International Journal of Innovation Studies

REINFORCING SOFTWARE VERIFICATION: STATIC CODE ANALYSIS FRAMEWORKS AND TOOLCHAINS

Dr. Pallavi Mandhare
Department of Computer Science, SPPU. Email: mandharepa@gmail.com

Abstract

Static code analysis tools are pivotal for identifying and mitigating software vulnerabilities, which significantly reduces development costs and enhances efficiency. By automating the detection of potential issues, these tools eliminate the need for extensive manual code reviews and streamline the development process, allowing programmers to focus on creating robust and secure software solutions. This paper explores the capabilities of static code analysis tools, particularly their role in detecting common software vulnerabilities. A comparative study evaluates various tools based on scalability, accuracy, usability within integrated development environments (IDEs) and optimizing software verification processes. The findings reveal that combining multiple static analysis techniques, such as abstract interpretation, data flow analysis, and program slicing can enhance software reliability and security. Machine learning approaches, including clustering for categorizing similar bugs and supervised learning for identifying vulnerabilities can be the part of tool to enhance the accuracy and effectiveness of these tools.

Keywords: Abstract Interpretation, Program Slicing, Data Flow Analysis, Static Analysis, Software Security, Machine Learning, Program Verification.

1 INTRODUCTION

In era of digital connectivity, ensuring correctness and security of software systems is most important, especially as software becomes the backbone of critical applications such as nuclear plant monitoring, autonomous vehicles, medical devices and financial systems. Unlike hardware components, which undergo rigorous physical testing, software often presents more significant challenges in guaranteeing quality due to its inherent complexity and susceptibility to vulnerabilities [1]. Software vulnerabilities can result in severe security failure, potentially causing system failures, unauthorized access and data leaks. For example, buffer overflow vulnerabilities typically arise from improper memory handling, such as failing to implement boundary checks for fixed-size buffers [4,7,10,15]. Such issues underscore the importance of systematic methods to detect and eliminate software vulnerabilities early in the development lifecycle. This necessity has led to the development of various program analysis techniques, broadly categorized into two main types: static program analysis and dynamic program analysis.

- Static program analysis (SPA) inspects code without executing it, focusing on its structure and behaviour by analyzing the source code or bytecode. This method is particularly effective in detecting vulnerabilities, coding errors, and quality issues before the software runs [11,20].
- Dynamic Analysis, in contrast, entails executing the program and observing its behaviour during runtime. Dynamic analysis tools inspect the actual behaviour of a program, often looking for memory leaks, performance bottlenecks, or runtime exceptions.

By applying methods like control flow analysis and data flow analysis, SPA tools can identify critical issues—such as uninitialized variables, resource leaks, and infinite loops during the development phase. These techniques reduce the efforts on time-consuming manual code reviews, enabling developers to improve software quality and enhance security. By detecting vulnerabilities early, SPA mitigates the risks associated with runtime errors and external attacks, ultimately reducing development and maintenance costs [12]. However, SPA is not without its limitations. For instance, static tools can produce false positives, flagging issues that do not actually pose a risk, which may lead to unnecessary debugging efforts. Additionally, they sometimes struggle to fully analyze dynamic aspects of the code, such as runtime dependencies and input/output interactions, which are often critical in complex applications [27].

Static Program Analysis (SPA) tools like Clang (built on LLVM), FindBugs, and SonarQube are integral to modern software development. They analyze code prior to execution, providing immediate feedback and enabling early error detection within continuous integration (CI) workflows [14].

Clang, a widely adopted open-source compiler, includes built-in static analysis to detect issues such as memory leaks and uninitialized variables at compile time. FindBugs targets Java applications, identifying potential flaws—particularly concurrency and thread safety problems—before runtime. SonarQube, widely used in CI environments, continuously assesses code quality, flags security vulnerabilities, and identifies code smells.

SPA offers a proactive alternative to traditional testing by catching issues early, reducing the effort and cost of debugging during later development stages or post-deployment. This is particularly valuable in agile development, where rapid and iterative changes are frequent. As shown in Fig. 1, development cycles now involve multiple feedback loops and minor iterations, reflecting increasing complexity. Beyond security, SPA complements compiler optimizations by exposing structural inefficiencies in the code. While compilers apply techniques like loop unrolling and memory optimization to improve runtime performance, SPA helps developers refactor code that might otherwise block such optimizations.

Together, static program analysis (SPA) techniques enhance both software performance and security [16]. In high-performance systems, SPA identifies redundant or inefficient code patterns that compilers can subsequently optimize for faster execution [3].

Recent advancements in machine learning have further strengthened SPA by automating the detection of complex patterns, anomalies, and security vulnerabilities that traditional rule-based methods may missed or excluded.

Over time, SPA has become a core component of modern software development methodologies. Leading organizations such as NASA rely on static analysis tools to ensure software reliability and robustness.

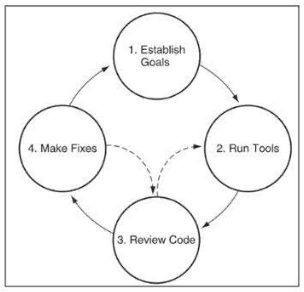


Figure 1 Code Review Cycle [21]

This paper focuses on SPA techniques, tools, and their impact on software quality and security. It highlights methods like abstract interpretation, program slicing, and data flow analysis, demonstrating how they reduce development costs and mitigate vulnerabilities. Additionally, the paper addresses key challenges in SPA adoption, such as managing false positives and integrating analysis within contemporary development workflows, offering practical strategies to overcome these issues.

The structure of the paper is as follows: Section 2 presents the fundamentals of SPA, Section 3 reviews key SPA techniques, Section 4 explores commonly used SPA tools, Section 5 discusses the integration of machine learning in static analysis, and Section 6 concludes the study.

2. FUNDAMENTALS OF STATIC PROGRAM ANALYSIS

Static Program Analysis (SPA) plays a vital role in software verification, helping detect vulnerabilities, enforce coding standards, enhance performance, and ensure code correctness prior to deployment. Rooted in formal methods, type theory, and mathematical abstractions, SPA enables automated reasoning about program behavior. Given that many behavior-related properties are undecidable or computationally hard, SPA aims to produce sound, efficient approximations that avoid misleading downstream analyses.

