*electronics*

MDPI

*Article*

# Static Analysis Techniques for Embedded, Cyber-Physical, and Electronic Software Systems: A Comprehensive Survey

**Maksim Iavich [1,*], Tamari Kuchukhidze [1] and Audrius Lopata [2]**

[1] Department of Computer Science, Caucasus University, Tbilisi 0102, Georgia; tkuchukhidze@cu.edu.ge
[2] Faculty of Informatics, Kaunas University of Technology, 44249 Kaunas, Lithuania; audrius.lopata@ktu.lt
* Correspondence: miavich@cu.edu.ge

**Abstract**

Static analysis is a critical methodology for ensuring the quality, security, and safety of embedded, cyber-physical, and electronic software systems, particularly as such systems become increasingly complex and tightly coupled with hardware and real-time constraints. Through a systematic study of the literature, this paper summarizes the State-of-the-Art in static program analysis. We develop a comprehensive taxonomy of fundamental techniques, including model checking, abstract interpretation, data-flow analysis, and symbolic execution, and examine their application in modern analysis tools used in electronic and safety-critical systems. The survey thoroughly reviews applications across key domains, including vulnerability detection, automotive and embedded software verification, smart contract auditing, and AI-enabled electronic systems. We also critically analyze persistent challenges, including tool integration, scalability limitations, and the trade-off between analysis precision and soundness. Finally, by discussing emerging trends and future research directions—such as machine-learning-enhanced analysis and hybrid static–dynamic techniques—this work provides a structured framework to guide future research and industrial practice in the development of reliable electronic systems.

**Keywords:** static analysis; embedded software; electronic systems; cyber-physical systems; abstract interpretation; data-flow analysis; vulnerability detection

## 1. Introduction

Today, software is a fundamental component of modern electronic and cyber-physical infrastructure, controlling essential processes in energy distribution, healthcare, transportation systems, and financial platforms. As electronic systems become increasingly software-driven and interconnected, the reliability, security, and safety of the software that controls them have moved beyond purely technical concerns to become critical pillars of public welfare and economic stability [1]. The severe consequences of software failures are illustrated by several high-profile incidents. The 2017 Equifax breach, which exposed the personal data of 147 million individuals, resulted from a known but unpatched vulnerability in the Apache Struts framework that could have been detected using appropriate static code analysis techniques [2]. Similarly, the November 2020 outage of the Australian Securities Exchange, which halted trading for nearly four hours, was caused by technical failures in the ASX Trade platform, demonstrating how software defects can disrupt large-scale electronic systems [3]. These events highlight the urgent need for reliable methods that proactively identify and mitigate software defects before deployment.

Check for updates

Academic Editor: Alexander Barkalov

Received: 8 January 2026
Revised: 15 February 2026
Accepted: 20 February 2026
Published: 24 February 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

Static analysis has emerged as a foundational approach to addressing this challenge. Formally, it involves the examination of source code, binaries, or intermediate representations without executing the program, enabling early verification and validation of software behavior [4]. Unlike dynamic testing, which evaluates software behavior under specific test cases and execution paths, static analysis enables systematic exploration of a broader set of possible behaviors. By transforming programs into intermediate representations such as Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs), static analysis tools construct structured models of potential execution semantics [5,6]. These models can then be algorithmically analyzed to detect a wide range of issues, including security vulnerabilities, logic errors, resource mismanagement, and violations of coding standards. This capability is particularly valuable in embedded and electronic systems, where early defect detection significantly reduces development costs and mitigates safety risks [7].

The strategic relevance of static analysis is further amplified by its integration into modern software engineering workflows for electronic systems. It is a core component of the DevSecOps paradigm, which embeds security and quality assurance mechanisms directly into the development lifecycle. This approach enables a "shift-left" strategy, in which verification activities occur continuously rather than being deferred to final validation stages [8]. Within Continuous Integration and Continuous Deployment (CI/CD) pipelines, static analysis tools operate as automated quality gates, evaluating every code change to prevent defective or insecure software from reaching production environments. Empirical studies indicate that organizations employing Static Application Security Testing (SAST) techniques can remediate vulnerabilities significantly faster than those relying on post-development reviews alone [9].

In parallel, the growing adoption of artificial intelligence (AI) in electronic and edge-computing platforms has expanded the scope of static analysis. As machine-learning components are increasingly integrated into safety- and security-critical systems, researchers have adapted static analysis techniques to address new classes of risks. These include verifying the correctness of data preprocessing pipelines, identifying unintended data leakage, and detecting potential sources of bias that may compromise system behavior or trustworthiness [10]. Such developments position static analysis as an essential tool for ensuring dependable AI-enabled electronic systems.

The importance of static analysis can be understood through its contribution to three fundamental quality imperatives: reliability, security, and safety. From a reliability perspective, techniques such as data-flow analysis and model checking systematically identify defects that can cause runtime failures, including null pointer dereferences, race conditions, and resource leaks. In terms of security, static analysis underpins SAST methodologies, which use advanced techniques such as taint analysis and symbolic execution to trace the propagation of untrusted inputs through software logic and detect critical vulnerabilities, including those cataloged in the OWASP Top Ten [11]. Finally, in safety-critical domains such as automotive electronics and avionics—regulated by standards including ISO 26262 [12] and DO-178C [13]—static analysis is frequently mandated to enforce strict coding rules and to formally verify the absence of hazardous runtime behaviors using sound methods such as abstract interpretation [14].

The main contributions of this survey are as follows:

1. A thorough systematic examination of static analysis methods for electronic, embedded, and cyber-physical systems that follows PRISMA guidelines and transparently reports search strings, screening data, and quality evaluation (Section 2).
2. A new, analytically based taxonomy that includes evidence-based comparison tables and links key technique families (data-flow analysis, symbolic execution, abstract

interpretation, model checking, and constraint solving) to their technical trade-offs (soundness, precision, and scalability) and representative tools.

3. Static analysis applications from a variety of domains, including blockchain smart contracts, safety-critical automotive and aerospace systems, security vulnerability identification, and AI/ML pipeline verification, with a focus on embedded and cyber-physical settings (Section 4).
4. A critical examination of enduring problems and new lines of inquiry, such as automated program repair, hybrid static–dynamic approaches, machine-learning-enhanced analysis, and verification issues in cloud-native and autonomous systems, with a clear connection to the technical trade-offs noted in the taxonomy (Sections 5 and 6).

From the development of research topics and application domain selection to the examination of scalability and certification issues, this study gives special attention to embedded, cyber-physical, and electronic systems.

This study focuses on embedded, cyber-physical, and electronic systems, from the formulation of research questions and application domain selection to the analysis of scalability, real-time restrictions, and safety certification issues. Despite the broad applicability of static analysis techniques, this study methodically tackles the unique needs that emerge when they are used in embedded and cyber-physical environments.

**Motivation and Research Questions**.

Despite the proven advantages of static analysis and its growing maturity, there remain substantial, persistent obstacles to its general acceptance and optimal efficacy. The most fundamental trade-off is between soundness (the guarantee of detecting all true flaws of a specific class) and accuracy (the reduction in false positives), a theoretical limitation that affects all practical tools [15]. Scalability remains a crucial issue since the computational cost of deep, inter-procedural analysis can grow exponentially with codebase size and complexity [16]. The substantial operational difficulties involved in the initial setup, tuning, and integration of these tools into different development frameworks could discourage teams. In order to map the entire area of static analysis, from its fundamental algorithmic techniques and useful applications to its enduring flaws and prospective future direction, this article offers a systematic assessment and synthesis of recent research.

The following research questions (RQs) serve as the framework for this investigation:

- **RQ1:** What is a comprehensive taxonomy of the most popular static analysis approaches, and how do their underlying formal methods (such as data-flow analysis, symbolic execution, and abstract interpretation) compare in terms of precision, computational complexity, and soundness guarantees?
- **RQ2:** How is static analysis used, specialized, and verified in a number of important sectors, such as security vulnerability detection, automotive embedded systems, blockchain smart contract verification, and AI/ML pipeline testing?
- **RQ3:** What are the biggest shortcomings of the static analysis methods currently in use, and how are new advancements, like the use of machine learning, the development of hybrid static–dynamic methods, and automated program repair, resolving these problems and creating new research opportunities?

The rest of this paper is arranged in the following methodical manner. Section 2 gives important background information on static analysis concepts and describes our rigorous research methodology. A thorough taxonomy of analysis methods and their implementation in contemporary tools is provided in Section 3. While Section 5 offers an open discussion of the basic difficulties and constraints, Section 4 examines their applications across several important domains. Innovative future research directions are examined in Section 6, and

our contributions are summarized, with the most urgent research gaps highlighted in Section 7, which wraps up the paper.

## 2. Background and Methodology

Static analysis examines software code without actually running it [17] by carefully analyzing program behavior, enabling early identification of flaws. Analyzers often transform source code into intermediate representations such as Abstract Syntax Trees (ASTs), which capture logical structure, and Control Flow Graphs (CFGs), which depict branching and execution routes [18–20]. Program Dependence Graphs (PDGs), which facilitate techniques such as program slicing, can be employed in more intricate studies to manage dependencies and collect information [21,22].

Static analysis uses a range of analytical techniques with varying power and processing costs. To identify problems such as resource leaks, null pointer dereferences, and un-initialized variables, data-flow analysis solves equations over the CFG and gathers information about possible variable values throughout a program [23]. Symbolic execution, which evaluates programs using symbolic rather than actual inputs while maintaining path conditions that indicate the input space required to reach each program point, enables the discovery of complex, path-sensitive faults [24]. Abstract interpretation provides a strong mathematical foundation for sound exaggeration of program behavior, ensuring coverage of all conceivable executions at the cost of potential false positives [25]. Model checking is crucial for identifying concurrency problems like deadlocks in safety-critical systems because it rigorously explores the state space to verify finite-state systems against temporal-logic requirements [26].

### *2.1. Research Methodology*

To provide a comprehensive, impartial, and reproducible synthesis of the literature, this review followed the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines [27]. Figure 1 shows the PRISMA flow diagram that describes the study selection procedure. Three distinct and rigorous phases were used to operationalize this established methodology: the first involved the precise formulation of research questions based on identified gaps in the current body of knowledge; the second, the systematic identification and retrieval of relevant literature; and the third, the consistent application of predetermined inclusion criteria. The entire process of choosing the material was led by the particular research questions listed in Section 1.

To reduce bias and increase coverage, the search was conducted in two complementary phases. Keywords like “static code analysis” and “program verification” were used in an initial scoping search to define the survey’s limits and major issues, with a focus on embedded and electronic systems settings. An additional systematic search was then conducted to fully capture the scope of static analysis research and ensure thorough coverage of both fundamental approaches and specific applications.

Three main aspects were addressed by this enlarged search, which included a wide range of keyword combinations: targeted vulnerability classes or program features, particular application domains, and fundamental static analysis approaches. “Data flow analysis”, “abstract interpretation”, “symbolic execution”, “model checking”, “taint analysis”, “constraint-based analysis”, “type-state analysis”, “pointer analysis”, and “interval analysis” were among the major technical terms. “Embedded systems AND static analysis”, “cyber-physical systems AND verification”, “smart contract AND static analysis”, “ML pipeline AND static verification”, and “IoT software AND security analysis” were examples of domain-specific keyword pairs. “Memory safety AND verification”, “concurrency bug AND static detection”, “vulnerability detection AND static”, and “resource leak AND

static analysis" were among the keywords that concentrated on particular results. For each database, these keyword groupings were combined using Boolean operators to create the full Boolean search strings. For example, the search string for IEEE Xplore was: (("static analysis" OR "data flow analysis" OR "symbolic execution" OR "abstract interpretation" OR "model checking" OR "taint analysis" OR "constraint-based analysis") AND ("embedded systems" OR "cyber-physical systems" OR "smart contract" OR "ML pipeline" OR "IoT software") AND ("memory safety" OR "concurrency bug" OR "vulnerability detection" OR "resource leak")). For each database's syntax, equivalent strings were modified.

