Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

Tool or Toy: Are SCA tools ready for challenging scenarios?

Congyan Shu [a], Wentao Chen [a], Guisheng Fan [a,b,*], Huiqun Yu [a,*], Zijie Huang [c], Yuguo Liang [a]

[a] School of Information Science and Engineering, East China University of Science and Technology, Shanghai, China
[b] Shanghai Engineering Research Center of Smart Energy, Shanghai, China
[c] Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, China

A R T I C L E   I N F O

Dataset link: https://github.com/ErqiFang/Benchmarking-SCA-Tools

Keywords:
Software Composition Analysis (SCA)
Open source software governance
Supply chain security
Tool evaluation model
Dependency detection
Vulnerability identification
License recognition

A B S T R A C T

The widespread adoption of open-source software (OSS) has introduced new security challenges to the software supply chain. While existing studies confirm the basic capabilities of Software Composition Analysis (SCA) tools, such as vulnerability detection and dependency resolution. They often focus on single ecosystems or detection aspects. This limited scope overlooks real-world complexities, including multi-language ecosystems, source and binary dependencies, and adversarial threats. Without a comprehensive evaluation, SCA tools may perform well in controlled settings but struggle in more complex scenarios. To address this gap, this study proposes a evaluation framework centered on the core functionalities of SCA tools: dependency detection, vulnerability identification, and license inspection. It covers three key dimensions including multi-language ecosystems compatibility, build forms, and attack defense. Using standardized datasets and quantitative metrics, such as precision, recall, F1-score and standard deviation, we evaluate four representative SCA tools, including both open-source and commercial options. Results reveal significant limitations in binary dependencies, language coverage, and license consistency. SCA tools also face challenges in balancing precision, coverage and robustness. The study highlights systemic shortcomings in current SCA tools, revealing that many perform like limited-use toys under real-world conditions. It offers data-driven recommendations to guide the evolution of these tools into practical, reliable solutions for supply chain security governance.

1. Introduction

Open-source software (OSS) has become pivotal in software development, supporting systems across virtually all industries (Duan et al., 2017; Li et al., 2024; Ivanova et al., 2024). The OSS supply chain (Wermke et al., 2023) is composed of interdependent modules and libraries, forming a complex network of dependencies through code reuse (Ohm et al., 2020; Ma, 2018). While this complexity enables rapid integration of functional components and accelerates development, it also introduces significant security risks (Imtiaz et al., 2021; Zhao et al., 2023a).

Components in the OSS supply chain come from diverse sources and may pose risks like security vulnerabilities, license compliance issues and version compatibility problems (Zahan, 2023; Tang et al., 2022; Wu et al., 2023; Jiang et al., 2024; Dietrich et al., 2023). Once these risks propagate, they can severely affect software systems reliant on the affected components (Fourné et al., 2023).

For instance, in 2016, the npm package left-pad was deleted by its developer (Wikipedia contributors, 2025a). Given its widespread use in the front-end ecosystem, its removal caused disruptions in building and deploying numerous projects, leading to website failures. Another example is the Log4j2 remote code execution vulnerability in 2021 (Hiesgen et al., 2024; Zhao et al., 2023b; Wu et al., 2023; Wetter and Ringland, 2021). As one of the mainstream logging libraries for Java software projects, its vulnerability had a broad impact, causing significant losses to the information security of governments and enterprises.

Software Composition Analysis (SCA) tools analyze open-source components used in software projects. They are recommended as a key measure for managing open-source risks by assessing the security, quality and licensing of these components (Ladisa et al., 2023). An increasing number of enterprises and development teams have recognized the crucial role of SCA tools in safeguarding OSS supply chains and have begun to actively use them (Zhan et al., 2020; Zhao et al., 2023b; Prana et al., 2021; Dann et al., 2021; Jiang et al., 2024; Imtiaz et al., 2021). These tools can analyze software project dependencies, identify potential vulnerabilities and license issues and support developers in making timely corrections. However, it is important to note

* Corresponding authors.
E-mail addresses: gsfan@ecust.edu.cn (G. Fan), yhq@ecust.edu.cn (H. Yu).

https://doi.org/10.1016/j.cose.2025.104624
Received 8 April 2025; Received in revised form 2 July 2025; Accepted 4 August 2025
Available online 9 August 2025

that existing SCA tools vary significantly in terms of accuracy, coverage and usability (Dietrich et al., 2023). On the one hand, there are performance differences across tools, particularly in core functionalities such as dependency detection, vulnerability identification and license checking. On the other hand, frequent false positives and negatives lead to inaccurate security assessments, which in turn affect the overall quality and reliability of the software.

Although some studies compare the performance of SCA tools, most are limited to specific scenarios or single datasets, lacking systematic validation of multi-language compatibility, binary dependency detection and adversarial threats. Sharma et al. (2024) compared several popular SCA tools, but their evaluation mainly focused on vulnerability detection, failing to cover key functions like dependency management and license compliance. Similarly, Imtiaz et al. (2021) revealed differences in vulnerability tracking and operational efficiency in their study of the large-scale OpenMRS Web application, but did not evaluate multi-language support or binary dependency detection.

These limitations are especially evident in multi-language ecosystems. Zhao et al. (2023b) proposed a dependency resolution evaluation model that systematically revealed the SSM support deficiencies of SCA tools in the Maven ecosystem, but their findings do not generalize well to other technology stacks like Python and C/C++. Jiang et al. (2024) demonstrated that code cloning and feature redundancy can lead to significant misjudgments in third-party library detection by traditional SCA tools in the C/C++ ecosystem, but such studies often focus on single-language optimization, lacking validation across ecosystems. Without a comprehensive evaluation, key performance dimensions remain unexplored, which may lead to overestimating tool effectiveness in real-world use.

Additionally, few studies have systematically evaluated the impact of adversarial operations on SCA tools, a critical aspect of modern software supply chain security. Dietrich et al. (2023) revealed the interference of code shadowing and cloning operations on tool accuracy, but did not propose a quantitative adversarial testing framework. Zhan et al. (2020) proposed an extensible evaluation framework for binary obfuscation scenarios, but failed to analyze the stability differences across various build forms. Neglecting any detection dimension, whether dependency, vulnerability, or license, may lead to tools that perform well in specific tests but fail in diverse and complex environments.

To address this gap, this study proposes a quantitative evaluation model that integrates three core functions: dependency detection, vulnerability identification and license recognition. It also covers three key scenarios: multi-language ecosystem compatibility, source and binary forms and adversarial threats (Wang et al., 2023). The model employs a quantitative indicator system based on recall, precision, F1-score and standard deviation to assess performance. We evaluate four representative SCA tools using standardized datasets designed to reflect real-world complexity. Results reveal that, while these tools may perform adequately in controlled settings, they often fail to handle more demanding scenarios involving low-level and emerging languages, binary dependencies, adversarial threats, and license complexity. These findings highlight the gap between current capabilities and practical needs—suggesting that many SCA tools behave more like limited-use toys than reliable solutions when facing real-world software supply chain challenges. The study provides data-driven guidance for developers, users, and researchers seeking to improve tool robustness and applicability.

The main contributions of this paper are as follows:

1. This study proposes the first comprehensive evaluation model that integrates three core functions: dependency detection, vulnerability identification and license recognition. It covers multi-language ecosystems, source and binary forms and adversarial threats, addressing the limitations of previous research focused on single function or ecosystem.
2. We construct a standardized test suite encompassing Java datasets, multi-language projects, diverse build methods, and adversarial scenarios. The datasets are derived from academic literature and leading open-source repositories over the past five years. All datasets and ground-truth lists are open-sourced to support reproducibility.[1]
3. Experiments on six datasets using four state-of-the-art tools, including RA, CleanSource, OpenSCA, and Snyk, this study identifies weaknesses in both commercial and open-source solutions and proposes targeted optimizations.

The organization of this paper is as follows: Section 2 provides an overview of the background and technical workflow of SCA tools. Section 3 outlines the methodology, including the evaluation model, research questions, datasets, tools, and evaluation metrics. Section 4 presents the performance differences and limitations of the tools in dependency detection, vulnerability identification, license recognition and stability analysis. Section 5 discusses the findings from the experimental results and offers recommendations for tool developers, users and researchers. Section 6 addresses the threats to the validity of the experiments. Section 7 compares this study with existing research and reviews related work. Finally, Section 8 summarizes the research conclusions and implications for the industry.

2. Background

2.1. Terminology

Open Source Component: A software module or library released under an open-source license that permits anyone to use, modify and distribute it. These components are frequently integrated into projects during both development and operations (Open Source Initiative, 2025).

Dependency: A reference from one software module or component to another. In OSS, dependencies can be direct, where one component immediately relies on another, or indirect, where a component depends on another that, in turn, relies on a third-party component.

Vulnerability: A flaw or weakness in software that attackers can exploit. Such vulnerabilities in open-source components may compromise the security of an entire system, making their identification and management a key task for SCA tools.

License Compliance: The adherence to the terms set by OSS licenses, which define how the software may be used, modified and distributed. SCA tools help ensure that the open-source components in a project comply with these licensing terms, thereby reducing legal risks (Microsoft, 2025).

Binary Dependency: A runtime dependency on binary files, typically compiled code. Unlike source code dependencies, analyzing binary dependencies requires handling compiled files and libraries, which adds complexity to dependency resolution (Pei et al., 2022).

Multi-Language Dependency: Dependencies involving components written in multiple programming languages, such as Java, Python, or C. These dependencies are typically resolved through separate language-specific package managers, rather than through cross-language function calls (Yang et al., 2024).

Multi-Language Ecosystem: A software project composed of multiple programming languages and corresponding build or dependency management tools, such as Maven, npm, and pip. This evaluation dimension assesses whether SCA tools can consistently and accurately identify dependencies across diverse language ecosystems (Mayer et al., 2017; Feng et al., 2024).