Due to the diversity of approaches addressing various aspects of program behavior, representing the entire landscape of SPA techniques is inherently complex. Nonetheless, Fig. 2 provides a high-level overview of widely adopted categories, offering insight into the breadth of static analysis methods in use today.

Modern software industries increasingly adopt SPA tools to tackle challenges such as scalability, accuracy, and seamless integration with Integrated Development Environments (IDEs) and Continuous Integration (CI) pipelines.

Many modern programming languages, especially statically typed ones, can detect bugs during compilation. Although testing has long been a standard method for identifying defects, it cannot guarantee the discovery of all bugs and is typically applied later in the development cycle. In addition to testing, techniques such as Formal Verification and Data Flow Analysis are also employed to improve software reliability [28].

Static Program Analysis (SPA) refers to analyzing code without executing it, typically during compilation. SPA tools compute information about the program's structure and behavior, which can then be leveraged by dynamic analysis to better understand runtime behavior.

While static analysis operates at compile time to infer the potential behavior of code, dynami

c analysis is performed during execution to observe how the program behaves in real environments. Importantly, these techniques can complement each other: SPA can provide input to dynamic analysis, and in turn, dynamic analysis results can inform or enhance static predictions, such as estimating likely execution paths or runtime behavior [2].

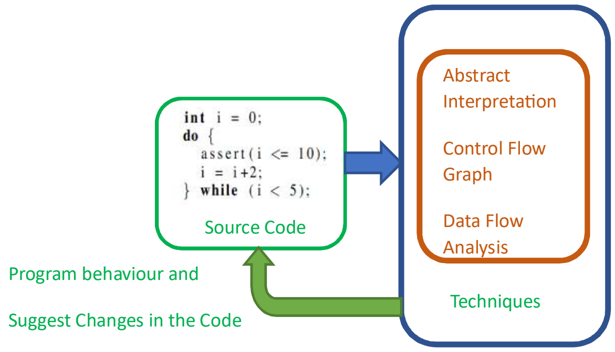


Figure 2 Static Program Analysis Technique's Overview

As per *Fig.3*, the input to the static analyzing techniques is the code/program. The output is shown such that, whether there are any mistakes in the given code without executing it. i.e. static analysis helps to fix the problem in the code to make the code safer and more reliable during execution. The primary function of static analysis is to detect and indicate code issues. Issues can include (not the coding standards) having dead code and unused data, dereference of a null or void pointer, security issues, infinite loops, and other arithmetic problems.

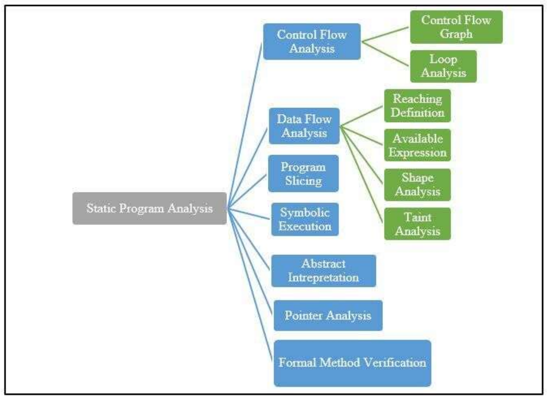


Figure 3 Static Program Analysis Overview

3. STATIC PROGRAM ANALYSIS TECHNIQUES

There are many run-time problems/issues, even compilers may not able to discover them. These problems/issues can be potentially identified by using SPA. Some of the examples are as follows: Resource Management, Wrong Operations, Dead Code, Incomplete Code.

These issues have a significant impact on the source code's operation and can lead to failures. Previously, the lexical analyzer was used to tackle static analysis, but now we have techniques and their tools to assist us in analysis, and they work better. In the following sections, we will discuss the different techniques available for automated static analysis, as well as certain metrics that can be used to examine these various tools.

3.1 Abstract Interpretation

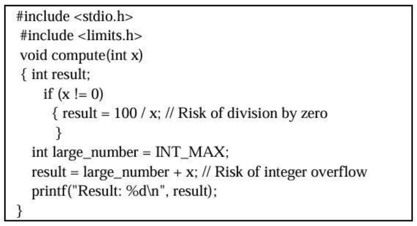
Abstract interpretation is a static program analysis (SPA) technique based on programming language semantics. It maps programs to mathematical abstractions that approximate input-output behavior, enabling the extraction of insights without requiring exact execution modeling [8, 22, 26].

This technique supports early detection of runtime errors, security vulnerabilities, memory safety issues, and performance bottlenecks. By choosing appropriate abstract domains, analysts can balance precision and efficiency, making the approach suitable for a wide range of verification tasks. Common domain types include: Numerical (e.g., interval, polyhedral), Logical and predicate-based, Memory and heap-focused, Control-flow, Security-oriented.

Despite its versatility, abstract interpretation faces challenges with scalability and computational overhead. Highly precise domains increase analysis time, while simpler abstractions may introduce false positives. To address this, recent work explores hybrid approaches, combining abstract interpretation with machine learning, model checking, and symbolic execution to improve both precision and practicality [23,24]. Abstract interpretation remains fundamental to formal verification, compiler optimization, and software security, valued for its theoretical rigor and adaptability to real-world systems.

A key application involves numerical abstractions, such as interval and polyhedral analysis, which track variable ranges to detect arithmetic errors like division by zero, integer overflows, and buffer overflows—frequent sources of system failures.

The technique is grounded in Galois connections, which formally relate concrete semantics (actual execution behavior) to abstract semantics (safe approximations). For example, in C programs involving arithmetic, interval analysis can statically detect errors that might otherwise cause unpredictable crashes at runtime.



```
#include <stdio.h>
 #include <limits.h>
 void compute(int x)
 { int result;
     if (x != 0)
       { result = 100 / x; // Risk of division by zero
       }
   int large_number = INT_MAX;
   result = large_number + x; // Risk of integer overflow
   printf("Result: %d\n", result);
}
```

Figure 4 C Program Code for Abstract Interpretation: division by zero and integer overflows

In the code (Fig. 4), x = 0 leads to **division by zero**, and x > 0 causes **integer overflow** on 32-bit systems. Manually detecting such errors at scale is impractical, highlighting the need for **static analysis** like **abstract interpretation**. **Interval analysis** abstracts numerical variables as ranges *[l,u]*, tracking value bounds instead of exact values. For compute(int x), these intervals are propagated through the program as shown in **Tab 1**.

