Continuous Static Assurance and Symbolic Execution for Secure Software Supply Chains

Konstantinos Giapantzis, Apostolis Zarras, Katerina Psychogyiou, Christos Xenakis
University of Piraeus, Greece

Abstract

Software supply chains increasingly depend on third-party and open-source components, yet these dependencies are often propagated without continuous, verifiable security evidence. To address this assurance gap, this work enhances supply-chain cybersecurity by integrating continuous static code security testing as a core component of a Trusted Bill of Materials (TBOM). Third-party and open-source components are treated as managed assets that must be accompanied by verifiable security evidence before they can be trusted and propagated through the chain. To operationalize this principle, the proposed Static Application Security Tester (SASTER) pipeline orchestrates multiple open-source static vulnerability analyzers. By combining complementary rule-based and pattern-matching techniques, SASTER enhances detection accuracy while substantially reducing false positives, a key obstacle to the adoption of static analysis in practice. The pipeline further integrates sensitive-data classification, ensuring that privacy risks and the misuse of confidential information are addressed as top-priority security findings early in development. To complement automated scanning, this work proposes the Intelligent Vulnerabilities Exposure Engine, a symbolic-execution-driven framework that can prove or refute embedded program properties and produce counterexamples when violations occur. Each assessment yields a Static Supply Chain Component Guarantee, which is required as a TBOM element and is regenerated whenever a component changes, enabling evidence-based lifecycle assurance.

1 Introduction

Modern software supply chains increasingly depend on dense, fast-evolving ecosystems of third-party and open-source components [1, 2]. In such ecosystems, a single vulnerable dependency can silently cascade into hundreds of downstream products [3]. Yet many components are introduced with little or no trustworthy security evidence (i.e., their code quality, vulnerability history, and patch status are often opaque to integrators, who may only observe a version string and a license) [4]. This opacity is especially risky as adversaries shift attention toward supply-chain vectors, such as dependency confusion, malicious package uploads, and subtle logic flaws that evade superficial checks [5]. Meanwhile, the scale and heterogeneity of modern systems (i.e., spanning multiple languages, building pipelines, and microservice architectures under continuous integration) make manual review impractical and render ad-hoc scanning brittle [6]. Consequently, organizations struggle to maintain a reliable, time-sensitive understanding of which components are vulnerable, the severity of those vulnerabilities, and whether a given artifact is trustworthy at a specific point in its lifecycle.

Various approaches have been proposed to mitigate these security challenges and improve the resilience and reliability of software supply chains against emerging threats [7]. One prominent direction is Cybersecurity Supply Chain Risk Management (C-SCRM), an emerging management construct that combines approaches, methods, and practices from cybersecurity, enterprise risk management, and supply chain management [8]. Within this context, National Institute of Standards and Technology (NIST) provides detailed guidance for managing supply chain cybersecurity risk, emphasizing governance structures, risk assessment, and controls over supplier and third-party channels [9]. Similarly, ENISA's guidance advocates a multifaceted approach that secures the software supply chain end-to-end through security-by-design, strong access controls, and robust risk management and documentation practices [10]. Complementing these frameworks, ISO/IEC 27036 focuses on strengthening supplier relationships at a global level, highlighting risk considerations throughout procurement and lifecycle activities [11]. Taken together, these initiatives provide a coherent foundation for managing systemic exposure in increasingly complex software environments.

Despite these advancements, many supply-chain security measures remain insufficient against sophisticated attack strategies that exploit vulnerabilities in widely used software components [12]. This gap has been repeatedly underscored by thousands of documented supply-chain incidents,

including several high-impact cases. In 2017, attackers compromised the build process of the popular CCleaner utility [13] and shipped a backdoored, vendor-signed release (v5.33) that was distributed as legitimate software, impacting an estimated 2.27 million users [14]. Three years later, the SolarWinds incident [15] illustrated how deeply a poisoned update channel can penetrate: adversaries inserted a backdoor into the Orion platform, modified the vendor's software, and induced victims to install it through routine updates, thereby obtaining privileged access within otherwise well-defended organizations. In 2021, a critical Remote Code Execution (RCE) vulnerability in Apache Log4j (Log4Shell) [16] escalated into a global supply-chain crisis, reportedly affecting 93% of enterprise cloud environments [17] and appearing as a direct dependency in roughly 70,000 open-source projects [18]. More recently, in 2024, a malicious actor gained maintainer-level trust in the XZ Utils project and surreptitiously introduced a backdoor with the potential to enable RCE via OpenSSH on many Linux systems; it was discovered only shortly before widespread distribution [19]. In 2025, Amazon Inspector researchers reported more than 150,000 npm packages linked to a coordinated token-farming campaign [20], representing one of the largest package-flooding incidents in the history of open-source registries. Collectively, these cases show how financially motivated attackers are scaling registry pollution and component compromise, and they underscore the need for stronger technical controls alongside sustained collaboration between industry and open-source communities to defend the software supply chain.

To address these challenges, we introduce two complementary state-of-the-art static analysis tools, Static Application Security Tester (SASTER) and Intelligent Vulnerabilities Exposure Engine (IVEE), designed to strengthen vulnerability detection and assurance across the software supply chain. SASTER aggregates multiple open-source static vulnerability analyzers within a unified, modular framework and normalizes their outputs into a common schema, yielding a single coherent security view per component. This enables rule-based and pattern-matching techniques to be applied consistently across diverse languages and codebases. In parallel, IVEE extends the pipeline from broad warning generation to deep property checking by applying symbolic execution at the binary level. By exploring feasible execution paths in compiled artifacts, IVEE can either prove that explicit security properties hold or produce concrete counterexamples when violations occur. Together, SASTER and IVEE transform pattern-based findings into property-oriented, binary-level evidence, providing end-to-end assurance that conventional static analyzers alone typically cannot achieve.

In summary, we make the following main contributions:

- **A unified, modular orchestrator for static code analysis.** We design and implement SASTER, a static-analysis orchestrator that integrates multiple open-source vulnerability analyzers within a single, modular framework. SASTER normalizes heterogeneous tool outputs into a common representation (SARIF), supports multi-language projects, and produces version-specific, component-level security reports that can be embedded into a Trusted Bill of Materials (TBOM) as structured, reusable evidence.

- **A symbolic-execution-driven framework for binary-level property checking.** We introduce IVEE, a verification framework operating at the binary level that complements source-based static analysis. IVEE systematically explores feasible execution paths in compiled artifacts, checks explicit security properties, and either proves that they hold (for the explored paths) or generates concrete counterexamples when violations occur, thereby turning potential weaknesses into demonstrable, property-oriented findings.