Adversarial Operations: Techniques that test the robustness and accuracy of SCA tools by simulating attacks. These methods, including

[1] The datasets and ground-truth lists are available at: https://github.com/ErqiFang/Benchmarking-SCA-Tools.

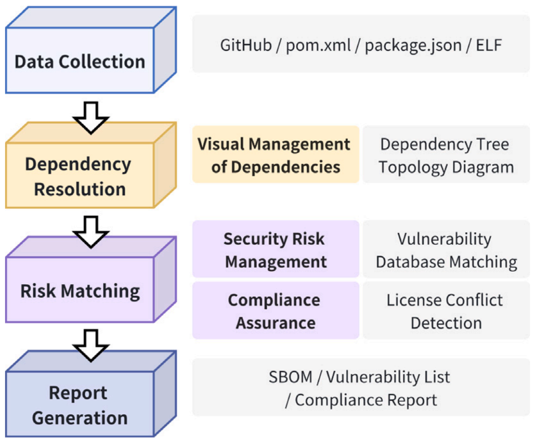
Data Collection
GitHub / pom.xml / package.json / ELF
Dependency Resolution
Visual Management of Dependencies
Dependency Tree Topology Diagram
Risk Matching
Security Risk Management
Vulnerability Database Matching
Compliance Assurance
License Conflict Detection
Report Generation
SBOM / Vulnerability List / Compliance Report

Fig. 1. SCA workflow.

code obfuscation and dynamic loading, aim to bypass detection mechanisms and reveal potential weaknesses (Bacci et al., 2018; Wang et al., 2023).

2.2. Overview of SCA

SCA is a key technique for identifying, tracking, and managing open-source components, third-party libraries (TPLs), and dependencies in software projects (Imtiaz et al., 2021; Dann et al., 2021). SCA tools support this process by mapping software components and associated risks, such as security vulnerabilities, license conflicts, and outdated or redundant dependencies. They provide developers and security teams with data-driven insights for risk management and compliance decisions (Sharma et al., 2024).

From technical process perspective, the SCA tools execution pipeline consists of four stages (Ponta et al., 2018; Decan et al., 2019; Microsoft, 2025; Wikipedia contributors, 2025b), as shown in Fig. 1. In the **data collection** stage, the tool scans the project's source code repository, builds configuration files such as `pom.xml` and `package.json` and binary files in JAR or ELF format. In the **dependency resolution** stage, the tool parses dependency declarations, builds tool configurations and runtime environments to generate a complete dependency tree (Decan et al., 2019). In the **risk matching** stage, the parsed results are compared with vulnerability databases, license repositories and version compatibility rules to identify high-risk components. Finally, in the **report generation** stage, the tool produces a Software Bill of Materials (SBOM) (Sorocean et al., 2024; O'Donoghue et al., 2024), a list of vulnerabilities and compliance recommendations and formulates remediation strategies.

Existing studies primarily examine the functionality of SCA tools from a single perspective, such as dependency management (Zhao et al., 2023b; Ombredanne, 2020; Jiang et al., 2023, 2024) or vulnerability detection (Kengo Oka, 2021; Imtiaz et al., 2021; Prana et al., 2021), without integrating multiple detection capabilities into a comprehensive assessment. Moreover, evaluating license compliance is crucial for a comprehensive analysis of SCA tools (Ombredanne, 2020; Duan et al., 2024).

The technical value of SCA tools is reflected in three dimensions: **dependency visualization and management**, which involves constructing multilevel dependency topology maps to reveal direct and transitive dependencies, assisting in optimizing dependency versions or removing redundant components; **security risk management**, which enables vulnerability reachability analysis and impact assessment by correlating with vulnerability databases such as NVD and Snyk Intel; and **compliance assurance**, which involves parsing open-source licenses like Apache-2.0 and GPL-3.0 and detecting potential constraints of contagious licenses on commercial code.

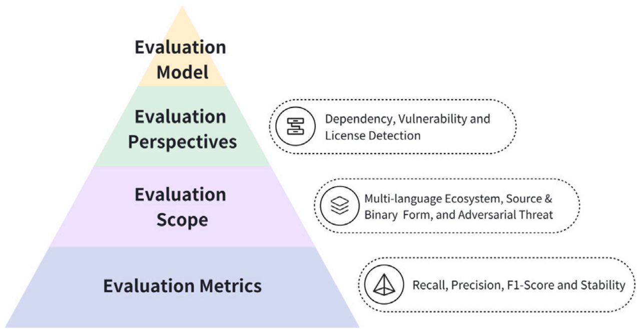
Evaluation Model
Evaluation Perspectives
Dependency, Vulnerability and License Detection
Evaluation Scope
Multi-language Ecosystem, Source & Binary Form, and Adversarial Threat
Evaluation Metrics
Recall, Precision, F1-Score and Stability

Fig. 2. Evaluation model.

Core functionalities, including dependency detection, vulnerability identification, and license inspection, constitute the foundation of modern SCA tools. However, their effectiveness depends not only on functional accuracy but also on adaptability to challenging scenarios. Specifically, the ability to support key dimensions, including multi-language ecosystems, source and binary forms and adversarial threats, is essential for ensuring robustness and practical applicability in complex software supply chains.

Although SCA tools are essential for software supply chain security, their effectiveness is limited by challenges such as multi-language ecosystem, binary dependency parsing and adversarial threat. Therefore, developing an evaluation framework that addresses multiple scenarios and systematically assesses these tools' performance is crucial for advancing both technology and industry applications.

3. Study methodology

3.1. Evaluation model

The evaluation model provides a structured framework for assessing the performance of SCA tools across diverse scenarios. Building on prior criteria proposed in related studies (Zhan et al., 2020; Zhao et al., 2023b, 2021), the model is structured around three core dimensions: evaluation perspectives, evaluation scope, and evaluation metrics, as illustrated in Fig. 2. It defines three key scenarios that SCA tools must address, and aligns them with the core functionalities. These elements together support a comprehensive and systematic evaluation of tool capabilities.

- **Evaluation Perspectives** focus on three core functions of SCA tools: **Dependency Detection**, **Vulnerability Identification**, and **License Inspection**. Dependency detection evaluates the tool's capability to accurately construct a SBOM, vulnerability identification assesses its effectiveness in detecting security risks, and license inspection examines its ability to manage legal risks associated with open-source licenses.
- **Evaluation Scope** consists of three key dimensions: **Multi-Language Ecosystem**, **Source and Binary Form**, and **Adversarial Threat**. These dimensions address practical challenges in software supply chains, including multi-language development, diverse build environments, and adversarial attack techniques. The model integrates insights from development practices, technological diversity, and security threats to provide a comprehensive evaluation framework covering both routine detection and complex adversarial scenarios.
- **Evaluation Metrics** include **Recall**, **Precision**, **F1-Score**, and **Standard Deviation** to assess detection accuracy and robustness. Recall measures a tool's ability to identify all existing vulnerabilities and dependencies, while precision indicates its effectiveness in minimizing false positives. F1-score provides a balanced metric of precision and recall. Additionally, standard deviation quantifies the variability of results under adversarial operations through repeated experiments, indicating the tool's consistency under varying conditions.

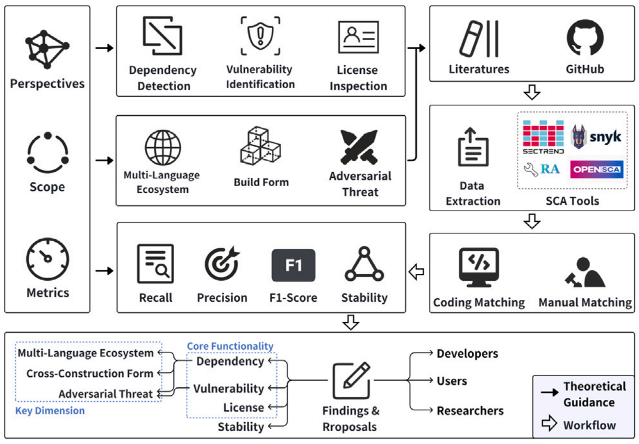

Fig. 3. The workflow of our study.

3.2. Workflow

Fig. 3 illustrates the workflow of our study, structured around the proposed evaluation model. We compile benchmark datasets from the past five years' academic studies and official open-source repositories such as GitHub. These datasets cover multi-language projects, source and binary form, and adversarial threat scenarios. To ensure broad applicability and reflect diverse technical approaches, we include both commercial and open-source SCA tools, selecting four representative ones for evaluation.

Across multiple scenarios, we systematically test these tools by extracting ground-truth data from the datasets and computing relevant metrics. By comparing evaluation results across different scenarios, we identify limitations in dependency detection, vulnerability identification, and license inspection. Based on these observations, we summarize our findings and provide recommendations for tool developers, users, and researchers.

3.3. Research questions

This study evaluates the overall effectiveness of SCA tools, focusing on four key research questions (RQs) that examine their performance limits, particularly in **core functions**. The specific RQs are:

- **RQ1: How effective are SCA tools in detecting dependencies?** This question evaluates dependency detection from three perspectives: **multi-language ecosystem**, **build form** and **adversarial threats**.
  - For multi-language ecosystem, DS1 on the Java Maven database and DS4 on multi-language projects are used to compare how effectively the tools resolve Java and multi-language dependency chains.
  - For build form, the differences in dependency resolution between source and binary code are evaluated by comparing the results of DS2 on binary files with those of DS1 and DS3-5 on source dependencies.
  - For adversarial threats, false negative rates are measured for dependency hijacking and code obfuscation using DS1 and DS3 for attacks on Maven projects, and using DS4 and DS5 for multi-language obfuscation.
- **RQ2: Can SCA tools effectively identify vulnerabilities in the supply chain attack scenarios?** Due to incomplete vulnerability data and limited details in some datasets (Dietrich et al., 2023), this study uses DS3 to analyze attacks on Maven projects, testing the tools' ability to detect known vulnerabilities across 11 adversarial scenarios.
- **RQ3: How well do SCA tools recognize licenses in complex licensing scenarios?** Using DS6 based on the SPDX license, we assess the tools' ability to identify licenses with naming discrepancies, such as different versions and full names versus abbreviations like "GNU General Public License v2.0" and "GPL-2.0". Since regex-based methods struggle with these variations, a semi-automated process was used to evaluate 663 licenses.
- **RQ4: How stable are SCA tools in different detection scenarios?** Dependency detection involves multi-language ecosystems, source and binary form and adversarial threat scenarios. Datasets DS1 to DS5 include large, well-annotated samples. We calculate the standard deviation (σ) of average recall, precision and F1-score across the datasets to assess the consistency of tool performance in complex software supply chain environments.

