efficiency and impact on the compilation process (i.e., transforming readable C++ code into machine-executable instructions).

3.1.1 CPPCHECK

Penttila (2014) analyzes the capability of ***CppCheck*** in analyzing C/C++ programs. He observed some of the functionalities for CppCheck using abstract syntax tree analysis, data-flow analysis, severity categorization, and an inconclusive flag check that can be enabled for specific checks.

3.1.2 CLANG STATIC ANALYZER

Kovacs & Porkolab (2019) introduces symbolic execution in C/C++ programs by focusing on the methods used by *CSA*. They implemented a module called *Inner Pointer Checker* that tracts raw inner pointers of strings and recognizes operations on the string that may corrupt the inner buffer.

[2] An abstract syntax tree is a data structure used to reason about the grammar of a programming language in the context of the instructions provided in the given source code. Static analysis tools utilize abstract syntax trees for tasks such as linting, refactoring, and optimization.

Balogh & Szalay (2024) reviewed *CSA* by analyzing reports from five products: Linux, FreeBSD, SerenityOS, systemd, and QEMU. They examined reports with both short and long bug paths. They found that longer bug paths made bugs harder to understand, indicating a more significant divergence between the analyzer's assumptions and reality. While they identified many true positives, such as in QEMU's switch statement, they also encountered numerous false positives, especially in low-level system software, where memory manipulation is complex for the current checkers to model. Despite this, the "core." checkers, essential for building exploded graphs, cannot be disabled without compromising analysis accuracy. However, the **unix.Malloc** checker, which struggles with dynamic memory handling in low-level code, can be safely disabled. Ultimately, Balogh and Szalay recommend keeping most default checkers in v6.22.1 but suppressing particular "core." checkers like **core.NullDereference** due to their high false positive rate in low-level contexts.

3.1.3 LANGUAGE-INDEPENDENT TOOL FOR C#

In the implementation made by Arusoaie (2015), we are presented with a ***language-independent symbolic execution framework for C#*** programs using Reachability Logic, featuring a special rule called Circularity for handling loops and recursion. They identify that it may have performance limitations compared to language-specific tools. Still, it counteracts that by offering a generic and robust tool for automated symbolic execution and program verification to ensure soundness through the Circularity Principle.

3.1.4 SHARPCHECKER

(Ignatyev & Mitrofanov, 2024) introduce large language models (LLMs) and identify that no known industrial static analysis tools successfully utilize LLMs for error detection. So, for their proposed approach, they conduct an initial evaluation of the applicability of LLMs in real-world projects on an existing set of 2230 tests. The tests were designed explicitly for ***SharpChecker***.

3.1.5 SYMBOLIC JPF

Păsăreanu & Visser (2009) developed a source-to-source translation method that introduces nondeterminism and supports path condition manipulation in Java programs, allowing Java PathFinder (JPF), a model-checking tool to explore the symbolic execution tree of an analyzed program, to perform symbolic execution by exploring symbolic state spaces. To simplify the process, they developed the ***Symbolic JPF*** framework, which enables symbolic execution directly on Java bytecodes without requiring code transformation, effectively finding errors in safety properties and generating test inputs. This approach helps address issues in Java programs related to safety and correctness.

3.1.6 FINDBUGS

Holsinger (2008) set up an assessment of ***FindBugs***, where they applied the tool to the open-source Java project, *JEdit*. They use this assessment to determine FindBugs' strengths and weaknesses in detecting **false positives** and **true positives**.

3.1.7 CHECKSTYLE

Oskouei & Kalıpsız (2018) used ***CheckStyle*** to identify defects in four open-source projects written in Java programs. The projects chosen are development projects from the company, *Sahand Iran*, with various development efforts and sizes. An experienced developer carefully examined each warning FindBugs generated to determine its validity as a defect.