- **An end-to-end pipeline for supply-chain-oriented security guarantees.** We combine SASTER and IVEE into a coordinated analysis pipeline that treats vulnerability assessment as a continuous process rather than an isolated scan. The pipeline links multi-tool source analysis with binary-level verification and consolidates outcomes into Static Supply Chain Component Guarantees (SSCGs), enabling lifecycle-aware, evidence-based assurance for software components that can be consumed by C-SCRM and TBOM-driven supply-chain risk-management processes.

The remainder of this chapter is organized as follows. Section 2 introduces the necessary background on Trusted Bills of Materials, Static Supply Chain Component Guarantees, and the underlying analysis techniques. Section 3 presents our static-assurance pipeline, detailing the design and implementation of SASTER and IVEE. Section 4 discusses current limitations and outlines directions for future work. Section 5 reviews related work on static analysis, symbolic execution, and hybrid approaches for vulnerability detection, while Section 6 concludes the chapter.

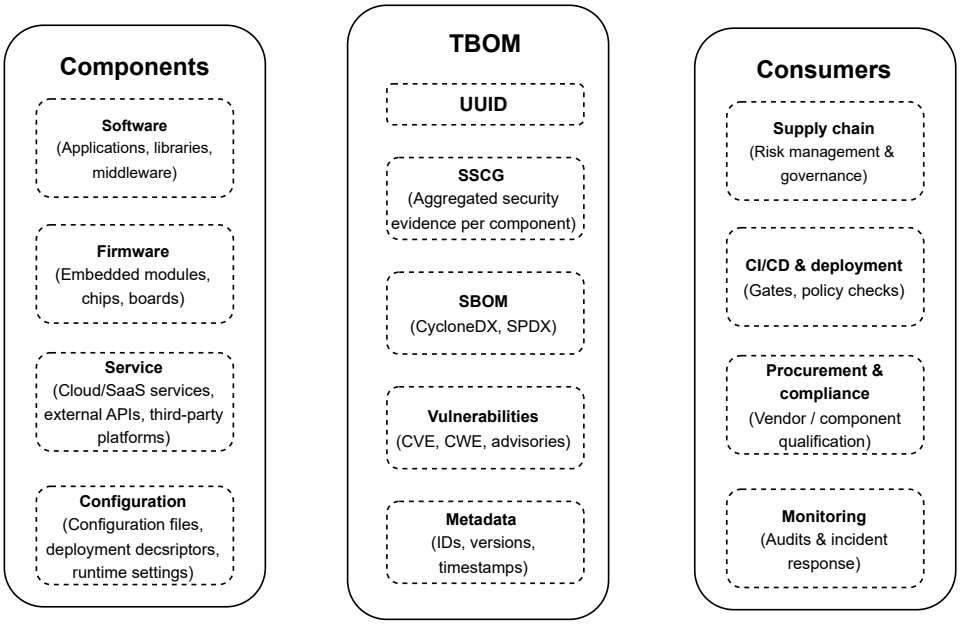

Figure 1: Trusted Bill of Material architecture

2 Background

This section presents the necessary background for our work. We begin by introducing the concept of a TBOM, which extends traditional Bill of Materials (BOM) formats to include structured security and trust information for software components. Next, we describe the SSCG as a machine-processable attestation that encapsulates the results of static analysis and formal verification at the component level. Finally, we review the core principles of static code analysis and symbolic code execution, the foundational technologies that generate the security evidence embedded in SSCGs and subsequently aggregated within the TBOM.

2.1 Trusted Bill of Materials

A TBOM is a structured security-and-trust artifact that extends the traditional BOM concept beyond a mere component list, as depicted in Figure 1 [21]. Whereas an Software Bill of Materials (SBOM) primarily records which software or firmware components are present in a product [22], a TBOM couples this inventory with machine-readable security evidence, vulnerability references, and assurance data. Specifically, a TBOM links to

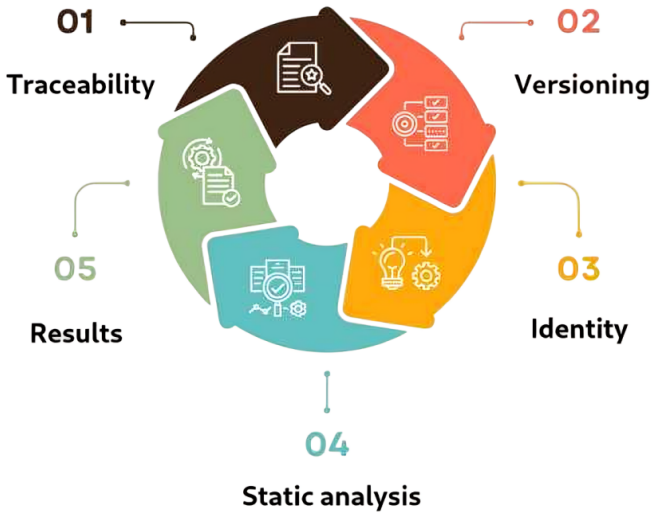

Figure 2: Essential elements of static supply chain component guarantee

the underlying SBOM and enriches it with (*i*) detailed metadata (e.g., producer/consumer identities, lifecycle state, and versioning information), (*ii*) references to known vulnerabilities, and (*iii*) embedded assurance artifacts such as SSCG that summarize static security testing results for each component. In practice, a TBOM is typically realized by extending an existing BOM format such as CycloneDX [23] or System Package Data Exchange (SPDX) [24] with additional sections for trust assurance, testing evidence, and extensible security attributes. As a result, a consumer can evaluate not only what a system contains, but also how, and to what extent, its components have been assessed. Overall, the TBOM functions as a central, verifiable evidence artifact that unifies inventory, security posture, and provenance, thereby enabling automated supply-chain risk management across the software lifecycle.

2.2 Static Supply Chain Component Guarante

A SSCG is a machine-processable security attestation that binds a software component to the results of static analysis and formal verification performed on a specific version of that asset [25]. It is generated after a dedicated analysis run and records what was assessed, how it was assessed, and by whom. As shown in Figure 2, an SSCG captures (*i*) a unique identifier and timestamp for traceability, (*ii*) the exact component version under evaluation,

(*iii*) the identity of the testing entity, (*iv*) the executed static-analysis and verification procedures, and (*v*) the resulting vulnerabilities and weaknesses, including severity, in a structured and machine-consumable form. It also includes environment and configuration metadata (e.g., toolchain versions, execution platform, and analysis parameters) to support reproducibility and longitudinal comparison of assessments [26]. In doing so, the SSCG elevates tool-specific logs into a standardized, component-level unit of security evidence that can be attached to supply-chain artifacts and consumed by downstream risk-management workflows.

