

Software Vulnerability Analysis Across Programming Language and Program Representation Landscapes: A Survey

ZHUOYUN QIAN and FANGTIAN ZHONG*, Montana State University, USA

QIN HU and YILI JIANG, Georgia State University, USA

JIAQI HUANG, University of Central Missouri, USA

MENGFEI REN, University of Alabama, US

JIGUO YU, Universitiy of Electronic Science and Technology of China, China

Modern software systems are developed in diverse programming languages and often harbor critical vulnerabilities that attackers can exploit to compromise security. These vulnerabilities have been actively targeted in real-world attacks, causing substantial harm to users and cyberinfrastructure. Since many of these flaws originate from the code itself, a variety of techniques have been proposed to detect and mitigate them prior to software deployment. However, a comprehensive comparative study that spans different programming languages, program representations, bug types, and analysis techniques is still lacking. As a result, the relationships among programming languages, abstraction levels, vulnerability types, and detection approaches remain fragmented, and the limitations and research gaps across the landscape are not clearly understood. This article aims to bridge that gap by systematically examining widely used programming languages, levels of program representation, categories of vulnerabilities, and mainstream detection techniques. The survey provides a detailed understanding of current practices in vulnerability discovery, highlighting their strengths, limitations, and distinguishing characteristics. Furthermore, it identifies persistent challenges and outlines promising directions for future research in the field of software security.

CCS Concepts: • Software Security; • System security; • Vulnerability discovery;

ACM Reference Format:

Zhuoyun Qian, Fangtian Zhong, Qin Hu, Yili Jiang, Jiaqi Huang, Mengfei Ren, and Jiguo Yu. 2025. Software Vulnerability Analysis Across Programming Language and Program Representation Landscapes: A Survey. *J. ACM* 37, 4, Article 111 (August 2025), 35 pages. <https://doi.org/XXXXXX.XXXXXXX>

1 Introduction

1.1 Background

Software vulnerability discovery is the cornerstone of securing modern software systems. It plays a critical role in software security, as vulnerabilities can result in unauthorized access, data breaches, system crashes, privilege escalation, and degraded user experience. By systematically identifying weaknesses in code, configuration, and execution environments, vulnerability discovery enables

*Corresponding author

Authors' addresses: Zhuoyun Qian, zhuoyunqian19@gmail.com; Fangtian Zhong, fangtian.zhong@montana.edu, Montana State University, Bozeman, MT, USA; Qin Hu, qhu@gsu.edu; Yili Jiang, yjiang27@gsu.edu, Georgia State University, Atlanta, Georgia, USA; Jiaqi Huang, jhuang@ucmo.edu, University of Central Missouri, Warrensburg, Missouri, USA; Mengfei Ren, mengfei.ren@uah.edu, University of Alabama, Huntsville, Alabama, US; Jiguo Yu, jiguoyu17@uestc.edu.cn, Universitiy of Electronic Science and Technology of China, Chengdu, Sichuan, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0004-5411/2025/8-ART111 \$15.00

<https://doi.org/XXXXXX.XXXXXXX>

organizations to assess and mitigate risks before they are exploited by adversaries. In offensive security research, this process reveals exploitable code paths and vulnerable inputs, facilitating the development of targeted mitigations and comprehensive security evaluations. In defensive contexts, it supports proactive protection by uncovering previously unknown vulnerabilities, guiding patch prioritization, and enhancing system resilience against attacks.

Vulnerability analysis is typically conducted at three levels: *binary*, *intermediate representation (IR)*, and *source code*, with each offering unique advantages and facing specific challenges. Binary-level analysis operates directly on compiled executables, making it indispensable for assessing closed-source software and deployed applications. Techniques such as binary instrumentation, fuzzing, and taint analysis are commonly used to detect memory corruption, missing-check vulnerabilities, and side-channel leaks. However, this level suffers from limited semantic information, obfuscation challenges, and difficulties in reconstructing accurate control and data flow. IR-level analysis is performed on representations like low-level virtual machine (LLVM) IR, bytecode, assembly, and abstract syntax trees (ASTs), striking a balance between low-level execution details and high-level code semantics. Techniques such as flow-sensitive analysis, data-flow analysis, and symbolic execution are employed to detect concurrency bugs, missing-check vulnerabilities, and memory safety issues. While IR-level analysis benefits from platform independence and better analyzability than binaries, it may lack complete context in the source code. Source-code level analysis focuses on human-readable source code, allowing early detection of logic errors, insecure coding patterns, and API misuse. Common techniques include static analysis and machine learning-based vulnerability detection. This level provides rich syntactic information, enabling precise reasoning about code behavior. However, it may be limited in capturing vulnerabilities that manifest only during execution or after compiler transformations. Furthermore, the programming language itself significantly influences the nature of vulnerabilities and the detection strategies employed. For example, C/C++ programs are particularly prone to memory safety issues, performance bottlenecks, concurrency bugs, and logic flaws. In contrast, JavaScript frequently exhibits logic bugs, missing-check vulnerabilities, and concurrency-related issues due to its dynamic and event-driven nature. As such, a comprehensive understanding of software vulnerabilities requires cross-language, cross-representation, and multi-bug-type perspectives.

1.2 Classification by Bugs

The classification of software bugs, as illustrated in Fig. 1, spans seven major categories: performance bugs, memory safety bugs, concurrency bugs, missing-check bugs, logic bugs, resource management bugs, and side channel vulnerabilities, each representing distinct causes of system instability or inefficiency. Performance bugs occur when software exhibits slow execution or excessive resource use without functional errors, often due to synchronization issues, configuration problems, or inefficient algorithms, frequently observed in C/C++, Java, and Android. Memory safety bugs arise from improper memory operations like use-after-free, buffer overflows, and invalid access, which may lead to crashes or data corruption, especially in C/C++ and Rust. Concurrency bugs result from incorrect handling of multiple threads or processes, leading to race conditions, atomicity violations, or inconsistent states—common across C/C++, Java, Android, and JavaScript. Missing-check bugs stem from the failure to validate inputs, permissions, or security conditions, enabling vulnerabilities such as access control violations and component hijacking across C/C++, Java, Android, PHP, and JavaScript. Logic bugs are caused by flawed control or computation logic, including API misuse, semantic errors, and improper comparisons, impacting correctness in languages like C/C++, Java, JavaScript, Python, and PHP. Resource management bugs involve the incorrect handling or accounting of system resources (e.g., memory), leading to leaks or data loss in Android, C/C++, and Python. Side-channel vulnerabilities are security flaws where attackers infer

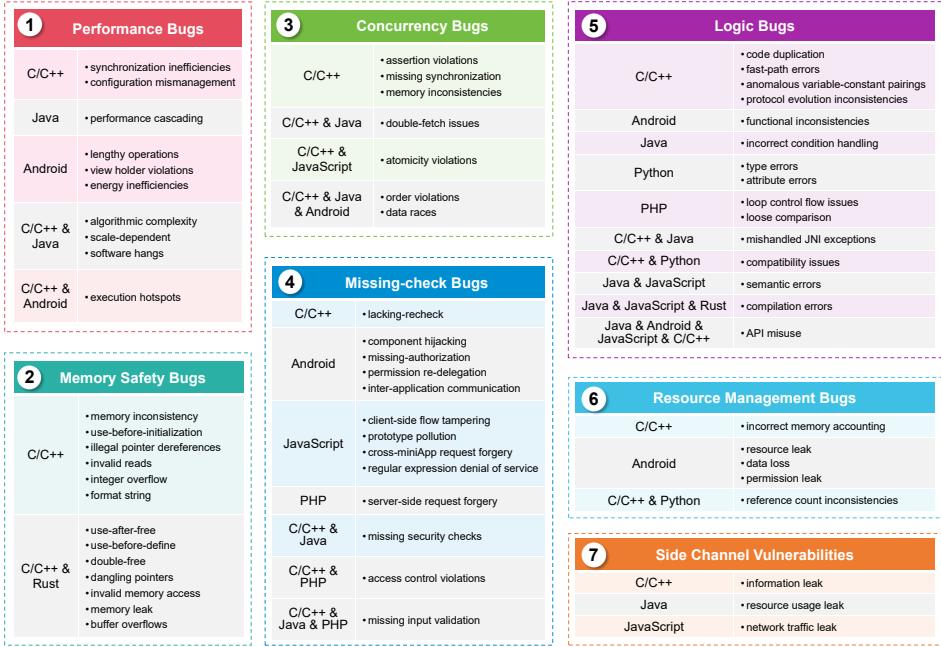


Fig. 1. Classification of Bugs

sensitive information through indirect behaviors such as timing, cache access, or traffic patterns, with examples like SSL/TLS leaks in C/C++, resource usage leaks in Java, and network traffic leaks in JavaScript.

1.3 Related Work

Prior surveys have explored specific aspects of software vulnerability detection, often focusing on narrow domains or limited bug types. For instance, [56] evaluates static and dynamic analysis techniques for detecting SQL injection (SQLi) and cross-site scripting (XSS) vulnerabilities, while [129] integrates rule-based filtering with machine learning to detect attacks in SS7 telecommunication networks. Fuzzing has been extensively studied in [168], which categorizes modern fuzzing strategies, including machine learning-assisted fuzzing, constraint-solving integration, and directed fuzzing for improving code coverage and vulnerability discovery efficiency. [53] provides an overview of automated techniques for vulnerability detection, exploitation, and patching, emphasizing the growing role of machine learning in software security automation. Similarly, [174] investigates the mining of patch commits to identify silent vulnerabilities in open-source software, leveraging NLP and graph-based machine learning approaches. Machine learning has indeed become a transformative force in vulnerability detection. [54, 172] surveys deep learning-based techniques, analyzing models such as CNNs, RNNs, and Transformers for software vulnerability detection. [61] focuses specifically on LSTM-based models for XSS detection. In contrast to these focused studies without generalizability and reusability of findings, our survey provides a broad, technique-driven overview of software vulnerabilities across multiple programming languages, program representations, and bug types. This holistic perspective fills an important gap in the current literature by integrating language diversity, representation levels, and vulnerability taxonomies

into a single cohesive framework. The differences between existing surveys and ours are presented in Table 1.

Table 1. Comparison of Surveys

Paper	Multi-Programming Languages	Multi-Program Representations	Multi-Bug Types	Multi-Analysis Methods
[56]	✓	✗	✗	✓
[129]	✗	✗	✗	✗
[168]	✗	✗	✓	✗
[53]	✗	✗	✓	✓
[174]	✗	✗	✓	✓
[172]	✗	✗	✓	✗
[61]	✗	✗	✗	✗
Ours	✓	✓	✓	✓

1.4 Contribution of the Survey

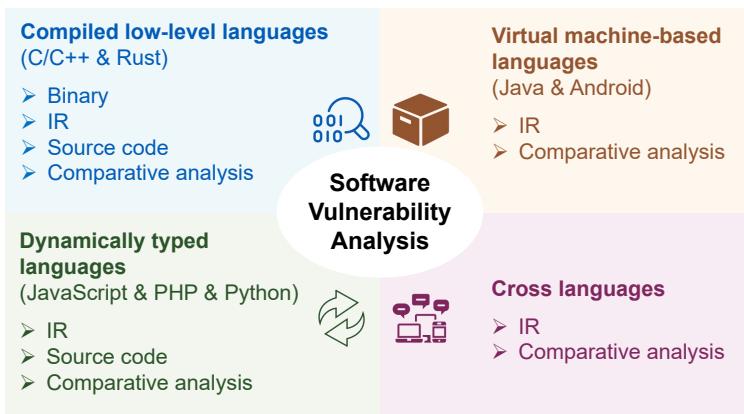


Fig. 2. Overview of The Survey

As shown in Fig. 2, our survey identifies the eight most vulnerability-prone programming languages, categorizing them into four groups: compiled low-level languages, virtual machine-based languages, dynamically typed languages, and cross-languages. The key contributions of this survey are outlined as follows:

- (1) We provide a modern, technique-driven overview of software vulnerabilities across various programming languages, program representations, and bug types, with a strong focus on detection methods, including static, dynamic, hybrid, and machine learning approaches.
- (2) We summarize the causes of vulnerabilities and categorize them into major bug types, offering a broad and unified view of existing software flaws. Our survey highlights practical detection techniques to each bug type and introduces a language-aware categorization framework.
- (3) We provide an in-depth discussion of both well-established and underexplored research directions in this domain. We conclude by identifying future opportunities in advancing vulnerability detection techniques.
- (4) We automate the retrieval of relevant publications on software vulnerabilities from major academic databases, including IEEE, ACM, and Google Scholar, to ensure the comprehensiveness and reproducibility of our survey. For each reviewed paper, we verify the availability of datasets and code, categorize the datasets into private, partially public, and public, and

compile both the collected metadata and our automation scripts in a public repository: <https://github.com/fangtian-zhong/SoftwareSecurity.git>.

1.5 Paper Organization

The remainder of this article is organized as follows. Sections 2–4 discuss bug detection techniques at the binary, IR, and source code levels along with a comparative analysis of static, dynamic, hybrid, and machine learning-based approaches for compiled, low-level languages such as C/C++ and Rust. Section 5 focuses on bug detection at the IR level for virtual machine-based languages, specifically Java and Android, including a corresponding comparison of detection methodologies. Sections 6–7 examine bug detection at both the IR and source code levels for dynamically typed languages (*i.e.*, JavaScript, PHP and Python), with comparative insights into their detection techniques. Section 8 explores bug detection at the IR level across multiple languages, again providing a comparison of different analysis approaches. Finally, Section 9 presents key challenges and open research gaps, followed by the conclusion in Section 10.