3.4. Tool selection

To ensure a balanced evaluation, we selected four representative SCA tools, based on differences in architecture, deployment, and adoption across industry and academia. This selection includes both commercial and open-source solutions, reflecting varied levels of technical maturity and usage contexts. RA is a commercial enterprise-grade tool with broad language support and strong detection capabilities. Snyk represents modern DevSecOps practices, offering risk-based vulnerability scanning and wide integration in development pipelines (Sushma et al., 2023). OpenSCA is an open-source project widely adopted in the Chinese security community, valued for its transparency, extensibility, and ecosystem coverage.

3.4.1. Tool 1: Commercial tool RA

RA, anonymized due to the request of its commercial provider, is an enterprise-grade SCA tool that supports dependency analysis, vulnerability detection and license compliance. Its strength lies in its comprehensive detection methods, which identify software components through dependency relationships, file structures, code snippets and binary signatures. RA generates detailed component inventories at the project and coordinate levels. Its broad language support and comprehensive features make it suitable for large-scale enterprise use.

3.4.2. Tool 2: Commercial tool CleanSource

CleanSource, developed by SecTrend, is known for its high detection accuracy and strong performance in adversarial scenarios, making it suitable for complex enterprise-level security and compliance needs. It has been deployed by major tech companies such as Tencent (Global TMT, 2025), reflecting its effectiveness in real-world. Its technical strengths include: (1) dependency tree visualization to map complex relationships; (2) high-precision vulnerability matching and risk assessment based on authoritative databases such as CVE and CNVD; (3) adaptive recognition algorithms for extracting component details, including versioning, licensing and encryption methods; (4) the ability to scan binary packages without requiring source code, enabling fast and passive analysis; and (5) license compatibility checks for open-source components.

3.4.3. Tool 3: Open-source tool OpenSCA

OpenSCA, the open-source version of Xmirror Security's Xcheck SCA, is widely used in small and medium-sized projects. In this study, we use version v3. OpenSCA supports dependency analysis and vulnerability detection for major programming languages such as Java, Python, PHP and Golang, integrating with the CVE database for basic vulnerability scanning and component-level license identification. However, it lacks file-level license detection, has limited binary analysis capabilities and struggles with obfuscation or encryption scenarios, making it more suitable for less complex environments.

Table 1
Dataset information and tags.

ID	Dataset description	Source	Tags
DS1	Classic Java Dependencies	Zhao et al. (2023b)	Dependency, Java
DS2	Binary Dependencies	Zhan et al. (2020)	Dependency, Binary
DS3	Maven Dependencies with Adversarial Modifications	Ivanova et al. (2024)	Dependency, Vulnerability, Java, Adversarial
DS4	Multi-Language Dependencies	SourceClear (2025)	Dependency, Multi-language
DS5	Multi-Language Dataset with Adversarial Modifications	Wu et al. (2023)	Dependency, Multi-language, Adversarial
DS6	License Dataset	SPDX (2025)	License

3.4.4. Tool 4: Commercial tool Snyk

Snyk is a commercial SCA tool designed for vulnerability detection and remediation. It supports a wide range of programming languages and integrates with platforms such as GitHub to perform automated dependency scanning and continuous monitoring. Snyk prioritizes high-impact vulnerabilities and provides actionable fix suggestions. In our experiments, we used version v1.1297.1 (Snyk Documentation, 2025; Snyk CLI Developers, 2025).

3.5. Dataset selection and construction

The basic information of the datasets is summarized in Table 1. The design logic, data sources and evaluation objectives are detailed below. Although all datasets are derived from benchmark-oriented scenarios, they are constructed based on patterns observed in real-world projects and threat reports. Each dataset is designed to reflect specific challenges faced by SCA tools in practice, including multi-language ecosystem, binary dependency, and attack defense.

3.5.1. Dataset 1 (DS1): Java dependency dataset

This dataset is based on the benchmark framework proposed by Zhao et al. (2023b), designed to model the complexity of dependency management within the Java-Maven ecosystem. It includes Maven modules and their dependency topologies extracted from real Java projects, covering eight Maven Dependency Features (MDF) and three Maven Dependency Settings (MDS). MDF includes dependency management, parent inheritance, exclusion, profiles, optional dependencies, version ranges and variable-based versioning, while MDS includes dependency type, classifier and scope. The dataset consists of 256 MDF combinations and 22 MDS instances, forming a standardized test set of 259 experimental projects. Each project is annotated with ground-truth dependency lists to ensure reliable benchmarking.

3.5.2. Dataset 2 (DS2): Binary dependency dataset

To evaluate SCA tools' ability to detect OSS reuse in binary form, this study uses a dataset from Zhan et al. (2020) consisting of 35 complex executables generated from GCC, Clang and MSVC. It includes 24 Linux ELF files (10 from Clang) and 11 Windows PE files from MSVC, covering more than one million assembly functions and 55 million lines of C/C++ code. These binaries, derived from large-scale applications such as physics engines and payment protocols, reflect real-world production complexity.

3.5.3. Dataset 3 (DS3): Java adversarial dataset

To test the robustness of SCA tools against supply chain attacks, the study uses an adversarial Maven POM dataset from Ivanova et al. (2024). This dataset includes 29 high-profile vulnerable dependencies sourced from the Maven Central Repository and simulates attack scenarios by modifying manifest characteristics, bundling methods and dependency configurations. It consists of 13 Maven projects, including 11 adversarial cases and 2 baselines used for comparative analysis.

To better reflect real-world threats, each scenario is mapped to establish software supply chain attack taxonomies (Ladisa et al., 2023; Ohm et al., 2020). Specifically, Scenarios 1–5 emulate metadata-level obfuscation techniques such as variable-based versioning, profile-activated dependencies, and parent–child inheritance. These mimic subtle configuration-based evasions that hinder accurate dependency resolution. Scenarios 6–9 simulate build-stage attacks by modifying Uber-JAR artifacts, including the use of shaded packages, stripped metadata, or forged manifest files, resembling techniques found in the SolarWinds incident (Martínez and Durán, 2021). Scenarios 10–11 reflect dependency confusion attacks, in which misleading group IDs and tampered versions are introduced through manual installation, representing typosquatting and hijacking behaviors commonly reported in open ecosystems.

3.5.4. Dataset 4 (DS4): Multi-language dataset

To assess SCA tools' compatibility and accuracy in multi-language projects, the study adopts the “Evaluation Framework for Dependency Analysis” (EFDA) dataset from SourceClear (SourceClear, 2025). This dataset spans 10 major programming languages, including Java, JavaScript, Python and Golang and integrates heterogeneous build systems such as Maven, npm, pip and Go Modules. It simulates real-world development environments with multi-language dependency chains and standardized ground-truth annotations.

3.5.5. Dataset 5 (DS5): Adversarial multi-language dependency dataset

To test SCA tools' ability to handle obfuscated dependencies in multi-language scenarios, the study uses a dataset from Wu et al. (2023), covering Python, Ruby, PHP, Java, Rust, Golang and JavaScript. C/C++ projects were excluded due to limited support in existing SBOM tools, which could introduce evaluation bias. The dataset introduces parser-level ambiguity by injecting non-standard syntax into files such as Python's requirements.txt, intentionally exploiting inconsistencies among language-specific dependency parsers to disrupt accurate resolution.

Beyond dependency obfuscation, DS5 also models the complexity of vulnerability propagation in real-world software supply chains. The design captures practical issues including how vulnerabilities are reached, the difficulty of triggering them, and the downstream responses of dependent projects. It reflects risks from deeply nested call chains, vulnerabilities requiring multi-layered triggering logic, and vulnerable functions invoked without sufficient validation or control. Additionally, it includes cases of misconfigured dependency management, such as version pinning failures and hidden profiles. These characteristics mirror the patterns of latent propagation risk and supply chain failure observed in recent incident analyses, such as those reported in Ladisa et al. (2023), and help reveal the limitations of existing SCA tools in identifying and mitigating such threats.

3.5.6. Dataset 6 (DS6): License benchmark

For license detection evaluation, the study employs a dataset based on the Software Package Data Exchange (SPDX) standard, which includes 663 licenses with detailed metadata and versioning information (SPDX, 2025). Provided in RDFa, HTML, Text and JSON formats, the dataset enables the evaluation of tools' ability to recognize common licenses and resolve versioning discrepancies.

3.6. Metric setup

To evaluate the performance of SCA tools across environments, we use three key metrics: **recall** for detection coverage, **precision** for result reliability and **standard deviation** for consistency across scenarios.

In the experimental design, ground-truth serves as the baseline, transforming detection tasks into binary classification problems. Each dataset directory records four key values: **matches**, representing correct detections where predictions align with the ground-truth; **misses**, indicating ground-truth elements that the tool failed to detect; **extras**, referring to incorrect predictions not present in the ground-truth; and **truths**, denoting the total number of ground-truth elements. These values form the basis for computing recall, precision and standard deviation. The methodology includes the following steps:

1. **Dynamic data definition**: Positive labels, including dependency names, vulnerability identifiers and license IDs, are defined by the dataset's objectives.
2. **Confusion matrix analysis**: Performance is evaluated using a confusion matrix with True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN).
3. **Cross-dataset comparison**: Tools' performance is tested across different datasets, including adversarial and regular scenarios, to evaluate generalization in diverse environments.