Table 1 Interval analysis data propagation

Variable	Concrete Value	Abstract Interpretation (Interval Analysis)
x	$x \in [-\infty, \infty]$ (user input)	$x \in [-\infty, \infty]$
100/x	Undefined for x = 0	Reports possible division by zero
result	INT_MAX x = 0 may be overflow	Reports integer overflow risk

Each operation modifies the interval as follows:

- **Addition:** [a, b] + [c, d] = [a + c, b + d]
- **Multiplication:** [a, b] * [c, d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]
- **Division:** [a, b] / [c, d] is **undefined if** 0 ∈ [c, d], triggering a **division by zero alarm**.

Applying these rules, Abstract Interpretation detects that:

- 100 / x is **unsafe** when x ∈ [-∞, ∞] (potential division by zero).
- INT_MAX + x **exceeds integer limits**, triggering an **overflow warning**.

This example demonstrated how **abstract interpretation using interval analysis** effectively detects **arithmetic errors** in C programs. By approximating variable ranges, it enables **early detection of division by zero and integer overflows**, improving software reliability in **safety-critical applications**.

3.2 Program Slicing

Program slicing identifies code statements relevant to a particular computation, defined by a slicing criterion ⟨p, v⟩, where *p* is a program point and *v* is a set of variables [37]. A slice includes all statements that potentially affect the value of *v* at *p*, aiding in error localization and debugging [33]. Using Weiser's algorithm, slices are created by removing irrelevant statements while preserving program executability. A valid slice must: remain executable, preserve the computation of *v* under the same input as the original program *p*.

Slicing is typically computed using the Program Dependence Graph (PDG), which combines: Control Flow Graph (CFG), Control Dependence Graph (CDG), and Data Dependence Graph (DDG).

These are derived from the program's Abstract Syntax Tree (AST) and used to analyze control and data dependencies. In PDG-based slicing, reachable nodes from the slicing criterion are included in the slice via graph traversal.

PDGs provide a unified representation of control and data dependencies, enabling not just slicing, but also code optimization and software testing. For example, in detecting SQL injection vulnerabilities, traditional static analysis may over-approximate taint propagation, causing false positives. Program slicing improves precision by isolating only the paths that influence both user input and database queries, as illustrated in the following C code example.

```
#include <stdio.h>
 #include <string.h>
 void get_user_input(char *input)
 {
  scanf("%s", input); // (S1) }
 void process_query(char *input)
 {
   char query[256];
  sprintf(query, "SELECT * FROM users WHERE name='%s'", input);
  // (S2) execute_query(query); // (S3) }
void unused_data()
{
 int a=10;
 int b=a*5;
  printf("print data %d",b);}
int main()
{
 char user_input[100];
 get_user_input(user_input); // (S4)
 process_query(user_input); // (S5)
 unused_data();//(S6)
 return 0; }
```

Figure 5 C program for SQL injection

Program slicing identify (Fig. 5) whether user_input at *S4* can reach the database query at *S2*, causing a potential SQL injection and eliminate unnecessary code to improve analysis efficiency. The PDG and analysis of the data and control dependencies in the program is shown as below:

In the Fig. 6(a) data dependencies are shown input at (S1) → query at (S2) (through sprintf); query at (S2) → execute_query at (S3) and in Fig. 6(b) c**ontrol dependencies are shown.** main() calls get_user_input() (*S4*) and process_query() (*S5*). After performing **backward slicing** from the database query at **(*S2*) following constraints are considered**:

- Slicing Criterion: (*V = input, L = S2*)
- Backward dependencies:
- query at (S2) depends on input at (*S5*).
- input at (S5) depends on user_input at (*S4*).
- user_input at (S4) comes from get_user_input() at (*S1*).The unrelated code is removed (e.g., other functions, unused variables) shown in *fig. 7.*

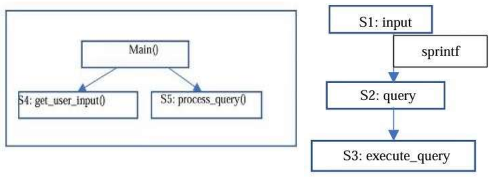
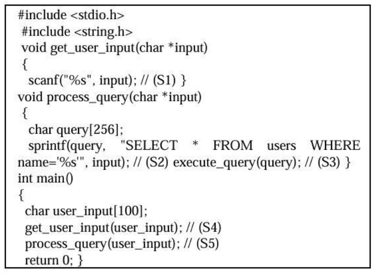


Figure 6(a) Control Dependencies; **(b)** Program Dependence Graph for Program Slicing



```
#include <stdio.h>
 #include <string.h>
 void get_user_input(char *input)
 {
  scanf("%s", input); // (S1) }
void process_query(char *input)
 {
  char query[256];
  sprintf(query,  "SELECT  *  FROM  users  WHERE
name='%s'", input); // (S2) execute_query(query); // (S3) }
int main()
{
 char user_input[100];
 get_user_input(user_input); // (S4)
 process_query(user_input); // (S5)
 return 0; }
```

Figure 7 Program after removing unrelated code

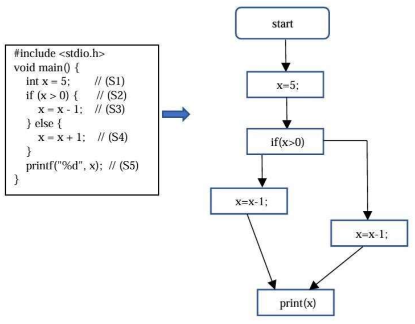
3.3. Control Flow Graph (CFG) and Data Flow Graph (DFA)

A Control Flow Graph (CFG) is a directed graph where nodes represent program statements or basic blocks, and edges indicate control flow. Each CFG has a single entry and exit node, modeling all possible execution paths from start to end [17]. CFGs support optimizations such as dead code elimination, loop unrolling, and branch prediction, and are valuable in security analysis (e.g., identifying paths to buffer overflows).

A Program Dependence Graph (PDG) extends CFGs by incorporating data and control dependencies between statements. While effective for intra-procedural slicing, PDGs are limited to single procedures [18]. To overcome this, researchers introduced the System Dependence Graph (SDG), which extends PDGs to support inter-procedural slicing by modeling dependencies across procedure boundaries. SDGs are widely used in program comprehension, optimization, and software testing [19].