2 C/C++ & Rust Bug Detection at Binary Level

2.1 Logic bugs

Logic bugs at the binary level include compatibility issues and semantic errors, detected through static and dynamic analysis. **Static analysis** identifies compatibility issues in external libraries via pattern matching, flagging modified interfaces while surrounding code remains unchanged [55]. It uses ABI-Tracker to detect backward and forward application binary interface (ABI) changes, such as added, removed, or altered interfaces. If an application relies on an incompatible library version, it is flagged as a compatibility issue. **Dynamic analysis** uses fuzzing to detect semantic inconsistencies in software verification tools, including code coverage tools [157], SMT solvers [93], model checkers [163], compilers [39], and type checkers [28]. For code coverage tools like gcov and llvm-cov, fuzzing with Csmith generates random C programs, detecting discrepancies in execution counts and line coverage. In SMT solvers, fuzzing breaks seed formulas into sub-formulas, assigns random truth values, mutates them, and checks if solvers misclassify satisfiable instances. For C model checkers, fuzzing mutates seed programs, inserts counter variables, and generates program-property pairs to detect inconsistencies. In Simulink compilers, fuzzing applies assertion injection and block mutation, selectively mutating unexecuted blocks using MCMC sampling while maintaining functional equivalence. Discrepancies between original and mutated outputs indicate compiler errors. For type checkers, fuzzing generates well-typed programs using constraint logic programming and type rule encoding, then mutates them to analyze checker behavior. It detects precision bugs (valid programs rejected), soundness issues (ill-typed programs accepted), and consistency bugs (conflicting results). Fuzzing, combined with taint analysis and field-sensitive data-flow analysis, detects configuration-related logic errors by identifying the internal and external effects[136]. It dynamically mutates configuration options, executes the program under startup-loaded and modified configurations, and compares their output to assess external effect. To track internal effects, taint analysis and field-sensitive data-flow analysis monitor global configuration variable values.

2.2 Missing-check bugs

Missing-check bugs at the binary level include missing security checks, detected through hybrid analysis. **Hybrid analysis:** It integrates intra-procedural data-flow analysis with fuzzing to detect access control violations [49]. Kernel source code is compiled into LLVM IR, constructing DFGs and performing interface analysis to identify permission checks. Intra-procedural data-flow analysis tracks privileged function calls, while constraint-based invariant analysis verifies permission

conditions. Greybox fuzzing with distilled seeds is applied, using a clustering-based seed distillation technique to prioritize test cases likely to expose vulnerabilities.

2.3 Concurrency bugs

Concurrency bugs at the binary level include order violations, data races, atomicity violations, missing synchronization, memory inconsistencies, double-fetch issues, and assertion violations, detected through dynamic and hybrid analysis. **Dynamic Analysis:** *Execution monitoring* detects concurrency-induced order violations by recording thread interactions and tracking memory operations, identifying null pointer dereferences (NPD), double pointers (DP), uninitialized reads, and buffer overflows [166]. It validates buggy interleavings by instrumenting and replaying them while also detecting non-interleaved concurrency bugs through different execution models. AsyncAsSync replaces asynchronous calls with normal procedure calls to simulate sequential execution, while AsyncAsEvents randomizes asynchronous execution order. AsyncGeneral combines both methods, first detecting non-interleaved concurrency bugs by collecting code traces from AsyncAsSync and comparing them with the original program, and then uncovering synchronization errors with AsyncGeneral [57]. *Controlled execution perturbation* detects exchangeable memory operations by forcing execution scheduling to identify UAF, NPD, and DF bugs. It checks if a free(p) operation and all accesses to p are exchangeable, profiles memory access traces to detect unserializable interleavings, and confirms atomicity violations by injecting delays to force rare interleavings leading to crashes or incorrect outputs [10, 102, 159]. Sequential tests are converted into concurrent client requests to explore interleavings, detecting bugs through output discrepancies [31]. *Fuzzing* generates high-coverage inputs and guides execution toward risky interleavings. It instruments memory and persistency instructions with LLVM and applies an AFL++-based mutator to prioritize read-after-write interleavings, exposing non-persisted data reads and durable side effects while monitoring synchronization variables for inconsistencies. Memory aliasing and branch coverage refine interleaving exploration[19]. Fuzzing also generates concurrent tests ensuring writer-reader pairs access the same memory, exposing data races, order violations, and atomicity violations [37]. It can be combined with side-channel analysis to detect double-fetch bugs, where memory is fetched multiple times without synchronization, leading to race conditions [112]. Flush+Reload flushes memory locations from the CPU cache and measures reload times to detect multiple fetches. Syscall fuzzing identifies privilege escalation cases when memory is modified between fetches. **Hybrid analysis:** Hybrid techniques integrate data-flow analysis, static slicing, fuzzing, execution monitoring, record-and-replay, and controlled execution perturbation. One approach combines *fuzzing* with *data-flow analysis* to detect concurrency relationships triggering buffer overflows, DF, or UAF errors [78]. It constructs data-flow graphs to track shared variable accesses across threads and manipulates execution order to expose concurrency-induced errors by fuzzing. Another approach integrates *static slicing* with *execution-trace analysis* to track shared-memory reads contributing to infinite loops, assertion violations, and memory errors [165]. It traces write operations one step back in the data dependency chain from read instructions and replays suspicious interleavings to confirm concurrency bugs. *Record-and-replay* is combined with *controlled execution perturbation* to detect data races [160]. Static analysis identifies locks based on intrinsic characteristics, and execution traces are replayed with a proof-of-concept (PoC) input. Shared variables accessed during execution are analyzed, and well-synchronized accesses are eliminated based on detected locks. Race conditions are then actively forced through thread scheduling perturbation.

2.4 Memory Safety Bugs

Memory safety bugs at the binary level include memory inconsistency, buffer-overflow, use-after-free (UAF) , use-before-define (UBD), double-free (DF), dangling pointers (DP), and invalid memory access errors, detected through dynamic analysis, hybrid analysis, and machine learning. **DYNAMIC**

analysis: *Failure injection* introduces controlled faults to evaluate memory state and recoverability [36, 82]. It tracks read/write operations, logging memory addresses before and after failure to detect memory inconsistencies. A failure point tree maps faults to call stacks, injecting faults at unvisited points and re-executing the program to test recovery. *Pattern matching* detects memory inconsistencies by tracking memory stores, cache flushes, and fences, mapping memory locations to operations and applying predefined rules (e.g., missing durability guarantees, multiple overwrites, ordering violations) [29]. *Fuzzing* generates test cases and apply backward taint analysis to track memory accesses [132]. Buffer overflows are identified by reconstructing valid memory layouts from successful executions and detecting boundary violations in failed executions, highlighting discrepancies between memory states. It detects UAF, UBD, DF, DP, and invalid memory access errors in library APIs by generating test cases that trigger undefined behaviors [124, 125]. It synthesizes valid test cases based on API typing specifications (input parameters, return types), maps raw byte data to API parameters, and inserts API call sequences. API specifications are refined by adjusting polymorphic function signatures to mitigate recurring compiler errors, and a coverage- and type-guided search algorithm optimizes API sequence generation. The test cases are executed with generated inputs, and undecided behaviors are flagged as bugs. **Hybrid analysis** This approach combines typestate, data flow analysis and taint analysis with fuzzing, runtime execution and record-and-replay to detect UAF and UBD bugs [18, 131]. Typestate-guided *fuzzing* constructs operation sequences such as $\langle \text{malloc} \rightarrow \text{free} \rightarrow \text{use} \rangle$ from *typestate analysis*, instruments the programs, and adaptively mutates inputs using flow-sensitive information to maximize sequences coverage. *Static taint analysis* applies track integer operations affecting memory allocations on GIMPLE IR, inserts overflow checks and guarding code at memory sites [123]. Memory allocation behavior is monitored at runtime to flag inputs causing overflows. *Record-and-replay* with *data-flow analysis* logs system calls and execution traces, identifying anomalies via crash detection and syscall interposition. Anomalous traces undergo dynamic binary translation, instrumenting the program to log runtime memory access. The instrumented program is replayed to extract data flow and compare it with a precomputed data flow graph, detecting uninitialized memory access. **Machine learning:** *Deep reinforcement learning* enhances *fuzzing* to detect buffer overflows via *concolic execution* [52]. Coverage-guided fuzzing explores programs, marking test cases that reach new basic blocks. These cases are prioritized by reinforcement learning with a Deep Q-network for unexplored path execution, with constraints solved using an SMT solver to guide fuzzing until a crash occurs.

2.5 Performance Bugs

Performance bugs at the binary level include synchronization, algorithm complexity, configuration, scale-dependent, and timeout-related issues, detected through dynamic analysis and machine learning. **Dynamic analysis:** *Synchronization analysis* detects inefficiencies in multithreaded applications by intercepting pthread synchronization primitives at runtime, tracking lock usage and contention points to identify improper primitives, granularity issues, over-synchronization, asymmetric contention, and load imbalance [2]. Resource-usage-guided *fuzzing* monitors functions, basic blocks, and edge execution in the control flow graph while tracking resource usage [104]. It mutates inputs using random and dictionary-based modifications, prioritizing those that consume excessive resources or cause slowdowns, exposing algorithmic complexity vulnerabilities. **Machine learning:** Configuration, scale-dependent, and timeout-related issues are detected through natural language processing (NLP), supervised learning, and unsupervised anomaly detection. *NLP* extracts configuration rules from documentation and compares runtime performance against expected outcomes under different configurations to identify anomalies [43]. Supervised anomaly detection trains a *regression model* on control features (e.g., input properties, configuration parameters) and

observational features (e.g., syscall counts, libc function calls) from small-scale, bug-free runs to predict large-scale behaviors [169]. Performance anomalies are detected by comparing predicted and observed system behavior at scale. Unsupervised anomaly detection applies a *Self-Organizing Maps model* to learn execution patterns from timeout-related system call traces, analyzing frequencies to identify anomalies causing server hangs or slowdowns [44].

2.6 Side-channel Vulnerability

Side-channel vulnerabilities at the binary level involve information leaks, detected through dynamic analysis and machine learning. **Dynamic analysis:** *Execution trace analysis* identifies unintended variations in execution [12, 147]. For SSL/TLS leaks, it compares execution differences between conformant and non-conformant packet sequences. Modified ClientKeyExchange messages and CBC-encrypted data are injected, and the target program executes while constructing dynamic control-flow graphs. Vulnerabilities are reported by comparing basic block traces. For TCP leaks, dynamic analysis simulates correct and incorrect secret guesses, sending identical packet sequences in both cases. A vulnerability is flagged if responses differ, indicating behavioral discrepancies that leak sensitive information. **Machine learning:** This method detects side-channel vulnerabilities by analyzing network traffic patterns and user interactions [15]. It simulates interactions, logs network traces, and constructs a finite-state machine where each state represents a saved DOM state, and transitions capture network behaviors. *leak quantifier* applies distance metrics and the Fisher criterion to classify traces by user actions, clustering them via a *multi-class classifier*. Unknown traces are assigned to the closest centroid using Hamiltonian distance. If traces for different user actions are easily distinguishable, a side-channel vulnerability is reported.

2.7 Multiple Types of Bugs

Different types of bugs detected simultaneously at the binary level include logic bugs, memory safety bugs, and missing-check bugs, identified through dynamic analysis, hybrid analysis, and machine learning. **Dynamic analysis:** Fuzzing and dynamic symbolic execution detect multiple bug types. Hydra [63] mutates OS images and syscalls by *fuzzing*, applies four dedicated checkers to detect bugs, validates issues via VM-based re-execution, and minimizes syscall sequences through delta debugging to produce concise PoC reports. Cerebro [75] optimizes seed selection by scoring inputs based on execution time, file size, edge coverage, and trace complexity, prioritizing high-impact seeds for mutation. It allocates execution cycles and mutates seeds to trigger crashes. VulScope [23] collects execution traces from a reference with known vulnerabilities and a target program version, mutating inputs to explore execution detours and prioritize paths with high trace similarity to trigger crashes. Fuzzing also aids bug localization. VulnLoc [113] tracks conditional statements in a vulnerable binary, generating a focused test suite through iterative input mutation, sensitivity map inference, and execution tracing. It computes sufficiency and necessity scores for conditional statements, ranks vulnerable locations, and maps binary instructions to source code statements using objdump. Symbolic execution explores complex paths where fuzzing struggles. BitFuzz [9] employs stitched *dynamic symbolic execution*, identifying encoding functions and their inverses via *trace-based dependency analysis*, bypassing difficult constraints and focusing on solvable execution path constraints. It re-stitches the output using encoding functions or their inverses, generating valid inputs to uncover vulnerabilities.

Hybrid analysis: it enhances fuzzing by optimizing input generation and prioritizing execution paths. One approach extracts sensitivity, complexity, depth, and rarity metrics from basic blocks via *intra-procedural analysis*, assigns weight values, and injects trampolines to track control-flow edges [116]. *Fuzzing* allocates execution energy to rare or high-weight regions, mutating inputs to maximize code coverage and trigger crashes. Another applies *path-sensitive AST analysis* and reflection-based techniques to generate semantically valid JavaScript seed inputs, mutating them to

explore JavaScript engines [45]. It modifies seeds by altering constants, inserting member operations, and injecting function calls, then executes them to detect bugs. Seeds that improve coverage without crashes are added for iterative fuzzing. A final approach extracts API descriptions from ASTs and applies fuzzing to iteratively generate Python applications around APIs [73]. It refines these applications based on CFG coverage feedback, executing them within a time-constrained budget using type-aware mutations. During execution, it monitors for unexpected behaviors, testing the Python runtime for vulnerabilities. DSFuzz [83] uncovers deep-state bugs by constructing a state dependency graph using *def-use analysis*, *inter-procedural data-flow analysis*, and *dynamic data-flow analysis* to capture direct/indirect control dependencies. It identifies deep states dependent on prior states and determines dependent input bytes for each state transition via *dynamic taint tracking*. Progressive micro-directed fuzzing modifies inputs to transition from the current state to target deep states, flagging mutated seeds causing crashes as bugs. 1dFuzz [154] introduces directed differential fuzzing to reproduce 1-day vulnerabilities. It disassembles patched and unpatched binaries, generates call graphs, and identifies patch locations by comparing trailing call sequence (TCS) features. Directed differential fuzzing then generates PoC inputs for these locations, refining fuzzing with a TCS-based distance metric to improve accuracy. RPG [150] detects logic and memory safety bugs in Rust libraries by generating fuzz targets based on API information extracted from rustdoc. It builds an API dependency graph using static analysis, applies Pool-based Sequence Generation to prioritize unsafe code while maintaining type consistency, and synthesizes fuzz targets by linking the library with AFL. **Machine learning:** ML improves fuzzing and symbolic execution by guiding execution-path exploration, optimizing input generation, and refining mutation strategies. CAMFuzz [114] trains a CNN on input seeds and CFGs to predict input bytes influencing unexplored paths and optimize *fuzzing* mutation strategies. It extracts constants such as file signatures and loop counts to refine mutations, improving coverage and bug detection. SyML [111] enhances *dynamic symbolic execution* by predicting high-risk execution paths. It extracts execution trace features (e.g., memory accesses, function complexity, system calls), trains a supervised model, and guides symbolic execution toward paths most likely to contain bugs.