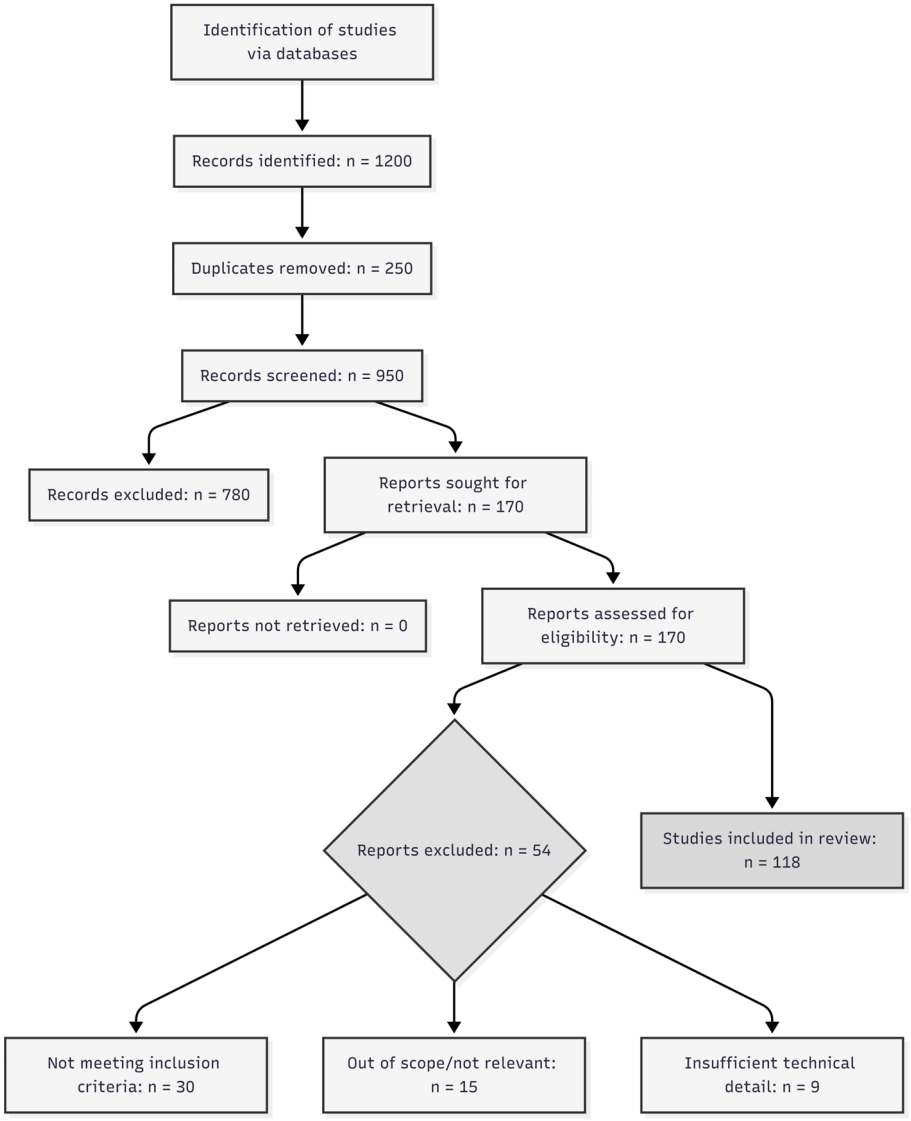


**Figure 1.** PRISMA flow diagram of the systematic literature selection process.

The search was temporally limited from 2015 to 2025 to focus on the contemporary evolution of the field; seminal pre-2015 works establishing foundational techniques (e.g., in abstract interpretation and model checking) were found through backward snowballing from recent surveys and included where relevant. These keyword strings were applied to the titles, abstracts, and keywords of publications across the major scientific databases: IEEE Xplore, ACM Digital Library, SpringerLink, Scopus, and Google Scholar. The information is reflected in Table 1.

**Table 1.** Reference Scientific Literature Databases and Academic Search Engines.

| Database/Search Engine | Type | URL |
|---|---|---|
| **Scopus** | Search Engine | https://www.scopus.com |
| Google Scholar | Search Engine | https://scholar.google.com |
| IEEE Xplore | Digital Library | https://ieeexplore.ieee.org |
| ACM Digital Library | Digital Library | https://dl.acm.org |
| SpringerLink | Digital Library | https://link.springer.com |
| arXiv | Preprint Repository | https://arxiv.org |

To ensure relevance and minimize bias, a comprehensive, multi-stage filtering approach was employed to choose papers for this evaluation based on predetermined inclusion and exclusion criteria. Works that did not meet the inclusion criteria were excluded.

The filtering process was a logical, multi-step procedure:

Step 1: Publications judged unrelated to the study topics were excluded based on information from titles and abstracts. If an article's abstract and title explicitly addressed at least one of the fundamental method types or research issues listed in Section 1, it was kept.

Step 2: The remaining articles' complete texts were assessed. Works that were deemed outside the purview of this study (for example, concentrating solely on dynamic analysis without a static component) or that just briefly discussed static analysis were eliminated.

Step 3: A quality review was conducted on the final set of papers, which had to meet the following requirements: the article presents a novel methodology or a significant improvement to an existing static analysis technique; the work provides enough technical detail to make the methodology understandable; the publication is either peer-reviewed or from a reputable preprint server; and the study directly and significantly contributes to the specified research questions. Each full-text publication was further assessed using a four-point quality rating checklist: methodological clarity (0–1), empirical evaluation (0–1), peer-review status (0–1), and direct relevance (0–1) in order to further evaluate methodological quality and reduce selection bias. The final synthesis only includes papers with a score of at least 2. The selected publications had an average quality score of 3.2, which indicates that their methodological rigor was typically strong.

A summary of the formal inclusion and exclusion criteria is provided in Table 2.

**Table 2.** Inclusion and Exclusion Criteria.

| Inclusion Criteria | Exclusion Criteria |
|---|---|
| Published in one of the selected databases (Table 1) | Not written in English |
| Published between 2015 and 2025 to ensure relevance. | Published before 2015 (with exceptions for seminal, foundational works). |
| At least one search term appears in the title, abstract, or keywords | Full text not available |
| Addresses or analyzes the stated research questions | Lacks technical detail or is a duplicate of another study |
| Directly addresses the techniques, applications, or evaluations of static analysis. | Focuses exclusively on dynamic analysis without a static analysis component. |
| Provides sufficient technical depth, empirical results, or a novel theoretical framework | Lacks technical detail, experimental validation or a clear methodology. |

Throughout the systematic literature selection process, the PRISMA standards were adhered to, as shown in Figure 1. In the initial search, 1200 records were located in the six preprint repositories and databases. 950 items remained for title and abstract screening after 250 duplicates were eliminated; 780 records were deemed superfluous. After evaluating the remaining 170 full-text publications for eligibility, 54 were rejected (30 because they did not fulfill inclusion standards, 15 because they were beyond the scope, and 9 because they did not provide enough technical depth). As a result, the final systematic review had 118 publications.

Figure 2 shows the distribution by year of publication of the 118 included publications. The number of publications remained very stable between 2015 and 2019, averaging 11 papers each year. Static analysis applications for emerging domains such as AI/ML pipelines and smart contracts have garnered increasing attention, as seen by the noteworthy peak of 14 articles in 2020. Following a little decline in 2022–2023, the number of publications rose to 9 papers in 2025, indicating continued research effort. The articles on machine-learning-based static analysis, which make up the majority of the output from 2020 to 2025, are noteworthy since they mostly appeared after 2020 and show a distinct shift in research focus toward learning-based verification methodologies.

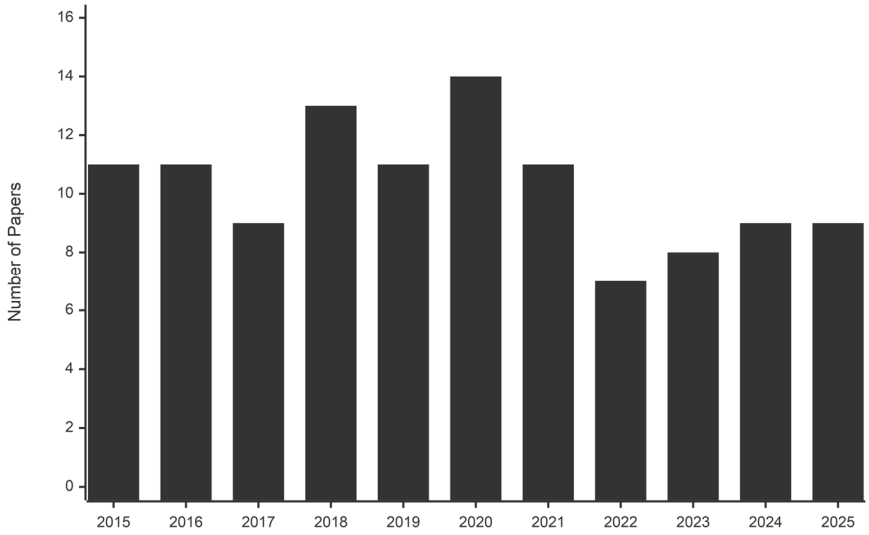


**Figure 2.** Distribution of included publications by year (2015–2025).

This rigorous, two-stage search and filtering process significantly increased the initial corpus and enabled the inclusion of significant academic publications specialized in new topics and specialist methodologies, ensuring a balanced synthesis of academic research and industry application. The sections that follow offer a comprehensive synopsis of the material selected through this systematic process. The primary focus was on advancements from 2015 to 2025, with special attention to innovative studies that established foundational concepts in static analysis. Papers published before 2015 were included if they provided basic techniques or were significant sources that continue to influence contemporary study. This approach ensured that both the current developments in static analysis research and its historical foundations were fully covered.

### *2.2. Related Work and Positioning*

Different aspects of static analysis have been the subject of several previous surveys. Beyer [15] offers empirical tool performance evaluations and presents an overview of software verification through the SV-COMP competition. An extensive textbook discussion of abstract interpretation is given by Rival and Yi [25], while Baldoni et al. [24] give a

thorough summary of symbolic execution strategies. In their discussion of model checking and verification of concurrent systems, Kant et al. [26] focus on issues related to mitigating false positives. The results of the yearly SV-COMP competition are used by Beyer [28] to provide an overview of the State-of-the-Art in software verification. All of these works have contributed significantly to the discipline.

This poll is different from previous research in three important ways. First, it uses a methodical, PRISMA-guided approach (Section 2.1) with well-documented screening statistics, inclusion criteria, and search strings, offering a degree of repeatability and transparency not usually found in narrative reviews. Second, it focuses on embedded, cyber-physical, and electronic systems in particular domains, where static analysis must address resource constraints, real-time constraints, and safety certification standards such as DO-178C and ISO 26262.

Although previous surveys touch on these areas in passing, they do not regard them as the core organizing factor. Third, this study offers an analytically supported taxonomy structured around basic engineering trade-offs (scalability vs. accuracy vs. soundness). It is backed up by evidence-based comparison tables that reference particular empirical findings from the collected literature. Instead of merely outlining methods and instruments, we methodically connect their technical attributes to their applicability in various application scenarios.

In conclusion, by providing a specific, methodologically sound, and application-focused synthesis of static analysis for embedded and cyber-physical systems, a focus that is both relevant and underrepresented in the current literature, this study enhances surveys that already exist.

## 3. Taxonomy of Static Analysis Techniques

### *3.1. Introduction to the Taxonomy*

Static analysis is a broad family of methods for determining software behavior without executing programs, and it is an essential part of modern software quality assurance. The field has evolved from simple pattern-matching linters to intricate frameworks with State-of-the-Art mathematical foundations, each with distinct trade-offs between accuracy, scalability, and soundness. A methodical taxonomy is necessary to comprehend when and how to apply these strategies successfully. This section provides a comprehensive classification of the most widely used static analysis paradigms, examining their theoretical underpinnings, practical uses, and representative tools that have gained popularity in both academic and commercial contexts [28]. By mapping analytical approaches to their real-world instantiations, we hope to provide practitioners with a structured framework for selecting appropriate analysis techniques based on their specific verification requirements and constraints.

A conceptual representation of the taxonomy created in this section is shown in Figure 3. Data-flow analysis, symbolic execution, abstract interpretation, model verification, and constraint solving are the five main families of static analysis techniques that are mapped to their main application fields. Key technical characteristics (such as soundness, accuracy, and scalability) and representative tools define each method family. In domains like security testing, safety-critical systems, and new fields like AI/ML and cloud-native systems, the links show how various technology capabilities handle particular verification demands.