3.1.8 PYTHON CHEF ENGINE

Bucur (2014) used CHEF, a tool that executes the target program by symbolically executing the interpreter's binary while exploiting inferred knowledge about the program's high-level structure, to develop a symbolic execution engine for *Python*. They began their testing phase by evaluating two engines on 11 popular Python library packages that generated up to 1000 times more test cases than applying plain symbolic execution on the interpreter executable. The tests generated acquired good coverage results and detected several bugs. They also compared the Python engine to other implementations.

3.1.9 PYLINT

Kiska (2021) used ***Pylint*** to determine its effect on given test cases. Examining the gathered results focuses on whether the tool discovered a specified error or resulted in a false positive or negative. We selected five of the twenty test cases that Kiska evaluated to observe and discuss: checking for an invalid number of arguments, membership support, undefined arguments, function argument type, and unused or missing imports.

4 DISCUSSION

4.1 C/C++

4.1.1 **COMPARISONS**: MACH7, CPPCHECK, AND THE CLANG STATIC ANALYZER

CppCheck and CSA allow custom checks, extendable frameworks, or modules to enhance code analysis and can detect issues such as dead code and uninitialized variables. Mach7 and CSA are made to implement standard static analysis techniques (e.g., pattern matching and symbolic execution).

Mach7, CppCheck, and CSA enhance code quality through different analysis methods: Mach7 relies on pattern matching, CppCheck uses data-flow analysis, and CSA uses symbolic execution. CppCheck amplifies error detection by categorizing errors and integrating a flag to alert for more checks in checks that would increase false positives. However, CppCheck doesn't provide as many checkers as CSA and executes them quickly.

We discussed two different observations/implementations of CSA: Kovacs et al. introduced a new module to enhance an established checker in CSA, and Balogh and Szalay evaluated the established checkers on different test cases. Balogh and Szalay observed that CSA has difficulty detecting false positives, especially in low-level contexts. At the same time, Kovacs et al. attempted to reduce false positives by enhancing their specialized checks. In evaluating CSA, Balogh and Szalay conclude that CSA offers comprehensive reporting, but long bug paths may hinder user comprehension and require more resources to analyze the code. On the other hand, Kovacs et al.'s approach aims to simplify bug tracing by providing more context on specific memory issues by implementing new modules.

4.2 C#

4.2.1 **COMPARISONS**: LANGUAGE-INDEPENDENT TOOL USING SYMBOLIC EXECUTION AND SHARPCHECKER

Arusoaie's language-independent tool aims to be broadly applicable. We assume that this includes implementing path-sensitive analysis, which can accurately track the state of the current program across different execution paths and is a standard analysis method in tools integrated with symbolic execution. Koshelev et al., as well, utilize *path-sensitive analysis* in SharpChecker for intra-procedural analysis, as it (Koshelev & Belevanstev, 2017) allows one to detect bugs such as resource leaks, NULL dereferences, and type casting errors.

In contrast, Arusoaie's framework aims for a language-independent approach to make it applicable to various programming languages, not just C#. Simultaneously, SharpChecker is tailored explicitly for C#, leveraging language-specific features. SharpChecker is more efficient for C# but lacks the flexibility of a language-independent framework.

4.3 JAVA

4.3.1 **COMPARISONS**: SYMBOLIC JPF, FINDBUGS, CHECKSTYLE

Symbolic JPF employs path-sensitive analysis, while FindBugs and CheckStyle rely on pattern matching, balance thoroughness, and stylistic checks without deep-path exploration (path-sensitive analysis). FindBugs offers a broad range of defect detection with a good balance of performance and meticulousness in Java bytecode. CheckStyle focuses on upkeeping code quality style and standard checks, ensuring a program's maintainability, not its functional accuracy, like Symbolic JPF.