Data Flow Analysis (DFA) tracks how data moves through a program and is fundamental to many compiler optimizations. Common DFA tasks include live variable analysis, constant propagation, common subexpression elimination, and dead code detection. DFA is also employed by editors and debuggers to identify static semantic errors. During inter-procedural optimizations, data-flow information must be updated to reflect changes across separately compiled modules.

As illustrated in Fig. 8, control flow transitions from S1 to S2, which branches to S3 or S4 based on a condition. Both paths converge at S5, which prints the final value. Such flow graphs aid in precise analysis of execution behavior.



```
#include <stdio.h>
void main() {
   int x = 5;       // (S1)
   if (x > 0) {     // (S2)
      x = x - 1;    // (S3)
   } else {
      x = x + 1;    // (S4)
   }
   printf("%d", x);  // (S5)
}
```

Figure 8 Control Flow Graph of the C program

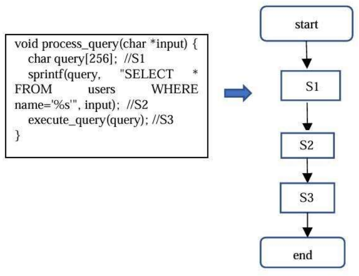
As shown in Fig. 8, S1 initializes x, followed by a conditional branch in S2 that leads to either S3 or S4. Both paths converge at S5, which prints x. Control Flow Analysis (CFA) helps identify dead code, unreachable branches, and supports compiler optimizations like loop unrolling, inlining, and branch prediction. It is also used in security analysis to detect paths leading to buffer overflows or privilege escalation.

Data Flow Analysis (DFA) tracks how data propagates through a program and is widely used in compiler optimizations, debugging, and static error detection. When inter-procedural optimizations are applied across separately compiled units, data-flow information must be updated to reflect changes.

DFA operates over a Control Flow Graph (CFG), where nodes represent statements and edges represent execution order. The analysis involves: Fact collection (via GEN/KILL sets), Equation formulation, based on direction (Forward analysis: data flows from entry to exit, Backward analysis: data flows from exit to entry). DFA answers questions like: Which variables are used or defined in each statement? Which are live at each point? Which statements affect a given variable? [14, 36].

While CFG represents execution flow, DFA models data dependencies between variables and operations. In Fig. 8, x defined in S1 flows into S2; S3 and S4 depend on S2; and S5 reads x, relying on all prior paths.

Examples of DFA applications include: Constant propagation: detects unused variables, Def-Use analysis: eliminates dead code, Taint analysis: tracks user inputs for vulnerabilities. Thus, CFG focuses on control, whereas DFA emphasizes data propagation—both crucial in static analysis and vulnerability detection, such as SQL injection.



```
void process_query(char *input) {
    char query[256];  //S1
    sprintf(query,     "SELECT     *
FROM        users        WHERE
name='%s'", input);  //S2
    execute_query(query); //S3
}
```

Figure 9 C Program with CFG for DFA

Fig. 9 shows CFG for process_query() i.e. ***S1*****:** input is passed into the function; ***S2*****:** the query is constructed using user input; ***S3*****:** the query is executed. **Tainted** input flows into query, making it vulnerable. **Executing** query with user-controlled data can lead to an **SQL injection attack**. **CFG** helps in **control structure analysis, optimizations, and vulnerability detection**. **DFG** tracks **data dependencies**, assisting in **constant propagation, taint analysis, and optimization**. **Together**, they form the backbone of modern **static program analysis techniques and tools**.

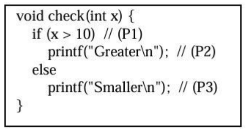
3.4 Symbolic Execution

Symbolic execution is a powerful program analysis technique that treats inputs as symbols rather than concrete values, enabling exhaustive exploration of execution paths. It is widely used in security analysis, test case generation, and formal verification.

Tools like KLEE, SAGE, and Angr apply symbolic execution to detect issues such as buffer overflows, integer overflows, null pointer dereferences, and other vulnerabilities in real-world software.

In this approach: Variables receive symbolic values, Execution generates path constraints per path, A constraint solver (e.g., Z3, STP) checks path feasibility.

Unlike concrete execution (which runs with specific inputs), symbolic execution explores multiple paths simultaneously. The symbolic execution of the code in Fig. 10 demonstrates this process.



```
void check(int x) {
   if (x > 10)  // (P1)
      printf("Greater\n");  // (P2)
   else
      printf("Smaller\n");  // (P3)
}
```

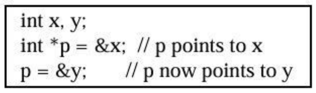
Figure 10 Program for Symbolic Execution

- **Symbolic Variable:** x is treated as a symbolic value, say α.
- **Path Constraints:**
- **Path 1**: If x > 10 → Constraint: α > 10 (Executes P2).
- **Path 2**: If x ≤ 10 → Constraint: α ≤ 10 (Executes P3).
- **Constraint Solving:** The solver finds concrete values satisfying these conditions (x = 11 for P2, x = 5 for P3). Generates test cases {x = 11, x = 5} ensuring both branches are covered, improving test coverage. Following steps are involved in symbolic execution. **Symbolic Variable Assignment:** Inputs (x, y, etc.) are assigned **symbolic variables** instead of fixed values.
- **Path Exploration:** The symbolic executor tracks all **feasible execution paths**.
- **Path Constraint Collection:** Conditions in branches (if, while, switch) form a **constraint system**.
- **Constraint Solving:** A **SMT solver (e.g., Z3, STP, CVC4)** checks if constraints are **satisfiable**.
- **Concrete Test Generation:** If constraints are solvable, concrete inputs are generated to **trigger each path**.

Symbolic execution helps detect security flaws such as: **Buffer Overflows**: Checking out-of-bounds memory access, **Integer Overflows**: Detecting arithmetic overflows in expressions, **Taint Analysis**: Tracing user inputs through the program to detect **SQL injection or command injection attacks**.