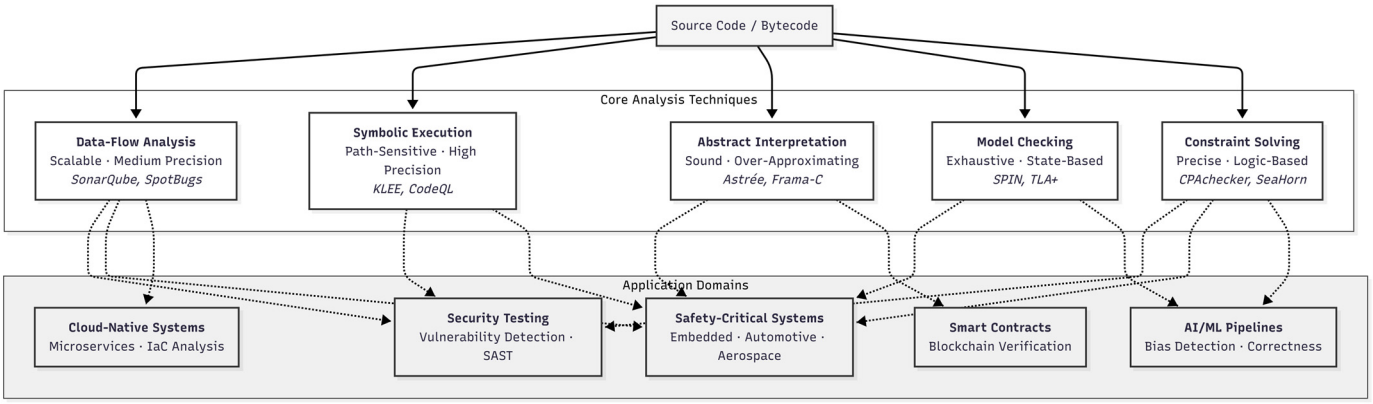


**Figure 3.** Taxonomy of static analysis techniques and their application domains.

### *3.2. Core Analysis Techniques*

#### 3.2.1. Data-Flow Analysis

Monitoring how data values propagate by using program variables along various execution paths is the fundamental principle behind data-flow analysis. This approach formulates program analysis as a series of equations that describe how information flows through a program's control flow graph by modeling possible data states using lattice theory [29]. The analysis calculates fixed-point solutions to these equations to find properties such as variable liveness, reaching definitions, and available expressions at each program point [30].

This approach is particularly good at identifying typical programming problems, such as use-before-initialization violations, assured null pointer dereferences, and resource leaks, where files or database connections may not be properly terminated. However, practical factors sometimes restrict its accuracy; many implementations adopt flow-insensitive or context-insensitive approximations to guarantee scalability in big codebases, which may lead to false positives [31]. Programs like SonarQube, which is used for continuous code quality inspection, and FindBugs (now superseded by SpotBugs), which employs intraprocedural analysis to identify bug patterns in Java bytecode, are built on this technique.

#### 3.2.2. Symbolic Execution

Symbolic execution allows for the simultaneous exploration of multiple execution paths by substituting symbolic input values for concrete ones. The method maintains path conditions that describe the input space needed to reach each program point by treating program inputs as symbolic variables and interpreting operations as constraints on these symbols rather than processing specific data [32]. Program analysis is successfully transformed into a constraint satisfaction problem using this method [33].

Buffer overflows, integer overflows, and injection vulnerabilities, all of which can go undiscovered during standard testing, are among the deep, path-sensitive flaws that symbolic execution excels at detecting. The KLEE tool, which employed extensive path exploration to identify several significant flaws in GNU Core Utilities [34], serves as an example of this tactic. CodeQL, which represents code as queryable databases and enables security researchers to design sophisticated queries that leverage symbolic execution principles for variant analysis across whole codebases, has made a related semantic code analysis technique more prominent in recent years [35].

The primary problem remains the path explosion problem, which necessitates sophisticated heuristics and state pruning strategies since the number of potential paths grows exponentially with program complexity and size.

#### 3.2.3. Abstract Interpretation

Abstract interpretation provides a rigorous mathematical foundation for producing static analyses that precisely approximate program behavior. The basic idea involves executing programs on abstract domains that offer simplified views of possible program states in order to guarantee that all concrete executions are covered while making analysis tractable [36]. For instance, rather than tracking exact integer values, an analysis may employ operations defined over this abstract domain to track just whether a variable is positive, negative, or zero. Congruence domains monitor periodic interactions, interval domains mimic variable ranges, and polyhedral domains represent linear inequalities between variables. Different program properties are captured by these abstract domains [37]. Widening operators ensure completion at the risk of potential precision loss, and fixed points are calculated over these abstract domains in the analysis.

Because of this formal foundation, abstract interpretation is particularly well-suited for safety-critical applications where proving the lack of particular mistake types is essential. An important development in this area is the Astrée analyzer, which has demonstrated the absence of runtime defects in flight control software for the Airbus A380 [38]. In a similar vein, the Clang Static Analyzer employs a checkers design that allows domain-specific analyses to leverage its basic symbolic reasoning skills to find problems in C, C++, and Objective-C code using path-sensitive abstract interpretation [39]. Even while the soundness guarantee is robust, it typically comes at the cost of false positives because the over-approximation may contain program behaviors that cannot truly occur during execution.

#### 3.2.4. Model Checking

Model checking automates the verification of finite-state systems against formal requirements expressed in temporal logic. Unlike other static analysis approaches that work directly on source code, model checking typically involves creating a formal model of system behavior and then carefully inspecting every conceivable state to confirm whether the model matches properties described in logics like LTL or CTL [40]. This comprehensive inquiry ensures that any violations of the designated attributes will be found.

The system is depicted as a Kripke structure, a directed graph with nodes representing system states and edges representing transitions, as part of the verification procedure. The model checker then carefully scans this state space to determine whether each state satisfies the temporal logic formulas. Sophisticated techniques like bounded model checking using SAT solvers and symbolic model checking using binary decision diagrams (BDDs) have been developed to address the inherent state explosion problem of this methodology [41]. Partial order reduction and symmetry reduction help manage complexity by eliminating pointless state explorations.

In distributed and concurrent systems, where minute timing issues can lead to livelocks, deadlocks, and race situations, this approach has proven to be quite beneficial. The SPIN model checker has been frequently used to confirm protocol correctness, but NASA has employed Java PathFinder (JPF) to validate critical spacecraft software [42]. To prevent minor defects in AWS services, companies such as Amazon have reported adopting TLA+ and its associated TLC model checker to design and validate distributed algorithms [43]. The primary barrier remains the state explosion problem, which restricts application to systems that can be effectively abstracted to manageable state spaces.

In order to verify satisfiability, contemporary bounded model checking tools use SAT and SMT solvers, demonstrating how constraint solving has evolved into a technology that makes scaled model checking possible.

#### 3.2.5. Constraint Solving

As mentioned in Sections 3.2.2 and 3.2.4, constraint solving is a fundamental engine for bounded model checking and symbolic execution, even if it is a separate verification paradigm. Constraint-based analysis reduces program verification problems to satisfiability queries, which are then handled by automated reasoning engines. Program semantics, pathways, and characteristics are encoded into logical formulas using Satisfiability Modulo Theories (SMT), which combine propositional logic with domain-specific theories for integers, arrays, and bit vectors [44]. The viability of particular program behaviors is then demonstrated by the satisfiability of these formulas.

As part of the analysis process, program structures are transformed into comparable logical constraints. For instance, operations become functions, control flow merges become disjunctions, and program variables become constants in the logic. Sophisticated methods like Horn clause solution, which enable the testing of complex program invariants, extend this approach to relational properties [45]. It is the responsibility of SMT solvers, such as Z3, CVC4, or Yices, to determine whether the resulting constraint systems are satisfiable and, in turn, provide models.

This formalism provides exceptional precision for checking complex, nonlinear program properties that are difficult to manage using traditional methods. The Infer tool, developed at Facebook and now widely used in mobile app development, uses separation logic with bi-abduction, a constraint-based method for identifying frame conditions, to identify memory safety issues in C, Java, and Objective-C with remarkably low false-positive rates [46]. The SeaHorn verification system, which uses limited Horn clauses to validate safety features of C programs, further demonstrates the scalability of these techniques to real-world software [47]. Similar to this, the CPAchecker framework [48] offers a highly flexible platform for software verification that supports a variety of analysis approaches, including explicit-state model checking and predicate abstraction. This framework has demonstrated great performance in international verification contests [49]. The primary challenge is the computational difficulty of solving these constraints, which might be prohibitive for complex or huge verification circumstances.

### *3.3. Tool-Based Categorization*

In order to put these theoretical approaches into practice, static analysis is practically applied through tools that often combine many ways to balance their benefits and drawbacks. Users need to have a deep awareness of the tool ecosystem in order to integrate static analysis into their development workflows.

Table 3 provides a detailed comparison of the primary static analysis tools' technological approaches, language support, analysis depth, and target user bases. This table is helpful when selecting a tool based on organizational limitations and technological specifications.

**Table 3.** Comprehensive Static Analysis Tool Comparison.

| Tool | Type | Core Technique | Soundness | Key Strength | Key Limitation | Primary Domain |
|---|---|---|---|---|---|---|
| **SonarQube** | Industrial/OSS | Data-flow, Pattern Matching | No [50] | Broad language support, CI/CD integration | Higher false positives (30–50% in Java projects [50]), limited depth | Multi-language code quality |
| **CodeQL** | Industrial/OSS | Symbolic Execution, Constraint Solving | Partial [35] | Deep semantic analysis, security-focused; used to find ~70% of critical CVEs at GitHub (CodeQL version 2.6) [35] | Steep learning curve, resource-intensive | Security vulnerability detection |

**Table 3.** *Cont.*

| Tool | Type | Core Technique | Soundness | Key Strength | Key Limitation | Primary Domain |
|---|---|---|---|---|---|---|
| **Clang Static Analyzer** | Academic/OSS | Abstract Interpretation, Symbolic Execution | Partial [39] | Path-sensitive, integrates with LLVM | C-family only, moderate false positives | C/C++ systems programming |
| **Astrée** | Academic/ Industrial | Abstract Interpretation | Yes [38] | Sound verification of runtime errors; proven absence of RTE in 132 K LoC Airbus A380 flight control software [38] | Requires expert configuration; primarily for C | Safety-critical aerospace/automotive |
| **KLEE** | Academic | Symbolic Execution | Partial [34] | Finds deep path-sensitive bugs. Achieved 90%-line coverage on GNU Coreutils [34] | Path explosion, scalability limits | C/LLVM systems testing |
| **CPAchecker** | Academic | Predicate Abstraction, Model Checking | Yes [48] | Configurable, strong competition results. (multiple SV-COMP awards [49]) | High memory usage | Software verification (C, Java) |
| **Frama-C** | Academic | Abstract Interpretation, Value Analysis | Yes [51] | Suite for critical C code analysis. Interoperable analyzers for value, dependency, and WP calculus [51] | Requires expertise to configure | Safety-critical C code |
| **SeaHorn** | Academic | Constrained Horn Clause Solving | Yes [47] | Scalable verification of systems code | Focuses on safety properties | C program verification |
| **Infer** | Academic/ Industrial | Separation Logic, Bi-Abduction | Yes [46] | Compositional analysis, low false positives (80%+ precision at Facebook scale [46]) | Focused on memory/resource bugs | Mobile/system memory safety |
| **Coverity** | Industrial | Advanced Inter-procedural Analysis | Yes [52] | Whole-program analysis, certifications | Closed source, limited extensibility | Enterprise, safety-critical |
| **Yama** | Academic | Opcode-based Data-Flow Analysis | Partial [53] | High precision for PHP vulnerabilities | PHP-specific | PHP web application security |
| **Tchecker** | Academic | Inter-procedural Taint Analysis | Partial [54] | Scalable taint tracking for PHP | PHP-specific | PHP security analysis |
| **ESLint/PMD** | OSS | Pattern Matching, AST Analysis | No [55] | Fast, IDE integration, customizable (100 K LoC in <10 s [55]) | Syntax-level only, no semantic analysis | Web development, code style |

SonarQube maintains its position as a leading platform for continuous code quality monitoring with regular updates that enhance its data-flow analysis and pattern-matching capabilities. The platform now supports more modern programming languages and frameworks and has better interoperability with popular CI/CD systems. Businesses managing large, multilingual codebases have embraced SonarQube's comprehensive approach to code quality management, which includes technical debt tracking and quality gate enforcement. Industry reports indicate that it has been widely implemented by firms seeking to maintain code quality across geographically dispersed teams [50].

CodeQL, a crucial component of GitHub's security ecosystem, provides semantic code analysis through its declarative query language. The platform continues to develop with enhanced support for new programming language features and frameworks. Findings from vulnerabilities discovered using CodeQL in significant open-source projects are regularly published by the GitHub Security Lab. The focus of recent developments has been on improving the precision and scalability of analysis. The tool's integration with GitHub's

developer workflow has made advanced static analysis accessible to a variety of software development teams [56].