4.3.2 FINDBUGS (NOW SPOTBUGS)

As previously mentioned, FindBugs, discontinued in 2015, was succeeded by *SpotBugs*, which inherited its features while adding new functionalities, bug detectors, and support for modern Java versions (Java 17 and beyond). SpotBugs retains FindBugs' existing bug detectors but introduces new categories like EXPERIMENTAL and SECURITY. It also enhances support for custom plugins, allowing users to tailor the tool to their specific analysis needs. This transition reflects FindBugs developers' success in creating a platform with a low false positive rate, particularly in correctness categories. It refines bug patterns for greater accuracy, focusing on detectors like CORRECTNESS and MT_CORRECTNESS. It also assigns specific designations to each bug to help developers prioritize issues.

4.4 PYTHON

4.4.1 **COMPARISONS**: PYTHON CHEF ENGINE AND PYLINT

The Python CHEF Engine and Pylint identify logical and syntax errors, using test cases to examine their ability to detect runtime issues. While they differ in their analysis methods, Python CHEF Engine uses path-sensitive analysis, and Pylint employs AST analysis, both of which offer extensive program coverage. Pylint is more experienced in handling false positives and negatives, making it suitable for routine code checks and detecting common mistakes. At the same time, Python CHEF Engine is ideal for thoroughly testing critical applications where reliability and correctness are the main factors.

5 CONCLUSION

The tools we observed, shown in Table 1, revealed that five of the ten static analysis tools (e.g., CppCheck, FindBugs, CheckStyle, Pylint, SharpChecker) we reviewed were independent tools. At the same time, the other five (e.g., Mach7, Clang Static Analyzer, Symbolic JPF, Language-Independent Tool, Python CHEF Engine) were implemented as solutions using features of the techniques: pattern matching and symbolic execution to make up for functionalities that the current tools did not possess.

However, there is a significant difference between the two identified techniques– pattern matching and symbolic execution. Pattern matching is commonly used in static analysis tools that identify specific bug patterns (e.g., CppCheck, FindBugs, SharpChecker) and uphold common coding standards. Those are valuable applications of pattern matching, but those applications are also limited. Pattern matching can detect common bug patterns only if predefined, making it easier to dismiss undefined ones. On the other hand, symbolic execution, while also utilized by static analysis tools, is more used as an implementation or branch of a tool rather than a function.

The other static analysis tools, while effective in identifying their designated issues, lack some crucial functions for memory management and error detection, making them unsuitable for a fully efficient static analysis tool. For instance, CheckStyle, despite preventing some common errors by enforcing coding standards that maintain readability and consistency, does not significantly contribute to memory management. FindBugs' detection of memory issues is limited to recognizable patterns, making it ineffective in detecting unknown bug patterns and upholding code quality.

CppCheck, while effective in detecting a wide range of memory-related issues (e.g., memory leaks, array-bound errors), is lacking due to its generation of false positives, a common problem for static analysis tools. Pylint, although helpful in maintaining code quality, like CheckStyle, cannot perform deep memory analysis and, therefore, cannot detect memory management issues like memory leaks.

Overall, our objective was to gather research on static memory analysis and identify the most efficient tools to utilize in our automated process to perform a memory analysis on a virtual machine. From our findings and observations, symbolic execution is the most effective technique because it is helpful in memory management and can be implemented with various static analysis tools. However, in terms of developing our automated process, it seems that it would be best to utilize symbolic execution with another static analysis so that a broader range of potential memory issues could be addressed.

5.1 Lessons Learned

Quite a few lessons have been learned in researching and examining the static analysis tools:

- We now understand the limits of static analysis more in-depth. Static analysis tools have proven worthwhile but are limited, as they can discover potential issues in source code pre-runtime but not all problems, especially those involving dynamic behavior.
- A few of the tools we observed, like CppCheck and CheckStyle, while differing in language support, are similar in functionality, like upholding coding standards. Style issues detected in CppCheck for C/C++ source code could also be detected in CheckStyle but for Java source code.
- The capabilities of each static analysis tool vary, as one may be more helpful in enforcing coding standards, while another may be more useful in detecting memory leaks. There is a more effective tool, but there is no right tool as some projects may require a combination of tools and others with just one specific feature.