2.8 Conclusion of Bug Detection at Binary Level

Table 2. Comparison of Bug Detection Techniques for C/C++ & Rust

Paper	Techniques	Bugs	Datasets	Accuracy&Overhead
[55]	static	C/C++, logic	public	-
[157],[93],[163],[39],[28],[136]	dynamic	C/C++, logic	private, public	-,-,-,68.8%,-,75%
[49]	hybrid	C/C++, missing-check	public	28.9% TP
[166],[57],[10],[102],[31],[159],[19],[37],[112]	dynamic	C/C++, concurrency	private ,public	90%,-,66.7%,-,-,-,97 %&0.8%,
[78],[165],[160]	hybrid	C/C++, concurrency	private,public	115.5 % -90.9% & 662.6% ,63.3 %
[82],[132],[36],[29]	dynamic	C/C++, memory safety	public	-,-,96% ,117.85% ,30.6%
[131],[123],[18]	hybrid	C/C++, memory safety	private, partially public	-,-,99.99% & 0.69%, 0.35%
[52]	ML	C/C++, memory safety	public	-
[125],[124]	dynamic	Rust, memory safety	private, public	-,-
[2],[104]	dynamic	C/C++, performance	private, public	2.3 %,-
[43],[169],[44]	ML	C/C++, performance	private, public	70.5%, 92.5% & 114.4%, 94.7% & 1%
[147],[12]	dynamic	C/C++, side channel	public	-,-
[15]	ML	C/C++, side channel	private	61.2%
[63],[75],[23],[113],[9]	dynamic	C/C++, logic, memory safety	private, public	79.6%,-,-,88.4%,-
[116],[45]	hyrbid	C/C++, logic, memory safety	partially public,public	-,-
[73]	hyrbid	C/C++, logic, memory safety, performance	public	-
[150]	hyrbid	Rust, logic, memory safety	public	-
[114]	ML	C/C++, unlisted	private	82%
[111]	ML	C/C++, logic, memory safety	public	78.6%
[83],[154]	hybrid	C/C++,unlisted	private,public	-,-

As shown in Table 2, static analysis is less common and only remains effective for detecting logic bugs such as ABI incompatibilities. Dynamic analysis is the dominant approach for binary-level bug detection, widely applied across logic, concurrency, memory safety, performance, and side-channel vulnerabilities. Tools like Csmith-based fuzzing [157], [163], and symbolic tracing

tools [136] detect semantic inconsistencies in verification tools, SMT solvers, and compilers with up to 75% precision. For concurrency bugs, execution perturbation and interleaving-aware fuzzing (e.g., [37]) reveal data races, atomicity violations, and synchronization errors, with tools like [112] achieving 97% accuracy using cache-based side-channel monitoring. Memory safety bugs, including UAF, DF, and buffer overflows, are tackled using fuzzing, fault injection, and taint tracking. For instance, typestate-guided fuzzing tools [131], [18] report up to 99.99% accuracy. Rust-specific tools such as [125] and [124] synthesize API sequences to expose undefined behaviors in libraries. Hybrid analysis combines static slicing, execution monitoring, and data-flow tracking to improve precision and bug coverage. Notable tools include [160] for concurrency bug reproduction and [83] for detecting deep-state multi-type vulnerabilities. ML-based approaches are increasingly used to enhance fuzzing and symbolic execution. DRL-based methods like DR [52] improve path exploration and crash detection, while [114] and [111] guide fuzzing using control-flow features and execution path risk prediction, achieving up to 82% accuracy.

3 C/C++ & Rust Bug Detection at IR Level

3.1 Logic Bugs

Logic bugs at the IR level include code duplication and semantic bugs, detected through static analysis. **Static analysis:** Buggy code clones are identified by extracting ASTs, transforming them into n-dimensional characteristic vectors, and clustering similar fragments using *locality-sensitive hashing* [33]. Structural differences between clones are analyzed to detect syntactic inconsistencies introduced during modifications. Recurring semantic bugs are identified by constructing data dependency graphs from vulnerable traces and applying *taint analysis* to extract feature vectors encoding low-level properties (e.g., operator usage frequencies) and high-level behavioral signatures (e.g., buffer overflow conditions) [59]. Vulnerability signatures are stored in a database, and new traces are flagged as potential vulnerabilities if their cosine similarity score exceeds 0.85.

3.2 Missing-check Bugs

Missing-check bugs detected at the IR level include missing security checks, lacking-recheck, missing input validation, detected through static and machine learning. **Static analysis:** This method constructs CGs, CFGs, and DFGs and applies data-flow, intra/inter-procedural, flow-sensitive, field-sensitive, taint, and alias analysis. One approach detects missing security checks by compiling kernel source code into LLVM IR, generating CGs, CFGs, and DFGs, and identifying security checks through conditional branches leading to error-handling statements [141]. A bidirectional *intra/inter-procedural data-flow analysis* tracks security-sensitive operations to ensure security enforcement on all execution paths. Another approach detects lacking-recheck bugs by constructing CGs, CFGs, and DFGs from LLVM IR and applying error-code inference to identify security checks [138]. CFG traversal tracks critical variables using *taint tracking* and *alias analysis* to form check-use chains, detecting modifications after the initial check via forward *interprocedural data-flow analysis*. **Machine learning:** this method integrates with taint analysis to analyze code dependencies and behaviors [152]. ASTs are extracted to identify sources, sinks, conditions, assignments, and API symbols. A bag-of-words model clusters functions with similar API symbols, while *taint analysis* tracks data flow from security-sensitive sources using dependency graphs. Normalized tainted conditions are embedded into vector representations, and a *normality model*, trained on correctly validated functions, detects anomalies by measuring deviation from expected patterns. Functions with high anomaly scores are flagged as missing input validation checks.

3.3 Concurrency Bugs

Concurrency bugs at the IR level include data race, order violations, synchronization violations, and atomicity issues, detected through static and hybrid analysis. **Static analysis:** It constructs

CGs, CFGs, and DFGs and applies flow-sensitive, inter-procedural, alias, and data-flow analysis. One approach detects data races by converting source code into Boogie IR, constructing control flow graphs, and applying *inter-procedural analysis* to identify available locks [26]. *Symbolic verification* analyzes concurrent entry points, tracking locksets on shared memory using *data-flow analysis*. A data race is flagged if two locksets do not intersect. Another approach detects sleep-in-atomic-context bugs by compiling the Linux kernel into LLVM IR, performing connection-based *alias analysis* to construct a full kernel call graph, and applying summary-based inter-procedural analysis to track atomic code paths leading to sleep function calls [5]. Path-checking is used to reduce false positives. Other approaches enforce synchronization constraints and memory access ordering. *Synchronization constraints* ensure correct interrupt-related synchronization rules using (Precondition, Postcondition) annotations [126]. It extracts interrupt-related statements from comments, assertions, and macros, propagates annotations across the call graph, and verifies whether root functions have conflicting preconditions and postconditions. Memory access ordering ensures correct execution order using *memory barriers* [68]. It extracts barriers from source code, constructs control-flow graphs, and classifies them into unpaired barriers, paired barriers for ordering violations, and multi-paired barriers for correct multi-threaded execution. Violations are flagged for misplaced memory accesses, incorrect barrier types, and redundant reads. **Hybrid analysis:** It collects execution traces and applies flow-sensitive and alias analysis to detect concurrency bugs. One approach tracks *pointer aliasing* across execution paths and mutates execution order to predict concurrency violations [41]. Programs are instrumented to collect event traces and LLVM IR slices, mapping each event to its corresponding instruction. Pointer flows are analyzed to detect conflicting event pairs, and sequential orderings are mutated to identify NPD, UPU, UAF, and DF due to order violations. Another approach integrates *flow analysis* with execution traces, collecting control flow traces from client programs and performing core analysis on the server when crashes or deadlocks occur [60]. It applies *points-to analysis*, type-based ranking, bug pattern classification, and statistical diagnosis to detect order violations and atomicity issues. Failure traces are compared with successful execution traces near failure points.

3.4 Memory safety bugs

Memory safety bugs at the IR level include use-before-initialization (UBI), memory leaks, illegal pointer dereferences, buffer overflows, invalid reads, UAF, integer overflow and format string vulnerabilities, detected through static, dynamic, hybrid analysis, and machine learning. **Static analysis:** It verifies memory integrity by converting code into IR, constructing CGs, CFGs and DFGs, and applying flow-sensitive, field-sensitive, alias, intra/inter-procedural, taint and data-flow analysis along with symbolic execution. UbiTect [161] constructs CGs, CFGs, and DFGs from LLVM IR and performs *flow- and field-sensitive intra-procedural points-to and alias analysis* to track variables and memory objects. It propagates function summaries across call sites and uses a type qualifier inference system to verify If variables or memory object fields in the caller functions exhibit weaker-than-expected qualifiers and employs under-constrained *symbolic execution* to validate UBI bugs. NDI [170] extracts security-critical variables using *intra-procedural field-sensitive data-flow analysis*, collects inconsistent path pairs leading to security state inconsistencies, and applies under-constrained *symbolic execution* to detect discrepancies at merging points. *Inter-procedural, flow-insensitive, and field-insensitive* program slicing at the function level refine the analysis scope along the paths from the merging point to critical variable usage. It then performs *inter-procedural analysis* to determine whether inconsistencies persist. Cod [40] resolves heap aliasing issues and distinguishes symbolic heap locations to detect memory corruption. It constructs CGs and CFGs, applies *intra-procedural analysis* to initialize function states, and generates function summaries capturing memory operations. *Inter-procedural analysis* propagates memory effects

across functions, inlining summaries when disjointness assumptions fail. *SMT solving* identifies memory safety violations. Rupair [50] detects unchecked arithmetic, unsafe memory access patterns, and lossy integer casting by converting source code into ASTs and MIR to collect information on risky operations such as raw pointer dereferences and direct memory manipulations. Using *intra-procedural data-flow and alias analysis*, Rupair identifies potential buffer overflow candidates by tracking live variables in unsafe blocks and checking their definition sites for matches with overflow patterns. It translates candidate overflows into symbolic constraints and uses *SMT solving* to generate concrete counterexamples to confirm buffer overflow occurrences. SafeDrop [21] applies *taint analysis*, compiling programs into MIR, constructing CGs, CFGs, and DFGs, and simplifying CFGs using the Tarjan algorithm to prioritize bug-prone paths. *Flow- and field-sensitive alias analysis* tracks composite-type variables, while *inter-procedural analysis* remains context-insensitive for efficiency. It caches alias sets for function arguments and return values, scans for unsafe patterns, and propagates taint through alias sets to detect memory corruption.

Dynamic analysis: it detects memory safety bugs through execution tracing. Safe Sulong [110] interprets LLVM IR in Java to detect out-of-bounds accesses, UAF errors, and invalid memory deallocation in C programs. It compiles source code into LLVM IR using Clang, interprets it with Truffle to generate execution traces, and compiles frequently executed ASTs to machine code. Execution is terminated upon detecting a memory violation, and an error is reported. **Hybrid analysis:** it integrates symbolic execution with execution tracing to verify memory integrity. Bunkerbuster [151] reconstructs symbolic program state traces from PT-enabled kernel drivers, filters potentially buggy traces, and applies *symbolic execution* on VEX IR to symbolize memory snapshots. By analyzing executed basic blocks, it detects buffer overflows, UAF errors, and format string vulnerabilities. Symbolic root cause analysis then identifies the precise origin of memory safety violations. **Machine learning:** It enhances inter-procedural analysis for detecting memory-related bugs at the statement level [11]. It extracts Program Dependence Graphs (PDGs) from source code, incorporating control-flow and data-flow graphs to capture execution dependencies. PDGs are extended with call relations and return values from call graphs for *inter-procedural analysis*. Backward and forward program slicing from key points, such as system API calls and pointer variables, generates focused program slices. Each statement node is vectorized using Doc2Vec, and a flow-sensitive graph neural network model is trained to identify bug patterns from these graph-based representations.

3.5 Resource Management Bugs

Resource management bugs at the IR level include inconsistent reference count and memory accounting issues. **Static analysis:** One approach employs inter-procedural analysis, field-sensitive analysis, and symbolic execution to detect inconsistent reference count bugs [94]. It ensures reference count values remain non-negative and eventually reach zero by compiling source code into Linux bitcode, constructing CGs and CFGs, and applying *summary-based inter-procedural analysis* to track modifications. The *Z3 solver* enforces constraints on function paths, flagging bugs when execution paths with identical constraints show conflicting reference count updates. Another approach detects memory accounting bugs caused by improper tracking or deallocation of allocated memory, leading to system memory counter imbalances [155]. It compiles the Linux kernel into LLVM bitcode, constructs a kernel call graph, and identifies memory accounting interfaces by matching counter variables within kernel IR instructions. *Data-flow analysis* maps allocation and free operations to corresponding accounting structures, while *inter-procedural bitwise data-flow tracing* analyzes accounting flag conditions to handle conditional memory tracking. Bugs are identified by detecting failures in updating system counters during allocation or deallocation.