Clang Static Analyzer remains a crucial part of the LLVM project's infrastructure for C-family languages with ongoing improvements to its path-sensitive analysis capabilities. The tool's checkers for detecting concurrency and memory safety issues have continuously improved because of active industry and academic collaboration. Because of its tight connection with the Clang compiler infrastructure, it is guaranteed to stay current with language standards, making it applicable to the embedded development and systems programming communities [57].

One well-known academic tool for examining safety-critical C code is the Frama-C platform [51]. It offers a set of compatible static analyzers based on formal techniques, such as weakest-precondition calculus, dependency analysis, and value analysis (to calculate variable ranges and demonstrate the lack of runtime mistakes). Within a single framework, these analyzers may be used to confirm functional features, demonstrate adherence to coding standards like MISRA C, and even produce verification conditions for interactive proof helpers like Coq [58]. Because of its modular design, it is a reference tool for essential systems in both academic research and industry applications, especially in the automobile and aerospace industries, where verification is necessary.

Infer is still being developed and used at scale to detect memory-safety and resource-management issues, with a focus on mobile and systems programming languages. The tool's utility in industrial contexts is demonstrated by published case studies, and its compositional analysis method based on separation logic has worked well for examining big codebases. Increasing the tool's accuracy and expanding its compatibility with modern programming paradigms are the aims of ongoing research and development [59].

Coverity is still a competitor in the commercial static analysis industry, with a constant focus on scalability and accuracy for business codebases. The tool's sophisticated inter-procedural analysis capabilities can be advantageous to organizations in safety-critical areas where certifications and standards compliance are required. Industry papers and user case studies highlight its application in large-scale software projects across a range of industries, including automotive and aerospace [52]. Recent empirical research has demonstrated its effectiveness in identifying complex security problems that traditional testing methods could miss [60].

### *3.4. Comparative Analysis*

The variety of static analysis approaches presents a challenging selection challenge for practitioners, requiring careful evaluation of organizational restrictions, integration requirements, and technical capabilities. Data-flow analysis offers the best scalability for huge codebases, even if it could miss subtle, path-dependent errors. Symbolic execution provides exceptional depth for security research, notwithstanding its difficulties with path explosion. Abstract interpretation offers soundness guarantees that are advantageous for important systems, despite the fact that it often produces false positives. Concurrent systems benefit greatly from model checking, but creating appropriate models requires a high level of expertise. Constraint solution enables precise verification of complex properties despite computational complexity issues [61].

Table 4 offers domain-specific suggestions for selecting appropriate instruments and analysis techniques. When mapping common application domains to recommended methodologies, it considers the unique verification requirements and restrictions of each domain. With this table, practitioners can quickly identify suitable solutions for their specific scenario.

**Table 4.** Technique Selection Guide by Application Domain.

| Application Domain | Recommended Techniques | Rationale | Representative Tools | Key Considerations | Typical Analysis Requirements |
|---|---|---|---|---|---|
| Enterprise Web Applications | Data-flow analysis, Pattern matching | Speed/coverage balance for DevOps | SonarQube, ESLint, PMD | CI/CD integration, multi-language support, fast feedback | Medium precision, High scalability, DevOps integration |
| Security-Critical Systems | Symbolic execution, Constraint solving | High precision for complex vulnerabilities | CodeQL, Coverity, Klocwork | Vulnerability detection, Low false negatives, Compliance | High precision, Deep analysis, Security rulesets |
| Safety-Critical Embedded | Abstract interpretation, Model checking | Soundness for standards compliance | Astrée, Clang Static Analyzer, Frama-C | Soundness guarantees, Certification support, MISRA compliance | High soundness, Standards compliance, Low false positives |
| Mobile Applications | Constraint solving, Data-flow analysis | Scalable memory safety analysis | Infer, Android Lint, SonarQube | Memory safety, Performance, Battery impact detection | Medium precision, Fast analysis, Resource leak detection |
| Systems Programming | Symbolic execution, Abstract interpretation | Deep bugs + sound verification | Clang Static Analyzer, KLEE, Coverity | Memory safety, Concurrency, Low-level code analysis | Deep analysis, Path sensitivity, Pointer analysis |
| Academic/ Research | All techniques (custom implementations) | Flexibility for algorithm research | CPA checker, SeaHorn, Frama-C | Extensibility, Algorithm experimentation, Formal methods | Research flexibility, Latest techniques, Customization |

Table 5 compares the fundamental technical characteristics of core analytical methodologies using key criteria, including false-positive rates, scalability, precision, and soundness. This technical comparison enables well-informed conclusions about which analytical techniques best address specific verification challenges.

When selecting a tool, the specific verification objectives should be considered. For example, enterprise teams should prioritize SonarQube's quality management features, embedded systems developers should benefit from Clang Static Analyzer's language expertise, and security-focused organizations should benefit from CodeQL's query-based approach. Using a range of tools at different stages of the development lifecycle, such as the most comprehensive analyses in nightly builds, more sophisticated tools in pull request validation, and fast, lightweight analyzers in IDEs and pre-commit hooks, organizations are increasingly implementing a defense-in-depth approach [62].

This taxonomy, which connects their theoretical foundations to practical applications, systematically classifies the dominant paradigms in static analysis. Each of the basic methods: data-flow analysis, symbolic execution, abstract interpretation, model verification, and constraint solving has a special location in the design space of soundness, precision, and scalability. Their use in applications such as SonarQube, CodeQL, Clang Static Analyzer, Infer, and Coverity demonstrates how these concepts are adapted to address specific verification problems across different domains. Understanding these links enables more informed tool selection and provides a foundation for evaluating new analysis techniques

as the field continues to evolve in response to new software development practices and application areas.

**Table 5.** Technical Characteristic Comparison of Analysis Techniques.

| Technique | Soundness | Scalability | Evidence/Key Study | Primary Use Cases |
|---|---|---|---|---|
| Data-Flow Analysis | Medium | High | Flow-insensitive approximations enable scalability for large codebases but limit precision [30]. SonarQube achieves analysis speeds of ~10 K LoC/s [50]. | Code quality, Bug patterns, Style checks |
| Symbolic Execution | High (on paths) | Low-Medium | The path explosion problem fundamentally limits scalability [33], but it is effective for deep bug finding in KLEE. KLEE achieved 90% line coverage of GNU Coreutils [34]. | Security vulnerabilities, Complex logic bugs |
| Abstract Interpretation | High | Medium | Sound over-approximation guarantees completeness at cost of false positives [36]; used in Astrée for avionics. Astrée demonstrated the absence of RTE in 132 K LoC of Airbus flight control software [38]. | Safety verification, Runtime error proof |
| Model Checking | High | Low | The state explosion problem restricts the application to abstracted models [41], which is the standard for protocol verification. State space limited to ~$10^6$ states without abstraction [41]. | Concurrency, Protocol verification |
| Constraint Solving | High | Low-Medium | SMT solving enables precise verification but faces computational complexity barriers [44]. Z3 solves constraints with $10^4$+ clauses in seconds [45]. | Complex properties, Type verification |
| Pattern Matching | Low | Very High | Syntax-level matching provides fast feedback but lacks semantic understanding [50]. ESLint analyzes 100 K LoC in <10 s [55]. | Syntax checks, Style enforcement |

The appropriateness of any approach to a family is strongly influenced by the verification situation. Data-flow analysis does not work well in large-scale industrial codebases where scalability is critical and superficial bug patterns are the primary concern, whereas path-dependent mistakes require deep semantic knowledge. It is not appropriate for whole-program analysis without severe pruning or compositional approaches due to the exponential growth in symbolic execution paths, but it is unparalleled at identifying complex, critical security flaws. The ideal method is abstract interpretation for safety-critical systems that require verifiable assurances of correctness (such as the absence of runtime mistakes in aircraft software). This method's over-approximation flaw, however, frequently leads to false positives that require manual triage. Model checking provides comprehensive verification of concurrent and reactive systems, but its state explosion problem restricts its use to abstracted models or components with constrained state spaces. Despite its exceptional accuracy in verifying complex functional properties, constraint solving's computational cost limits its use in big or loop-intensive applications. Hybrid techniques are growing in popularity since no single strategy is dominant in all sectors. This is not due to implementation errors, but to fundamental trade-offs.

## 4. Applications of Static Analysis in Embedded and Electronic Systems

Static analysis has evolved from a specialized verification technique into a central component of modern software engineering for embedded, cyber-physical, and electronic systems. Its rapid adoption is driven by the safety-critical nature of cyber-physical platforms—where software faults can directly lead to physical consequences—the growing complexity of software-controlled electronic devices, and increasingly stringent safety and security regulations. This transition from primarily academic research to widespread industrial deployment represents one of the most significant developments in contemporary software engineering.

The following subsections outline the primary domains in which static analysis has demonstrated substantial impact. Emphasis is placed on its practical industrial adoption in electronic and safety-critical systems, its role in supporting compliance with regulatory standards, and recent research advances that illustrate the maturity, scalability, and effectiveness of modern static analysis techniques.

Each domain's level of development in the static analysis literature is shown by the scope of coverage across domains. Since security and safety-critical automobile systems have been the subject of extensive study and industry adoption for decades, a more comprehensive analysis is required. Even though they are growing quickly, fresh topics like smart contract verification and AI/ML pipeline testing are still being explored in the literature; hence, our review provides a focused overview of the early research and current trends in these areas.

### *4.1. Static Application Security Testing (SAST)*

One of the most well-known uses of static analysis is Static Application Security Testing (SAST), which identifies security flaws before software is released [63]. The method looks for code patterns that indicate possible security flaws by examining software artifacts, like source code or binaries, without running them. Because it can examine the entire codebase instead of just the code paths used during a test, this offers a significant advantage over dynamic analysis.

The Common Weakness Enumeration (CWE) [64] and the OWASP Top Ten [65] are two of the major safety regulations that SAST tools systematically search for vulnerability types. They are particularly good at identifying serious vulnerabilities, such as buffer overflows, SQL injection, incorrect authentication, and sensitive data exposure [66].

Modern SAST tools use advanced methods that provide both deep inspection and broad code coverage to be effective. They mainly do this by using two specialized techniques: Symbolic Execution and Taint Analysis. By tracking untrusted user input from its entrance point (source) to any sensitive activity (sink), a technique known as "taint analysis", a specific type of data-flow analysis, finds security threats. If the data reaches the sink unsensitized, it notifies developers [67]. A technique called symbolic execution looks at a variety of possible routes a program could take while running [68]. It excels at identifying extensive vulnerability chains that use complex conditional logic and span procedural boundaries. These fundamental methods are still evolving in specific fields. For instance, opcode-based analysis for PHP vulnerability identification (Zhao et al., 2025) [53] and scalable inter-procedural taint tracking for PHP applications (Luo et al., 2022) [54] are two examples of how current research has expanded taint and data-flow analysis for contemporary online ecosystems.

Vulnerability detection in a variety of programming languages and application domains has significantly improved as a result of the integration of these techniques in industrial tools. For example, taint analysis has been effectively used to identify SQL injection and cross-site scripting vulnerabilities in web applications [69].

Despite their benefits, empirical research shows that SAST tools have significant practical challenges that could hinder their broad implementation. The main issues include a large number of false alarms and difficulties integrating the tools into developer workflows. These characteristics negatively impact developers' opinions of the tool's effectiveness [70]. Furthermore, these issues were confirmed by a thorough evaluation carried out in open-source projects by Beller et al., which also showed notable differences in the warnings produced and the difficulties in properly setting the tools [55]. All of these findings show that for SAST to be successful in industry, its accuracy must be balanced with usefulness [71].

Integrating SAST directly into modern software development workflows is a crucial evolution in SAST adoption. Instead of being a stand-alone, late-phase audit, the goal is to make analysis a crucial part of the coding process. We have lessons learnt from deploying static analysis at Google, emphasizing that for adoption to be successful, tools must be fast, perform incremental analysis, and give developers prompt, actionable warnings [72]. SAST's maturity is demonstrated by its official acknowledgment in security standards [73]. The National Institute of Standards and Technology's (NIST) Secure Software Development Framework (SSDF) expressly recommends it as an essential verification process [74]. The primary goal of current research is to solve persistent scalability and usability problems. As software systems grow increasingly complicated with cloud-native architectures and microservices, SAST's role remains vital, and efforts to adapt these techniques for new environments are ongoing.