Structurally, a SSCG is a signed attestation document that can be represented as a specialization or extension of established bill-of-materials formats such as CycloneDX [23]. Its core section consolidates the normalized outputs of one or more analyzers together with the associated metadata (e.g., tool invocations, targets, and configurations). Optional extensions enable the producer to reference specific security or quality standards and declare component requirements. These requirements are linked to *claims* (statements of compliance or violation) and to *evidence* (concrete, verifiable artifacts such as normalized SARIF [27] findings or formal-verification results), connected through stable identifiers so that automated tooling can trace which evidence substantiates each claim. This design supports comparability across components, aggregates multiple reports into a unified view, and ensures authenticity and integrity via digital signatures, thereby making the SSCG a reusable and trustworthy unit of security evidence in software supply chains.

2.3 Static Code Analysis

Static code analysis refers to the automated inspection of program artifacts (i.e., most commonly source code and bytecode) without executing them, to identify defects, security vulnerabilities, and violations of coding or compliance rules early in the software lifecycle [28]. In practice, static-analysis tools construct and traverse abstract program representations such as abstract syntax trees, control-flow graphs, and data-flow or taint models. These representations are evaluated against rules derived from language semantics, secure-coding guidelines, and vulnerability taxonomies (e.g., Common Weakness Enumeration (CWE)) to flag potentially unsafe patterns, suspicious data flows, or risky API usages [29]. Contemporary approaches span a spectrum from lightweight, rule-based scanners that match syntactic patterns, through interprocedural analyses that reason about information flow across functions and modules, to more formal techniques such as model

checking and abstract interpretation, which approximate program behavior [30].

In security-oriented settings, findings are often normalized and linked to external vulnerability knowledge bases (e.g., by mapping warnings to CWE categories and correlating them with known Common Vulnerabilities and Exposuress (CVEs) in affected components), which supports prioritization and integration with broader vulnerability-management processes [31]. Consequently, static code analysis has evolved from a compiler-adjacent technique into a core element of industrial secure development practices, routinely embedded in integrated development environment (IDE)s and Continuous Integration and Continuous Delivery (CI/CD) pipelines to provide systematic, repeatable feedback on vulnerability-prone constructs and to reduce the likelihood that exploitable weaknesses propagate into deployed systems and, ultimately, the software supply chain [32].

2.4 Symbolic Code Execution

Symbolic execution is a program-analysis technique in which program inputs are modeled as symbolic variables rather than concrete values, enabling many potential executions to be reasoned about within a single analysis run [33]. Rather than executing a function with fixed inputs, the analysis begins with symbolic inputs and tracks how expressions over these symbols are computed along each explored path [34]. At conditional branches, execution conceptually splits, and the analysis maintains for each branch a *path condition*: a logical formula over the symbolic inputs that must hold for the path to be feasible. The resulting collection of symbolic states and transitions forms a symbolic-execution tree whose leaves are annotated with path conditions characterizing the input values that reach those points. By discharging a path condition to a constraint solver, one can derive concrete test inputs that exercise a selected path, allowing symbolic execution to systematically generate tests that expose assertion violations, divide-by-zero errors, buffer overflows, and other runtime faults with high structural coverage [35].

From a foundational perspective, symbolic execution can be viewed as interpreting a program over a symbolic domain while accumulating constraints that capture the effects of branch predicates, arithmetic operations, and memory accesses [33]. While basic formulations operate on a single function or program fragment, practical engines extend the model to handle procedures, heap-allocated data, and complex control structures, often combining symbolic reasoning with concrete execution, environment models, or library summaries [36]. A central challenge is scalability: the number of fea-

sible paths can grow exponentially with program size, and path conditions may become large and complex to solve, leading to path explosion and solver bottlenecks. To mitigate these issues, modern approaches employ targeted search strategies, heuristics for state pruning and merging, and selective application of symbolic execution to security-critical regions identified by complementary analyses, enabling deep path-sensitive reasoning where it is most valuable while remaining computationally tractable [35].

3 Static Assurance and Symbolic Execution

This section presents the static-assurance pipeline and its two core building blocks. We first propose SASTER, a multi-analyzer static application security testing framework that aggregates open-source tools spanning more than thirty programming languages and exposes them through a unified Graphical User Interface (GUI)/Command Line Interface (CLI) workflow. We then introduce IVEE, a binary-oriented engine based on symbolic execution that performs property-driven analysis directly on compiled executables. Together, these components illustrate how broad, source-level vulnerability scanning and deep, path-sensitive binary verification can be combined into a coordinated workflow for generating SSCG evidence.

3.1 Static Application Security Tester

Static Application Security Tester (SASTER) is an application that centralizes static application security testing by orchestrating multiple open-source analyzers through a single, unified workflow. Rather than requiring developers to install, configure, and execute each tool independently, SASTER provides a single entry point for scanning codebases written in languages such as Python, Java, JavaScript, C, and C++, among others, using the analyzers listed in Table 1. All findings are normalized into a common results format (SARIF), which facilitates downstream consumption by other tools, long-term storage as security evidence, and integration into supply-chain artifacts such as SSCGs and TBOMs. Overall, SASTER reduces the operational friction of multi-tool static analysis and promotes consistent, repeatable security assessments across heterogeneous projects.