5.2 Future Work

Our future work will extend the concept of static memory analysis to a VM (virtual machine). We will develop an automated process to perform a complete memory analysis on a virtual machine. We will examine various aspects of its memory, including Internal (e.g., RAM, DRAM, SRAM, ROM, PROM, EPROM, Cache) and External (e.g., Optical Drive, Solid State Drives, virtual memory). We will also identify any memory-related issues the tools detect (e.g., uninitialized variables and memory leaks). We will compare and contrast the results and develop multiple data visualizations to improve understanding.

References

What is static code analysis? a comprehensive overview. *Parasoft*, 2024.

Andrei Arusoaie. A generic framework for symbolic execution: theory and applications. *Inria*, 2015.

Á. Balogh and R. Szalay. On the applicability of static analysis for system software using codechecker. *7th International Conference on Software and System Engineering (ICoSSE)*, pp. 15–22, 2024.

Kinder Johannes Candea George Bucur, Stefan. Prototyping symbolic execution engines for interpreted languages. *ACM Conferences*, pp. 239–254, 2014.

Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Science Direct*, 217:5–21, 2008.

Michael Ernst. Static and dynamic analysis: Synergy and duality. *WODA 2003*, pp. 24–27, 2003.

Pietro Ferrara. Static type analysis of pattern matching by abstract interprettion. *Formal Techniques for Distributed Systems*, 6117:186–200, 2023.

Hristina Gulabovska and Zoltan Porkolab. Survey on static analysis tools of python programs. *In Proceedings of the SQAMIA 2019: 8th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications*, 2508, 2019.

Fulzele Snehal Ramteke Smita Tamagawa Ken Wesaratchakit Sahawut Holsinger, Lyle. Prevent vs. findbugs application and evaluation. *Carnegie Mellon University*, 2008.

Shimchik N. V. Panov D. D. Ignatyev, V. N. and A. A. Mitrofanov. Large language models in source code static analysis. *Ivannikov Memorial Workshop (IVMEM)*, pp. 28–35, 2024.

Jakub Kiska. Static analysis of python code. *Theses*, 2021.

Ignatiev V. N. Borzilov A. I. Koshelev, V. K. and A. A. Belevanstev. Sharpchecker: Static analysis tool for c programs. *Springer*, 43:268–276, 2017.

Horvath Gabor Kovacs, Reka and Zoltan Porkolab. Detecting c++ lifetime errors with symbolic execution. *ACM Conferences*, pp. 1–6, 2019.

Percorelli Fabiano Saarimaki Nyyti Lujan Savanna Lenarduzzi, Valentina and Palomba Fabio. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, 2023.

Jonathan Moerman. Evaluating the performance of open source static analysis tools. *Radboud University*, 2018.

Elmira Hassani Oskouei and Oya Kalıpsız. Comparing bug finding tools for java open source software. figshare. *figshare*, 2018.

Elias Penttila. Improving c++ software quality with static code analysis. *Aaltodoc Repository*, 2014.

Corina Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11:339–353, 2009.

Xavier Rival and Kwangkeun Yi. Introduction to static analysis. *The MIT Press*, 2020.

Fang Jianhong Zhao Jianjun Shen, Haihao. Efindbugs: Effective error ranking for findbugs. *IEEE Fourth International Conference on Software Testing, Verification, and Validation*, 2011.

Reis Gabriel Dos Stroustrup Bjarne Solodkyy, Yuriy. Open pattern matching for c++. *ACM Conferences*, pp. 33–42, 2013.

Cai Yuandao Zhang Bin Su Jie Xu Zhiwu Liu Dugang Qin Shengchao Ming Zhong Cong Tian Wen, Cheng. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Conferences*, 18, 2024.

Jones Yeboah and Saheed Popoola. Efficacy of static analysis tools for software defect detection on open-source projects. *Cornell University*, 2023.