### *4.2. Smart Contract Analysis*

Blockchain smart contracts are a significant topic for specialist static analysis due to their unique characteristics. A smart contract cannot be altered or corrected once it has been implemented, in contrast to traditional software. To prevent irreparable financial loss, it is therefore essential to thoroughly review the code before it goes live. Furthermore, the blockchain platform itself creates special security flaws, including reentrancy attacks and math overflows, which have been utilized in serious financial crime events [75].

In response, static analysis tools designed specifically for smart contracts have quickly developed. Research by Durieux et al. found that the ability of nine different analysis methods to find vulnerabilities varied greatly. Since no single instrument was able to detect every type of problem, a comprehensive security review requires the employment of multiple tools in tandem [76].

The technology of these tools combines modified standard procedures with new blockchain-specific strategies. For instance, tools like Mythril use symbolic execution, which looks at a variety of possible contract execution scenarios to find inputs that can result in a vulnerability. Another method, formal verification, uses mathematical models to show that a contract complies with specific safety rules, thereby offering solid guarantees regarding its behavior [77].

Because smart contract vulnerabilities might have significant financial consequences, this in-depth investigation is required. In a thorough survey of attacks, Atzei et al. classified the main vulnerability patterns that modern analysis tools are intended to detect [78]. Because of these high-profile occurrences, thorough static analysis is now a routine process in professional blockchain security audits. Top auditing companies now use a number of static analysis techniques as part of their evaluation process.

The area is constantly changing to meet new difficulties, especially in decentralized finance (DeFi). For example, Tsankov et al. developed Securify, a scanner that matches contracts to known vulnerability patterns and confirms the absence of particular significant

flaws [79]. As blockchain technology and applications advance, static analysis techniques are evolving to provide ongoing security.

### 4.3. Automotive/Embedded Systems

Verification is particularly difficult for embedded and cyber-physical systems, in contrast to general-purpose software. These include tight hardware coupling, real-time execution limitations, restricted memory and computing resources, and the requirement to follow safety certification requirements such as ISO 26262 for automotive systems and DO-178C for aircraft. In these areas, static analysis approaches must prioritize soundness and determinism over sheer bug-finding performance. Additionally, these methods need to be able to reason about low-level system characteristics such as worst-case execution time (WCET), interrupt management, and memory layout.

Some examples of the increasing complexity of modern cyber-physical systems include advanced control architectures such as adaptive sliding-mode security control for inverted pendulums under false-data injection attacks [80] and state-constrained adaptive fuzzy exact tracking control for nonlinear strict-feedback systems [81]. Despite addressing control-theoretic concerns, these studies also highlight the growing need for formal software verification techniques to ensure the precision, safety, and security of embedded software that uses these controllers. Static analysis is crucial in this case because it enables the early detection of software defects before they compromise the system's stability or security.

In safety-critical industries such as automotive and aerospace, static analysis is not only recommended but also mandated by law. International standards such as DO-178C for aviation and ISO 26262 for vehicles set strict requirements to ensure that software behaves predictably in all situations. These guidelines formally recognize static analysis as an essential method for demonstrating compliance, especially in systems where a software defect could endanger human lives [82].

A technique called abstract interpretation has shown great success for this type of verification since it can provide mathematical guarantees about a program's behavior. The Astrée analyzer, developed by Blanchet et al., is among the best illustrations of this technique in large-scale industrial software. By simulating every possible program state using complex mathematical models, it can be shown that aviation control systems do not exhibit significant runtime faults, such as calculation overruns and unauthorized memory accesses [83].

Another crucial use of static analysis in embedded domains is the enforcement of coding standards, such as MISRA C, which provide rules that forbid error-prone language structures. Functional safety standards require these regulations to be enforced through static analysis, a fundamental component of development processes in these sectors. Studies have demonstrated that integrating these technologies into car development leads to early fault discovery and improved code quality, despite significant configuration and results management challenges [84].

The automobile industry's shift to software-defined cars has made static analysis more important and challenging than before. In current autos, many electronic control units (ECUs) contain large quantities of software. Advanced technologies, like automated driving aids, require extensive verification. The fundamental challenges of validating autonomous driving software, where static analysis is indispensable, have been systematically outlined in the literature [85].

New developments in embedded systems static analysis have concentrated on tackling the particular difficulties of cyber-physical systems. Static timing analysis, a method for figuring out worst-case execution durations (WCET) for crucial code segments, is crucial for fulfilling real-time restrictions in safety-critical systems. Static analysis approaches are

growing to meet the convergent objectives of cybersecurity and functional safety as cars become more autonomous [86].

### 4.4. AI/ML Software Testing

The rapid development of machine learning (ML) and artificial intelligence (AI) systems has created new verification challenges that require specialized static analysis tools. ML systems process data through a complex pipeline that includes preparation, training, and final deployment, unlike traditional software. This results in distinct failure types that traditional static analysis was not designed to detect, like data leaks between training and test sets, dimension mismatches, and biased results [87–89].

As a result, static analysis for AI/ML has become a separate field of study. The issue of "hidden technical debt" in machine learning has been recognized in foundational work, where important flaws often occur in data preparation, feature engineering, and integration code rather than in the models themselves. Data leakage remains a common problem that can invalidate performance measurements when information from the test set unintentionally affects model training. To find these incorrect information flows, static analysis may automatically audit data partitioning and transformation logic [90,91].

Another crucial use is model integration code verification. According to research on ML testing procedures, a sizable percentage of production errors come from the code that prepares inputs, calls the model, or post-processes outputs rather than the core model. For this, the integration layer is a perfect target for static analysis to find problems like tensor dimension mistakes and API abuse [92].

Detecting discrimination and guaranteeing fairness has become a crucial frontier as ML systems are used in socially significant fields, including criminal justice, lending, and employment. In order to find possible sources of bias prior to deployment, research in this field uses static analysis to examine how sensitive characteristics are handled along ML pipelines [93].

A continuing problem for analytic tools is the quick development of ML frameworks and architectures. Comprehensive taxonomies of deep-learning system-specific errors have been produced recently, offering an organized basis for creating more efficient testing and verification tools [94,95]. Static analysis's function in guaranteeing the dependability, equity, and security of these systems is set to rise significantly as ML use picks up speed in safety and security-critical industries like healthcare and autonomous systems [96].

The static analysis techniques reviewed here are expected to become more specialized and adapted as machine learning continues to permeate embedded and safety-critical systems. This trend is similar to the previous development of the security and automotive verification domains.

## 5. Challenges and Limitations

Despite significant advancements and extensive use, static analysis still has fundamental problems that limit its maximum effectiveness and broader use. These limits stem from theoretical constraints, real engineering trade-offs, and organizational adoption hurdles. These challenges must be fully understood by researchers working to advance the State-of-the-Art and practitioners hoping to successfully integrate static analysis into software development practices.

### 5.1. Scalability and Performance

The computational complexity of modern static analysis techniques is a significant barrier to their application on large-scale industrial codebases. Context-sensitive interprocedural analysis and path-sensitive symbolic execution are two examples of deep

semantic-comprehension techniques that often exhibit exponential time and space complexity with respect to program size. This scaling problem becomes particularly severe in modern software systems, which often have millions of lines of code [97]. The technical features of particular analysis families are the underlying source of this difficulty. Our comparative taxonomy (Section 3.4, Table 5) quantifies the exponential complexity (path or state explosion) that deep techniques like symbolic execution and model checking face, which makes them challenging to scale to such large codebases without aggressive abstraction, even though they provide unmatched path or state coverage.

The basic trade-offs between performance and analytical depth in industrial settings have been well studied. The most accurate static analyses frequently become computationally prohibitive for big codebases, necessitating a workable trade-off between soundness, accuracy, and performance, as experience from developing tools at scale shows [98]. Another significant scaling limitation is the memory use of complex static analyzers. In order to handle this complexity, compositional and incremental analysis approaches have been the subject of much study [99].

### 5.2. False Positives and Alert Fatigue

The prevalence of false positives, or warnings that mistakenly imply the presence of problems, continues to be one of the main barriers to the application of static analysis in industrial practice. Empirical research has consistently shown that high false-positive rates undermine developer trust in analysis tools and reduce the likelihood of continuous use. When developers obtain a large number of bogus warnings, they may become fatigued and begin to disregard even legitimate alerts [100].

Extensive studies of developers' experiences with static analysis tools have revealed that tool adoption and developer satisfaction were greatly influenced by false-positive rates that exceeded specific criteria. Tools that use over-approximation techniques to guarantee soundness make this issue worse since these methods purposefully lean on the side of reporting possible problems that might not really be flaws in practice. This is not an issue with tool implementation, but rather a direct consequence of the soundness-precision trade-off described by abstract interpretation (Section 3.2.3). Our technical comparison (Table 5) clearly scores abstract interpretation as "High" soundness but "Low-Medium" accuracy, with a "High" false-positive rate, illustrating this intrinsic trade-off. However, techniques such as pattern matching (Table 5) do not ensure soundness and have a few false positives. Managing false positives is crucial to preserving developer confidence in the practical use of static analysis [101].

### 5.3. Soundness vs. Precision Trade-Off

The inherent trade-off between accuracy and soundness in static analysis is a basic theoretical and practical restriction. While precision shows that the majority of issued warnings match real faults (few false positives), soundness refers to the property that an analysis reports all true defects of a given class (no false negatives). This contradiction is a basic design constraint that fundamentally defines all static analysis tools and approaches, rather than just an implementation difficulty [102]. Each approach is positioned along this spectrum in our taxonomy of key techniques (Section 3.2). For instance, symbolic execution (Section 3.2.2) maximizes precision on investigated pathways, but abstract interpretation (Section 3.2.3) is intended for soundness.

Practical analytical design decisions are directly affected by this limitation. Over-approximation, which unavoidably adds false positives, is how sound studies, which ensure no false negatives, accomplish this completeness. On the other hand, precision-optimized studies increase accuracy by concentrating on probable fault patterns at the expense of

possibly overlooking real problems. The fundamental formal basis for managing this trade-off is the discipline of abstract interpretation, which enables the creation of analyzers that strategically manage precision loss through carefully crafted abstract domains while guaranteeing soundness.

### 5.4. Benchmarking and Evaluation Challenges

The absence of recognized norms for assessing static analysis tools is a significant obstacle to technical progress and comparative evaluation. Due to the lack of set standards and the disparate assessment techniques used by tool makers and researchers, it is difficult to directly compare analytic capabilities. This difficulty is made worse by the different fundamental capabilities of the approaches themselves. Benchmarking must consider whether a tool is intended for sound verification (e.g., those using abstract interpretation, as in Section 3.2.3) or high-precision bug finding (e.g., symbolic execution engines, as in Section 3.2.2), as these goals and acceptable trade-offs differ. This variation makes it challenging for practitioners to select appropriate tools and to objectively assess developments in the area [103]. A crucial foundation for the assessment of competing tools is provided by standardized benchmark efforts such as the International Competition on Software Verification (SV-COMP) [104], which have emerged to provide common benchmark suites and evaluation methodologies. In a similar vein, other programs, such as the DARPA Space/Time Analysis for Cybersecurity (STAC) program [105], have highlighted the need and challenges of developing benchmark suites that accurately replicate real-world software with clearly documented flaws.

Methodological flaws in evaluation methods for static analysis have been examined. A large-scale study revealed widespread issues with metric and benchmark selection, making it more challenging to assess the true advantages and disadvantages of different static analysis approaches [106].

### 5.5. Integration into Development Workflows

There are several organizational and technological obstacles to successfully incorporating static analysis into contemporary software development workflows. Static analysis is most valuable when it is smoothly integrated into development processes; however, this necessitates resolving problems with results management, tool configuration, and workflow disturbance. Despite their technical advantages, poorly integrated analytic tools frequently have limited uptake, according to research [107].

Static analysis integration presents both opportunities and challenges as a result of the move toward DevOps and continuous integration. These procedures demand stringent performance criteria, even though they offer obvious opportunities for automated analysis. To prevent hindering quick development cycles, tools must finish deeper analysis within the strict time limitations of CI/CD pipelines and offer nearly immediate feedback within the developer's IDE [108].

Configuration complexity has been found to be a major obstacle to successful integration in empirical investigations of the adoption of static analysis. With the hundreds or thousands of customizable tests that modern static analyzers usually offer, choosing the right settings for particular projects requires a great deal of experience. This setup complexity is influenced by the specialization of the instruments described in our survey (Section 3.3, Table 3). An engineer must choose between using a sound, safety-focused analyzer like Frama-C (suited for embedded systems) or a precision-oriented security tool like CodeQL for their specific verification requirement. Each requires separate rule sets and tweaking. Another crucial issue is managing and prioritizing analytical results. Effective integration necessitates the provision of context-aware findings that individual developers

can promptly evaluate and act upon, according to investigations on how developers utilize these tools [109]. Additionally, providing collaborative defect management and tracking becomes necessary when this procedure is scaled across teams.