3.6 Multiple Types of Bugs

Different types of bugs detected at the IR level include logic bugs, memory safety bugs, missing-check bugs, resource management bugs and side-channel vulnerabilities. Detection techniques rely on static analysis, and machine learning. **Static analysis:** it detects various bugs by constructing CFGs and DFGs and applying taint analysis, data-flow analysis, and signature matching. One approach applies *signature matching* to detect unpatched binaries [148], disassembling binaries into assembly code, generating CFGs, and comparing vulnerable and patched binaries to extract patch signatures. If a function in the target binary resembles the vulnerable version more than the patched one, it is flagged as unpatched. Another approach combines taint data-flow analysis and pattern matching to detect panic safety bugs, higher-order safety invariant violations, and concurrency issues [4]. Rust’s HIR is analyzed for function declarations, trait implementations, and unsafe blocks, while MIR enables *taint tracking* to detect lifetime violations and unsafe function assumptions. It also flags incorrect Send/Sync implementations by analyzing type definitions and API signatures for non-thread-safe methods. YUGA [98] detects UAF and data races caused by incorrect lifetime annotations in Rust. It converts source code into HIR to extract structure definitions, function signatures, and lifetime constraints, infers borrow times, and uses pattern-based detection to flag lifetime inconsistencies. *Flow- and field-sensitive intra-procedural alias analysis* on MIR confirms whether flagged value pairs alias the same memory location. **Machine learning:** it detects various bugs by learning context information from individual statements to entire programs. At the statement level, it models execution dependencies to identify vulnerabilities. One approach extracts control- and data-flow dependencies, structuring statement embeddings into a graph where nodes represent statements and edges capture dependencies [46]. A *graph attention network* learns topological structures, while an MLP classifier classifies vulnerable statements. At the function level, ML captures syntax, execution dependencies, structural dependencies, and historical bug indicators. Some models use syntax information, execution dependencies, and structural dependencies to detect vulnerabilities [74, 81, 139]. They extract features from statements, variables, types, function calls, AST paths, and control/data dependencies, transforming them into embeddings via tokenization. Sequence models like LSTMs and GRUs process embeddings, while ASTs, post-dominator trees, and exception flow graphs are embedded using tree-based (Tree-LSTM) or graph-based models (GNNs). These embeddings train classifiers such as *GNNs*, *KNNs*, or *MLPs* for vulnerability prediction. Other models incorporate historical bug indicators, including function complexity metrics and repository metadata, to train classifiers or compute vulnerability scores for function-level detection [30, 96]. At the program level, one approach applies static analysis to extend or simplify structural graphs (e.g., ASTs, CGs, CFGs, DFGs, or PDGs), constructing extended flow graphs used as *GNN* inputs for vulnerability detection at scale [20, 142, 144]. Another approach leverages *data mining* to identify frequent patterns and derive positive/negative rules for vulnerability detection. It extracts syntax features (e.g., variables, expressions) and semantic information (e.g., vulnerable code slices, control flow, function calls, condition checks) [7, 76]. These features are analyzed to detect recurring vulnerability patterns and derive association rules.

3.7 Conclusion of Bug Detection at IR Level

Table 3 indicates that static analysis is the most widely used approach for detecting logic, memory safety, missing-check, concurrency, and resource management bugs in C/C++ and Rust at the IR level. Techniques rely on CGs, CFGs and DFGs, and flow-sensitive, field-sensitive, alias, intra/inter-procedural, taint and data-flow analysis along with symbolic execution. Notable tools like UbiTect [161], AMCHEX [141], and RUDRA [4] effectively detect bugs such as use-before-initialization, missing security checks, and unsafe Rust patterns, with precision rates up to 87.5%. Hybrid techniques like Reorder [41] and FCatch [60] combine static and runtime analysis to

Table 3. Comparison of Bug Detection Techniques for C/C++ & Rust

Paper	Techniques	Bugs	Applications	Accuracy&Overhead
[33],[59]	static	C/C++, logic	private,public	-,-
[141],[38]	static	C/C++, missing-check ,	public	46.2% FP,-
[152]	ML	C/C++, missing-check	public	-
[26],[5],[126],[68]	static	C/C++, concurrency	partially public, public	-,.8.8% FP,75%,-
[41],[60]	hybrid	C/C++, concurrency	private,public	,100% & 0.97%
[161],[170],[40]	static	C/C++, memory safety	public	59% FP, .37%
[110]	dynamic	C/C++, memory safety	public	-
[151]	hybrid	C/C++, memory safety	partially public, public	-
[11]	ML	C/C++, memory safety	public	56.7%
[21],[50]	static	Rust, memory safety	private	-,48.3%
[94],[155]	static	C/C++,resource management	public	-,-
[148]	static	C/C++,unlisted	public	93.31%
[46],[74],[81],[139],[96],[142], [20],[144],[76], [7]	ML	C/C++, unlisted	public	36% F1,17.6% F1,80.9% F1,28.9% recall,-,66.94% F1,97.4%,92.7% F1,-,
[4]	static	Rust, logic, memory safety, concurrency	public	-
[98]	static	Rust, memory safety, concurrency	public	87.5% precision

uncover concurrency and order violations with higher accuracy. Dynamic analysis, e.g., Safe Sulong [110], captures runtime memory issues like UAFs via execution tracing. Machine learning is increasingly used for scalable detection. Models such as LineVD [46], DeepWukong [20], and YUGA [98] apply GNNs or LSTM-based architectures to learn vulnerability patterns from control/data dependencies, achieving high F1 scores (up to 97.4%). However, the rest of solutions, except for ML-based techniques, do not report accuracy or overhead metrics. Instead, they typically present only the number of bugs detected.

4 C/C++ & Rust Bug Detection at Source Code Level

4.1 Logic Bugs

Logic bugs at the source code level, including fast-path errors, anomalous variable-constant pairings, and protocol evolution inconsistencies, are detected using static analysis and machine learning. **Static analysis:** it identifies logic bugs through specification checking and symbolic execution. PALLAS [51] detects fast-path logic errors by merging fast-path code and headers, constructing a control-flow graph, and applying symbolic execution with user-specified annotations (e.g., @immutable for immutable variables, @cond for necessary condition checks, and @order for proper conditional evaluation ordering). It also detects mismatched fast-path and slow-path return values, missing fault-handling, and uncoordinated updates between path states and associated data structures. **Machine learning:** it enhances static analysis through data mining [66] and NLP [17]. *Data-mining-assisted* static analysis detects variable-constant pairing bugs by constructing a bipartite graph linking constants to variables and clustering constants with similar usage profiles from flow-sensitive analysis. Clustering is performed using Euclidean and Jaccard similarity metrics, and anomalies are ranked by suspiciousness metrics such as cluster size and association strength, classifying them as name bugs, value bugs, or context bugs. *NLP-assisted* static analysis detects RFC-evolutionary bugs by constructing evolutionary trees from RFC documents, extracting packet field meta-information (e.g., name, offset, size), and reconstructing packet hierarchies. It applies a predominator-based rule violation algorithm to detect implementation failures that do not adapt to new RFC conditions or lack necessary error handling.

4.2 Multiple types of bugs

Machine learning detects source code-level bugs at the function and program levels. Function-level detection identifies vulnerable functions by analyzing high-level structures and execution contexts. TROVON [35] focuses on pattern-based vulnerability detection by comparing vulnerable code fragments with their security fixes. It abstracts functions by replacing user-defined names and comments with generic identifiers, then processes vulnerable-fixed function pairs using an *LSTM* model to detect potential bugs based on code differences. Program-level detection identifies bugs by analyzing entire programs using graph-based machine learning, inter-procedural slicing, and representation learning. VCCFinder [103] detects vulnerability-contributing commits by extracting

repository-, author-, commit-, and function-level features from version control systems, classifying them using an SVM trained on a Bag-of-Words model. LSTMF [77] applies deep-learning-based representation learning with *Bi-LSTM* networks to learn vulnerable programming patterns. It converts code into vector embeddings using Word2Vec with a Continuous Bag-of-Words model, processes them with two *Bi-LSTM* networks trained on different datasets, and trains a *random forest* classifier. LSTMF also applies transfer learning to adapt to new projects and predict vulnerability likelihood.

4.3 Conclusion of Bug Detection at Source Code Level

Static analysis methods like PALLAS [51] employ symbolic execution and user-annotated specifications on source code to identify errors in fast-path code logic. It reports 69% detection accuracy. Machine learning-assisted analysis improves detection by identifying structural and semantic patterns within the source code. For example, data-mining approaches [66] use clustering and statistical similarity metrics to detect variable-constant mismatch bugs, while NLP-based techniques [17] extract and analyze protocol specifications to find RFC-evolution inconsistencies. These methods offer strong performance, with accuracies around 70% and manageable computational overhead (e.g., 37.3 seconds).

5 Java & Android IR-Level Bug Detection

5.1 Logic Bugs

Logic bugs at the IR level, including functional inconsistencies, API misuses, incorrect condition handling and semantic errors, are typically detected using static, dynamic, hybrid analysis and machine learning. **Static analysis:** it detects logic bugs by constructing ASTs, CFGs and DFGs and applying data-flow analysis, flow-sensitive, context-sensitive, field-sensitive, and intra- and inter-procedural analysis. Seader [167] detects and repairs security API misuses in Java libraries by analyzing secure and insecure code pairs, extracting ASTs, and detecting statement- and expression-level edits. It derives vulnerability-repair patterns using *intra-procedural data flow analysis* and stores security APIs with abstract fixes as JSON templates. *Inter-procedural backward slicing* matches WALA bytecode against stored patterns, automatically generating fixes for security API misuses. CryptoGuard [108] detects cryptographic and SSL/TLS API misuse bugs by applying *flow-, context-, and field-sensitive data-flow analysis* combined with forward and backward program slicing. It constructs CGs, CFGs, and DFGs from Jimple, tracing inter-procedural dependencies. Backward slicing detects hardcoded secrets, weak encryption, and improper SSL/TLS validation, while forward slicing tracks insecure configuration propagation. Sensitive parameters (e.g., cryptographic keys, PRNG seeds, SSL configurations) are monitored across method calls to flag weak cryptographic keys, insecure random number generators, and missing certificate validation. **Dynamic analysis:** it detects logic bugs by fuzzing. Genie [120] detects functional bugs in Android apps using independent view *fuzzing*. It builds a GUI transition model by generating random test seeds, exploring the app, and extracting GUI trees. Independent views, where interacting with one GUI element should not alter unrelated UI elements, are identified. Genie mutates interactions with inactive views and detects logic bugs by analyzing discrepancies between the original and mutated runs. Odin [134] identifies non-crashing functional bugs by detecting deviant behaviors. It executes the target app with generated event sequences, captures GUI execution traces, and constructs a GUI model ensuring deterministic state transitions. Odin improves test coverage using a random walk simulation algorithm to prioritize under-explored GUI states and generate additional test inputs using *fuzzing*. A hierarchical clustering algorithm groups observed GUI states, flagging unexpected small clusters as potential anomalies.

Hybrid analysis: it detects logic bugs by pattern matching with dynamic validation. DCDroid [140] detects SSL/TLS API misuse vulnerabilities in Android apps by generating call graphs from

Smali code to trace SSL-handling functions (e.g., X509TrustManager, HostnameVerifier). Static analysis identifies entry points in Activities and Services where vulnerable code could execute. Automated UI interactions dynamically test execution paths. To confirm vulnerabilities, DCDroid uses mitmproxy to perform Man-in-the-Middle attacks, intercepting HTTPS traffic and using VPNService, UsageStatsManager, and PackageManager to correlate app behavior with network traffic. If HTTPS traffic observed on the mobile device matches intercepted mitmproxy traffic, the vulnerability is confirmed. **Machine learning:** it analyzes control/data dependencies and syntactic structures. CFGNN [164] applies a CFG-based *GNN model* to detect condition-related bugs. It extracts CFGs from methods, tokenizes statements, and encodes them with a *BiLSTM* to preserve sequential context. A graph-structured *LSTM* captures long-range control-flow dependencies, while an API-aware attention mechanism highlights nodes containing API-related condition expressions. CFGNN classifies methods associated with condition-handling errors. Bugram [135] detects semantic errors using n-gram language models. It parses source code into ASTs, converts them into token sequences, and trains *n-gram models* (2 to 10 tokens in length) to learn probability distributions over token patterns. Token sequences with low probability scores across multiple models are flagged as potential semantic errors. LineFlowDP [153] enhances semantic error detection by applying *GNNs* to PDGs. It categorizes PDG nodes into variable definitions, conditions, and execution nodes while classifying control-flow edges as True, False, and Next to reflect execution semantics. To preserve execution order, it prioritizes control dependencies (True>Next>False) and extends code line semantics through backward and forward control/data flow dependencies. Doc2Vec encodes code lines into low-dimensional embeddings, which are processed by a *Relational GNN* for file-level defect classification. Social network centrality ranks code lines by risk scores for defect localization.