Recall, **precision**, and **F1-score** quantify detection effectiveness, balancing coverage and false positive control:

- **Recall** is the proportion of true positives among all actual positives, reflecting the tool's ability to detect dependencies or vulnerabilities. Higher recall indicates fewer missed detections:

$$\textbf{Recall} = \frac{TP}{TP + FN}. \tag{1}$$

- **Precision** measures the proportion of true positives among all predicted positives, reflecting the accuracy of detection results. Higher precision reduces false positives, which is critical in adversarial scenarios:

$$\textbf{Precision} = \frac{TP}{TP + FP}. \tag{2}$$

- **F1-score** is the harmonic mean of precision and recall, providing a single metric that balances both aspects. It is particularly useful when evaluating performance under class imbalance or adversarial conditions:

$$\textbf{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{3}$$

To evaluate stability, the **standard deviation** of recall and precision across datasets is calculated as:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n}}, \tag{4}$$

where σ is the standard deviation, x_i is the value for each dataset, $\overline{x}$ is the average value, and n is the number of datasets. A smaller standard deviation indicates more consistent performance across different scenarios, reflecting the robustness of the tools.

4. Empirical study

4.1. RQ1: How effective are SCA tools in detecting dependencies?

To evaluate the performance of SCA tools across diverse dependency scenarios, this study uses datasets covering programming languages, dependency types, and adversarial conditions. The evaluation framework examines three key dimensions: multi-language ecosystem compatibility, build form compatibility and robustness to adversarial threats.

- **Multi-Language Ecosystem Dimension** focuses on the ability of tools to handle multi-language ecosystems, including Java, Python, and JavaScript, reflecting the increasing use of multi-language frameworks in modern development. This dimension examines whether SCA tools can overcome single-language limitations and accurately detect dependencies across complex, multi-stack systems.
- **Build Form Dimension** considers two types of dependencies: source-level and binary-level. The former refers to structured dependencies declared during development, while the latter involves implicit runtime dependencies of compiled artifacts. The former relies on syntax analysis and semantic reasoning, while the latter requires reverse engineering and symbolic matching. Together, they form a complete dependency map of the software supply chain.
- **Adversarial Threat Dimension** evaluates the tools' robustness to supply chain attacks and multi-language attacks. These scenarios simulate real-world tactics such as code obfuscation and syntactic manipulation.

Experimental results indicate substantial performance variations across datasets. Table 2 shows the average number of ground-truths, matches, misses, extras and the average recall, precision and F1-score for each tool across 5 dependency datasets.

4.1.1. Multi-language ecosystem

To assess the generality and robustness of SCA tools in multi-language environments, this study compares **deep single-language support** using DS1, which follows Java Maven standards, with **broad multi-language coverage** using DS4, which includes 10 languages such as Java, Python, and C/C++. The number of ground-truth files in each dataset and the number of files each tool successfully scanned are shown in Table 3.

For DS1, Snyk and OpenSCA show the best performance with high recall and precision, followed by RA with moderate results. Snyk achieves the highest F1-score, demonstrating strong accuracy and low false positives in detecting Java dependencies. OpenSCA also performs well, with a recall of 74.86% and precision of 99.84%. RA achieves 65.76% recall and 82.58% precision but fails to detect certain files, possibly due to complex Maven dependency features or MDS projects exceeding its detection scope. CleanSource scans all files but shows very low recall, averaging only three correct matches per file compared to 41 ground-truth values.

DS4, which includes ten programming languages, reveals further differences among tools, as shown in Fig. 4. RA identifies dependencies in eight languages, missing only Scala and C#. Snyk supports seven languages and is the only tool capable of detecting dependencies in Scala and C#, though it fails to handle C, Objective-C, and Ruby. None of the tools cover all ten languages. OpenSCA performs best in Python and Ruby, with consistently high recall and precision, and shows moderate results in Java, Golang, JavaScript, and PHP. However, its performance drops sharply in C and Objective-C, where both recall and precision are near zero. RA achieves strong recall in Objective-C, Golang, Java, and JavaScript, but its precision is highly variable, remaining low in PHP and C. CleanSource demonstrates limited multi-language capabilities, performing relatively well only in Objective-C and Python. It fails to detect dependencies in Java, Ruby, or C, and its results in Golang, PHP, and JavaScript are inconsistent. Snyk achieves the highest recall across most languages, reflecting strong multi-language coverage, yet its lower precision results in only moderate F1.

Finding 1: Existing SCA tools exhibit **clear differences** in multi-language ecosystem. While detection techniques are mature for traditional environments like Java, they remain weak for emerging languages like Go and low-level languages like C and C++, highlighting the challenge of achieving compatibility in multi-language dependency analysis.

4.1.2. Build form

The diversity of software build patterns imposes distinct technical adaptation requirements on component analysis tools. From a full lifecycle perspective, dependency detection can be categorized into two types: **source-level** explicit dependencies and **binary-level** implicit

Table 2
Average dependency detection performance of five datasets.

Tool	Dataset	Matches	Misses	Extras	Truths	Recall (%)	Precision (%)	F1 (%)
RA	DS1	24.00	17.18	5.97	41.18	65.76	82.58	70.84
	DS2	0.56	6.06	1.81	6.63	7.21	24.22	10.46
	DS3	51.00	15.00	53.63	66.00	77.58	54.84	62.60
	DS4	3.33	0.72	19.62	4.05	74.48	36.42	42.63
	DS5	6.00	33.50	20.25	39.50	19.65	44.41	25.02
CleanSource	DS1	3.01	37.95	1.02	40.97	8.52	73.99	15.01
	DS2	0.74	7.04	1.96	7.78	9.37	28.41	12.62
	DS3	22.33	43.00	23.11	65.33	34.16	52.77	38.84
	DS4	1.96	2.30	14.78	4.26	32.96	22.71	24.13
	DS5	2.40	110.60	2.00	113.00	9.43	36.33	14.26
OpenSCA	DS1	28.13	12.83	0.08	40.97	74.86	99.84	84.68
	DS2	/	/	/	/	/	/	/
	DS3	44.50	20.80	10.30	65.80	68.02	70.40	68.42
	DS4	3.86	0.82	20.14	4.68	85.25	48.12	49.82
	DS5	15.00	83.43	1.86	98.43	33.38	66.69	40.96
Snyk	DS1	29.88	11.09	0.13	40.97	81.21	99.80	88.34
	DS2	/	/	/	/	/	/	/
	DS3	39.30	26.00	2.90	65.30	60.00	56.96	58.43
	DS4	2.86	0.45	22.77	3.83	86.84	23.14	32.52
	DS5	15.17	81.50	4.17	96.67	32.54	49.62	37.95

Note: All metrics are reported as the average values computed across different files within the same dataset.

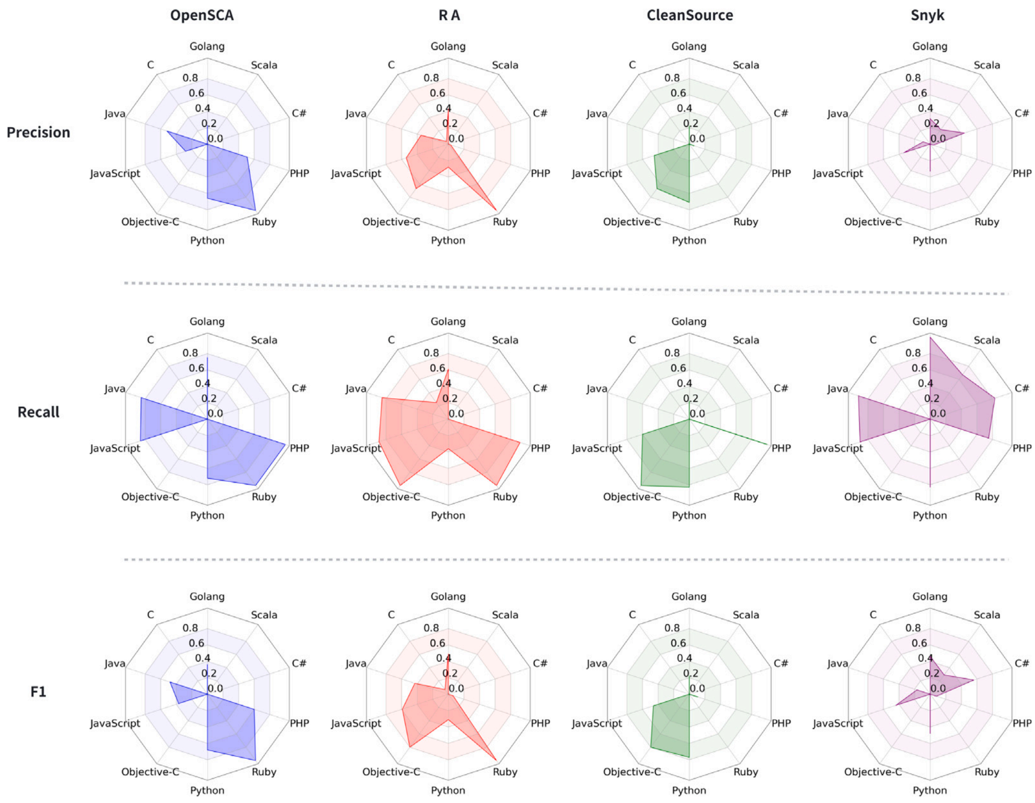

Fig. 4. Tool performance on DS4 in multi-language analysis.

Table 3
Number of files effectively scanned by tools (Total per dataset in parentheses).

Tool	DS1(258)	DS2(32)	DS3(10)	DS4(54)	DS5(12)
RA	131	32	8	40	4
CleanSource	258	23	9	27	5
OpenSCA	258	/	10	29	7
Snyk	258	/	10	22	6

dependencies. Source-level detection is evaluated using DS1, DS3-5 to assess the ability to parse structured dependency declarations. Binary-level detection is tested with DS2 (binary files compiled with gcc, clang and MSVC) to evaluate the ability to trace dependencies without source code.