The requirement for static analysis integration, built for the developer workflow and combining capabilities that facilitate team-scale management with actionable detection, is highlighted by these technical and usability issues. The field's continued vitality and its crucial role in creating more dependable, secure, and trustworthy software systems are demonstrated by the continuous advancement and specialization of static analysis, as seen in its application to security, smart contracts, embedded systems, and AI/ML.

### 5.6. Synthesis: Challenges as Manifestations of Core Trade-Offs

The limits examined in Sections 5.1–5.5 are examples of three fundamental, interrelated problems that characterize the boundaries of static analysis research rather than separate barriers:

1. The Balance Between Scalability and Precision. Because of their exponential complexity, sophisticated semantic analyses (symbolic execution, model checking) are only applicable to large-scale industrial codebases, notwithstanding their great precision. Scaling these techniques without unacceptable accuracy loss remains challenging; incremental and compositional approaches show promise but require further development.
2. The Conflict Between Soundness and False Positives. While sound analyses (e.g., abstract interpretation) guarantee thoroughness, they are too imprecise and yield costly false positives for developers. Faulty analyses (such as pattern matching, common in industrial SAST tools) can miss significant problems while reducing false positives. Closing this gap, whether through machine-learning-assisted triage, probabilistic ranking, or better abstract domains, has always been a top research focus.
3. The Gap Between Adoption and Usability. Even the most technically advanced tools are ineffective if they interfere with developer workflows, require extensive configuration, or display results without providing context that can be used. Translating analytical advancements into real-world impact requires research into collaborative defect management, intelligent defaults, and seamless CI/CD integration.

The need for fundamental improvements in the composition, design, and integration of studies, rather than minor adjustments to specific approaches, is a recurrent topic throughout the new paths discussed in Section 6.

### 5.7. Complementary Approaches: Dynamic Analysis and Fuzzing

While the primary emphasis of this examination is static analysis, it is important to note that complementary approaches are often used alongside static techniques. Dynamic analysis runs programs with concrete inputs to detect runtime faults, providing greater accuracy for observable behaviors but less coverage. Fuzzing automatically generates test inputs to trigger crashes or violations and is highly effective at detecting security flaws. Hybrid static–dynamic techniques combine the benefits of both paradigms by using static analysis to guide dynamic exploration or dynamic feedback to remove unrealistic static pathways. As discussed in Section 6.2, combining approaches can help overcome the limitations of each tactic alone.

## 6. Trends and Future Directions

Static analysis is being significantly impacted by the emergence of new programming paradigms, changes in software design, and advances in machine learning. Although traditional approaches based on data-flow analysis and abstract interpretation remain important, researchers and practitioners are exploring novel methods to improve accu-

racy, scalability, and usability. This section examines the major developments shaping the field's future, with a focus on integrating machine learning, complementing static and dynamic techniques, automating remediation, and adapting to new languages and distributed systems.

### 6.1. Machine-Learning-Enhanced Static Analysis

Machine learning offers promising solutions to recurring issues in static analysis, particularly in reducing false positives, tuning configurations, and approximating semantics. Machine-learning models may be trained on past warning data to forecast which alerts developers are likely to act on, thereby learning to distinguish between real issues and misleading warnings. Graph neural networks (GNNs) operating on code property graphs have shown particular promise for learning defect-relevant code representations, as they can capture syntactic and structural properties that conventional studies may miss [110].

Large language models (LLMs) that directly learn program semantics from massive code corpora, such as CodeBERT, offer a further advancement [111]. These models may discover subtle issue patterns and code smells that are difficult to capture in traditional rule-based systems. A significant unsolved problem is the semantic gap: LLMs exhibit remarkable pattern recognition but lack the proven soundness guarantees of formal techniques. Future studies should explore neuro-symbolic approaches that combine learned pattern recognition with verifiable correctness frameworks, including training GNNs to guide symbolic execution toward likely problematic routes or employing LLMs to generate formally verified candidate invariants.

### 6.2. Hybrid Static–Dynamic Analysis Methods

The complementary benefits of static and dynamic analysis have led to increased interest in hybrid methodologies that combine the two paradigms. Static analysis may investigate a large number of potential program states, whereas lightweight dynamic instrumentation confirms these discoveries at runtime by rejecting statically reasonable but dynamically impracticable pathways. An example of this synergy is the usage of sanitizers (such as AddressSanitizer) in combination with static checkers, where static analysis identifies potential memory corruption areas and dynamic testing confirms exploitability.

Static analysis, especially symbolic execution, generates high-value seed inputs for symbolically guided fuzzing, a particularly promising technique that directs fuzzers into deep, challenging-to-reach code pathways. Tools such as SAGE [112] have demonstrated the effectiveness of this approach by detecting small security vulnerabilities in large applications. Using runtime coverage data to enhance static models and dynamically remove unfeasible pathways, future research will likely focus on improving the feedback loop between static and dynamic components. Static analysis provides broad structural understanding, while dynamic testing provides concrete behavioral validation. The ultimate goal is a seamless integration.

### 6.3. Automated Program Repair

Automatic remediation, as opposed to defect identification, represents a paradigm shift that could transform software maintenance. Candidate patches are generated using generate-and-validate approaches (e.g., GenProg), which use genetic programming or template-based transformations, and are then validated against test suites [113]. Both the thoroughness of the test suite and the variety of its patch templates limit the scalability of these solutions. Though they have scalability problems repairing big systems, semantic-based approaches use constraint solving and program synthesis to produce fixes that are guaranteed to be accurate with respect to a formal specification.

The incorporation of LLMs has led to the emergence of a third paradigm, wherein models trained on massive datasets of human-written patches offer context-aware fixes for common issue patterns [114]. Despite the promising initial results, three key challenges remain: generalization (resolving bugs that require deep semantic knowledge), patch readability (ensuring generated code is human-maintainable), and patch correctness (moving beyond test-suite adequacy to semantic validity). Future research must integrate the formal guarantees of semantic approaches, the scalability of generate-and-validate, and the pattern detection capabilities of LLMs.

### *6.4. Language and Platform Evolution*

The landscape of static analysis is evolving with the advent of WebAssembly (Wasm), Go, and Rust. Rust's ownership model statically enforces memory safety without garbage collection, shifting the analysis away from traditional memory corruption toward logical mistakes, performance inefficiencies, and violations of safe concurrency patterns. Go's built-in concurrency primitives (goroutines, channels) introduce additional classes of potential problems, including races, deadlocks, and channel misuse. These specialized static checks can reason about lightweight thread interactions that are needed [115].

WebAssembly offers a new target: a small, structured bytecode format designed for safe execution in sandboxed contexts (browsers, plugin systems, blockchain smart contracts). Prior to deployment, Wasm static analysis must verify module isolation, resource consumption restrictions, and the absence of harmful code patterns [116]. Since Wasm is a compilation target for several source languages, analyses must operate at the bytecode level to recover higher-level semantics that are lost during compilation. Language-aware analyses that capitalize on language guarantees rather than undercut them must be developed in future study. Examples include using Rust's type system to reduce the scope of required checks or reconstructing control-flow and data-flow from Wasm bytecode for security auditing.

### *6.5. Cloud-Native and Distributed Systems Analysis*

As the architecture moves toward microservices, serverless operations, and distributed systems, networks of loosely connected, interacting services have supplanted single monolithic programs as the unit of analysis. Traditional intra- and even inter-procedural analysis is insufficient for detecting issues that transcend service boundaries, such as broken API contracts, inconsistent data formats, insecure communication channels, or cascading failures, in which a downstream service disruption propagates throughout the system.

In recent years, research on architecture-level static analysis has focused on creating system graphs with nodes representing databases, message queues, services, and other infrastructure components and edges indicating their interactions (API calls, message forwarding, data dependencies). The analysis of these graphs can identify circular dependencies, security vulnerabilities (such as unauthenticated service-to-service calls), and resilience issues (such as single points of failure) [117]. Static analysis must also be performed on the cloud-native environment-specific infrastructure-as-code (IaC) setups (CloudFormation, Terraform, and Kubernetes YAML). For example, tools that statically scan IaC templates for misconfigurations, such as dangerous default settings, wasteful resource allocations, or overly permissive access restrictions, offer an essential expansion of static analysis beyond application code to the whole software supply chain [118].

A key difficulty in cross-service data-flow research is tracking how data from one service moves across another, potentially revealing sensitive information or compromising trust boundaries. To achieve this, models of service interactions need to incorporate not

only control flow but also propagation semantics and data transformation, an area where research is still in its early stages.

## 7. Conclusions

This systematic study has thoroughly investigated the state of static analysis, charting its evolution from a basic academic field to a crucial component of modern software engineering. The rising criticality of software across all socioeconomic sectors, as seen by high-profile systemic failures, has transformed static analysis from a best practice to an essential component of responsible development.

Through a synthesis of a substantial body of literature, this work has mapped the basic techniques, numerous applications, persistent challenges, and novel directions that define this dynamic area. This survey provides a specialized reference for researchers and practitioners working at the intersection of critical infrastructure and static analysis, focusing on embedded, cyber-physical, and electronic systems, where software correctness is not only a quality concern but also a safety and reliability imperative.

This work's main contribution is an organized taxonomy that connects theoretical formal approaches with their real-world industrial applications. From scalable data-flow analysis to the intricate yet computationally demanding fields of symbolic execution and abstract interpretation, this review has covered a wide range of fundamental techniques and demonstrated their application in programs such as SonarQube, CodeQL, and the Clang Static Analyzer. This paradigm provides a rational foundation for method and tool selection by elucidating the underlying engineering trade-offs among soundness, accuracy, and scalability, as summarized in our comparative study (Section 3.4, Table 5).

Another significant contribution is the detailed analysis of domain-specific specializations, which demonstrates how static analysis has been painstakingly adjusted to handle the specific verification issues in application security, blockchain smart contracts, safety-critical automotive systems, and AI/ML pipelines. This cross-domain perspective emphasizes the field's remarkable growth and flexibility.

The study questions presented in Section 1 are directly addressed by our survey. The results show that choosing a static analysis method is a conscious decision along several competing dimensions for RQ1 (taxonomy of techniques): the scalability of data-flow analysis, the path-sensitive precision of symbolic execution, and the soundness guarantees of abstract interpretation. Evidence for RQ2 (domain applications) indicates that effective static analysis is now highly specialized, with emerging fields like smart contracts and AI/ML creating their own customized verification ecosystems, safety-critical domains requiring sound abstract interpretation (e.g., Astrée, Frama-C), and security-critical systems utilizing symbolic execution (e.g., CodeQL). The ongoing issues listed in Section 5, scalability, false positives, and integration barriers, are real-world examples of the fundamental trade-offs noted in RQ1 for RQ3 (limitations and future directions). New developments such as hybrid static–dynamic approaches (Section 6.2) and machine-learning-enhanced analysis (Section 6.1) are examples of research endeavors aimed at navigating rather than eradicating these intrinsic constraints.

Significant research gaps still need to be addressed despite these advancements. Hybrid systems that employ learning within a framework of verifiable correctness must be the focus of future study. A major problem in ML-based analysis is the semantic gap, where models exhibit remarkable pattern recognition but lack formal methods' provable soundness guarantees. The study of distributed, cloud-native systems is still in its early stages, as current approaches struggle to reason about system-wide features such as cross-service data flows. Furthermore, the usability and accessibility of sophisticated static analyzers remain a major adoption hurdle, necessitating research into intelligent default configura-

tions and minimal-disruption workflow integration. Finally, the persistent speed of change in software development presents a constant challenge that requires static analysis tools to become more organically flexible and extensible. As electronic systems continue to evolve toward software-defined, autonomous, and AI-enabled architectures, static analysis will remain a cornerstone technology for ensuring the safety, security, and reliability of next-generation electronic platforms.

In conclusion, static analysis is an essential part of modern software quality assurance. Its movement from abstract theory to practical application emphasizes its crucial and expanding worth. The role of static analysis will only expand as the complexity, ubiquity, and importance of software systems increase. Our taxonomy shows how the area has developed from fundamental methods to specific applications, demonstrating both its maturity and versatility. By building on its solid theoretical foundations and proactively embracing new paradigms in machine intelligence and systems architecture, the field is strategically positioned to address the difficult verification challenges that lie ahead, ensuring the dependability, security, and safety of the digital infrastructure on which the world increasingly depends.