3.5 Pointer Analysis

Pointer analysis is a **fundamental static program analysis technique** used in **compiler optimization, security analysis, and formal verification**. It aims to determine the possible memory locations (variables, heap objects, or arrays) that a pointer can reference at runtime **without actually executing the program**. Pointer-related issues such as **memory leaks, null pointer dereferences, aliasing, and buffer overflows** can cause serious software vulnerabilities, especially in low-level languages like **C and C++ [32, 36]**. Pointer analysis answers the question: **"Given a pointer variable, what memory locations can it point to?"**



```
int x, y;
int *p = &x;  // p points to x
p = &y;       // p now points to y
```

Figure 11 Program for Pointer Analysis

In the code (Fig. 11), pointer analysis reveals that p may point to both x and y during execution. Pointer analysis varies by precision and complexity:

- Flow-sensitive: Considers control flow.
- Flow-insensitive: Ignores statement order.
- Context-sensitive: Differentiates across function calls.
- Context-insensitive: Merges all function calls.
- Field-sensitive: Tracks individual struct fields.
- Field-insensitive: Treats entire structs as single units.

Alias analysis determines if multiple pointers refer to the same memory: May-alias: Pointers could point to the same location and Must-alias: Pointers always point to the same location.

In this example, p initially points to x, then to y, causing aliasing. A flow-sensitive analysis tracks this accurately, while a flow-insensitive one assumes p may alias either at any point. Tools like LLVM's Clang Analyzer detect such aliasing risks and pointer misuse in real-world software.

3.6. Formal Verification

Formal verification is a mathematically rigorous approach in static program analysis used to prove program **correctness**, **safety**, and **security**. Unlike testing, which checks specific inputs, formal verification ensures that **all possible executions** of a program meet its specifications. It is critical in **safety-critical systems** such as aviation, medical devices, cryptography, and embedded systems.

Formal verification is based on techniques like:

- **Mathematical logic** and **automata theory**
- **Abstract interpretation** and **model checking**
- **Theorem proving** and **constraint solving**

Formal verification is a **mathematically rigorous method** used in **static program analysis** to prove **correctness, safety, and security properties** of programs. Unlike traditional testing, which checks specific inputs, formal verification ensures that **all possible executions** of a program **satisfy given specifications**. It is widely used in **safety-critical systems** such as **aviation, medical devices, cryptographic protocols, and embedded systems** to guarantee reliability. Formal verification involves **mathematical proofs** to verify whether a program **meets its specifications**. It is based on: **Mathematical logic and automata theory, Abstract**

interpretation and model checking, Theorem proving and constraint solving. For example, sorting function always produces a correctly ordered array. **Formal verification uses Model Checking which is exhaustively explores the state space** of a program to verify if it satisfies a given **temporal logic property** (e.g., safety or liveness) which uses following steps:

Model Checking

- Model checking explores the program's entire **state space** to verify properties expressed in **temporal logic** (e.g., LTL, CTL).
- **Model representation**: Convert the program into a finite-state model (e.g., Kripke structure).
- **Property specification**: Define correctness using temporal logic.
- **State exploration**: Check if all reachable states satisfy the properties.
- **Counterexample generation**: If verification fails, produce an execution trace showing the error.

Theorem Proving (Deductive Verification)

Theorem proving uses **mathematical logic and inference rules** to prove program correctness. Unlike model checking, it does **not require exhaustive state exploration** but instead constructs **proofs using symbolic reasoning**. Mathematical logic means view the program P as a relation $[P] \subseteq stores \times stores$, so that $(s, t) \in [P]$ iff it is possible to start P in the state s and terminate in state t. Theorem proving is a **deductive reasoning approach** where properties of a program are proven using **formal logic and axioms**. The program's behaviour is defined using **Hoare Logic, Separation Logic, or First-Order Logic**.

Hoare Logic is a **formal system** used in **theorem proving** to verify the **correctness of programs.** It provides a mathematical way to reason about a program's behaviour using **preconditions and postconditions.** A triple is written as:$\{P\}S\ \{R\}$, where P is precondition which is TRUE before executing S (code), S is the code to be analyzed and R is the postcondition must be TRUE after executing S. If P holds before execution and S executes correctly then then R must holds. It helps verify **loop correctness, variable assignments, and control flow** using **preconditions and postconditions.** Hoare logic uses following weakest precondition (wp() function) rules for programs partial and total correctness verification.

- $P \Rightarrow wp(S, R)$
- $P \wedge B \Rightarrow wp(S, I)$
- $P \wedge \neg B \Rightarrow R$
- $P \wedge B \Rightarrow t > 0$
- $P \wedge B \Rightarrow wp(\text{t1:=1;S}, t < t1)$

where P is invariant predicate derived for the code, B is the guard command of the loop in the code and t is the bound function condition to verify termination of the loop.

4. STATIC PROGRAM ANALYSIS TOOLS

Different programming languages have unique vulnerabilities, requiring specialized static analysis tools. This section outlines several widely used tools:

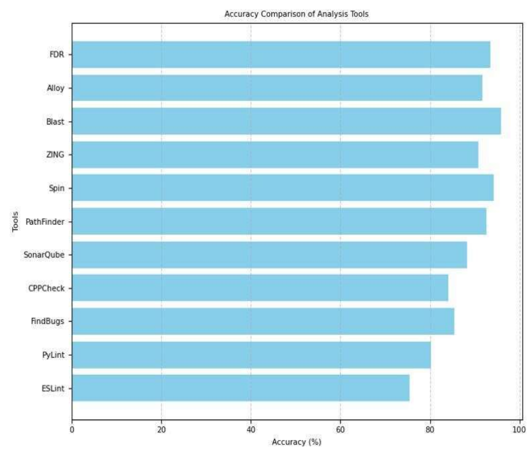
- Alloy Analyzer: A model-based tool using first-order logic to verify system properties. If unsatisfiable, it generates counterexamples.

- ESLint: A static analysis tool for JavaScript/TypeScript. It enforces coding standards, detects bugs, and integrates with IDEs for real-time feedback. Supports auto-fixes for style issues.
- Pylint: Analyzes Python source code to enforce coding standards and detect errors, bugs, and refactoring opportunities.
- FindBugs: Targets Java bytecode to uncover flaws missed in source code, offering deep static analysis post-compilation.
- Cppcheck: Designed for C/C++, detects bugs and security issues without executing code.
- BLAST: Verifies safety properties in C using lazy abstraction and formal verification techniques.
- SonarQube: A comprehensive platform for code quality monitoring across multiple languages, integrating into CI pipelines.