5.2 Performance Bugs

Performance bugs at the IR level include software hangs, algorithmic complexity issues, performance cascading, lengthy operations, view holder violations, energy inefficiencies, and execution hotspots, detected through static, dynamic, hybrid analysis and machine learning. **Static analysis:** it detects performance bugs by constructing CGs, CFGs and DFGs and applying intra/inter-procedural, flow sensitive, data-flow sensitive analysis with pattern matching. DScope [24] detects software hangs caused by data corruption by decompiling bytecode into Jimple using Soot, constructing control and data-flow graphs, and identifying loops with exit conditions dependent on external data (e.g., I/O operations) using *flow sensitive, data-flow sensitive analysis*. *Loop-bound and stride analysis* filter out irrelevant loops. PerfChecker [84] identifies lengthy main-thread operations and view holder violations by analyzing lifecycle and event handler checkpoints, constructing call graphs, and detecting transitive calls to resource-intensive APIs such as networking and file I/O using *intra/inter-procedural analysis*. It also verifies view reuse correctness in getView() callbacks using program dependency graphs. **Dynamic analysis:** it identifies performance issues through real-time execution monitoring [58] or fuzzing to maximize resource consumption [8]. *Execution monitoring* analyzes method invocations, return values, and long-latency calls, profiling method executions and using selective call pruning to isolate performance bottlenecks. *Fuzzing* detects algorithmic complexity vulnerabilities in Java libraries by treating each method in the target Java library as an independent entry point. It uses the Class Hierarchy Graph (CHG) for seed generation, instantiating concrete classes for abstract classes and interfaces. Random walks over CHG ensure diverse class selection. Genetic algorithms are then applied to mutate inputs to maximize execution time and memory consumption. A combined approach integrates execution monitoring and fuzzing to detect energy bugs and hotspots[6], building event flow graphs from user interactions, selecting energy-intensive traces, and *monitoring* I/O system calls and power usage. By dynamically adjusting inputs, *fuzzing* explores new traces, identifying system call sequences linked to abnormal power

consumption and confirming energy bugs, while hotspots are flagged by comparing energy usage across traces. **Hybrid analysis:** it combines loop-bound analysis, concolic execution, runtime profiling, and fuzzing. detects performance cascading bugs in distributed systems [69] by analyzing small-scale execution traces to designate critical execution areas as sinks, tracking execution dependencies and resource contention, and integrating static *loop-bound analysis* with *runtime profiling* to identify non-scalable code regions. Another approach detects algorithmic complexity vulnerabilities using *fuzzing and concolic execution* [99], generating inputs, constructing symbolic execution trees, identifying worst-case nodes with increasing code coverage or computational cost, and refining inputs iteratively by symbolic execution. **Machine learning** it detects energy bugs by modeling system operations and analyzing deviations in resource consumption patterns [171]. A Lasso-based *linear regression model* is trained on labeled datasets with system call features such as context switches, page faults, and file/socket operations. The trained model analyzes commit histories in updated software versions to detect abnormal energy spikes.

5.3 Missing-check Bugs

Missing-check bugs at the IR level, including missing security checks, missing input validation, component hijacking bugs, permission re-delegation vulnerabilities, Inter-Application Communication bugs, and missing-authorization, are detected through static, dynamic, hybrid analysis and machine learning. **Static analysis** it detects missing-check bugs by constructing CGs, CFGs and DFGs, and applying inter-procedural, context-, field-, flow- and alias, data-flow analysis and symbolic execution. MPChecker [86] and WeChecker [22] identify missing security checks by analyzing unprotected privileged operations. MPChecker applies *inter-procedural and field-sensitive data-flow analysis* to detect privileged operations and system-critical variables from logs, access patterns, and error conditions. It analyzes access patterns to user and system data and identifies permission-checking APIs using post-dominance relationships in control flows. It applies *inter-procedural data-flow analysis* to detect unguarded privileged operations. Similarly, WeChecker constructs CFGs from Jimple IR, extracts components, intent filters, and callbacks from Manifest and XML files, and builds a CG. It marks unprotected components as entry points, analyzes the CG, and applies SUSI-based *inter-procedural source-sink analysis* for capability leak detection. *Alias-, flow-, context-, and field-sensitive analysis* identify sensitive data returned by permission-protected calls. ACO-Solver [127] detects missing input sanitization checks by using *symbolic execution* to solve supported attack constraints and Ant Colony Optimization to address unsupported constraints. ACO-Solver simulates artificial ants, each representing a candidate attack string, to explore paths likely to bypass sanitization. If a SAT solver confirms feasibility, the system flags the vulnerability. CHEX [87] and ARF [38] detect component hijacking and permission re-delegation vulnerabilities, respectively. CHEX constructs CGs and DFGs on WALA IR, extracts entry points from Manifest and overridden framework methods, and applies *context- and field-sensitive analysis* to generate split data-flow summaries (SDSs). These track intra-split flows, linking them across execution sequences to detect unprotected source-to-sink flow via *inter-procedural analysis*. ARF identifies cases where a deputy entry point invokes a target entry point without enforcing necessary permission checks by constructing call graphs, extracting authorization checks, and identifying call paths between entry points accessible to third-party apps. **Dynamic analysis:** it identifies missing-check bugs using dynamic taint analysis, fuzzing and execution monitoring. Rivulet [47] and IntentDroid [42] detect input validation and IAC vulnerabilities via *taint analysis* and *bytecode instrumentation*. Rivulet tracks user-controlled input propagation in Java bytecode using ASM instrumentation, flagging cases where untrusted data reaches sensitive sinks. It then generates attack payloads (e.g., XSS, SQL injection) and verifies whether sanitization checks are missing. IntentDroid instruments Dalvik bytecode, identifies IAC entry points from Manifest files, and executes crafted intents and

attack payloads to assess enforcement of security checks, detecting potential user data leaks. PAIR [85] and MoSSOT [115] detect authorization bugs by modeling Remote Procedure Call (RPC) and SSO authorization mechanisms, respectively. PAIR analyzes Ant Group’s RPC system using cross-system tracing to construct behavioral dependency graphs (BDGs) capturing service dependencies. It decomposes RPCs into RPClets, prioritizing those handling sensitive data using entropy analysis of encrypted payloads. To detect authorization flaws, PAIR modifies high-risk RPClet requests by replacing user identifiers with test identifiers and replaying them. If the BDG remains unchanged, the system flags a bug. MoSSOT automates UI interactions to reach SSO login pages, captures network traces between mobile apps and identity provider servers, and applies differential analysis to infer authentication parameters. It generates HTTP(S) test cases, *fuzzes* login credentials (e.g., access tokens, user IDs), and compares UI/network behaviors to detect unauthorized login attempts. AuthScope [173] builds on MoSSOT’s approach by substituting protocol fields (e.g., security tokens) with values of small Euclidean distances. Modified requests are sent to the server, and server responses are analyzed to identify authorization flaws.

Hybrid analysis: it integrates static analysis with fuzzing for detecting missing-check bugs. AuthDroid [133] detects authorization bugs using a five-party model (user, SP server, RP server, RP app, and SP app). It reverse engineers RP apps, applies static *pattern matching* to extract participants, and uses *dynamic traffic analysis* to monitor request/response interactions. Differential *fuzzing* modifies parameters (e.g., app ID, access token), replays requests, and detects weak credential handling and token-user binding flaws. Demissie [27] detects second-order permission re-delegation vulnerabilities, where a malicious app exploits a privileged app to dispatch privileged Intents to system components. It extracts public Android components from the app’s Manifest, analyzes bytecode to identify entry points, and detects inter-component communication (ICC) method invocations (e.g., `startActivity()`) dispatching privileged Intents. Constant propagation using privileged Action strings and *data-flow analysis* collect additional sinks. A call graph maps paths from sources to sinks, which are verified via *fuzzing*. **Machine learning:** it analyzes semantic and syntactic patterns for inconsistencies. One approach integrates *NLP* with static analysis to detect missing security checks [101]. It constructs a method-term matrix from extracted API calls, conditions, and loop keywords, applying document frequency weighting and *Latent Semantic Analysis* for dimensionality reduction. Clustering groups similar functions, and inter-cluster comparisons identify missing checks. An inter-procedural control-flow graph verifies whether missing checks in a method are compensated by caller methods. Another approach leverages AST-based pattern recognition for multi-class classification of missing-check bugs [1]. It extracts ASTs from methods, parses CVE records for labels, and uses depth-first search to generate AST n-grams. These n-grams are converted into a bag-of-words representation, labeled with CWE types or non-vulnerable tags, and analyzed using a *random forest classifier* to assess security risks.

5.4 Concurrency Bugs

Concurrency bugs at the IR level include time-of-fault errors and data races, detected through static and hybrid analysis. **Static analysis:** it detects concurrency bugs by applying context-sensitive alias analysis. nAdroid [32] identifies UAF bugs in Android applications by modeling event-driven and multi-threaded concurrency. Given an APK package, it unifies event callbacks and posted callbacks as threads to preserve causal dependencies. It then detects UAF ordering violations using context-sensitive heap object naming and *alias analysis* to track object lifetimes and identify access-after-free scenarios across Android-specific components such as Handlers, AsyncTasks, and Services. **Hybrid analysis** it combines flow-sensitive, data-flow-sensitive, intra-procedural, and inter-procedural analysis with enforced execution. FCatch [80] detects time-of-fault bugs, which occur when component failures interfere with concurrent operations at critical

moments. It instruments Java programs using WALA for function tracing and Javassist for inserting monitoring functions before heap and static variable accesses. By observing both fault-free and faulty runs in distributed systems, FCatch tracks resource access and fault tolerance operations, detecting conflicting read-write operations on shared resources. *Control and data dependency analysis* identifies unprotected conflicts, and *enforced execution* manipulates timing to trigger bugs. DCatch [79] detects data concurrency bugs by analyzing correct execution traces and constructing a happens-before (HB) model to capture concurrency and communication patterns. It tracks inter-node communication, intra-node asynchronous events, and multi-threaded synchronization to identify execution sequences vulnerable to data races and deadlocks. DCatch collects traces of memory accesses, HB operations, and synchronization events, then builds an HB graph to detect vulnerable access pairs. It applies static pruning at *intra-procedural, inter-procedural, and inter-node levels* before re-executing the target system under controlled conditions to confirm bugs.

5.5 Side Channel Vulnerabilities

Side-channel vulnerabilities, particularly resource usage leaks, are detected using dynamic and hybrid analysis. **Dynamic analysis:** it detects side channel vulnerabilities by using fuzzing and runtime analysis. DIFFUZZ [97] uses resource-guided *fuzzing* to identify inputs that maximize differences in resource consumption between secret-dependent execution paths. It iteratively executes a program with varying secret inputs while measuring execution costs (e.g., time, memory, response size). Mutation-based fuzzing generates new inputs that increase cost variations or improve program coverage, with high-difference inputs re-added to the fuzzing queue. The process continues until a significant, exploitable side-channel leak is detected, indicating that an attacker could infer secret data from execution cost variations. ProcHarvester [118] detects procfs information leaks in Android by identifying and exploiting side-channel vulnerabilities. It operates in four phases: (1) Exploration scans procfs resources (e.g., /proc/meminfo, /proc/stat) to identify system files that change in response to events like app launches or keyboard inputs. (2) Profiling collects time-series data by logging fluctuations in these files while triggering events via ADB commands, MonkeyRunner scripts, or manual interactions. (3) Analysis applies dynamic time warping (DTW) to detect correlations between system resource variations and triggered events, even in noisy data. (4) Attack exploits identified leaks to infer user actions, operating in single-resource mode (analyzing individual procfs files) or multi-resource mode (combining multiple sources for higher accuracy). **Hybrid analysis:** it detectsside-channel vulnerabilities by constructing call graphs and integrating pointer analysis, alias analysis, and runtime analysis. Themis [16] detects resource-based side-channel vulnerabilities by verifying ϵ -bounded non-interference in resource consumption. It analyzes Java bytecode using pointer and alias analysis on a call graph to track variables influencing secret-dependent operations, identifies hot spots where resource consumption depends on secret data, and instruments the code with consume statements to measure execution time and response size. Themis evaluates whether these hot spots adhere to ϵ -bounded non-interference, ensuring that resource usage variations remain within an acceptable threshold. If deviations exceed the threshold, it indicates that secret-dependent execution differences could be exploited.

5.6 Resource Management Bugs

Resource management bugs, including resource leaks and data loss, are detected through static and dynamic analysis. **Static analysis:** it identifies resource management bugs by constructing data-flow graphs and applying context-sensitive *data-flow analysis* [86]. It tracks resource usage violations caused by lifecycle variations in two scenarios: (1) Skip scenarios, where lifecycle methods like onStop are skipped, leading to memory leaks when resources allocated in onCreate are never released. (2) Swap scenarios, where changes in lifecycle execution order (e.g., onStop executing before onSaveInstanceState) cause data loss when edit operations occur without a

corresponding save operation. **Dynamic analysis:** it detects data loss by *injecting faults* and monitoring applications' behavior [109]. It constructs a GUI model to track visited states, prioritizing unexplored states prone to data loss. To expose faults, it injects fault-revealing actions such as filling input fields and triggering stop-start events (e.g., screen rotations) to simulate interruptions. It compares screenshots before and after an interruption to detect visual discrepancies and analyzes GUI properties for unexpected changes. If an app fails to persist user input, resets UI elements unexpectedly, or modifies the interface inconsistently, it flags the issue as a potential data loss fault.

5.7 Multiple Types of Bugs

Various bugs are simultaneously detected at the IR level through similarity comparison and multi-tier static analysis. One approach detects vulnerable third-party libraries by extracting control-flow and opcode features from Smali and Jimple IRs and matching them against a database of known vulnerabilities and CVE records [162]. It isolates library code from host applications, applies *fuzzy hashing and edit distance comparisons* to determine library versions, and cross-references them with CVE records to generate vulnerability reports. Another approach integrates multiple static analysis techniques to detect and patch vulnerabilities across six categories: inter-component communication, storage, web, cryptographic, runtime-permission, and networking issues [34]. It constructs control-flow and data-flow graphs from Jimple IR and applies *flow-sensitive, data-flow, taint, parameter, return-value, and API analysis* to detect security flaws. It also examines Manifest files to identify weak permission enforcement and multitasking issues. Beyond detection, it automatically generates patches by instrumenting Jimple code and modifying control flow, issuing warnings for unresolved vulnerabilities.