This survey offers researchers and practitioners working at the nexus of static analysis and critical infrastructure a specialized reference by methodically concentrating on embedded, cyber-physical, and electronic systems domains where software correctness is not only a quality concern but also a safety and reliability imperative.

**Author Contributions:** Conceptualization, M.I.; methodology, M.I. and T.K.; software, M.I. and T.K.; validation M.I., T.K., and A.L.; formal analysis, M.I. and T.K.; investigation, M.I. and T.K.; resources, M.I. and T.K.; data curation, T.K.; writing—original draft preparation, T.K.; writing—review and editing, M.I., T.K., and A.L.; visualization, T.K.; supervision, M.I., T.K., and A.L.; project administration, A.L.; funding acquisition, A.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** The article was prepared under Project No. 101244751, titled AISSAM—"Automated Vulnerability Detection in Software Development Using AI Techniques" (under the call HORI-ZON-WIDERA-2024-TALENTS-02, type of action: HORIZON TMA MSCA Postdoctoral Fellowships).

**Data Availability Statement:** No new data were created or analyzed in this study. Data sharing is not applicable to this article.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Anderson, R.; Barton, C.; Bölme, R.; Clayton, R.; Ganán, C.; Grasso, T.; Levi, M.; Moore, T.; Vasek, M. Measuring the changing cost of cybercrime. In Proceedings of the 2019 Workshop on the Economics of Information Security, Boston, MA, USA, 3–4 June 2019.
2. US Government. *The Equifax Data Breach*; Majority Staff Report 115th Congress, US; House of Representatives Committee on Oversight and Government Reform: Washington, DC, USA, 2018.
3. Youvan, D.C. Anatomy of a Financial Collapse: The Role of Technical Glitches in Modern Financial Systems. 2024. Available online: https://www.researchgate.net/profile/Douglas-Youvan/publication/382968606_Anatomy_of_a_Financial_Collapse_The_Role_of_Technical_Glitches_in_Modern_Financial_Systems/links/66b52c858f7e1236bc459eea/Anatomy-of-a-Financial-Collapse-The-Role-of-Technical-Glitches-in-Modern-Financial-Systems.pdf (accessed on 10 February 2026).
4. Møller, A.; Schwartzbach, M.I. *Static Program Analysis*; Department of Computer Science, Aarhus University: Aarhus, Denmark, 2020. Available online: https://cs.au.dk/ amoeller/spa/ (accessed on 10 February 2026).
5. Zhou, Y.; Sharma, A. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 914–919.
6. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 297–308.

7. Sadowski, C.; Aftandilian, E.; Eagle, A.; Miller-Cushon, L.; Jaspan, C. Lessons from building static analysis tools at google. *Commun. ACM* **2018**, *61*, 58–66. [CrossRef]
8. Myrbakken, H.; Colomo-Palacios, R. DevSecOps: A multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination*; Springer International Publishing: Cham, Switzerland, 2017; pp. 17–29. [CrossRef]
9. Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Zaidman, A.; Gall, H.C. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*; IEEE: New York, NY, USA, 2018; pp. 38–49.
10. Hu, C. Machine Learning Slashes the Testing Needed to Work Out Battery Lifetimes. 2026, pp. 41–42. Available online: https://www.nature.com/articles/d41586-026-00168-w (accessed on 10 February 2026).
11. Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Gall, H.C.; Zaidman, A. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* **2020**, *25*, 1419–1457. [CrossRef]
12. Li, Y.; Liu, W.; Liu, Q.; Zheng, X.; Sun, K.; Huang, C. Complying with ISO 26262 and ISO/SAE 21434: A safety and security co-analysis method for intelligent connected vehicle. *Sensors* **2024**, *24*, 1848. [CrossRef] [PubMed]
13. Rierson, L. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*; CRC Press: Boca Raton, FL, USA, 2017.
14. Liu, M.; Wang, J.; Lin, T.; Ma, Q.; Fang, Z.; Wu, Y. An empirical study of the code generation of safety-critical software using llms. *Appl. Sci.* **2024**, *14*, 1046. [CrossRef]
15. Beyer, D. Software Verification and Verifiable Witnesses: (Report on SV-COMP 2015). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 401–416.
16. Lenarduzzi, V.; Pecorelli, F.; Saarimaki, N.; Lujan, S.; Palomba, F. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.* **2023**, *198*, 111575. [CrossRef]
17. Feist, J.; Grieco, G.; Groce, A. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*; IEEE: New York, NY, USA, 2019; pp. 8–15.
18. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2019; pp. 783–794.
19. Rimsa, A.; Nelson Amaral, J.; Pereira, F.M. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* **2021**, *51*, 353–384. [CrossRef]
20. Nandi, A.; Mandal, A.; Atreja, S.; Dasgupta, G.B.; Bhattacharya, S. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2016, San Francisco, CA, USA, 13–17 August 2016*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 215–224.
21. Johnson, A.; Waye, L.; Moore, S.; Chong, S. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Not.* **2015**, *50*, 291–302. [CrossRef]
22. Liu, Z.; Tang, Z.; Zhang, J.; Xia, X.; Yang, X. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *IEEE/ACM 46th International Conference on Software Engineering*; IEEE: New York, NY, USA, 2024; pp. 1–13.
23. Khedker, U.; Sanyal, A.; Sathe, B. *Data Flow Analysis: Theory and Practice*; CRC Press: Boca Raton, FL, USA, 2017.
24. Baldoni, R.; Coppa, E.; D'elia, D.C.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–39. [CrossRef]
25. Rival, X.; Yi, K. *Introduction to Static Analysis: An Abstract Interpretation Perspective*; MIT Press: Cambridge, MA, USA, 2020.
26. Kant, G.; Laarman, A.; Meijer, J.; Van de Pol, J.; Blom, S.; Van Dijk, T. LTSmin: High-performance language-independent model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 692–707.
27. Moher, D.; Liberati, A.; Tetzlaff, J.; Altman, D.G.; Prisma Group. Preferred reporting items for systematic reviews and meta-analyses: The PRISMA statement. *Int. J. Surg.* **2010**, *8*, 336–341. [CrossRef]
28. Beyer, D. Software verification: 10th comparative evaluation (SV-COMP 2021). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer International Publishing: Cham, Switzerland, 2021; pp. 401–422. [CrossRef]
29. Mongiovì, M.; Giannone, G.; Fornaia, A.; Pappalardo, G.; Tramontana, E. Combining static and dynamic data flow analysis: A hybrid approach for detecting data leaks in Java applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing 2015*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1573–1579.
30. Sherman, E.; Dwyer, M.B. Structurally defined conditional data-flow static analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer International Publishing: Cham, Switzerland, 2018; pp. 249–265. [CrossRef]
31. Späth, J.; Ali, K.; Bodden, E. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
32. Trabish, D.; Mattavelli, A.; Rinetzky, N.; Cadar, C. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 350–360.

33. Chen, Z.; Chen, Z.; Shuai, Z.; Zhang, G.; Pan, W.; Zhang, Y.; Wang, J. Synthesize solving strategy for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 348–360.
34. Cadar, C.; Nowack, M. KLEE symbolic execution engine in 2019. *Int. J. Softw. Tools Technol. Transf.* **2021**, *23*, 867–870. [CrossRef]
35. Han, H.; Kyea, J.; Jin, Y.; Kang, J.; Pak, B.; Yun, I. Queryx: Symbolic query on decompiled code for finding bugs in COTS binaries. In *2023 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2023; pp. 3279–3295.
36. Mirman, M.; Gehr, T.; Vechev, M. Differentiable abstract interpretation for provably robust neural networks. In Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018.
37. Darais, D.; Might, M.; Van Horn, D. Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis. *ACM SIGPLAN Not.* **2015**, *50*, 552–571. [CrossRef]
38. Bertrane, J.; Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Rival, X. Static analysis and verification of aerospace software by abstract interpretation. *Found. Trends® Program. Lang.* **2015**, *2*, 71–190. [CrossRef]
39. Babati, B.; Horváth, G.; Májer, V.; Pataki, N. Static analysis toolset with Clang. In Proceedings of the 10th International Conference on Applied Informatics 2017, Eger, Hungary, 30 January–1 February 2017.
40. Biallas, S. Verification of Programmable Logic Controller Code Using Model Checking and Static Analysis. Ph.D. Thesis, RWTH Aachen University, Aachen, Germany, 2016.
41. Nejati, F.; Abd Ghani, A.A.; Yap, N.K.; Jafaar, A.B. Handling state space explosion in component-based software verification: A review. *IEEE Access* **2021**, *9*, 77526–77544. [CrossRef]
42. Kunze, S. Automated Test Case Generation for Function Block Diagrams Using Java Path Finder and Symbolic Execution. Master's Thesis, Mälardalen University, Västerås, Sweden, 2015.
43. Konnov, I.; Kukovec, J.; Tran, T.H. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.* **2019**, *3*, 123. [CrossRef]
44. Van Antwerpen, H.; Néron, P.; Tolmach, A.; Visser, E.; Wachsmuth, G. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation 2016*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 49–60.
45. Liang, T.; Reynolds, A.; Tsiskaridze, N.; Tinelli, C.; Barrett, C.; Deters, M. An efficient SMT solver for string constraints. *Form. Methods Syst. Des.* **2016**, *48*, 206–234. [CrossRef]
46. Harmim, D.; Marcin, V.; Pavela, O. Scalable Static Analysis Using Facebook Infer. I, VI-B. 2019. Available online: https://d1wqtxts1xzle7.cloudfront.net/101074818/59-libre.pdf?1681478089=&response-content-disposition=inline%3B+filename%3DScalable_Static_Analysis_Using_Facebook.pdf&Expires=1771624343&Signature=V3uuHpCj-uJ-Qu22t1AX1McDuj6F4ue-Oz9A5oNw3GMz0djPDv1uaDnUNkGGXYgGXAekxlORexbEcsjCtRGEUrvRnlSBhjRf QYERVJUWdc528rUegdZkQNT2nE4PrTfc IIOOEojR5ZfGqr yGvdhRmt AG8FUoMQsRo9hIgpwZFyc2OntK4IhKqA9TopSvqW5twWPl-OBlXS28SPGciHt-4D0EYf Q-ddza9mBHfzjUvozkJ85GLhZZdq5qYx6q yiYNu7LxcYXNxztgG7FBWpqNvRyOrZQTo4UwU9atgZP4yG2cEDp5VzduOkXwFDD QbY1e4RG3UdrHAf5wOIPdg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA (accessed on 10 February 2026).
47. Priya, S.; Zhou, X.; Su, Y.; Vizel, Y.; Bao, Y.; Gurfinkel, A. Verifying verified code. *Innov. Syst. Softw. Eng.* **2022**, *18*, 335–346. [CrossRef]
48. Baier, D.; Beyer, D.; Chien, P.C.; Jakobs, M.C.; Jankola, M.; Kettl, M.; Wendler, P. Software verification with CPAchecker 3.0: Tutorial and user guide. In *Formal Methods, Proceedings of the 26th International Symposium (FM 2024), Milan, Italy, 9–13 September 2024*; Springer Nature Switzerland: Cham, Switzerland, 2024; pp. 543–570. [CrossRef]
49. Beyer, D.; Lingsch-Rosenfeld, M. CPAchecker 4.0 as Witness Validator: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer Nature Switzerland: Cham, Switzerland, 2025; pp. 192–198. [CrossRef]
50. Marcilio, D.; Bonifácio, R.; Monteiro, E.; Canedo, E.; Luz, W.; Pinto, G. 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)*; IEEE: New York, NY, USA, 2019; pp. 209–219.
51. Kirchner, F.; Kosmatov, N.; Prevosto, V.; Signoles, J.; Yakobowski, B. Frama-C: A software analysis perspective. *Form. Asp. Comput.* **2015**, *27*, 573–609. [CrossRef]
52. Bessey, A.; Block, K.; Chelf, B.; Chou, A.; Fulton, B.; Hallem, S.; Henri-Gros, C.; Kamsky, A.; McPeak, S.; Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* **2010**, *53*, 66–75. [CrossRef]
53. Zhao, J.; Zhu, K.; Yu, L.; Huang, H.; Lu, Y. Yama: Precise Opcode-Based Data Flow Analysis for Detecting PHP Applications Vulnerabilities. *IEEE Trans. Inf. Forensics Secur.* **2025**, *20*, 7748–7763.
54. Luo, C.; Li, P.; Meng, W. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 2175–2188. [CrossRef]