Fig. 5 illustrates the overall performance of the four tools across the five datasets, where areas enclosed by the same color block represent the same dataset. A comparison between the red and other colored regions reveals significant differences in handling source-level and binary-level dependencies. In source-level scenarios, the tools achieve an average recall of 48.7%, and an average precision of 57.4% (with OpenSCA reaching 99.84%), indicating strong capability in managing

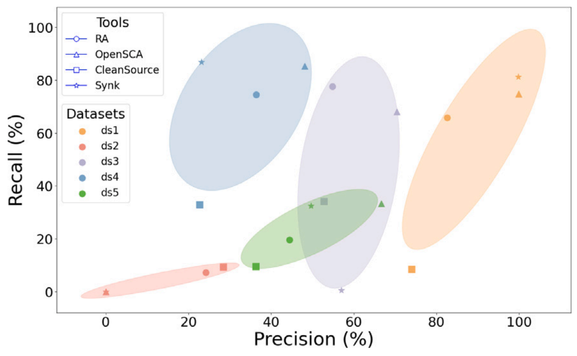

Fig. 5. Performance of tools across DS1-5. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Tools' dependency detection in attack scenarios on DS3.

Attack scenarios	RA	CleanSource	OpenSCA	Snyk
0-baseline1	✓	✓	✓	✓
0-baseline2	/	/	/	/
1-version-variable	✓	✓	✓	✓
2-dependency-management	✓	✓	✓	✓
3-profiles	×	✓	✓	✓
4-parent–child-version-variable	✓	∘	✓	✓
5-parent–child-groupid-variable	×	×	✓	✓
6-uber-jar	✓	✓	✓	∘
7-shaded-uber-jar	✓	✓	✓	∘
8-bare-uber-jar	✓	∘	∘	∘
9-uber-jar-modified-metadata	✓	✓	∘	∘
10-manual-install-modified-groupid	/	/	/	/
11-manual-install-wrong-version	/	/	/	/

[1] Baseline 1 is utilized for comparison with manifest-related and bundling attacks.
[2] Baseline 2 serves as a reference for comparison with dependency modification attacks.
Symbols: ✓ = correct identification; × = identification failure; ∘ = report generated, all matches fail; / = no ground-truth data.

explicit dependencies. However, in binary-level scenarios, the average recall drops sharply to 8.7% (RA: 7.21%, CleanSource: 9.37%) with precision falling below 30%. Notably, OpenSCA and Snyk lack binary-level detection capabilities entirely.

Finding 2: SCA tools show **significantly weaker performance** on binary datasets compared to source code scenarios, highlighting limitations in handling source and binary forms. Some tools are unable to detect any binary dependencies at all, exposing a critical blind spot. This gap in binary-level build detection limits comprehensive security across the software supply chain lifecycle.

4.1.3. Adversarial threat

This study evaluates the robustness of SCA tools in threat detection using two adversarial datasets: **basic supply chain attacks** (DS3) and **complex multi-language adversarial scenarios** (DS5). DS3 simulates traditional dependency hijacking in Maven projects, while DS5 tests multi-language dependency confusion. These datasets assess the tools' defense capabilities and ecosystem adaptability.

Table 4 and Fig. 6 present the tools' performance boundaries, including recall and precision, under different attack scenarios in DS3. The "✓" indicates that the tools can effectively identify and generate prediction reports. In the basic attack scenario, OpenSCA demonstrates strong defense, with a recall of 68.02% and precision of 70.40%, though it fails under complex build attacks involving bare uber-jar files. In contrast, RA, CleanSource, and Snyk exhibit varying degrees of weakness. RA achieves the highest recall at 77.58% but suffers a precision drop to

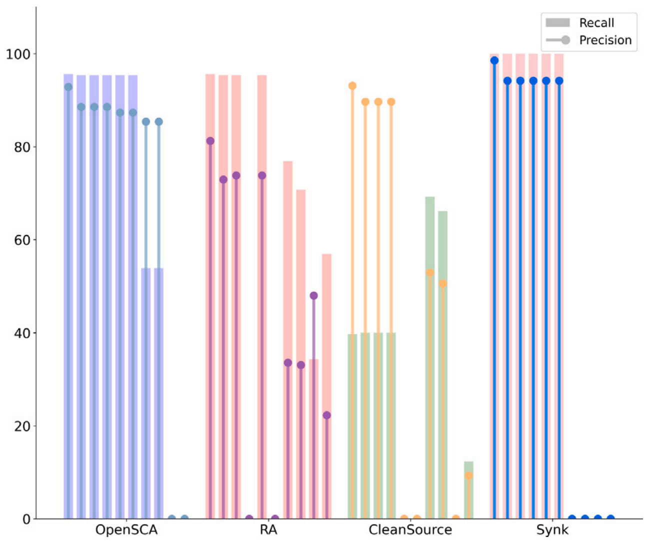

Fig. 6. Tools' dependency detection in DS3 attack scenarios.

33.56% under parent–child variable substitution, indicating difficulty in resolving modified dependency structures. CleanSource shows severe instability, with recall fluctuating from 9.3% to 93.1% and a low F1 score of 38.84%, reflecting poor robustness against structural attacks. Snyk maintains moderate yet stable performance, with an F1 score of 58.43%. It handles basic and multi-module cases adequately, but fails to match any predictions under uber-jar and metadata manipulation scenarios, revealing clear limitations in build-stage obfuscation defense.

The complex multi-language adversarial scenario on DS5 (as shown in Fig. 7) further amplified performance gaps among tools. All tools struggled with adversarial operations involving Ruby and Rust. OpenSCA maintained high accuracy in certain PHP, Golang and Java scenarios but showed a sharp decline in Python and JavaScript compared to DS3. RA's overall performance was weaker, with reliable detection only in Golang, Java and Python, while performance in PHP, Ruby and JavaScript deteriorated significantly. CleanSource performed moderately, successfully identifying some obfuscated dependencies in Golang, Java, JavaScript and Python, but with limited consistency. Its detection capability in Python, PHP and JavaScript dropped significantly under complex adversarial conditions. Snyk exhibited strong detection in Golang, achieving high F1 scores, but performed poorly in JavaScript where key metrics approached zero.

Finding 3: Existing SCA tools exhibit **limited resilience** and **poor adaptability** to adversarial threats. They struggle to handle bundling modification attacks and multi-language obfuscation, revealing critical gaps in attack surface coverage and inconsistencies in semantic analysis.

4.2. RQ2: Can SCA tools effectively identify vulnerabilities in supply chain attack scenarios?

In software security, the vulnerability identification capability of SCA tools is essential. This study evaluates the vulnerability and dependency identification performance of RA, CleanSource, OpenSCA and Snyk using DS3. The assessment covers both **simple baseline scenarios**, such as standard dependency injection and **complex adversarial scenarios**, such as Uber-JAR metadata tampering. Table 5 presents the tools' vulnerability identification performance, including recall and precision for both vulnerability and vulnerable dependency identification. Table 6 illustrates the tools' performance limits across different attack scenarios.

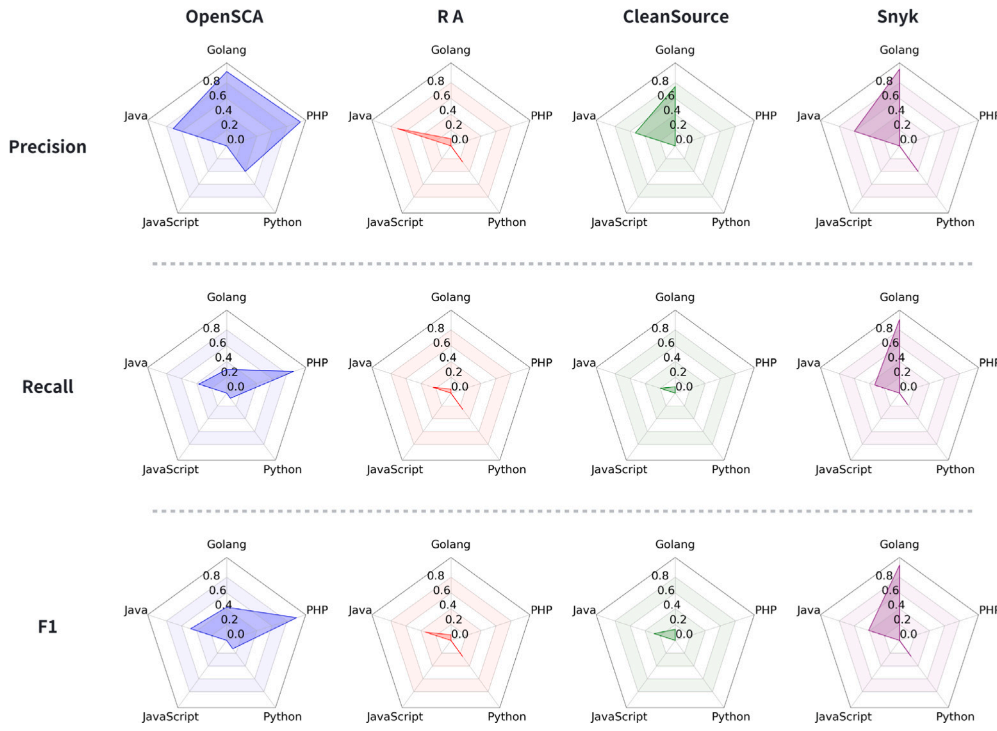

Fig. 7. Tool performance on DS5 in multi-language analysis.

Table 5
Vulnerability and affected dependency detection performance (Average Across Datasets).

Tool	Vul.Rec (%)	Vul.Prec (%)	Vul.F1 (%)	Vul.Dep.Rec (%)	Vul.Dep.Prec (%)	Vul.Dep.F1 (%)
RA	62.12	21.59	30.94	88.40	53.92	66.15
OpenSCA	24.25	24.46	24.04	62.86	31.27	41.26
CleanSource	34.59	43.74	38.09	56.90	68.10	61.10
Snyk	23.47	16.16	18.45	87.07	65.55	74.42

Table 6
Vulnerability identification of SCA tools in DS3 attack scenarios.