5.8 Conclusion of Bug Detection at IR Level

Table 4. Comparison of Bug Detection Techniques Across Java & Android

Paper	Techniques	Bugs	Applications	Accuracy&Overhead
[167],[108]	static	Java, logic	public	95% precision&72% recall, 98.61% precision&12.7s
[120],[134]	dynamic	Android logic	partially public,public	40.9% TP,-
[140]	hybrid	Android, logic	private	54.3% TR
[24]	static	Java, performance	public	53.2% TR
[84]	static	Android, performance	public	54% TR
[58],[8]	dynamic	Java, performance	private	-,-
[69],[99]	hybrid	Java, performance	partially public,public	-,-
[171]	ML	Android performance	public	94%
[88],[22]	static	Android, missing-check	public	87.5%&30s, 96% precision&96% recall&30s
[127]	static	Java, missing-check	public	4.7%-100%
[87],[38]	static	Android, missing-check	private,public	81% TR&37s,-
[47],[85]	dynamic	Java, missing-check	private, public	-,0 FN
[42],[115],[173]	dynamic	Android, missing-check	private	92% recall,-,-
[133],[27]	hybrid	Android, missing-check	private,public	86.2%,-
[32]	static	Android, concurrency	public	-
[80],[79]	hybrid	Java, concurrency	private, public	-,-
[97]	dynamic	Java, side channel	public	-
[118]	dynamic	Android, side channel	private	73%
[16]	hybrid	Java, side channel	private	-
[88]	static	Android, resource management	public	-
[109]	dynamic	Android, resource management	public	75%
[162]	static	Java, unlisted	public	90.55% precision& 88.79% recall
[34]	static	Android, logic, missing-check	private	95.23% precision&97.5% F1
[164],[135],[153]	ML	Java, logic bugs	public	46.2% F1,71.2% precision,37.6%
[101],[1]	ML	Java, missing-check bugs	partially public,public	33% F1,75% F1

As summarized in Table 4, static analysis remains a widely used approach for detecting logic, missing-check, and performance bugs, leveraging control-flow graphs (CFGs), data-flow graphs (DFGs), alias analysis, and symbolic execution. Tools such as Seader [167] and CryptoGuard [108]

identify API misuse and cryptographic vulnerabilities in Java libraries, achieving high precision (95–98.61%) with reasonable computational overhead. MPChecker [86] and WeChecker [22] effectively analyze missing security checks in Android applications, reporting 87.5%–96% precision with execution times around 30s. Dynamic analysis is primarily used for detecting runtime vulnerabilities, including functional logic errors, concurrency issues, and side-channel attacks. Tools such as Genie [120] and Odin [134] detect GUI-related logic bugs in Android apps, achieving 40.9% true positive (TP) rates. For missing-check vulnerabilities, IntentDroid [42] and MoSSOT [115] apply taint tracking and fuzzing, achieving 92% recall. Dynamic execution monitoring tools, such as ProcHarvester [118], successfully identify side-channel vulnerabilities in Android with 73% detection accuracy. Hybrid analysis, combining static and dynamic techniques, offers enhanced precision and validation mechanisms for detecting logic, missing-check, concurrency, and side-channel vulnerabilities. DC-Droid [140] detects SSL/TLS API misuses using a mix of static call graph construction and dynamic HTTPS traffic interception, achieving 54.3% true recall (TR). AuthDroid [133] and Demissie et al. [27] apply hybrid differential fuzzing and pattern matching to detect Android authentication flaws, achieving 86.2% accuracy. ML-based approaches are increasingly being used for semantic-aware bug detection, particularly for logic and missing-check bugs. CFGNN [164] and LineFlowDP [153] utilize graph neural networks (GNNs) to analyze control/data dependencies and program structures, achieving 46.2% F1-score and 71.2% precision. For missing-check vulnerabilities, Confiance [101] applies NLP-based semantic analysis, achieving 33–75% F1-scores, depending on dataset granularity.

6 JavaScript, PHP & Python Bug Detection at IR Level

6.1 Logic Bugs

Logic bugs such as loop control issues, API misuse, type errors, attribute errors, and loose comparisons are detected through static, dynamic, and hybrid analysis. **Static analysis:** it detects logic bugs using taint analysis, symbolic execution, and pattern comparison. Torpedo [100] identifies loop control flow bugs in PHP web applications by using *taint analysis* and *symbolic execution* to track tainted database attributes and detect loops dependent on tainted attributes. It performs backward *symbolic execution* to generate attack vectors for verification. SAFEWAPI [3] identifies API misuse in JavaScript by analyzing Web API function calls against specifications. It builds a CFG, a DOM tree, extracts *Web API specifications* from HTML documents, traverses the CFG to identify API-related nodes, retrieves type information from the heap, and validates calls based on types, argument counts, and return values against specifications. **Dynamic analysis:** it detects logic bugs by runtime monitoring and pattern matching. AsyncG [122] detects asynchronous API misuse by running Node.js applications and constructing an Async Graph (AG) representing event-driven execution. AG nodes represent callback registration, execution, triggering, and object binding, while edges capture execution flow and object bindings. It applies patterns to detect scheduling bugs (e.g., recursive micro-tasks), emitter bugs (e.g., dead emits), and promise bugs (e.g., double resolve). **Hybrid analysis:** it integrates taint tracking, symbolic execution, and runtime validation. LChecker [70] detects loose comparison bugs in PHP applications using static taint analysis and dynamic validation. It defines three conditions: untrusted input in comparisons (Cond1), implicit type conversion (Cond2), and inconsistent type behavior (Cond3). It constructs CFGs and CGs, applies *taint analysis* to identify untrusted comparisons, and infers operand types. An enhanced PHP interpreter monitors loose comparisons at runtime, performing parallel strict type comparisons to detect inconsistencies. Xu et al. [149] detect type and attribute errors in Python by combining dynamic tracing with symbolic execution. The system executes test inputs, converts statements to Static Single Assignment form, and expands execution states via *constraint solving*. Type errors arise from failed subtype assertions, while attribute errors occur when accessing nonexistent attributes

due to inconsistent initialization. An SMT solver verifies constraint violations and reports bugs with triggering inputs.

6.2 Missing-check bugs

Missing-check bugs at the IR level are detected through static, dynamic analysis and machine learning. **Static analysis:** it includes alias, interprocedural, taint, data-flow, flow-sensitive analysis, and subgraph matching. CmrfScanner [156] detects Cross-MiniApp Request Forgery in JavaScript mini-apps using *alias analysis*. It scans for cross-miniapp API calls (e.g., `navigateToMiniProgram`), converts relevant code into ASTs, and checks for missing appID verification. TChecker [89] and HiddenCPG [143] detect missing input validation in PHP applications. TChecker performs *context-sensitive interprocedural taint analysis*, constructing a call graph via backward *data-flow analysis* to link call sites to targets and check if tainted variables reach sensitive operations without sanitization. HiddenCPG extracts vulnerable snippets from CVEs and GitHub, converts them into Code Property Graphs (CPGs) using Joern, and applies subgraph matching with the VF2 algorithm to detect unsanitized inputs. ObjLupAnsys [71] identifies prototype pollution in Node.js using *flow-, context-, and branch-sensitive taint analysis*. It constructs an Object Property Graph from the AST to trace attacker-controlled inputs that may override system object properties, checking if built-in functions can be redefined. **Dynamic analysis:** it uses taint analysis, fuzzing, and runtime monitoring. SSRFuzz [130] targets Server-Side Request Forgery (SSRF) in PHP by analyzing sinks and URL schemes that process user-supplied URLs. It generates probe payloads to test these sinks and flags functions as SSRF sinks if a crafted URL triggers a server-side request. Dynamic taint inference hooks these sinks at runtime, while a web crawler identifies unsanitized input points. SSRFuzz then injects fuzzed payloads into these points, sends test HTTP requests, and detects SSRF vulnerabilities using six monitoring strategies: HTTP/DNS out-of-band tracking, port monitoring, HTTP response/log analysis, and file interaction tracking. Steinhauser *et al.* [119] propose a graybox analyzer for detecting reflected context-sensitive, stored context-insensitive, and stored context-sensitive XSS flaws. It mutates HTTP request components (e.g., GET/POST parameters, cookies, headers) with XSS payloads and submits modified requests to observe whether injected values appear in responses (reflected XSS). For stored XSS, it injects payloads into both HTTP requests and database responses via a custom interception protocol, detecting whether the input is executed later. Context-sensitive analysis identifies failures in input transformation that lead to execution. **Machine learning:** it enhances static/dynamic analysis by filtering false positives. Kim *et al.* [62] detect client-side business flow tampering by recording user interactions and identifying DOM elements tied to business logic. They generate automation scripts to replay navigation, monitor DOM mutations, collect function call traces, and construct business control flow graphs. A machine learning model prioritizes functions based on execution frequency and call stack position. Automated tampering tests modify JavaScript bytecode at runtime to bypass or alter function calls and branch outcomes. The results are clustered using screenshot similarity and Tree Edit Distance for HTML, verifying whether business logic (e.g., paywalls, ads) is bypassed. WAP [95] detects missing input validation in PHP (e.g., SQL Injection, XSS, OS Command Injection) by combining *static taint analysis* with machine learning. It converts source code into ASTs and trace control-flow paths from entry points (e.g., `$_GET`) to sensitive sinks (e.g., database queries), and extracts attributes related to string manipulation, input validation, and SQL query handling. *Random Forest and Multi-Layer Perceptron* classifiers, trained on historical vulnerabilities, filter false positives.

6.3 Concurrency bugs

At the IR level, concurrency bugs such as data races and atomicity violations can be detected through **dynamic analysis**. It monitors execution traces, constructs happens-before (HB) graphs, and applies heuristics to identify conflicts. NRace [14] detects data races by executing test cases while

tracking lifecycle events, resource operations, and control flow. It builds an HB graph where nodes represent tasks and edges capture execution order based on program structure, event registration, promise resolution, and FIFO semantics. A race condition is flagged when two tasks perform conflicting operations on the same resource without an HB relationship. NodeAV [13] targets atomicity violations in Node.js by instrumenting source code and collecting execution traces that capture events and read/write operations on shared resources. It constructs an HB graph consistent with Node.js's event-driven model and identifies atomic event pairs expected to run without interference. Violations are detected by observing interleaved events that match four patterns: Read-Write-Read, Write-Read-Write, Read-Write-Write, and Write-Write-Read. Node.fz [25] uncovers hidden concurrency bugs in Node.js by perturbing the execution order of asynchronous events. It operates in three phases: (1) Hooking and Interception to capture events (e.g., timers, I/O) into controllable queues; (2) Schedule Fuzzing to simulate rare interleavings via shuffling, delaying, and de-multiplexing; and (3) Execution and Monitoring to run perturbed schedules while detecting crashes, assertion failures, or incorrect outputs.

6.4 Conclusion of Bug Detection at IR Level

Table 5. Comparison of Bug Detection Techniques Across JavaScript, PHP & Python

Paper	Techniques	Bugs	Datasets	Accuracy&Overhead
[100]	static	PHP, logic	public	58.3 TR
[3]	static	JavaScript, logic	private	7.9% FP
[122]	dynamic	JavaScript, logic	public	-
[70]	hybrid	PHP, logic	public	89%-95% , 68% TP , 20 ms
[149]	hybrid	Python, logic	public	95.97% & 99.80% accuracy
[89],[143]	static	PHP, missing-check	private, public	46.1% precision&661m,-
[130]	dynamic	PHP, missing-check	partially public	46.1% precision, 661m
[95]	ML	PHP, missing-check	public	92.1% accuracy, 92.5% precision
[156],[71]	static	JavaScript, missing-check	private	95.97% & 99.80%,-
[119]	dynamic	JavaScript, missing-check	private	-
[62]	ML	JavaScript, missing-check	private	98.15% precision&85.01% recall
[14],[13],[25]	dynamic	JavaScript, concurrency	partially public, public	0.9s-17.0s, -92.1%&92.5% precision

As shown in Table 5, static analysis remains the most prevalent technique for detecting logic bugs and missing-check bugs at the IR level, especially in PHP and JavaScript. Tools such as Torpedo [100], SAFEWAPI [3], and TChecker [89] leverage techniques like taint analysis, symbolic execution, and data-flow tracking to detect security vulnerabilities with varying levels of precision. Dynamic analysis is particularly effective in identifying runtime-dependent vulnerabilities, including concurrency issues and server-side request forgery. Tools such as SSRFuzz [130], NodeAV [13], and NRace [14] employ runtime monitoring, fuzzing, and schedule perturbation to uncover subtle bugs, such as race conditions and atomicity violations. While dynamic analysis typically achieves high detection precision, with Kim *et al.* [62] reporting 98.15% precision and 85.01% recall, it can also introduce significant runtime overhead. For example, SSRFuzz [130] reports an execution time of 661 minutes. Hybrid analysis, which integrates static and dynamic approaches, offers a balanced trade-off between accuracy and efficiency. Tools such as LChecker [70] and the method proposed by Xu *et al.* [149] apply taint analysis, symbolic execution, and runtime validation to detect logic bugs in PHP and Python, achieving accuracy rates as high as 99.8%. Besides, many solutions do not provide the information about their dataset or give the application names without specific versions.

7 JavaScript, PHP & Python Bug Detection at Source Code Level

7.1 Logic bugs

Logic bugs at the source code level include API misuse, cross-project compatibility, and name bugs, which are detected using static analysis and machine learning. **Static analysis:** it detects

logic bugs via pattern matching and symbolic execution. WeDetector [137] detects WeChat Mini-Program bugs by analyzing JavaScript for three patterns: incorrect platform-dependent API usage, incomplete layout adaptation, and improper asynchronous return handling. It constructs ASTs from index.js and util.js to extract function definitions and calls. It detects sensitive API misuse (e.g., `wx.makeBluetoothPair`), layout adaptation issues (e.g., improper use of `wx.getSystemInfo`), and improper handling of asynchronous calls (e.g., `wx.request`). NSBAP [91] detects Python cross-project bugs by analyzing how upstream bugs affect downstream modules. It constructs a dependence network mapping inter-project call relations and tracks version-specific dependencies. Intra-module analysis encodes function call paths, applying *constraint solving* to assess whether a bug propagates across modules. **Machine learning:** it detects logic bugs by learning from positive-negative code pairs. DeepBugs [107] targets name-related bugs in JavaScript by analyzing variable names, function arguments, and operators using semantic representations. It trains a neural network on correct code and buggy samples generated by swapping parameters, altering operators, or misusing variables from correct code. Code is embedded using Word2Vec to preserve semantic relationships between identifiers and literals, and a feedforward neural network classifies code as correct or buggy.