Table 1 summarizes the open-source static analyzers integrated into SASTER and the programming language families they support. Each row corresponds to an analyzer and each column to a language family, illustrating that most tools are specialized to a limited subset of ecosystems (e.g.,

Table 1: Open-source analyzers integrated with SASTER

Analyzers	C/C++	Python	Java	JavaScript	Go	PHP	Ruby	Rust	Other
Flawfinder [37]	✓	✗	✗	✗	✗	✗	✗	✗	✗
Semgrep [38]	✓	✓	✓	✓	✓	✓	✓	✓	✓
Bandit [39]	✗	✓	✗	✗	✗	✗	✗	✗	✗
Bearer [40]	✗	✓	✓	✓	✓	✓	✓	✗	✗
Horusec [41]	✓	✓	✓	✓	✓	✓	✓	✗	✗
Cppcheck [42]	✓	✗	✗	✗	✗	✗	✗	✗	✗
SASTER	✓	✓	✓	✓	✓	✓	✓	✓	✓

Flawfinder and Cppcheck for C/C++, Bandit for Python). In contrast, Semgrep, Bearer, and Horusec provide broader multi-language coverage. By composing these analyzers within a single workflow, SASTER effectively combines their capabilities and achieves coverage across all listed language categories. Overall, it supports more than 30 programming languages, including Apex, Bash, C, C++, C#, Clojure, Dart, Dockerfile, Elixir, HTML, Go, Java, JavaScript, JSX, JSON, Julia, Jsonnet, Kotlin, Lisp, Lua, OCaml, PHP, Python, R, Ruby, Rust, Scala, Scheme, Solidity, Swift, Terraform, TypeScript, TSX, YAML, XML, as well as generic templating languages (ERB, Jinja, etc.).

SASTER provides a GUI that serves as an intuitive, centralized environment for managing and analyzing source code, as depicted in Figure 3. Through this web interface, users can authenticate, create projects, upload code, and launch static analyses without needing to interact directly with individual tools. Based on the uploaded artifacts, SASTER automatically detects the relevant programming languages and selects appropriate analyzers from Table 1. Results are presented in a 0 that organizes findings by project and severity, enabling users to quickly review reported vulnerabilities and navigate to the corresponding code locations. The GUI additionally enforces user-based access control, allowing multiple users or teams to share a deployment while maintaining project isolation and protecting intellectual property.

Under the hood, SASTER follows a modular, containerized architecture that decouples the web interface from the analysis logic. A Flask-based backend handles authentication and session management, orchestrates projects and analysis jobs, persists configuration in a database, and exposes RESTful endpoints to the front end. Each supported analyzer is packaged as an independent Docker module, comprising a Dockerfile that defines the execution environment, a loader script that registers the tool, an analysis script that executes the scan with suitable parameters, and a result handler that collects

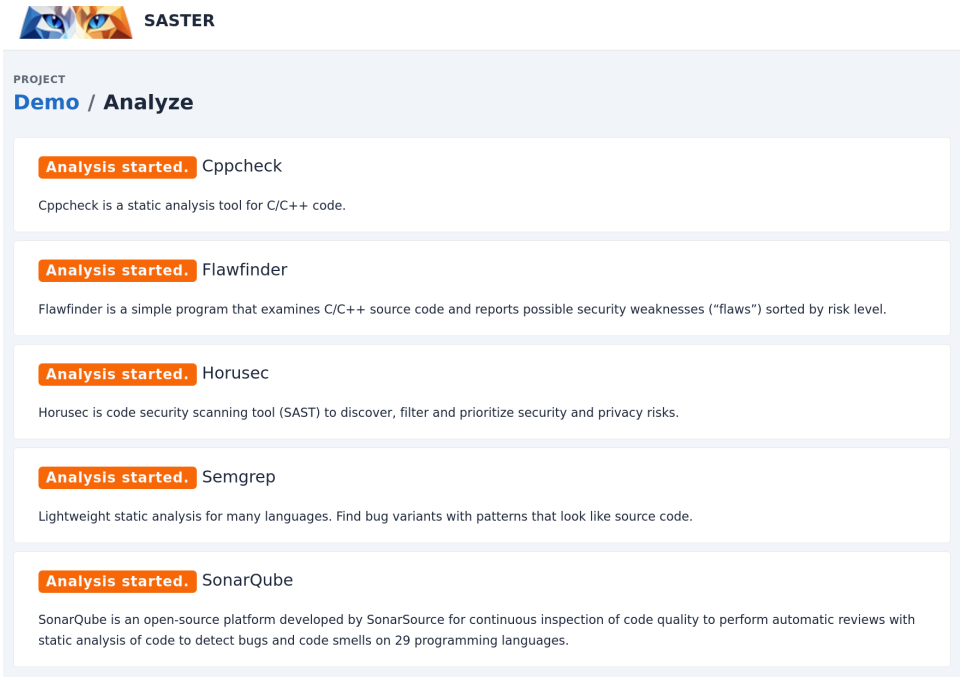

Figure 3: SASTER’s graphical interface while an analysis is running

and post-processes the output. Where necessary, native tool outputs (typically JSON or XML) are converted into the SARIF standard, allowing heterogeneous findings to be aggregated and consumed uniformly. This design facilitates extension with additional analyzers, ensures reproducible executions across environments, and supports integration with external workflows such as CI/CD and supply-chain evidence generation. Figure 4 presents the results interface, where users can inspect identified vulnerabilities after analysis is complete; findings are displayed in a structured form that highlights the type and severity to support assessment and prioritization.

Beyond web-based deployment, SASTER also includes a lightweight CLI designed for developers and CI/CD pipelines that require scriptable interactions. Unlike the GUI-based workflow, the current offline SASTER-CLI operates entirely on the local machine. A small command processor reads the project configuration, dispatches the relevant analyzers via a dedicated execution engine, and records progress and errors through an integrated logging subsystem, while supporting the concurrent processing of multiple files and languages within a single scan.