Attack scenarios	RA	CleanSource	OpenSCA	Snyk
0-baseline1	✓	×	✓	✓
0-baseline2	✓	✓	✓	✓
1-version-variable	✓	×	✓	✓
2-dependency-management	✓	×	✓	✓
3-profiles	×	×	✓	✓
4-parent–child-version-variable	✓	×	✓	✓
5-parent–child-groupid-variable	×	×	✓	✓
6-uber-jar	✓	✓	✓	×
7-shaded-uber-jar	✓	✓	✓	×
8-bare-uber-jar	✓	×	∘	×
9-uber-jar-modified-metadata	✓	∘	∘	×
10-manual-install-modified-groupid	✓	✓	✓	×
11-manual-install-wrong-version	✓	✓	✓	∘

[1] Baseline 1 is utilized for comparison with manifest-related and bundling attacks.
[2] Baseline 2 serves as a reference for comparison with dependency modification attacks.
Symbols: ✓ = correct identification; × = identification failure; ∘ = report generated, all matches fail; / = no ground-truth data.

RA demonstrates solid overall performance, with a vulnerability recall of 62.12%, indicating good coverage in identifying vulnerabilities. It handles most adversarial scenarios in DS3, failing only in a few cases such as profiles and parent–child group ID variation. For vulnerable dependency identification, RA performs well, achieving an F1 score of 66.15%, though its low precision of 21.59%.

CleanSource favors precision over recall, achieving the highest vulnerability precision at 43.74% and an F1 score of 38.09%. However, its scenario coverage is limited, performing effectively in only five scenarios (including one baseline), and failing completely in complex cases involving version variable parsing and dependency management configuration tampering.

OpenSCA shows balanced but limited performance. Its vulnerability recall and precision remain low, around 24%, and it fails in specific bundling attacks such as bare Uber-JAR and metadata modification, where it generates reports without correct matches. Nevertheless, it handles a wide range of scenarios stably, detecting attacks in 11 out of 13 DS3 cases.

Snyk shows the weakest performance in vulnerability identification, with a recall of 23.47%, precision of 16.16%, and an F1 score of just 18.45%. However, it performs best in detecting vulnerable dependencies, achieving the highest F1 score of 74.42%. While Snyk identifies early-stage attacks such as baseline and version variation effectively, it fails in build-stage and manual installation scenarios. These results indicate strong dependency-level detection but limited capability in mapping vulnerabilities to specific identifiers, reducing its adaptability to complex attack patterns.

Through systematic mapping of each tool's vulnerability reports to the NVD database and extraction of corresponding CWE entries, we found that the top five most frequently detected CWE types are highly consistent across tools, including CWE-502, CWE-400, CWE-770, CWE-787, and CWE-20.

Table 7
License recognition performance of tools.

	Average per-sample true value						Global dataset					
	Matches	Misses	Extras	Recall	Precision	F1	Matches	Misses	Extras	Recall	Precision	F1
RA	0.75	0.26	0.61	65.04%	64.83%	64.90%	179.00	484.00	202.00	27.00%	46.98%	34.29%
CleanSource	0.47	0.53	0.56	45.56%	45.56%	45.56%	23.00	640.00	7.00	3.47%	76.67%	6.64%

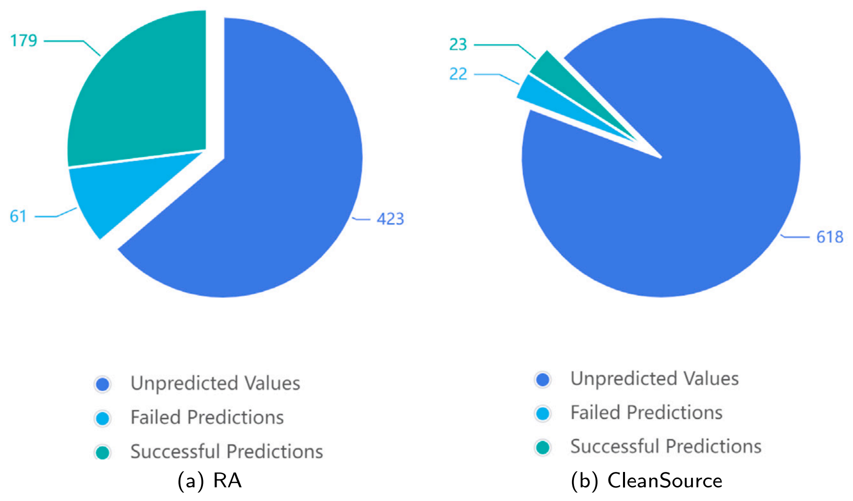

Fig. 8. Performance of tools in license recognition.

> **Finding 4**: The vulnerability detection performance of SCA tools is highly variable, often showing **high false positive rates** and **poor adaptability to complex scenarios**. This reflects a trade-off in traditional detection methods between precision and coverage, as well as limitations in handling attacks like bundling modifications and manifest features changes.

4.3. RQ3: How well do SCA tools recognize licenses in complex licensing scenarios?

The **diversity of open-source licenses** and the **complexity of license expressions**, including the mixed use of full names and abbreviations, present key challenges for SCA tools in license recognizing. This study evaluates the license identification and matching performance of SCA tools using DS6, the SPDX standardized license list, which includes 663 licenses. The analysis focuses on the semantic parsing accuracy (precision) and the coverage (recall). Since other datasets lack standardized ground-truth, SPDX serves as the only quantifiable benchmark.

Technical limitations cause OpenSCA and Snyk to fail entirely, generating no valid predictions. This indicates their dependency-based matching mechanism is incompatible with SPDX's file-level detection. The license recognition results for RA and CleanSource are in Table 7, where "Average Per-Sample True Value" denotes the single license prediction and "Global Dataset" refers to the entire dataset. Fig. 8 shows the success-failure match ratio reported by RA and CleanSource.

RA exhibited mixed performance, combining high per-sample accuracy with low global coverage. At the individual license level, it achieved over 65% precision and recall, averaging 0.75 matches per license. However, at the global level, it matched only 179 licenses (27% recall) with a 73% false negative rate. RA generated 202 predictions, exceeding the number of valid matches, indicating systematic blind spots for certain licenses.

In contrast, CleanSource showed more pronounced limitations. Its global recall was below 7%, leaving 93.21% of licenses unidentified. Even in successful cases, its precision reached 76.67%, averaging only 0.47 matches per license. The low recall and high rate of missed detections suggest that CleanSource's algorithm struggles to handle complex license patterns.

Table 8
Standard deviation of dependency recall, precision, and F1 across datasets.

Tool	Recall (%)	Precision (%)	F1 (%)
RA	29.51	19.77	22.53
OpenSCA	19.47	18.53	16.90
CleanSource	11.99	18.57	9.79
Snyk	21.32	27.53	21.89

> **Finding 5**: SCA tools have **limitations** in license recognition. These gaps underscore the systemic limitations of traditional methods in handling **expression diversity** (e.g., "`GPL-2.0-only`" vs. "`GNU GPL v2`") and **version granularity** (e.g., "`Apache-2.0`" vs. "`Apache-1.1`").

4.4. RQ4: How stable are SCA tools in different detection scenarios?

The **stability** of SCA tools directly impacts their reliability and deployment feasibility across various environments. Dependency detection involves multiple language, source and binary form and adversarial threat, increasing its complexity. Using DS1–5, this study evaluates stability based on the consistency of dependency detection results. The standard deviation (σ) of the average recall, precision and F1 across datasets is calculated for each tool as a measure of stability. The experimental results are shown in Table 8.

RA indicates unstable performance across datasets, with standard deviations of 29.51% in recall, 19.77% in precision, and 22.53% in F1. OpenSCA is more consistent but still fluctuates, with deviations of 19.47% in recall, 18.53% in precision, and 16.90% in F1. CleanSource demonstrates the most stable performance, particularly in F1 with a deviation of only 9.79%. Snyk shows high inconsistency overall, with precision varying by 27.53%, and substantial fluctuations in recall and F1, at 21.32% and 21.89% respectively.

Finding 6: SCA tools show **poor consistency** across datasets, especially in recall and F1. Tools with broader coverage tend to be less stable, exposing a trade-off between coverage and consistency.

5. Discussion and implication

5.1. Discussion

Empirical findings reveal significant **capability gaps** and **adaptation challenges** in existing SCA tools. While some tools perform well in controlled, source-level scenarios, they struggle in multi-language, binary, and adversarial contexts. Performance declines sharply in binary scenarios, as shown in DS2, due to algorithmic limitations in addressing compilation optimizations and dependency obfuscation. Moreover, SCA tools exhibit weak resistance to advanced threats, such as post-compilation attacks and multi-language obfuscation techniques, as shown through comparative analysis on DS1 and DS3, DS4 and DS5.

SCA tools struggle to detect metadata poisoning and obfuscated dependency relationships, consistent with past security failures. Our attack modeling in DS3 and DS5 reflects realistic supply chain threats based on established scenarios. DS3 focuses on build-stage and metadata-layer vulnerabilities, including configuration obfuscation and tampered binary artifacts. It draws on real-world cases such as the SolarWinds incident (Martínez and Durán, 2021), where compromised build outputs evaded standard scanning, and on typosquatting and metadata manipulation attacks reported in earlier studies (Ohm et al., 2020). DS5 models downstream risks caused by structural propagation of vulnerabilities and misconfigured dependencies, rather than direct attacks. Aligned with prior software supply chain attack taxonomies (Ladisa et al., 2023), these datasets go beyond synthetic constructs to replicate the ambiguity and complexity encountered in actual ecosystems. The reduced precision and recall in these scenarios highlight the difficulty of detecting complex threats across metadata, artifacts, and runtime layers.