Tool-language associations are summarized in Table 2, which compares ESLint, PyLint, FindBugs, Cppcheck, SonarQube, PathFinder, SPIN, Zing, BLAST, Alloy, and FDR on features like language support and verification capabilities. Additional tools include CodeSonar for interprocedural analysis in C/C++, and PolySpace, which uses abstract interpretation to verify arithmetic correctness and variable relationships.

Numerous studies have compared static analysis tools across languages (e.g., C/C++, Java), evaluating parameters such as detection accuracy and execution time. Researchers often test tools on custom applications seeded with known vulnerabilities to assess their effectiveness in detecting real-world security flaws [10, 17, 24, 37].

The JULIET Test Suite (version 1.3) is used as a dataset to evaluate vulnerability detection tools and conduct a comparative analysis of their effectiveness. To compare the performance of these tools using the JULIET Test, we evaluated them based on key performance metrics used in vulnerability detection and program analysis. The evaluation of these tools is based on the following performance metrics:



- True Positives (TP): Correctly identified vulnerabilities,
- False Positives (FP): Incorrectly flagged issues
- False Negatives (FN): Missed vulnerabilities
- Precision = TP / (TP + FP), Recall = TP / (TP + FN), False Positive Rate = FP / (FP + TN).

The tools analysis can be divided into **static analysis tools** and **model checking tools**.

Static Analysis Tools : These tools detect vulnerabilities in source code **without executing it**.

- **ESLint** (JavaScript)
- **PyLint** (Python)
- **FindBugs** (Java)
- **CPPCheck** (C/C++)
- **SonarQube** (Multi-language)

Model Checking and Formal Verification Tools: These tools **simulate program execution** or perform **symbolic execution** to detect vulnerabilities.

- **PathFinder** (Java)
- **Spin** (Concurrent System Verification)
- **ZING** (Model Checking for C#)
- **Blast** (C Code Verification)
- **Alloy** (Formal Specification)
- **FDR** (Refinement Checking for CSP)

The JULIET Test Suite contains C/C++, Java, and other language-based vulnerability test cases, categorized into: Buffer Overflows, SQL Injections, Cross-Site Scripting (XSS), Null Pointer Dereferences, Memory Leaks, Race Conditions, Authentication Bypass, Other Security Flaws. The tools were tested against 5,000 selected vulnerabilities from the JULIET dataset. The comparative table 3 is as shown below:

Table 3 Comparative Analysis of tools based on performance measures

Tool	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	FPR (%)	FNR (%)	Execution Time (s)
ESLint	75.5	82.3	74.6	73.3	14.5	25.3	3.2
PyLint	80.2	85.1	77.8	81.3	12.3	22.2	3. 5
Find-bugs	85.4	88.2	81.7	84.8	10.6	18.3	4.8
CPP-check	84.1	86.5	79.3	82.7	11.5	20.7	4.1
SonarQube	88.3	90.7	85.6	88.1	9.1	14.4	5.2
PathFinder	92.5	95.1	90.3	96.6	6.8	9.7	7.3
Spin	94.2	96.5	92.8	94.6	5.5	7.2	8.1
ZING	90.7	92.8	88.4	90.5	7.3	11.6	6.7

Blast	95.8	97.1	94.5	95.8	4.1	5.5	8.9
Alloy	91.6	94.2	89.1	91.6	6.1	10.9	7.5
FDR	93.5	95.9	91.2	93.5	5.2	8.8	7.9

Figure 12 Comparison of Accuracy across the Tools

Tool accuracy depends on how well false positives and false negatives are balanced. As shown in **Fig. 12, BLAST** and **SPIN** offer the highest accuracy, effectively detecting vulnerabilities with minimal misclassifications. In contrast, **ESLint** and **PyLint** show lower accuracy, either missing issues or generating excessive alerts. When **execution speed** is a priority, tools like **SonarQube** or **PathFinder** offer a practical trade-off between accuracy and performance.

Figure 13(a) illustrates the precision–recall trade-off for SPA tools.

Tools with high precision, low recall (e.g., conservative) detect few vulnerabilities but with high confidence. Those with high recall, low precision catch more issues but generate more false positives. Tools in the top-right quadrant (Blast, Spin, PathFinder) strike the best balance—detecting most vulnerabilities with few false alarms. ESLint and PyLint, in the bottom-left quadrant, show low precision and recall, missing vulnerabilities and producing excessive alerts. SonarQube and FindBugs offer a balanced trade-off.

Figure 13(a) Comparison of Precision Vs Recall of the tools

As shown in Figure 13(b), Blast, Spin, and FDR demonstrate the highest reliability, with minimal false positives and false negatives. In contrast, ESLint and PyLint show higher misclassification rates.

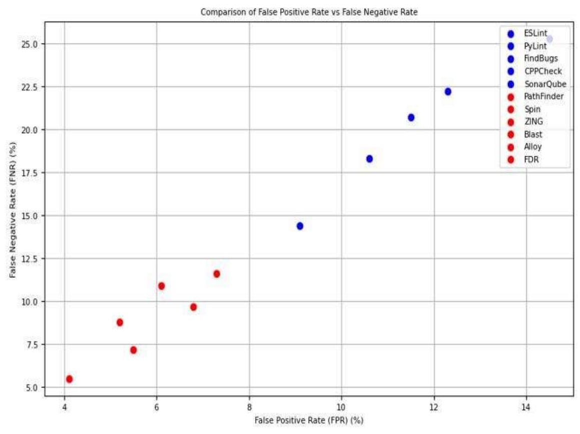
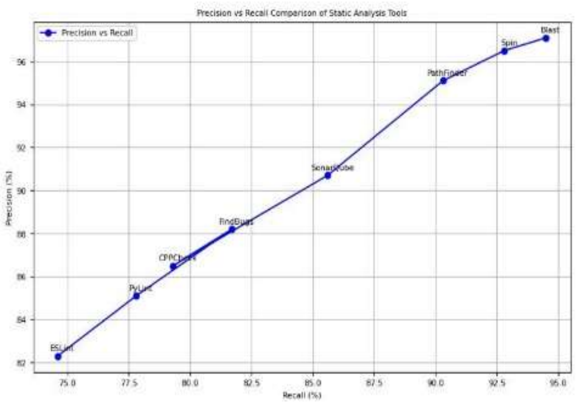


Figure 13 (b) Comparison of False Positive Rate vs False Negative Rate of the tools



Static Program Analysis (SPA) tools like ESLint, PyLint, FindBugs, Cppcheck, SonarQube, PathFinder, Spin, Zing, Blast, Alloy, and FDR face several limitations. Many suffer from high false positive rates (e.g., ESLint, PyLint, SonarQube), lack runtime analysis (FindBugs, Cppcheck, Blast), or struggle with scalability due to state explosion (PathFinder, Spin, Zing, FDR). Tools like FindBugs and Alloy lack integration with modern CI/CD workflows.