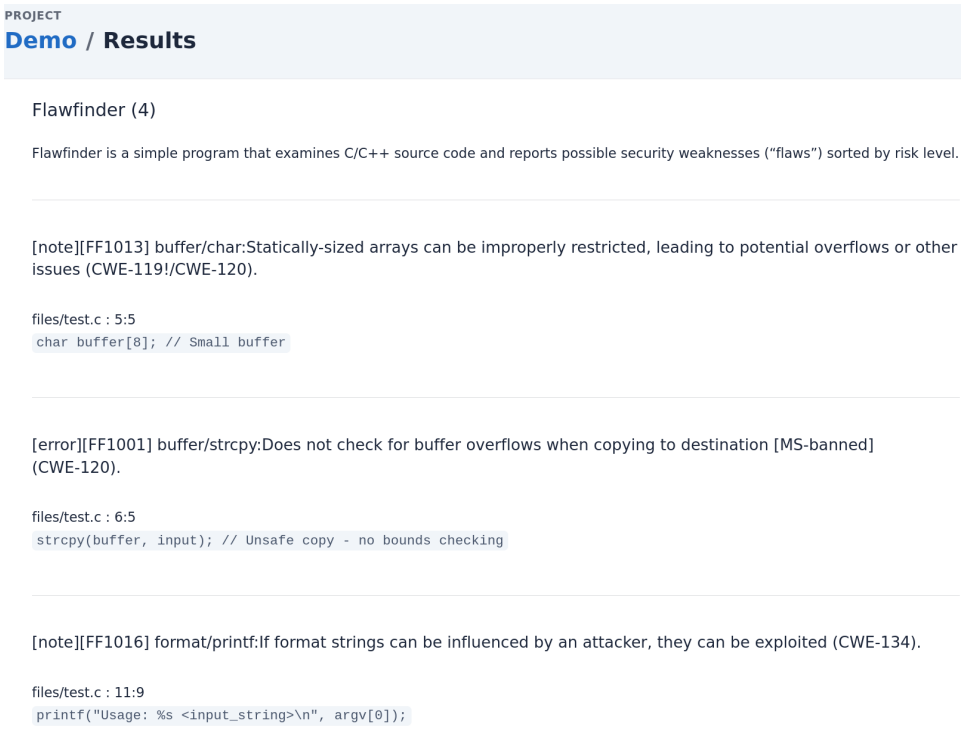

Figure 4: SASTER’s graphical interface displaying the analysis results

The CLI mirrors the platform’s modular design by treating each analyzer as a pluggable module and emitting results using the same schema as the GUI workflow, allowing outputs to be forwarded into evidence-generation processes without additional post-processing. For ease of adoption, the CLI is also distributed as a minimal Debian-based Docker image that bundles the tool and its dependencies, enabling consistent invocation across developer workstations, build servers, and other automated environments and thereby simplifying integration into TBOM and SSCG generation workflows. This is illustrated in Figure 5, where the SASTER CLI performs an offline scan using multiple analyzers (including Horusec, Bandit, Flawfinder, and Semgrep) while logging tool versions and configurations for transparency.

3.2 Intelligent Vulnerabilities Exposure Engine

To complement the source-level perspective offered by SASTER, which aggregates multiple static analyzers over program artefacts, we introduce the

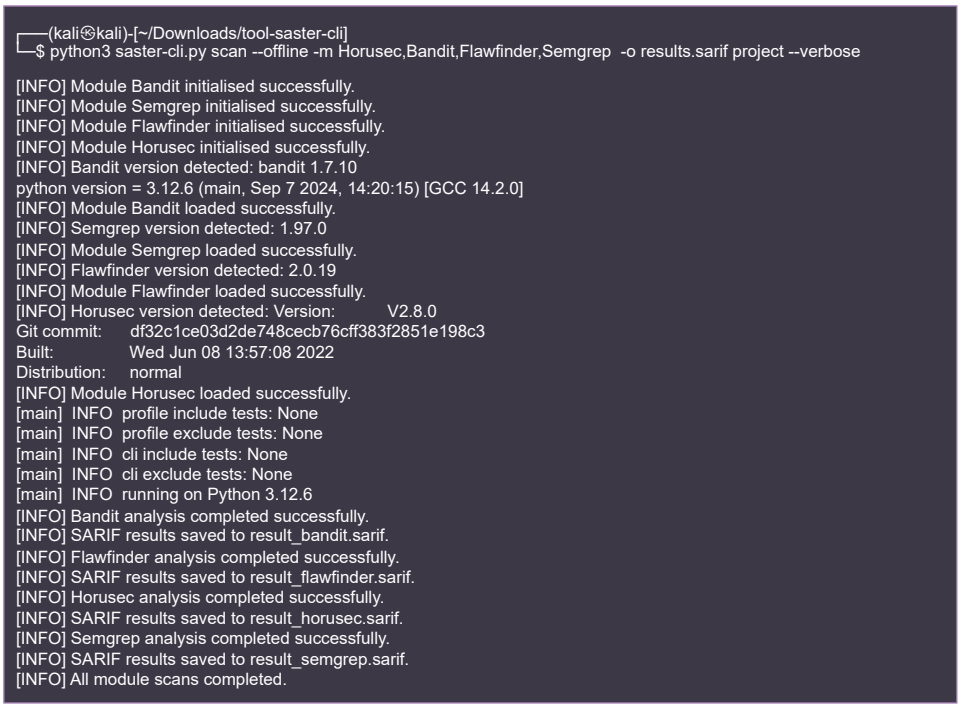

Figure 5: SASTER's CLI executing the static code analysis

Intelligent Vulnerabilities Exposure Engine (IVEE) as a binary-oriented analysis component. Whereas SASTER targets vulnerability patterns in source and intermediate representations, IVEE analyses the compiled executables themselves, enabling property-driven reasoning over the binaries that are ultimately shipped and deployed. Together, these components form a coordinated pipeline in which SASTER provides broad, multi-language coverage at the code level. In contrast, IVEE delivers deep, path-sensitive reasoning at the binary level, strengthening the assurance evidence that underpins SSCG and TBOM-driven supply-chain workflows.

More concretely, IVEE is a binary-focused security analysis tool that addresses limitations of traditional static analyzers by applying symbolic execution directly to compiled executables. Rather than relying on source-level syntactic and semantic checks, it reasons about low-level program behavior in formats such as Executable and Linkable Format (ELF) (Linux) and Portable Executable (PE) (Windows), treating external inputs as symbolic variables and systematically exploring feasible execution paths to determine whether specified security properties hold. Built on top of the angr

framework [43], IVEE leverages angr's symbolic execution engine and simulation manager for path exploration, constraint solving, and counterexample generation. This enables the detection of highly path-dependent vulnerabilities (e.g., buffer overflows, use-after-free, and integer overflows) and the synthesis of concrete inputs that trigger observed property violations. By combining symbolic execution with complementary static analyses, such as Control Flow Graph (CFG) construction, IVEE supports property-oriented assessment of compiled components without requiring access to their source code.

At the core of IVEE is a set of analyses that structure and guide symbolic exploration over complex binaries. Using angr's CFGFast analysis, IVEE builds CFGs that capture branches, loops, and function calls, and then uses these graphs to identify and prioritise regions of interest where vulnerabilities are more likely to arise. Properties are encoded as assertions or constraints that are evaluated along the explored paths. When symbolic execution discovers a feasible path that violates a property, the underlying solver produces a counterexample in the form of concrete input values that drive the binary into the erroneous state, providing actionable evidence about exploitability conditions.