Vulnerability detection remains inconsistent, with high false positives and poor handling of bundling changes and manifest modifications. Similarly, license detection struggles with expression differences and version details. The challenge of balancing detection breadth and stability remains unresolved. Broader coverage often leads to greater variation in recall and precision, making it difficult to achieve both sensitivity and consistency.

These results suggest that current SCA tools may function as effective **tools** in well-structured, single-language, source-level environments, where dependency information is explicit and standardized. However, in more realistic and challenging scenarios involving binary artifacts, multi-language architectures, and adversarial threats, they fail to maintain accuracy, coverage, and stability, acting more like **toys** that lack reliable and comprehensive results.

Overall, existing SCA tools have yet to balance **precision**, **coverage** and **robustness**. These issues show systemic weaknesses in managing **heterogeneous build forms**, **multiple adversarial threats** and **deep multi-language semantic analysis**, limiting their ability to address complex software supply chain risks effectively.

5.2. Implication

5.2.1. For evaluated SCA tool developers

Our evaluation reveals distinct capability gaps in each SCA tool when applied to multi-language, binary form, and adversarial scenarios. Addressing these limitations is essential to enhance their reliability in practical software supply chain environments.

OpenSCA lacks binary analysis capabilities and struggles with license recognition. As shown in Sections 3.4 and 4.1, it fails to detect any dependencies in the binary dataset DS2, and performs poorly on DS6, where embedded or non-standard license texts are missed. To improve, OpenSCA should integrate binary fingerprinting for compiled artifact analysis and adopt SPDX-based license parsing with semantic normalization.

Snyk shows limited support for binary and license-level detection, and performs inconsistently under adversarial conditions. It relies solely on manifest-based scanning, leading to zero detection in DS2. In DS3 and DS5, recall and precision drop sharply (Section 4.1), and standard deviation increases across metrics (Section 4.4). Snyk should extend its scope to binary and container artifacts, strengthen resilience to obfuscation, and stabilize results through iterative validation mechanisms.

CleanSource achieves high precision but consistently lower recall, particularly in multi-language and adversarial datasets (Table 2). Its conservative detection strategy avoids false positives but overlooks valid dependencies. Adopting fuzzy matching and feature-tolerant analysis could help capture implicit or partially modified components.

RA provides accurate results but suffers from limited scanning coverage. In DS1, it scans significantly fewer files than other tools (Table 3), due to strict assumptions about project structure (Section 4.2). To improve coverage, RA should relax layout constraints and implement dynamic project boundary inference for broader applicability.

5.2.2. General recommendations for SCA tool developers

Beyond tool-specific issues, our findings highlight common limitations in existing SCA tools, especially in handling **non-standard scenarios** such as binary dependency. Compilation optimizations often obscure dependencies, leading to reduced recall and precision. Addressing these challenges and expanding support for diverse compilers and platforms like GCC, Clang, and MSVC are essential for improving dependency tracing in source-less environments.

Improving **multi-language support** is equally important. As multi-language projects are becoming more common, SCA tools need better handling of diverse programming languages. Flexible dependency resolution algorithms that support heterogeneous build systems like Maven, npm and pip would enhance the resolution of multi-language dependency chains, aligning with contemporary hybrid software development.

Optimizing **license detection** is another critical need. Existing SCA tools struggle to identify complex license patterns and distinguish between different license versions. Developers should refine license detection algorithms to improve accuracy and coverage.

Promoting **standardization** among SCA tools is crucial. Inconsistent license names and versions in SCA reports hinder cross-tool comparison. Establishing unified reporting standards would improve interoperability and consistency across different tools.

Finally, deeper **integration of SCA tools** into the development lifecycle should be prioritized. Developing plugins for real-time scanning and automated triggering mechanisms could enable immediate detection and risk alerts, strengthening supply chain security.

5.2.3. For SCA tool users

Selecting the appropriate SCA tool depends on **specific project requirements** since tools differ in functionality, language support and detection accuracy. For Java-based projects requiring high detection accuracy, OpenSCA is likely the best option. For multi-language projects, RA's broad coverage makes it a viable choice despite its varying performance across languages. For enterprise-level projects involving binary files, RA and CleanSource may offer a more effective solution. Additionally, combining **multiple tools** and **manual review** can further enhance project compliance and improve overall detection accuracy.

5.2.4. For researchers

Develop a more comprehensive **evaluation framework** for SCA tools, as current research mainly focuses on specific scenarios or single datasets. A broader framework should assess tool performance across diverse programming languages and build environments. Moreover, exploring **advanced detection techniques** is essential, as traditional SCA methods may face challenges with future threats. Using AI or machine learning can improve dependency detection and help identify complex attacks by uncovering deeper patterns. Monitoring supply chain attack trends and developing targeted defense strategies are also crucial.

6. Threats to validity

6.1. Internal validity

Subjectivity in data processing may introduce bias. Constructing ground-truth sets and prediction reports typically requires manual intervention. For instance, when ground-truth information is distributed across README files and documentation, data must be collected using a web crawler and manually organized into standardized reports. Similarly, processing tool outputs such as JSON or CSV, requires manually extracting key fields and defining uniform rules for dependency name normalization and versioning removal. Despite employing a semi-automated process to handle 663 licenses, which consumed approximately 50 h, potential errors remain.

Configuration differences across tools may also affect the replicability of results. Since configuration parameters affect detection performance, this study uses default settings to reduce variability.

6.2. External validity

Selection bias in dataset choice may affect the generalizability of the results. While this study uses six distinct datasets covering Java projects, binary files, multi-language projects and adversarial threats, these datasets may not fully capture the diversity of real-world software projects.

Temporal factors may also affect the validity of the findings. Since SCA tools' vulnerability databases and detection algorithms are continuously updated, the results reflect the tools' performance at a specific point in time. Subsequent improvements could render some conclusions outdated.

Finally, **excluding failed scans during data collection** may weaken result validity. Projects that failed to be successfully scanned were excluded from the study, which may introduce inconsistency in tool evaluation across the same dataset and weaken the overall reliability of the findings.

6.3. Construct validity

Limitations in evaluation metrics may restrict a comprehensive assessment of tool performance. This study primarily uses recall, precision, and standard deviation to evaluate performance. While these metrics capture core aspects, they may overlook factors such as usability, runtime efficiency and integration with development workflows.

Furthermore, **limited tool representation** may reduce the generalizability of the findings. The study evaluates only RA, CleanSource and OpenSCA, which are representative within the commercial and open-source domains. However, excluding widely used tools like Black Duck may limit the results' applicability to other SCA tools.

6.4. Conclusion validity

The **limitations of the experimental design** may restrict the applicability of the conclusions. Although the study tests the robustness of SCA tools against supply chain attacks using adversarial datasets, these operations may not capture the full complexity of real-world attacks, limiting the practical relevance of the findings. Moreover, while the study includes multi-language datasets, limited support for languages like C/C++ and Objective-C may affect the accuracy of tool performance evaluation in multi-language environments.

7. Related work

7.1. Research on existing SCA tools

In research on SCA tools, numerous studies have explored their strengths and weaknesses. These tools differ in functional features, operational mechanisms and detection accuracy. For instance, some tools scan code repositories to identify vulnerabilities. OWASP Dependency Check leverages multiple third-party data sources to detect publicly disclosed vulnerabilities in project dependencies (Imtiaz et al., 2021). In contrast, Snyk Open Source uses its proprietary Snyk Intel vulnerability database to scan project manifest files, build a dependency tree and detect vulnerabilities (Sharma et al., 2024). However, existing SCA tools still struggle with complex software environments, leading to issues such as inaccurate dependency resolution and false positives in vulnerability detection (Zhao et al., 2023b; Wu et al., 2023; Imtiaz et al., 2021; Jiang et al., 2024). Dietrich et al. (2023) showed that shading and cloning operations significantly reduce the accuracy of SCA tools, highlighting their weakness in handling dependency modifications.

Despite extensive research on SCA tools, a comprehensive evaluation that spans functionality and environmental complexity remains absent. Prior studies often focus solely on vulnerability detection outcomes (Prana et al., 2021), neglecting variations in dependency resolution and complex scenarios, or are limited to specific ecosystems (Zhan et al., 2020; Jiang et al., 2024; Imtiaz et al., 2021) and programming languages (Zhao et al., 2023b; Dann et al., 2021), which reduces their applicability in broader contexts. Our study addresses this gap by jointly evaluating detection capabilities under realistic, cross-dimensional conditions.

Zhan et al. (2020) investigated binary scanning approaches and identified common steps in feature-based detection, covering aspects such as version inference and obfuscation recovery. However, their comparison was limited to tool-level scanning strategies and did not analyze how different build forms within the same tool impact detection outcomes. Our work extends this by comparing the stability and effectiveness of the same SCA tool under both source-based and binary-based modes, revealing internal inconsistencies that affect accuracy.

Prana et al. (2021) conducted a comparative analysis of popular SCA tools based on vulnerability detection and CI integration. They found that even tools using the same database, such as Snyk and Red Hat, report different vulnerabilities, indicating variations in algorithms or implementation. We extend this work by examining how tool performance changes under adversarial conditions like metadata poisoning and dependency obfuscation, showing that most tools lack robustness in such scenarios.

Unlike previous studies that rely heavily on average precision or recall, we introduce detection stability as a critical metric, using standard deviation across datasets to assess the consistency of tool performance. This reveals that tools with broader coverage often suffer from high performance variance, a trade-off not captured by existing work.

Imtiaz et al. (2021) benchmarked nine SCA tools on OpenMRS, focusing on runtime, vulnerability count, and dependency tracking. While valuable, their study centered on a single application and lacked coverage of multi-language support and adversarial behavior. In contrast, our experiments span six datasets, including both curated real-world projects and adversarially designed cases (e.g., DS3, DS5), which reflect common patterns in real attacks such as SolarWinds and typosquatting (Ohm et al., 2020; Martínez and Durán, 2021).