55. Beller, M.; Bholanath, R.; McIntosh, S.; Zaidman, A. 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)*; IEEE: New York, NY, USA, 2016; Volume 1, pp. 470–481.
56. Hajdu, Á.; Marescotti, M.; Suzanne, T.; Mao, K.; Grigore, R.; Gustafsson, P.; Distefano, D. InfERL: Scalable and extensible Erlang static analysis. In *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang 2022*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 33–39.
57. Youn, D.; Lee, S.; Ryu, S. Declarative static analysis for multilingual programs using CodeQL. *Softw. Pract. Exp.* **2023**, *53*, 1472–1495. [CrossRef]
58. Kosmatov, N.; Prevosto, V.; Signoles, J. *Guide to Software Verification with Frama-C*; Springer: Berlin/Heidelberg, Germany, 2024. [CrossRef]
59. Umann, K.; Porkoláb, Z. Towards Better Static Analysis Bug Reports in the Clang Static Analyzer. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*; IEEE: New York, NY, USA, 2025; pp. 170–180.
60. Bhutani, V.; Toosi, F.G.; Buckley, J. Analysing the Analysers: An Investigation of Source Code Analysis Tools. *Appl. Comput. Syst.* **2024**, *29*, 98–111. [CrossRef]
61. Thomson, P. Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity. *Queue* **2021**, *19*, 29–41. [CrossRef]
62. Samarasekara, P.; Hettiarachchi, R.; De Silva, D. A Comparative Analysis of Static and Dynamic Code Analysis Techniques. *TechRxiv* **2023**. [CrossRef]
63. Li, J. Vulnerabilities mapping based on OWASP-SANS: A survey for static application security testing (SAST). *arXiv* **2020**, arXiv:2004.03216. [CrossRef]
64. Wang, T.; Qin, S.; Chow, K.P. Towards vulnerability types classification using pure self-attention: A common weakness enumeration based approach. In *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*; IEEE: New York, NY, USA, 2021; pp. 146–153.
65. Mateo Tudela, F.; Bermejo Higuera, J.R.; Bermejo Higuera, J.; Sicilia Montalvo, J.A.; Argyros, M.I. On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Appl. Sci.* **2020**, *10*, 9119. [CrossRef]
66. Li, Y.; Yao, P.; Yu, K.; Wang, C.; Ye, Y.; Li, S.; Luo, M.; Liu, Y.; Ren, K. Understanding Industry Perspectives of Static Application Security Testing (SAST) Evaluation. In *Proceedings of the ACM on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 3033–3056.
67. Marashdih, A.W.; Zaaba, Z.F.; Suwais, K. An enhanced static taint analysis approach to detect input validation vulnerability. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 682–701. [CrossRef]
68. Stoenescu, R.; Popovici, M.; Negreanu, L.; Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 314–327.
69. Goseva-Popstojanova, K.; Perhinschi, A. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.* **2015**, *68*, 18–33. [CrossRef]
70. Kang, H.J.; Aw, K.L.; Lo, D. Detecting false alarms from automatic static analysis tools: How far are we? In *Proceedings of the 44th International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 698–709.
71. Wadhams, Z.D.; Izurieta, C.; Reinhold, A.M. Barriers to using static application security testing (SAST) tools: A literature review. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 161–166.
72. Jaspan, C.; Jorde, M.; Knight, A.; Sadowski, C.; Smith, E.K.; Winter, C.; Murphy-Hill, E. Advantages and disadvantages of a monolithic repository: A case study at google. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Gothenburg, Sweden, 27 May–3 June 2018; pp. 225–234.
73. Elder, S.; Zahan, N.; Kozarev, V.; Shu, R.; Menzies, T.; Williams, L. Structuring a comprehensive software security course around the OWASP application security verification standard. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*; IEEE: New York, NY, USA, 2021; pp. 95–104.
74. Lee, S.U.; Dong, L.; Xing, Z.; Ahmed, M.E.; Avgoustakis, S. Software Security Mapping Framework: Operationalization of Security Requirements. *arXiv* **2025**, arXiv:2506.11051.
75. Kim, S.; Ryu, S. Analysis of blockchain smart contracts: Techniques and insights. In *2020 IEEE Secure Development (SecDev)*; IEEE: New York, NY, USA, 2020; pp. 65–73.
76. Durieux, T.; Ferreira, J.F.; Abreu, R.; Cruz, P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 530–541.

77. Grishchenko, I.; Maffei, M.; Schneidewind, C. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*; Springer International Publishing: Cham, Switzerland, 2018; pp. 243–269. [CrossRef]
78. Atzei, N.; Bartoletti, M.; Cimoli, T. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust 2017*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 164–186.
79. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, F.; Vechev, M. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 67–82.
80. Zhang, Q.; He, D.; Liu, H.; Shao, X. Adaptive Sliding Mode Security Control for Rotary Inverted Pendulum Against Randomly Occurring False Data Injection Attacks. *IEEE Trans. Autom. Sci. Eng.* **2025**, *22*, 17988–17999. [CrossRef]
81. Zhang, Q.; He, D.; Li, X.; Liu, H.; Shao, X. Enhanced state-constrained adaptive fuzzy exact tracking control for nonlinear strict-feedback systems. *Fuzzy Sets Syst.* **2025**, *522*, 109598. [CrossRef]
82. Panchal, P.; Hein, L.; Myschik, S.; Holzapfel, F. *A Systematic and Agile Approach to Developing DO-178C Compliant Model-Based Safety-Critical Software*; Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV: Bonn, Germany, 2024.
83. Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Monniaux, D.; Blanchet, B. The Astrée Static Analyzer. 2015. Available online: http://www.astree.ens.fr (accessed on 10 February 2026).
84. Voelter, M.; Kolb, B.; Birken, K.; Tomassetti, F.; Alff, P.; Wiart, L.; Wortmann, A.; Nordmann, A. Using language workbenches and domain-specific languages for safety-critical software development. *Softw. Syst. Model.* **2019**, *18*, 2507–2530. [CrossRef]
85. Araujo, H.; Mousavi, M.R.; Varshosaz, M. Testing, validation, and verification of robotic and autonomous systems: A systematic review. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–61. [CrossRef]
86. Giannaros, A.; Karras, A.; Theodorakopoulos, L.; Karras, C.; Kranias, P.; Schizas, N.; Kalogeratos, D.; Tsolis, D. Autonomous vehicles: Sophisticated attacks, safety issues, challenges, open topics, blockchain, and future directions. *J. Cybersecur. Priv.* **2023**, *3*, 493–543. [CrossRef]
87. Drobnjaković, F.; Subotić, P.; Urban, C. Abstract interpretation-based data leakage static analysis. *arXiv* **2022**, arXiv:2211.16073. [CrossRef]
88. Dong, Z.; Andrzejak, A.; Shao, K. Practical and accurate pinpointing of configuration errors using static analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*; IEEE: New York, NY, USA, 2015; pp. 171–180.
89. Mitchell, S.; Potash, E.; Barocas, S.; D'Amour, A.; Lum, K. Algorithmic fairness: Choices, assumptions, and definitions. *Annu. Rev. Stat. Its Appl.* **2021**, *8*, 141–163. [CrossRef]
90. Zhang, J.M.; Harman, M.; Ma, L.; Liu, Y. Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. Softw. Eng.* **2020**, *48*, 1–36. [CrossRef]
91. Barham, P.; Isard, M. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 177–183.
92. Zuo, C.; Lin, Z.; Zhang, Y. Why does your data leak? Uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2019; pp. 1296–1310.
93. Humbatova, N.; Jahangirova, G.; Bavota, G.; Riccio, V.; Stocco, A.; Tonella, P. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1110–1121.
94. Zafar, M.B.; Valera, I.; Rogriguez, M.G.; Gummadi, K.P. Fairness constraints: Mechanisms for fair classification. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, Lauderdale, FL, USA, 20–22 April 2017; pp. 962–970.
95. Dwarakanath, A.; Ahuja, M.; Sikand, S.; Rao, R.M.; Bose, R.J.C.; Dubash, N.; Podder, S. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 118–128.
96. Kafai, Y.; Biswas, G.; Hutchins, N.; Snyder, C.; Brennan, K.; Haduong, P.; DesPortes, K.; Fong, M.; Flood, V.J.; Aalst, O.W.-v.; et al. Turning bugs into learning opportunities: Understanding debugging processes, perspectives, and pedagogies. In *Proceedings of the 14th International Conference of the Learning Sciences (ICLS 2020)*; Gresalfi, M., Horn, I.S., Eds.; International Society of the Learning Sciences: Bloomington, IN, USA, 2020; pp. 374–381. Available online: https://repository.isls.org/bitstream/1/6661/1/374-381.pdf (accessed on 10 February 2026).
97. Belcastro, L.; Cantini, R.; Marozzo, F.; Orsino, A.; Talia, D.; Trunfio, P. Programming big data analysis: Principles and solutions. *J. Big Data* **2022**, *9*, 4. [CrossRef]
98. Muske, T.; Serebrenik, A. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*; IEEE: New York, NY, USA, 2016; pp. 157–166.
99. Gu, R.; Zuo, Z.; Jiang, X.; Yin, H.; Wang, Z.; Wang, L.; Wang, L.; Li, X.; Huang, Y. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 867–883. [CrossRef]

100. Guo, Z.; Tan, T.; Liu, S.; Liu, X.; Lai, W.; Yang, Y.; Li, Y.; Chen, L.; Dong, W.; Zhou, Y. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Trans. Softw. Eng.* **2023**, *49*, 5154–5188. [CrossRef]
101. Habib, A.; Pradel, M. How many of all bugs do we find? A study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 317–328.
102. Zhang, H.; Yu, Y.; Jiao, J.; Xing, E.; El Ghaoui, L.; Jordan, M. Theoretically principled trade-off between robustness and accuracy. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019.
103. Beyer, D.; Strejček, J. Improvements in software verification and witness validation: SV-COMP 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer Nature Switzerland: Cham, Switzerland, 2025; pp. 151–186. [CrossRef]
104. Amadini, R.; Gange, G.; Schachte, P.; Søndergaard, H.; Stuckey, P.J. Abstract interpretation, symbolic execution and constraints. In *Recent Developments in the Design and Implementation of Programming Languages*; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Wadern, Germany, 2020; pp. 1–7.
105. Hancock, T.M.; Gross, S.; McSpadden, J.; Kushner, L.; Milne, J.; Hacker, J.; Walsh, R.; Hornbuckle, C.; Campbell, C.; Kobayashi, K. The DARPA millimeter wave digital arrays (MIDAS) program. In Proceedings of the 2020 IEEE BiCMOS and Compound Semiconductor Integrated Circuits and Technology Symposium (BCICTS), Monterey, CA, USA, 16–19 November 2020; pp. 1–4.
106. Zampetti, F.; Scalabrino, S.; Oliveto, R.; Canfora, G.; Di Penta, M. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*; IEEE: New York, NY, USA, 2017; pp. 334–344.
107. Christakis, M.; Bird, C. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 332–343.
108. Hilton, M.; Tunnell, T.; Huang, K.; Marinov, D.; Dig, D. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2016; pp. 426–437.
109. Gopstein, D.; Iannacone, J.; Yan, Y.; DeLong, L.; Zhuang, Y.; Yeh, M.K.C.; Cappos, J. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2017; pp. 129–139.
110. Liu, R.; Wang, Y.; Xu, H.; Sun, J.; Zhang, F.; Li, P.; Guo, Z. Vul-LMGNNs: Fusing language models and online-distilled graph neural networks for code vulnerability detection. *Inf. Fusion* **2025**, *115*, 102748. [CrossRef]
111. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
112. Godefroid, P.; Levin, M.Y.; Molnar, D. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. *Queue* **2012**, *10*, 20–27. [CrossRef]
113. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **2011**, *38*, 54–72. [CrossRef]
114. Chen, M. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
115. Liu, Z.; Zhu, S.; Qin, B.; Chen, H.; Song, L. Automatically detecting and fixing concurrency bugs in go software systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 616–629.
116. Watt, C. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*; Association for Computing Machinery: New York, NY, USA, 2018; pp. 53–65.
117. Hassan, S.; Bahsoon, R. Microservices and their design trade-offs: A self-adaptive roadmap. In *2016 IEEE International Conference on Services Computing (SCC)*; IEEE: New York, NY, USA, 2016; pp. 813–818.
118. Alonso, J.; Piliszek, R.; Cankar, M. Embracing IaC through the DevSecOps philosophy: Concepts, challenges, and a reference framework. *IEEE Softw.* **2022**, *40*, 56–62. [CrossRef]


**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.