Formal verification tools (Spin, Alloy, FDR) also require manual rule specification and complex logic, making them less accessible. Additionally, several tools are ineffective with multi-threaded code or dynamically typed languages (e.g., PyLint, Cppcheck).

While SPA tools offer rule-based detection, they lack AI capabilities to adapt to evolving code patterns. Integrating machine learning can improve accuracy, reduce false positives, enhance scalability, and automate rule generation—making them more effective for large-scale, modern software systems.

5. MACHINE LEARNING INTEGRATION WITH STATIC PROGRAM ANALYSIS

In recent years, Machine Learning (ML) has significantly enhanced program analysis by identifying patterns in large codebases that traditional static analysis may overlook. ML models can detect bugs, vulnerabilities, and performance issues, analyze coding styles, and suggest optimizations. Tools like Facebook's Infer and DeepCode use real-world data to move beyond rule-based analysis, reducing false positives and improving prioritization [25].

Unsupervised learning, such as clustering, can group similar bugs, while supervised models like SVM classify issues using labeled data. Traditional analysis techniques (e.g., DFA, CFG) rely on semantics and are limited by undecidability, often leading to approximations and false positives. ML helps mitigate this by learning from historical patterns and refining analysis precision.

A method in [35] proposes automatic classification of static analyzer warnings using ML. Code metrics are extracted during analysis and used as features for classifiers. The approach improves result quality by suppressing false positives and simplifying evaluations. For industrial adoption, tools must handle multiple languages, large datasets, and maintain a balance between precision and recall.

In safety-critical systems like power grid automation [29, 31], combining tools (e.g., CppCheck, TscanCode, Flawfinder) with ML algorithms (e.g., Naïve Bayes, Random Forest, KNN) improves defect classification. This integration boosts precision by filtering false positives with minimal impact on recall.

Advances in ML/DL, driven by open-source resources, have extended to software engineering tasks such as code quality analysis, testing, refactoring, code summarization, and vulnerability detection [38].

However, challenges remain—most notably the lack of standard, well-annotated datasets, which hinders model generalization and reproducibility [30]. To ensure practical effectiveness, future research must address data imbalance, dataset transparency, and benchmarking standards.

As per the different literature, the better approach for ML-integrated static program analysis is to combine lightweight models (Random Forest, Decision Trees) for rule-based refinement with deep learning models (Transformers, GNNs) for advanced code understanding. Hybrid models (ML + Static Rules) provide optimal results in real-world scenarios. Following framework proposed with integration of ML model and static analysis techniques together.

The first step in framework is to transform source code into structured representations as shown in *fig.*14. This step converts source code into **ASTs, CFGs, and DFGs** for ML processing. Using ensemble learning by applying the Boosting refinement of rule-based predictions can be done to reduce false positives. To this result by applying Stacking (meta model), result can be finalized for the prediction. This framework combines static rule-based analysis with machine learning (Boosting & Stacking) for accurate vulnerability detection in source code. By integrating AST, CFG, Graph Neural Networks, and XGBoost, it minimizes false positives and improves static analysis performance. Though we used the JULIET dataset for tools comparison, but for the analysis of the proposed framework we found the following challenges: although the volume of publicly available software engineering artifacts is continuously growing, the absence of high-quality, well-annotated datasets remains a significant challenge in the field. The lack of standardized datasets has been cited as a primary reason for low performance, poor generalizability, and overfitting in various studies.

Without clean and reliably labeled data, many models struggle to achieve **consistent and reproducible results,** emphasizing the urgent need for the development of **validated benchmark datasets** to improve research and practical applications in software engineering. Imbalanced datasets are a common challenge in software engineering applications. Handling imbalanced datasets is a major concern in software engineering. While over-sampling and under-sampling methods help during training, test datasets should reflect real-world class distributions to prevent biased evaluations. Models tested on artificially balanced datasets may appear more effective but often fail in practical deployment. Furthermore, the lack of transparency in reporting dataset size and class ratios affects research reproducibility and model generalization. Hence, in the future considering these facts analysis of the proposed framework will be carried out.

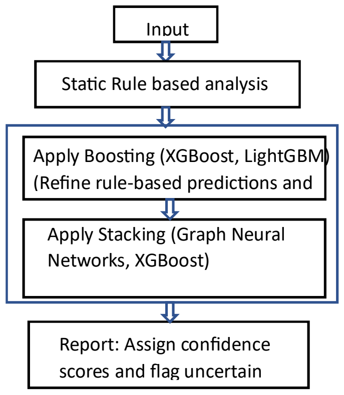

Input
Static Rule based analysis
Apply Boosting (XGBoost, LightGBM) (Refine rule-based predictions and
Apply Stacking (Graph Neural Networks, XGBoost)
Report: Assign confidence scores and flag uncertain

Figure 14 Proposed Hybrid ML Approach for Static Program Analysis

6. CONCLUSION

Static program analysis is essential for detecting vulnerabilities early, improving software security and reliability. This study explored various analysis techniques, evaluated popular static analysis tools, and highlighted their strengths and limitations. While traditional methods are effective, they often suffer from false positives and lack runtime context, limiting their accuracy.

To address these challenges, machine learning (ML) integration has been proposed, enhancing static analysis with Boosting, Stacking, and Graph Neural Networks to refine predictions and reduce false positives. However, the lack of standardized datasets and imbalanced data issues remain key obstacles, affecting reproducibility and model generalization.

Future research should focus on developing high-quality benchmark datasets and analyzing proposed frameworks, and optimizing computational efficiency. By combining static analysis with AI-driven techniques, the accuracy and adaptability of vulnerability detection can be significantly improved, making software more secure and resilient.