7.2 Multiple Types of Bugs

Logic errors, concurrency issues, and missing checks can be simultaneously detected at the source code level using multi-tier **static analysis** and pattern matching. RADAR [92] detects event-handling errors in Node.js by analyzing an event-based call graph composed of direct call edges, emit edges for event triggers, listen edges for listener registrations, and may-happen-before edges for execution order. It detects dead listeners when a listener is never triggered, dead emits when an event has no listeners, mismatched synchronous and asynchronous calls when improper mixing causes race conditions, and unreachable functions when they are never invoked or referenced. VuDeFr [64] detects prototype pollution and Regular Expression Denial of Service vulnerabilities in JavaScript using custom Semgrep patterns to identify unsafe object property assignments and malicious regular expressions. It enhances efficiency with *textual similarity search*, comparing function hashes against a vulnerability dataset and flagging functions that match predefined patterns or known vulnerability hashes. KIDvD [65] builds on VuDeFr by identifying and verifying these vulnerabilities. It uses Semgrep patterns and hashing to identify near-duplicate vulnerabilities, and verifies exploitability via *static taint analysis*. File dependency graphs are constructed to track taint propagation from entry points to vulnerable functions.

7.3 Conclusion of Bug Detection at Source Code Level

Static analysis remains the dominant approach, detecting logic, concurrency, and missing-check bugs primarily in JavaScript and Python source code, with some works explicitly addressing multiple types of bugs. Only one work uses ML-based techniques to detect logic bugs in JavaScript, due to a lack of available buggy open source projects. Performance evaluation reveals that static analysis techniques report a 7.9% false positive rate for Python logic bugs [91] and 58.3% true positive rate for JavaScript logic bugs, while only reporting execution times for JavaScript multi-type bug detection range from 0.9s to 17.0s. ML-based techniques, in contrast, achieve a 68% true positive rate with a 20ms overhead, showing potential in reducing detection time.

8 Cross Language IR-Level Bug Detection

8.1 Logic Bugs

Logic bugs spanning multiple languages, such as mishandled JNI exceptions and compiler errors, can be detected using static analysis and machine learning. **Static analysis:** it employs interprocedural, data-flow, and taint analysis. SBSAf [72] detects and recovers from mishandled JNI

exceptions in Java Native Interface (JNI) programs, where exceptions in native code (C/C++) are not automatically handled by the JVM. Mishandling occurs when a pending exception is followed by an unsafe operation (e.g., `strcpy`). SBSAf tracks exception states via *interprocedural data-flow analysis* and applies *static taint analysis* on a pointer graph to detect unsafe operations near exception states. Unsafe operations not on a whitelist trigger warnings, and `ExceptionClear` statements are inserted to recover from mishandling. **Machine learning:** it enhances logic bug detection by generating diverse test cases. ComFuzz [158] applies a *Transformer-based neural model* to generate syntactically valid, semantically diverse test programs for Java and JavaScript compilers. It trains on GitHub repositories and buggy test suites, extracting ASTs, tokenizing function-level blocks using Byte Pair Encoding, and using MCMC sampling to prioritize test cases. Differential analysis detects parsing or optimization bugs by comparing compiled test outputs, while mutation-based iterative testing refines programs that cause crashes or incorrect outputs.

8.2 Memory safety bug

Memory safety bugs, such as double free, use-after-free, and buffer overflow, can span multiple programming languages and are detectable through static analysis. **Static analysis:** it converts different programming languages into the same IR and applies inter-procedural flow analysis. ACORN [145] detects memory safety bugs in multilingual Rust programs by using Wasm as a unified IR for cross-language analysis. It translates Rust and C components into Wasm using formalized rules that map types, memory operations, and control flow constructs. ACORN then builds CFGs and CGs to model Rust-C interactions. It applies Wasmati and Wasabi to traverse CFGs *interprocedurally* and analyze memory operations, detecting double-free, use-after-free, and buffer overflow. CRUST [48] translates Rust and C into a common intermediate representation, CRUSTIR, which integrates control flow, data flow, type information, lifetime management, and ownership semantics, enabling precise memory safety analysis. It applies existing bug detection tools (rustc, Miri, MirChecker, and Rudra) to identify issues such as use-after-free, double-free, out-of-bounds access, stack overflow, and integer overflow.

8.3 Multiple Types of Bugs

Various types of bugs across languages can be detected at the IR level using static analysis, dynamic analysis, and machine learning. **Static analysis:** it uses pattern matching and structural similarity. Pewny *et al.* [105] introduced tree edit distance-based equational matching to detect vulnerabilities in C/C++ and Objective C. It extracts disassembly and CFGs, summarizes basic blocks as expression trees, and matches them to known buggy code signatures using tree edit distance. Functions are ranked by similarity to reference signatures, highlighting likely vulnerable code regions. **Dynamic analysis:** it utilizes runtime monitoring and constraint validation. Jinn [67] identifies foreign function interface (FFI) violations in JNI and Python/C programs using context-specific dynamic analysis based on state machine specifications. It models FFI constraints using state machines that define state transitions and language interactions, ensuring proper Java-Python-C integration. Jinn intercepts Java-C transitions, monitors runtime states, and validates FFI constraints by analyzing JNI function calls, exception state changes, resource management, and thread switches. **Machine learning:** it identifies bugs by analyzing historical data. Luo *et al.* [90] propose Compact Abstract Graphs (CAGs) for vulnerability detection in Java and C. CAGs are derived from Abstract Graphs (AGs), built from ASTs with nodes representing tokens or syntactic properties and edges defining relationships. AGs are built by reversing AST edges, adding edges between sequential tokens, and linking terminal nodes to the root as a global sink. AGs are compressed by merging node sequences and aggregation structures, then embedded into a vector space for *GNN-based* vulnerability classification. Scaffle [106] detects bugs in Android, IOS and PHP codebases by combining machine learning with an information retrieval-based search engine.

Using historical crash-fix data, it trains a *bi-directional RNN* at the line level to generate vector representations of words within each line and a *bi-directional RNN* at the trace level to predict the relevance score of each line. The information retrieval-based search engine then queries the highest-ranking lines to retrieve buggy files.

8.4 Conclusion of Bug Detection at IR Level

Static techniques remain the predominant approach, particularly effective in cross-language scenarios involving Rust and C memory safety bugs, where they achieve high precision and recall (e.g., [145], [48]). In contrast, dynamic analysis offers enhanced coverage for concurrency and runtime resource management bugs, demonstrating strong accuracy with manageable overhead in cross-language settings, as exemplified by [67]. ML-based approaches (e.g., [90], [106]) often operate on large-scale, real-world datasets, including industrial codebases, but typically do not explicitly delineate the specific bug types they detect. While these methods show potential for cross-language generalization, their detection performance tends to vary across different language and bug contexts.

9 Challenges, and Research Gaps

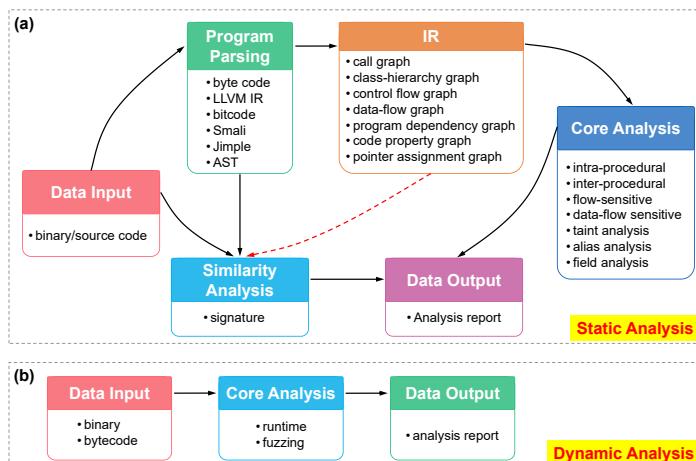


Fig. 3. Overview of Analysis Techniques

Despite growing interest in software vulnerability detection, our survey identifies several critical research gaps. Java and Android, though widely used in enterprise and mobile applications, have received limited attention at the source code level, with only one notable work addressing this space [121] because of a limitation compounded by the scarcity of open-source applications for these languages. Performance bug detection remains largely limited to C/C++, Java, and Android, with few efforts targeting dynamically typed languages or cross-language performance issues, aside from [128, 146] for JavaScript and Python.

Furthermore, at the IR level, research remains fragmented, with very few efforts targeting multiple bug types in languages like PHP, JavaScript, and Python—notably, only one work exists for PHP [117]. Similarly, source code-level multi-bug detection in PHP and Python is severely underexplored. Among these, Python remains the least tested programming language though it is the most popular language for machine learning applications. More generally, most ML-based techniques still struggle to learn semantically meaningful patterns from raw source code, limiting their pre-compilation bug detection capabilities across programming languages. Perhaps most notably, no current approach

unifies multiple programming languages for source-level bug detection, revealing a fundamental gap in cross-language analysis at the source code level. This shortfall is particularly problematic as modern applications are inherently polyglot, often comprising components written in diverse languages that interact via APIs, system calls, or network protocols. In such systems, vulnerabilities originating in one component can propagate, posing system-wide security risks. Effective cross-language and cross-representation analysis is therefore not optional but essential.

Finally, we observe notable concerns around reproducibility and transparency. Approximately 37.1% of surveyed works fail to disclose dataset details or specify application versions, and only 45.7% release their source code, making comparative evaluations difficult and hindering progress in the field. These findings collectively highlight an urgent need for comprehensive, reproducible, and language-agnostic vulnerability detection techniques, especially at the source code level, to support scalable, multi-bug, and cross-language analysis in real-world, heterogeneous software systems.

10 Conclusion

The main analysis pipelines for vulnerability detection across programming languages are illustrated in Fig.3. Static analysis (a) often relies on intermediate representation languages, such as LLVM IR, or AST, to construct structural, control, or data dependencies (e.g., call graphs), and apply static analysis algorithms to identify potential vulnerabilities. Alternatively, some approaches directly extract meta-information from binaries, bytecode, or source code to generate signatures, which are then matched against known vulnerable patterns. Machine learning (red dashed line) uses features derived from various IRs to train models for bug detection. Dynamic analysis (b) relies on runtime monitoring to capture buggy execution traces or uses fuzzing to trigger crashes. Hybrid analysis integrates static and dynamic techniques to enhance detection capability. Modern software systems are written in diverse languages, such as C/C++ for low-level system modules, and Java for enterprise applications. Each language introduces unique syntax, semantics, and vulnerability surfaces. Consequently, purely language-agnostic techniques may fail to capture language-specific bug patterns and nuances in program behavior. Furthermore, applications may be analyzed at different levels, source code, IR, or binary, depending on the development or deployment context, and each representation supports different forms of analysis. Effective vulnerability detection must therefore align with both the programming language characteristics and the analysis granularity. This survey underscores the pressing need for more robust, general-purpose tools and benchmarks that are capable of analyzing heterogeneous codebases across programming languages, while also adapting to the specific requirements of different bug types and representations.

References

- [1] Fahad Al Debeyan, Tracy Hall, and David Bowes. 2022. Improving the performance of code vulnerability prediction using abstract syntax tree information. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2–11.
- [2] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*. 298–313.
- [3] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 507–517.
- [4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.
- [5] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2020. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS)* 36, 4 (2020), 1–30.
- [6] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 588–598.

- [7] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. Nar-miner: discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422.
- [8] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2022. Hotfuzz: Discovering temporal and spatial denial-of-service vulnerabilities through guided micro-fuzzing. *ACM Transactions on Privacy and Security* 25, 4 (2022), 1–35.
- [9] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. 2010. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*. 413–425.
- [10] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 706–717.
- [11] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th international conference on software engineering*. 1456–1468.
- [12] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, and Paul Yu. 2019. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 211–224.
- [13] Xiaoning Chang, Wensheng Dou, Yu Gao, Jie Wang, Jun Wei, and Tao Huang. 2019. Detecting atomicity violations for event-driven node.js applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 631–642.
- [14] Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. 2021. Race detection for event-driven node.js applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 480–491.
- [15] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*. 263–274.
- [16] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 875–890.
- [17] Jingting Chen, Feng Li, Qingfang Chen, Ping Li, Lili Xu, and Wei Huo. 2023. EBugDec: Detecting Inconsistency Bugs caused by RFC Evolution in Protocol Implementations. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 412–425.
- [18] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 334–345.
- [19] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 873–887.
- [20] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.
- [21] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–21.
- [22] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. 2015. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 1–12.
- [23] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3300–3317.
- [24] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 313–325.
- [25] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*. 145–160.
- [26] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. 2015. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 166–177.
- [27] Biniam Fisseha Demissie and Mariano Ceccato. 2020. Security testing of second order permission re-delegation vulnerabilities in android apps. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 1–11.