4 Limitations and Future Work

For SASTER, the primary limitations arise from its dependence on existing open-source analyzers and the current maturity of its orchestration workflow. Although it unifies multiple tools and languages behind a single GUI/CLI, its detection capability is ultimately bounded by the rulesets, precision, and coverage of the underlying analyzers. In practice, such analyzers may miss a non-trivial fraction of real-world vulnerabilities, and certain CWE classes remain comparatively underserved. As future work, SASTER will incorporate machine-learning-based post-processing to mitigate these shortcomings by refining and prioritizing analyzer outputs, reducing noise, and providing a more reliable vulnerability signal for downstream consumption in SSCG and TBOM generation.

In contrast, IVEE 's main limitation concerns the scalability of binary-level symbolic execution. The number of feasible paths can grow rapidly with program size and complexity. Constraint solving becomes costly in the presence of indirect control flow and complex data structures, and faithfully modeling libraries, system calls, and operating-system interactions remains a challenging task. Moreover, the current analysis relies on explicitly encoded

properties and harnesses, which constrains coverage when specifications are incomplete, implicit, or informal. Planned enhancements include integrating dynamic analysis to capture runtime-dependent behaviors more effectively and applying machine-learning-based heuristics for path prioritization to scale to larger binaries while preserving the ability to uncover subtle, low-level vulnerabilities.

5 Related Work

A range of techniques has been proposed to detect vulnerabilities that may enter through the software supply chain. One family of approaches performs *pre-ingestion* vetting of third-party and open-source components using static code analysis, aiming to catch recurring weakness patterns before a dependency is integrated. A second family applies symbolic execution to obtain path-sensitive evidence of exploitability, which is particularly useful for memory-corruption vulnerabilities in binaries or firmware where source code is unavailable or incomplete. More recent work explores *hybrid* pipelines that combine these strengths: static analysis is used to localize or prioritize suspicious program regions, and symbolic execution is then targeted at those regions to achieve deeper, path-sensitive exploration while mitigating the scalability limits of unconstrained symbolic analysis.

5.1 Static Code Analysis

A first line of work asks how far static analyzers can go in detecting vulnerabilities on their own. Kaur and Nayyar [44] conduct a comparative study of several open-source static analysis tools for vulnerability detection in C/C++ (Flawfinder [37], Rough Auditing Tool for Security (RATS) [45], Cppcheck [42]) and Java (SpotBugs [46], PMD [47]). Using Juliet v1.3 [48] and Apache Tomcat [49] as benchmarks, they compare tools based on the covered CWE categories and false-positive behavior. Their findings indicate that tool effectiveness is highly uneven across weakness classes, with no single analyzer providing comprehensive coverage across all classes.

Building on benchmark-based evaluation, Alqaradaghi and Kozsik [50] study five free, widely used Java analyzers (Infer [51], SonarQube [52], SpotBugs [46], Find Security Bugs [53], and PMD [47]) on Juliet 1.3 [48], measuring recall, false-alarm rate, precision, and F-score per CWE. They similarly observe incomplete coverage: no tool detects all vulnerability categories, and 13% of the evaluated CWEs are missed by every tool. Overall, these results

reinforce the view that static analysis alone remains insufficient for comprehensive vulnerability coverage, motivating the use of tool combinations and complementary techniques.

Complementing synthetic benchmarks with real-world evidence, Lipp, Banescu, et al. [54] evaluate five analyzers (Flawfinder [37], Cppcheck [42], Infer [51], CodeChecker [55], and CodeQL [56]) on a CVE-based benchmark comprising 27 open-source projects (1.15M LoC) and 192 validated vulnerabilities. They also propose a function-level methodology based on fault–error location conformity and map findings to CWEs for consistent comparison. Despite strong performance on synthetic suites, the studied analyzers miss between 47% and 80% of real-world vulnerabilities, with non-memory-related weaknesses particularly underdetected. While combining tools improves recall by 21–34 percentage points, it also increases the proportion of functions flagged for review by about 15 percentage points, highlighting an inherent trade-off between coverage and triage burden in multi-tool static pipelines.

5.2 Symbolic Code Execution

Where static analysis prioritizes scalability, a second line of work leverages symbolic execution to provide path-sensitive reasoning about program behavior, often targeting binaries or firmware. Jayasena and Mishra [57] introduce FirmWall, a directed symbolic execution framework for Root-of-Trust (RoT) firmware that analyzes security-critical system calls directly at the binary level. FirmWall combines lazy CFG construction [58], system-call-centric regions of interest, path-vector-guided symbolic execution, and Large Language Model (LLM)-assisted translation of natural-language security specifications into formal rules. Evaluated on multiple Advanced RISC Machine (ARM) Trusted Firmware configurations, FirmWall reports improved coverage over traditional symbolic execution and confirms several real-world CVEs, illustrating the potential of directed exploration for pre-deployment firmware validation.

Focusing on heap-related weaknesses, Tu [59] proposes HEAPX, a boosted symbolic execution engine designed to improve detection and exploitation of heap-based vulnerabilities (e.g., buffer overflows and use-after-free). The work identifies three recurring obstacles in symbolic engines (i.e., path explosion, imprecise memory modeling, and unrealistic environment modeling) and addresses them via an unsafe-pointer-oriented exploration strategy, a symbolic-address memory model with concretely mapped symbolic locations, and dynamic infrastructure to support native heap addresses. To-

gether, these techniques bias exploration toward vulnerability-prone behavior and enable the generation and validation of working exploits.

At the executable level, Baradaran et al. [60] present UbSym, a unit-based symbolic execution method for detecting memory corruption vulnerabilities (heap/stack buffer overflow, use-after-free, and double-free) in binaries. UbSym first statically identifies *test units* likely to contain vulnerable statements based on formal specifications, then symbolically executes only these units to derive path and vulnerability constraints. Finally, it employs machine learning to relate unit inputs to system-level inputs, enabling the creation of concrete test cases that trigger the vulnerable instructions. This staged design aims to improve both accuracy and efficiency by restricting symbolic exploration to vulnerability-relevant fragments.

5.3 Hybrid Code Analysis

Bridging scalable yet approximate static analysis and precise yet expensive symbolic execution, hybrid approaches utilize static analysis to focus symbolic exploration on high-value regions or paths. Busse et al. [36] investigate trace-guided dynamic symbolic execution: they take error traces from Clang Static Analyzer [61] and Infer [51], and then guide KLEE [62] with a targeted search heuristic that prioritizes paths consistent with the static trace. Their evaluation yields two negative results that are informative for the community: (*i*) current static analyzers rarely produce non-trivial bugs on benchmarks suitable for KLEE, and (*ii*) for injected bugs, full traces provide limited benefit over using only the reported bug location as a guidance signal. The study, therefore, highlights practical friction points in trace-guided hybridization and motivates the improvement of trace quality and the development of more representative benchmarks.