REFERENCES

[1] Ahmed, N. (2013, August). Verifying abstract data types: A hybrid approach. *2013 International Conference on Computing, Electrical and Electronic Engineering (ICCEEE)*, 634–639. IEEE. https://doi.org/10.1109/ICCEEE.2013.6634007

[2] Ashish, A. K., & Aghav, J. (2013, July). Automated techniques and tools for program analysis: Survey. *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 1–7. IEEE. https://doi.org/10.1109/ICCCNT.2013.6726508

[3] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static

analysis to find bugs in the real world. *Communications of the ACM, 53*(2), 66–75. https://doi.org/10.1145/1646353.1646374

[4] Ayewah, N., & Pugh, W. (2008). Using FindBugs on production software. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 345–348. ACM. https://doi.org/10.1145/1453101.1453155

[5] Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering, 35*(5), 684–702. https://doi.org/10.1109/TSE.2009.28

[6] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., et al. (2001). *The Agile Manifesto*. Agile Alliance.

[7] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't people use static analysis tools to find bugs? *IEEE Software, 30*(4), 22–28. https://doi.org/10.1109/MS.2013.72

[8] Cousot, P., & Cousot, R. (1977, January). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 238–252. https://doi.org/10.1145/512950.512973

[9] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2010). Business logic vulnerabilities in web applications. *Black Hat DC*.

[10] Duanzhi, C. (2010). A collection of program slicing. *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*. IEEE. https://doi.org/10.1109/ICCASM.2010.5620371

[11] Lokuciejewski, P., Cordes, D., Falk, H., & Marwedel, P. (2009). A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27*(7), 1165-1178. https://doi.org/10.1109/TCAD.2008.923407

[12] Clarke, E., Grumberg, O., & Peled, D. (1999). Model checking. In *Encyclopedia of Computer Science*. Wiley.

[13] Emanuelsson, P., & Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science, 217*, 5-21. https://doi.org/10.1016/j.entcs.2008.06.045

[14] Fosdick, L. D., & Osterweil, L. J. (1976). Data flow analysis in software reliability. *ACM Computing Surveys (CSUR), 8*(3), 305-330. https://doi.org/10.1145/356725.356727

[15] Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations*. IT Revolution Press.

[16] Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM, 53*(2), 66–75. https://doi.org/10.1145/1646353.1646374

[17] Hoffner, T. (1995). *Evaluation and comparison of program slicing tools*. Linköping University, Department of Computer and Information Science.

[18] Horwitz, S., Reps, T., & Binkley, D. (1988, June). Interprocedural slicing using dependence graphs. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 35-46. https://doi.org/10.1145/53990.53994

[19] Hong, H. S., Lee, I., & Sokolsky, O. (2005, September). Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, 25-34. IEEE. https://doi.org/10.1109/SCAM.2005.10

[20] Ilyas, B., & Elkhalifa, I. (2016). Static code analysis: A systematic literature review and an industrial survey. *Journal of Software Engineering*, *30*(4), 22-38.

[21] West, J., & Chess, B. (2007). *Secure programming with static code analysis*. Pearson Education.

[22] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385–394. https://doi.org/10.1145/365228.365236

[23] Limin, J., Hongqiang, J., & Jie, L. (2010, August). A dynamic program slice algorithm based on simplified dependence. *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 4, V4-356. IEEE. https://doi.org/10.1109/ICACTE.2010.5579340

[24] Lokuciejewski, P., Cordes, D., Falk, H., & Marwedel, P. (2009, March). A fast and precise static loop analysis based on abstract interpretation, program slicing, and polytope models. *2009 International Symposium on Code Generation and Optimization*, 136-146. IEEE. https://doi.org/10.1109/CGO.2009.41

[25] Li, X., Li, Y., Liu, F., & Zeng, F. (2024). Research on the integration method of software static testing tools based on machine learning. *2024 IEEE QRS-C Conference*, 920-925. https://doi.org/10.1109/QRS-C63300.2024.00122

[26] Musumbu, K. (2008, August). Static checking by means of abstract interpretation. *2008 International Conference on Computer Science and Information Technology*, 107-112. IEEE. https://doi.org/10.1109/ICCSIT.2008.115

[27] Ohtsuki, Y. (2017). Limitations of static analysis tools. *Journal of Computer Science and Technology, 32*(4), 540-551. https://doi.org/10.1007/s11390-017-1756-x

[28] Ouimet, M., & Lundqvist, K. (2007). Formal software verification: Model checking and theorem proving. *Embedded Systems Laboratory Technical Report ESL-TIK-00214*, Cambridge, USA.

[29] Payet, É., & Spoto, F. (2012). Static analysis of Android programs. *Information and Software Technology, 54*(11), 1192-1201. https://doi.org/10.1016/j.infsof.2012.05.002

[30] Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., & Sarro, F. (2021). A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*. https://doi.org/10.48550/arXiv.2110.09610

[31] Sgandurra, D., Muñoz-González, L., Mohsen, R., & Lupu, E. (2016). Automated dynamic analysis of ransomware: Benefits, limitations, and use for detection. *arXiv preprint arXiv:1609.03020*. https://doi.org/10.48550/arXiv.1609.03020

[32] Singer, J. (2006). *Static program analysis based on virtual register renaming* (No. UCAM-CL-TR-660). University of Cambridge, Computer Laboratory.

[33] Surendran, A., & Samuel, P. (2012, October). Partial slices in program testing. *2012 35th Annual IEEE Software Engineering Workshop*, 82–89. IEEE. https://doi.org/10.1109/SEW.2012.20

[34] Thomson, P. (2021). Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue, 19*(4), 29–41. https://doi.org/10.1145/3470540.3470545

[35] Tsiazhkorob, U. V., & Ignatyev, V. N. (2024). Classification of static analyzer warnings using machine learning methods. *2024 Ivannikov Memorial Workshop (IVMEM)*, 69–74. IEEE. https://doi.org/10.1109/IVMEM63006.2024.10659704

[36] Vollmer, J. (1995, June). Data flow analysis of parallel programs. *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, 168–177.

[37] Zhang, Y. Z. (2021). SymPas: Symbolic program slicing. *Journal of Computer Science and Technology, 36*, 397–418. https://doi.org/10.1007/s11390-021-0952-8

[38] Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., & Sarro, F. (2021). A survey on machine learning techniques for source code analysis. arXiv preprint arXiv:2110.09610.

[39] Fan, G., Xie, X., Zheng, X., Liang, Y., & Di, P. (2023). Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. arXiv preprint arXiv:2310.08837.