- [28] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493.
- [29] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–516.
- [30] Xiaoming Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 60–71.
- [31] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the sixth conference on computer systems*. 215–228.
- [32] Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAndroid: statically detecting ordering violations in Android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 62–74.
- [33] Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, and Zhendong Su. 2010. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 175–190.
- [34] Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, Gaurav Soman, Akka Zemmari, and Manoj Singh Gaur. 2020. Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation. *Digital Threats: Research and Practice* 1, 2 (2020), 1–25.
- [35] Aayush Garg, Renzo Degiovanni, Matthieu Jimenez, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2022. Learning from what we know: How to perform vulnerability prediction using noisy historical data. *Empirical Software Engineering* 27, 7 (2022), 169.
- [36] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 734–750.
- [37] Sishuai Gong, Deniz Altınbükən, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 66–83.
- [38] Sigmund Albert Gorski III and William Enck. 2019. Arf: identifying re-delegation vulnerabilities in android system services. In *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*. 151–161.
- [39] Shikai Guo, He Jiang, Zhihao Xu, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Rong Chen. 2022. Detecting Simulink compiler bugs via controllable zombie blocks mutation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1061–1072.
- [40] Yiyuan Guo, Peisen Yao, and Charles Zhang. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 63–75.
- [41] Yuqi Guo, Shihao Zhu, Yan Cai, Liang He, and Jian Zhang. 2024. Reorder Pointer Flow in Sound Concurrency Bug Prediction. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
- [42] Roei Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 118–128.
- [43] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. Cp-detector: Using configuration-related performance properties to expose performance bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 623–634.
- [44] Jingzhu He, Ting Dai, and Xiaohui Gu. 2018. Tscope: Automatic timeout bug identification for server systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 1–10.
- [45] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. 2021. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2229–2242.
- [46] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*. 596–607.
- [47] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing injection vulnerabilities by leveraging existing tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 284–296.
- [48] Shuang Hu, Baojian Hua, Lei Xia, and Yang Wang. 2022. Crust: Towards a unified cross-language program analysis framework for rust. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 970–981.
- [49] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. 2021. ACHyb: A hybrid analysis approach to detect kernel access control vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 316–327.

- [50] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. 2021. Rupair: towards automatic buffer overflow detection and rectification for rust. In *Proceedings of the 37th Annual Computer Security Applications Conference*. 812–823.
- [51] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 709–722.
- [52] Seungho Jeon and Jongsub Moon. 2022. Dr. PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search. *Neural Computing and Applications* 34, 13 (2022), 10731–10750.
- [53] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. 2018. The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques. In *2018 IEEE third international conference on data science in cyberspace (DSC)*. IEEE, 53–60.
- [54] Yetao Jia, Honglin Zhuang, Zhechao Lin, and Yangyang Meng. 2021. Machine learning for software vulnerability analysis: A survey. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 396–402.
- [55] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: detecting dependency bugs to prevent compatibility failures. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 86–98.
- [56] Rahul Johari and Pankaj Sharma. 2012. A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. In *2012 international conference on communication systems and network technologies*. IEEE, 453–458.
- [57] Saurabh Joshi, Shuvendu K Lahiri, and Akash Lal. 2012. Underspecified harnesses and interleaved bugs. *ACM SIGPLAN Notices* 47, 1 (2012), 19–30.
- [58] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 155–170.
- [59] Wooseok Kang, Byounguh Son, and Kihong Heo. 2022. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1695–1708.
- [60] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 582–598.
- [61] Jasleen Kaur, Urvashi Garg, and Gourav Bathla. 2023. Detection of cross-site scripting (XSS) attacks using machine learning techniques: a review. *Artificial Intelligence Review* 56, 11 (2023), 12725–12769.
- [62] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 222–233.
- [63] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2020. Finding bugs in file systems with an extensible fuzzing framework. *ACM Transactions on Storage (TOS)* 16, 2 (2020), 1–35.
- [64] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2022. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia conference on computer and communications security*. 917–930.
- [65] Maryna Kluban, Mohammad Mannan, and Amr Youssef. 2024. On Detecting and Measuring Exploitable JavaScript Functions in Real-world Applications. , 37 pages. <https://doi.org/10.1145/3630253>
- [66] Julia Lawall and David Lo. 2010. An automated approach for finding variable-constant pairing bugs. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 103–112.
- [67] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S McKinley. 2010. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 36–49.
- [68] Baptiste Leper, Josselin Giet, Willy Zwaenepoel, and Julia Lawall. 2023. OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 33–45.
- [69] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.
- [70] Penghui Li and Wei Meng. 2021. Lchecker: Detecting loose comparison bugs in php. In *Proceedings of the Web Conference 2021*. 2721–2732.
- [71] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 268–279.
- [72] Siliang Li and Gang Tan. 2009. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*. 442–452.

- [73] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1659.
- [74] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.
- [75] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544.
- [76] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering*. 333–344.
- [77] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2469–2485.
- [78] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. 2018. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th annual computer security applications conference*. 529–541.
- [79] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 677–691.
- [80] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices* 53, 2 (2018), 419–431.
- [81] Shigang Liu, Guanjun Lin, Lizhen Qu, Jun Zhang, Olivier De Vel, Paul Montague, and Yang Xiang. 2020. CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. *IEEE Transactions on Dependable and Secure Computing* 19, 1 (2020), 438–451.
- [82] Sihang Liu, Korakit Seemakupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1187–1202.
- [83] Yinxi Liu and Wei Meng. 2023. DSFuzz: Detecting Deep State Bugs with Dependent State Exploration. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1242–1256.
- [84] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.
- [85] Zhuotao Liu, Hao Zhao, Sainan Li, Qi Li, Tao Wei, and Yu Wang. 2021. Privilege-escalation vulnerability discovery for large-scale RPC services: Principle, design, and deployment. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 565–577.
- [86] Jie Lu, Haofeng Li, Chen Liu, Lian Li, and Kun Cheng. 2022. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2145–2158.
- [87] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 229–240.
- [88] Yifei Lu, Minxue Pan, Yu Pei, and Xuandong Li. 2022. Detecting resource utilization bugs induced by variant lifecycles in Android. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 642–653.
- [89] Changhua Luo, Penghui Li, and Wei Meng. 2022. 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* (Los Angeles, CA, USA) (CCS ’22). Association for Computing Machinery, New York, NY, USA, 2175–2188. <https://doi.org/10.1145/3548606.3559391>
- [90] Yu Luo, Weifeng Xu, and Dianxiang Xu. 2022. Compact abstract graphs for detecting code vulnerability with GNN models. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 497–507.
- [91] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. 2020. Impact analysis of cross-project bugs on software ecosystems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 100–111.
- [92] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications (*OOPSLA 2015*). Association for Computing Machinery, New York, NY, USA, 505–519. <https://doi.org/10.1145/2814270.2814272>

- [93] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholtz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 701–712.
- [94] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 531–544.
- [95] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*. 63–74.
- [96] Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2021. Bran: Reduce vulnerability search space in large open source repositories by learning bug symptoms. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 731–743.
- [97] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. 2019. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 176–187.
- [98] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. 2024. Y uga: Automatically Detecting Lifetime Annotation Bugs in the Rust Language. *IEEE Transactions on Software Engineering* (2024).
- [99] Yannic Noller, Rody Kersten, and Corina S Pásăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 322–332.
- [100] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 616–628.
- [101] Léopold Ouairy, Hélène Le Bouder, and Jean-Louis Lanet. 2020. Confiance: detecting vulnerabilities in Java Card applets. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 1–10.
- [102] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 25–36.
- [103] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 426–437.
- [104] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2155–2168.
- [105] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 406–415.
- [106] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 225–236.
- [107] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.
- [108] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarciooglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.
- [109] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.
- [110] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. *ACM SIGPLAN Notices* 53, 2 (2018), 377–391.
- [111] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding symbolic execution toward vulnerable states through pattern learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. 456–468.
- [112] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 587–600.

- [113] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 537–549.
- [114] Ji Shi, Wei Zou, Chao Zhang, Lingxiao Tan, Yanyan Zou, Yue Peng, and Wei Huo. 2022. CAMFuzz: Explainable Fuzzing with Local Interpretation. *Cybersecurity* 5, 1 (2022), 17.
- [115] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. 2019. MoSSOT: An automated blackbox tester for single sign-on vulnerabilities in mobile applications. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 269–282.
- [116] Ling-Yun Situ, Zhi-Qiang Zuo, Le Guan, Lin-Zhang Wang, Xuan-Dong Li, Jin Shi, and Peng Liu. 2021. Vulnerable region-aware greybox fuzzing. *Journal of Computer Science and Technology* 36 (2021), 1212–1228.
- [117] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. 1–13.
- [118] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on asia conference on computer and communications security*. 749–763.
- [119] Antonn Steinhauer and Petr Tma. 2020. Database traffic interception for graybox detection of stored and context-sensitive XSS. *Digital Threats: Research and Practice* 1, 3 (2020), 1–23.
- [120] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31.
- [121] Kazi Zakia Sultana, Charles B Boyd, and Byron J Williams. 2023. A software vulnerability prediction model using traceable code patterns and software metrics. *SN Computer Science* 4, 5 (2023), 599.
- [122] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. 2019. Reasoning about the node.js event loop using async graphs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 61–72.
- [123] Hao Sun, Xiangyu Zhang, Chao Su, and Qingkai Zeng. 2015. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 483–494.
- [124] Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S Păsăreanu. 2024. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 618–647.
- [125] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S Păsăreanu. 2021. Syrust: automatic testing of rust libraries with semantic-aware program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 899–913.
- [126] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd international conference on software engineering*. 11–20.
- [127] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 198–208.
- [128] Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 189–199.
- [129] Kaleem Ullah, Imran Rashid, Hammad Afzal, Mian Muhammad Waseem Iqbal, Yawar Abbas Bangash, and Haider Abbas. 2020. SS7 vulnerabilities—a survey and implementation of machine learning vs rule based filtering for detection of SS7 network attacks. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 1337–1371.
- [130] Enze Wang, Jianjun Chen, Wei Xie, Chuhan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. 2024. Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 216–216.
- [131] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 999–1010.
- [132] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 718–728.
- [133] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. 2015. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st annual computer security applications conference*. 1–10.

- conference. 61–70.
- [134] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 434–446.
 - [135] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 708–719.
 - [136] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 628–639.
 - [137] Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, et al. 2022. Characterizing and detecting bugs in wechat mini-programs. In *Proceedings of the 44th International Conference on Software Engineering*. 363–375.
 - [138] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
 - [139] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. 2023. DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2249–2261.
 - [140] Yingjie Wang, Xing Liu, Weixuan Mao, and Wei Wang. 2019. Dcdroid: Automated detection of ssl/tls certificate verification vulnerabilities in android apps. In *Proceedings of the ACM Turing Celebration Conference-China*. 1–9.
 - [141] Ying-Jie Wang, Liang-Ze Yin, and Wei Dong. 2021. Amchex: Accurate analysis of missing-check bugs for linux kernel. *Journal of Computer Science and Technology* 36 (2021), 1325–1341.
 - [142] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M Zhang, and Qing Liao. 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2275–2286.
 - [143] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *Proceedings of the ACM Web Conference 2022*. 755–766.
 - [144] Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1371–1383.
 - [145] Lei Xia, Baojian Hua, and Yang Wang. 2023. Acorn: Towards a Holistic Cross-Language Program Analysis for Rust. (2023).
 - [146] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript performance code smells relevant to type mutations. In *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30–December 2, 2015, Proceedings 13*. Springer, 335–355.
 - [147] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 859–874.
 - [148] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387.
 - [149] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 121–132.
 - [150] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
 - [151] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated bug hunting with data-driven symbolic root cause analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 320–336.
 - [152] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 499–510.
 - [153] Fengyu Yang, Fa Zhong, Guangdong Zeng, Peng Xiao, and Wei Zheng. 2024. LineFlowDP: A Deep Learning-Based Two-Phase Approach for Line-Level Defect Prediction. *Empirical Software Engineering* 29, 2 (2024), 50.
 - [154] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 867–879.

- [155] Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. 2022. Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 869–880.
- [156] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. 2022. Cross miniapp request forgery: Root causes, attacks, and vulnerability detection. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3079–3092.
- [157] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 488–499.
- [158] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1127–1139.
- [159] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. 2021. Detecting concurrency vulnerabilities based on partial orders of memory and thread events. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 280–291.
- [160] Ming Yuan, Yeseop Lee, Chao Zhang, Yun Li, Yan Cai, and Bodong Zhao. 2021. Raproducer: Efficiently diagnose and reproduce data race bugs for binaries via trace analysis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 593–606.
- [161] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel (*ESEC/FSE 2020*). Association for Computing Machinery, New York, NY, USA, 221–232. <https://doi.org/10.1145/3368089.3409686>
- [162] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707.
- [163] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 763–773.
- [164] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Chunming Hu, and Yang Liu. 2023. Detecting condition-related bugs with control flow graph neural network. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1370–1382.
- [165] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: detecting concurrency bugs through sequential errors. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 251–264.
- [166] Wei Zhang, Chong Sun, Junghee Lim, Shan Lu, and Thomas Reps. 2013. Conmem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 1–33.
- [167] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, and Na Meng. 2022. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 190–201.
- [168] Xiaoqi Zhao, Haipeng Qu, Jianliang Xu, Xiaohui Li, Wenjie Lv, and Gai-Ge Wang. 2024. A systematic review of fuzzing. *Soft Computing* 28, 6 (2024), 5493–5522.
- [169] Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. 2013. WuKong: automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 131–142.
- [170] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-distinguishable inconsistencies as a deterministic oracle for detecting security bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3253–3267.
- [171] Chenyang Zhu, Zhengwei Zhu, Yunxin Xie, Wei Jiang, and Guiling Zhang. 2019. Evaluation of machine learning approaches for android energy bugs detection with revision commits. *IEEE Access* 7 (2019), 85241–85252.
- [172] Yuhui Zhu, Guanjun Lin, Lipeng Song, and Jun Zhang. 2023. The application of neural network for software vulnerability detection: a review. *Neural Computing and Applications* 35, 2 (2023), 1279–1301.
- [173] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 799–813.
- [174] Fei Zuo and Junghwan Rhee. 2024. Vulnerability discovery based on source code patch commit mining: a systematic literature review. *International Journal of Information Security* 23, 2 (2024), 1513–1526.