Targeting firmware analysis, Shafiuzzaman et al. [63] propose STASE, which combines rule-based static analysis with guided symbolic execution to detect and characterize vulnerabilities in Unified Extensible Firmware Interface (UEFI) firmware. Static analysis over LLVM bitcode [64], expressed as reusable Datalog rules, identifies potential vulnerability sites and attacker-controlled inputs; this information is then used to generate symbolic-execution harnesses automatically. STASE produces vulnerability signatures (preconditions, code segment, and postconditions) and aims to reduce both false positives and constraint-solving effort compared to purely static or purely symbolic techniques. Applied to EDKII firmware [65] and PixieFail-style bugs [66], it demonstrates the effectiveness of rule-guided harness generation for firmware vulnerability discovery.

In a similar hybrid spirit, Lacombe et al. [35] present a toolchain that combines static analysis and dynamic symbolic execution to detect fault-injection vulnerabilities in C programs. Their approach uses a Frama-C [67] plugin with dependency analysis to identify and prune fault injection points that cannot affect specified security properties, thereby reducing the analysis perimeter without eliminating relevant attack paths. The remaining injection points are then explored with Lazart [68], a KLEE-based [62] dynamic symbolic execution engine that simulates faults and searches for property-violating executions. Evaluations of FISSC [69], `sudo` [70], and WooKey components [71] demonstrate substantial reductions in injection points and explored paths, while still uncovering realistic single- and multi-fault attacks, thereby improving scalability for fault-oriented security analysis.

6 Conclusion

In this work, we addressed the challenge of enhancing vulnerability detection in software supply chains by integrating comprehensive, multi-tool static analysis with in-depth, binary-level property checking. We introduced SASTER, a modular orchestrator that unifies several open-source static analyzers behind a single GUI/CLI. SASTER supports more than thirty programming languages and normalizes heterogeneous findings into a common, SARIF-based format, enabling developers and CI/CD pipelines to obtain consistent, repeatable source-level security assessments without the overhead of managing individual tools. We complemented this capability with IVEE, a symbolic-execution-driven engine that operates directly on compiled binaries. IVEE systematically explores feasible execution paths against explicit security properties and produces concrete counterexamples when violations occur. This allows it to confirm, refine, or refute suspicions raised by pattern-based analysers and to uncover subtle low-level defects that may evade source-based techniques. Together, SASTER and IVEE form a coordinated pipeline in which multi-analyzer source warnings are paired with path-sensitive binary verification, yielding richer and more defensible evidence of a component's security posture than either approach can provide in isolation.

Acknowledgments

This research has received funding from the European Commission's Horizon Europe research and innovation programs under grant agreement No.

101120962 (RESCALE) and from the European Commission's Digital Europe program under grant agreements No. 101145872 (NITRO) and No. 101128024 (CYBERUNITY). The authors acknowledge the use of Grammarly for language polishing of the manuscript.

References

[1] J. Hejderup, "On the use of tests for software supply chain threats," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pp. 47–49, 2022.

[2] S. Anasuri, "Secure software supply chains in open-source ecosystems," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 1, pp. 62–74, 2023.

[3] L. Williams, G. Benedetti, S. Hamer, R. Paramitha, I. Rahman, M. Tamanna, G. Tystahl, N. Zahan, P. Morrison, Y. Acar, *et al.*, "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–38, 2025.

[4] Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding vulnerabilities in software supply chains," *Empirical Software Engineering*, vol. 30, no. 1, p. 20, 2025.

[5] R. Shariffdeen, B. Hassanshahi, M. Mirchev, A. El Husseini, and A. Roychoudhury, "Detecting python malware in the software supply chain with program analysis," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 203–214, IEEE, 2025.

[6] V. Botta, L. Fusco, A. Mondelli, and I. Visconti, "Secure blockchain-based supply chain management with verifiable digital twins," in *Proceedings of the 2023 ACM Conference on Information Technology for Social Good*, pp. 291–298, 2023.

[7] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1509–1526, IEEE, 2023.

[8] S. Boyson, "Cyber supply chain risk management: Revolutionizing the strategic control of critical it systems," *Technovation*, vol. 34, no. 7, pp. 342–353, 2014.

[9] J. Boyens, A. Smith, N. Bartol, K. Winkler, A. Holbrook, and M. Fallon, "Cybersecurity supply chain risk management practices for systems and organizations," tech. rep., National Institute of Standards and Technology, 2022.

[10] ENISA, "Website." https://www.enisa.europa.eu/sites/default/files/publications/Good%20Practices%20for%20Supply%20Chain%20Cybersecurity.pdf. (Accessed: February 2026).

[11] ISO-IEC, "Website." https://www.iso.org/standard/82905.html. (Accessed: February 2026).

[12] D. Yan, Y. Niu, K. Liu, Z. Liu, Z. Liu, and T. F. Bissyandé, "Estimating the attack surface from residual vulnerabilities in open source software supply chain," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 493–502, IEEE, 2021.

[13] Ccleaner, "Website." https://www.ccleaner.com/. (Accessed: February 2026).

[14] C. Clancy, J. Ferraro, R. Martin, A. Pennington, C. Sledjeski, and C. Wiener, "Deliver uncompromised: Securing critical software supply chains," *MITRE Technical Papers*, vol. 24, no. 01, 2021.

[15] IBM, "Website." https://www.ibm.com/docs/en/randori?topic=2022-solarwinds-orion-cve-2020-10148. (Accessed: February 2026).

[16] Y. S. Fasale, R. R. Halwai, V. S. Galphade, and P. William, "Analysis on log 4j vulnerability and its severity," in *2023 4th International Conference on Computation, Automation and Knowledge Management (ICCAKM)*, pp. 1–5, IEEE, 2023.

[17] Ami Luttwak, Alon Schindel, "Log4shell 10 days later: Enterprises halfway through patching." https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell. (Accessed: February 2026).

[18] Menghan Xiao, "Digging into the numbers one year after log4shell." https://www.scworld.com/feature/digging-into-the-numbers-one-year-after-log4shell. (Accessed: February 2026).

[19] P. Przymus and T. Durieux, "Wolves in the repository: A software engineering analysis of the xz utils supply chain attack," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 91–102, IEEE, 2025.

[20] Chi Tran and Charlie Bacon, "Amazon inspector detects over 150,000 malicious packages linked to token farming campaign." `https://aws.amazon.com/blogs/security/amazon-inspector-detects-over-150000-malicious-packages-linked-to-token-farming-campaign`. (Accessed: February 2026).

[21] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, 2024.

[22] J. Bonacci and R. Martin, "Software bill of materials (sbom) approach to iot security vulnerability assessment," in *International Conference on Information Technology-New Generations*, pp. 57–62, Springer, 2024.

[23] Ecma International, "CycloneDX Bill of Materials Specification." ECMA-424, 1st Edition, 2024. Source: Link: Ecma International https://ecma-international.org/publications-and-standards/standards/ecma-424/.

[24] SPDX, "Overview." `https://spdx.dev/learn/overview/`. (Accessed: February 2026).

[25] D. Ivanov, "What network and performance indicators can tell us about supply chain and sourcing resilience (and what they cannot)," *Journal of Purchasing and Supply Management*, p. 101014, 2025.

[26] N. K. Tran, S. Pallewatta, and M. A. Babar, "An empirically grounded reference architecture for software supply chain metadata management," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pp. 38–47, 2024.

[27] Github, "Sarif support for code scanning." `https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning`. (Accessed: February 2026).

[28] P. Louridas, "Static code analysis," *Ieee Software*, vol. 23, no. 4, pp. 58–61, 2006.

[29] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015.

[30] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević, and S. Ristić, "Static code analysis tools: A systematic literature review," in *Ann. DAAAM Proc. Int. DAAAM Symp*, vol. 31, pp. 565–573, 2020.

[31] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[32] Z. Wadhams, A. M. Reinhold, and C. Izurieta, "Automating static code analysis through ci/cd pipeline integration," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*, pp. 119–125, IEEE, 2024.

[33] C. S. Pasareanu, "Symbolic execution: The basics," in *Symbolic Execution and Quantitative Reasoning: Applications to Software Safety and Security*, pp. 5–20, Springer, 2022.

[34] D. Lind and B. Marcus, *An introduction to symbolic dynamics and coding*. Cambridge university press, 2021.

[35] G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, "Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities," *Journal of Cryptographic Engineering*, vol. 14, no. 1, pp. 147–164, 2024.

[36] F. Busse, P. Gharat, C. Cadar, and A. F. Donaldson, "Combining static analysis error traces with dynamic symbolic execution (experience paper)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 568–579, 2022.

[37] David A. Wheeler, "Github." `https://github.com/david-a-wheeler/flawfinder`. (Accessed: February 2026).

[38] Semgrep, "Github." `https://github.com/semgrep/semgrep`. (Accessed: February 2026).

[39] Python Code Quality Authority, "Github." https://github.com/PyCQA/bandit. (Accessed: February 2026).

[40] Bearer by Cycode, "Github." https://github.com/Bearer/bearer. (Accessed: February 2026).

[41] Zup, "Github." https://github.com/ZupIT/horusec. (Accessed: February 2026).

[42] Daniel Marjamäki, "Github." https://github.com/danmar/cppcheck. (Accessed: February 2026).

[43] angr, "Github." https://github.com/angr/angr. (Accessed: February 2026).

[44] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020.

[45] Andrew Dunham , "Github." https://github.com/andrew-d/rough-auditing-tool-for-security. (Accessed: February 2026).

[46] SpotBugs Team, "Github." https://github.com/spotbugs/spotbugs. (Accessed: February 2026).

[47] PMD Team, "Github." https://github.com/pmd/pmd. (Accessed: February 2026).

[48] Alexander Richardson, "Github." https://github.com/arichardson/juliet-test-suite-c. (Accessed: February 2026).

[49] Apache, "Website." https://tomcat.apache.org/. (Accessed: February 2026).

[50] M. Alqaradaghi and T. Kozsik, "Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code," *IEEE Access*, vol. 12, pp. 55824–55842, 2024.

[51] Meta, "Github." https://github.com/facebook/infer. (Accessed: February 2026).

[52] Sonar, "Github." https://github.com/SonarSource/sonarqube. (Accessed: February 2026).

[53] OWASP, "Github." `https://github.com/find-sec-bugs/find-sec-bugs`. (Accessed: February 2026).

[54] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 544–555, 2022.

[55] Ericsson's Open Source software, "Github." `https://github.com/Ericsson/codechecker`. (Accessed: February 2026).

[56] CodeQL, "Github." `https://codeql.github.com/`. (Accessed: February 2026).

[57] A. Jayasena and P. Mishra, "Firmwall: Directed symbolic execution of firmware binaries for defending against unauthorized system calls," *IEEE Transactions on Information Forensics and Security*, 2025.

[58] S. Bardin, P. Herrmann, and F. Védrine, "Refinement-based cfg reconstruction from unstructured programs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 54–69, Springer, 2011.

[59] H. Tu, "Boosting symbolic execution for heap-based vulnerability detection and exploit generation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 218–220, IEEE, 2023.

[60] S. Baradaran, M. Heidari, A. Kamali, and M. Mouzarani, "A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes," *International Journal of Information Security*, vol. 22, no. 5, pp. 1277–1290, 2023.

[61] Clang project, "Website." `https://clang-analyzer.llvm.org/`. (Accessed: February 2026).

[62] C. Cadar and M. Nowack, "Klee symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 6, pp. 867–870, 2021.

[63] M. Shafiuzzaman, A. Desai, L. Sarker, and T. Bultan, "Stase: Static analysis guided symbolic execution for uefi vulnerability signature generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1783–1794, 2024.

[64] LLVM Project, "Website." https://llvm.org/docs/BitCodeFormat.html. (Accessed: February 2026).

[65] Tianocore, "Github." https://github.com/tianocore/edk2. (Accessed: February 2026).

[66] Quarkslab, "Github." https://github.com/quarkslab/pixiefail. (Accessed: February 2026).

[67] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, pp. 573–609, 2015.

[68] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 213–222, IEEE, 2014.

[69] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "Fissc: A fault injection and simulation secure collection," in *International Conference on Computer Safety, Reliability, and Security*, pp. 3–11, Springer, 2016.

[70] linux-pam , "Github." https://github.com/linux-pam/linux-pam. (Accessed: February 2026).

[71] ANSSI-FR, "Github." https://github.com/wookey-project/libiso7816/blob/master/smartcard_iso7816.c. (Accessed: February 2026).