Zhao et al. (2023a) proposed a model to evaluate dependency resolution in the Maven ecosystem (SSM), reporting average F1 scores and accuracy limitations in the build and pre-build phases. While their results provide insight into Maven-specific tool performance, our study expands this to multi-language ecosystems and demonstrates that tools often perform inconsistently when faced with language-specific build systems and binary-only releases.

Overall, our contribution lies not only in aggregating detection results but in uncovering interaction effects between detection functions and real-world scenarios—such as the observed degradation of license recognition in binary mode, or the instability of vulnerability detection under adversarial construction. This integrated view enables us to distinguish between tools that merely function under ideal settings and those capable of handling complex, evolving software supply chains.

7.2. Research on open-source dependencies, vulnerabilities, and license detection

In research on open-source dependencies, vulnerabilities and license detection, scholars have proposed various methods and tools to address the security and compliance risks associated with OSS. Vulnerabilities in open-source components often stem from TPLs or frameworks, typically due to coding errors, misconfigurations, or outdated dependencies. For example, when the Apache Log4j2 vulnerability was disclosed, it had a significant impact, affecting over 35,000 Java packages and more than 8% of the Maven ecosystem (Zhao et al., 2023b; Wu et al., 2023; Wetter and Ringland, 2021). The scope of these vulnerabilities is broad, particularly when popular libraries are affected, as all projects depending on those libraries are at risk (Zhao et al., 2023a).

To detect and mitigate these risks, researchers have proposed various strategies and tools. Tang et al. (2022) introduced LibDB, a framework for detecting TPLs in C++ binaries. LibDB includes three main modules: the feature extraction module, which identifies basic and function vector features from binary files and constructs function call graphs; the fast detection module, which searches for candidate libraries in a local database using feature channels; and the FCG filtering module, which compares the function call graphs of candidate libraries and the detection target using a graph embedding network to filter out incorrectly reported libraries (Tang et al., 2022). Similarly, Duan et al. (2017) proposed OSSPolice, a tool for detecting dependencies in Java and C/C++ binaries. OSSPolice generates software feature signatures using syntactic features like string literals and functions and achieves high scalability and accuracy through a hierarchical indexing scheme. However, its performance is limited by control flow obfuscation (Duan et al., 2017). Additionally, Li et al. (2024) presented a method for analyzing TPLs based on pre-built dependency graphs. By employing a Common Platform Enumeration (CPE) transformation algorithm, this method constructs localized dependency graphs, enabling teams to perform TPL analysis and vulnerability scanning within their local environments. Experimental results showed that this approach is efficient while maintaining high precision. Zhang et al. (2018) introduced LibPecker, which uses class dependencies as code features and employs fuzzy class matching to identify TPLs. Although this method has high time complexity during the feature matching phase, its similarity-based feature calculation based on package structure provides valuable insights for future research.

In vulnerability detection, Zhao et al. (2023b) conducted a large-scale empirical study to explore dependency vulnerabilities. Using the Veracode SCA tool, they analyzed 450 projects in Java, Python and Ruby, revealing that "denial of service" and "information leakage" are common vulnerability types across these languages. They also found that high-severity vulnerabilities are more frequent in Java and Python projects. Additionally, the study showed that the number of direct dependencies in a project has a greater impact on vulnerabilities than on factors such as project age or commit frequency. Furthermore, Ivanova et al. (2024) highlighted that even small metadata modifications can significantly affect SCA tool detection, leading to false positives or negatives.

In license detection, Zhao et al. (2023b) proposed an efficient and accurate model for identifying license texts, which includes an extraction module and a recognition module. Experimental results demonstrated that this model outperforms typical existing license detection tools in terms of both accuracy and recall. However, current SCA tools still fall short in license detection, especially in complex dependency networks, where identifying which dependencies pose genuine risks remains challenging (Dann et al., 2021).

Building on these findings, our study expands the evaluation scope by incorporating six curated datasets across multiple dimensions, including Java and binary dependencies, multi-language projects, complex licensing, and adversarial scenarios. Unlike prior studies that focused on a single detection capability (Tang et al., 2022; Zhao et al., 2023b; Zhang et al., 2018; Ivanova et al., 2024; Dann et al., 2021), our evaluation integrates dependency detection, vulnerability identification, and license recognition into a unified assessment using both commercial and open-source tools. This approach enables a comprehensive examination of SCA tool performance under realistic conditions that better reflect modern software development practices.

8. Conclusion

This study systematically evaluates the capabilities of SCA tools in OSS governance by introducing a comprehensive evaluation model covering dependency detection, vulnerability identification, and license recognition. The framework focuses on multi-language ecosystems, source and binary dependency tracking, and adversarial threat. Using standardized test sets and quantitative metrics, the results reveal substantial performance gaps across tools and scenarios. While some tools perform well in source-based, single-language settings, they often struggle in more complex scenarios. These challenging conditions demonstrate that SCA tools are not yet ready to meet the demands of real-world software supply chain environments. These findings suggest that current SCA tools function as practical tools in idealized conditions, but remain limited toys when exposed to the real-world software supply chain risks. The study underscores the need for better multi-language support, more reliable binary analysis, and stronger resilience against evolving threats. Although it offers targeted recommendations for tool developers, users, and researchers, the study has certain limitations, including dataset diversity and tool selection constraints. Additionally, temporal factors and evolving attack techniques may affect the generalizability of the results. Our future research will prioritize enhancing SCA tool readiness for these challenging scenarios to refine evaluation methodologies and improve tool effectiveness, ultimately advancing OSS security and governance.

CRediT authorship contribution statement

Congyan Shu: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Wentao Chen:** Validation, Supervision, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Guisheng Fan:** Supervision. **Huiqun Yu:** Supervision. **Zijie Huang:** Supervision, Methodology, Conceptualization. **Yuguo Liang:** Supervision.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT-4o in order to manuscript polishing. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China[http://dx.doi.org/10.13039/501100001809] (No. 62372174 and NO. 62276097), the Computational Biology Program of Shanghai Science and Technology Commission (No. 23JS1400600), the Research Programme of National Engineering Laboratory for Big Data Distribution and Exchange Technologies (No. 2021-GYHLW-01007), the China Postdoctoral Science Foundation, China under Grant 2024M761927, and the Shanghai 2024 Science and Technology Innovation Action Plan Star Cultivation (Sailing Program, 24YF2720000).

Data availability

All datasets and ground-truth annotations used in this study are publicly available at: https://github.com/ErqiFang/Benchmarking-SCA-Tools.

References

Bacci, Alessandro, Bartoli, Alberto, Martinelli, Fabio, Medvet, Eric, Mercaldo, Francesco, Visaggio, Corrado Aaron, 2018. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In: ICISSP. pp. 379–385, https://doi.org/10.5220/0006642503790385.

Dann, Andreas, Plate, Henrik, Hermann, Ben, Ponta, Serena Elisa, Bodden, Eric, 2021. Identifying challenges for OSS vulnerability scanners: A study & test suite. IEEE Trans. Softw. Eng. 48 (9), 3613–3625, https://doi.org/10.1109/tse.2021.3101739.

Decan, Alexandre, Mens, Tom, Grosjean, Philippe, 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng. 24 (1), 381–416, https://doi.org/10.1007/s10664-017-9589-y.

Dietrich, Jens, Rasheed, Shawn, Jordan, Alexander, White, Tim, 2023. On the security blind spots of software composition analysis. In: Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses. pp. 77–87, https://doi.org/10.1145/3689944.3696165.

Duan, Ruian, Bijlani, Ashish, Xu, Meng, Kim, Taesoo, Lee, Wenke, 2017. Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2169–2185, https://doi.org/10.1145/3133956.3134048.

Duan, Moming, Li, Qinbin, He, Bingsheng, 2024. ModelGo: A practical tool for machine learning license analysis. In: Proceedings of the ACM Web Conference 2024. pp. 1158–1169, https://doi.org/10.1145/3589334.3645520.

Feng, Qiong, Ji, Huan, Ma, Xiaotian, Liang, Peng, 2024. Cross-language dependencies: An empirical study of kotlin-java. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 189–199, https://doi.org/10.1145/3674805.3686680.

Fourné, Marcel, Wermke, Dominik, Enck, William, Fahl, Sascha, Acar, Yasemin, 2023. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In: 2023 IEEE Symposium on Security and Privacy. SP, pp. 1527–1544, https://doi.org/10.1109/sp46215.2023.10179320.

Global TMT, 2025. Successful deployment of SecTrend CleanSource SCA tool at tencent. https://www.tmtnews.tech/archives/29062. (Accessed 25 March 2025).

Hiesgen, Raphael, Nawrocki, Marcin, Schmidt, Thomas C., Wählisch, Matthias, 2024. The log4j incident: A comprehensive measurement study of a critical vulnerability. IEEE Trans. Netw. Serv. Manag. https://doi.org/10.1109/tnsm.2024.3440188.

Imtiaz, Nasif, Thorn, Seaver, Williams, Laurie, 2021. A comparative study of vulnerability reporting by software composition analysis tools. In: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–11, https://doi.org/10.1145/3475716.3475769.

Ivanova, Ekaterina, Stakhanova, Natalia, Sistany, Bahman, 2024. Adversarial analysis of software composition analysis tools. In: International Conference on Information Security. pp. 161–182, https://doi.org/10.1007/978-3-031-75764-8_9.

Jiang, Ling, An, Junwen, Huang, Huihui, Tang, Qiyi, Nie, Sen, Wu, Shi, Zhang, Yuqun, 2024. BinaryAI: Binary software composition analysis via intelligent binary source code matching. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13, https://doi.org/10.1145/3597503.3639100.

Jiang, Ling, Yuan, Hengchen, Tang, Qiyi, Nie, Sen, Wu, Shi, Zhang, Yuqun, 2023. Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we? In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1383–1395, https://doi.org/10.1145/3597926.3598143.

Kengo Oka, Dennis, 2021. Software composition analysis in the automotive industry. In: Building Secure Cars: Assuring the Automotive Software Development Lifecycle. pp. 91–110, https://doi.org/10.1002/9781119710783.ch6.

Ladisa, Piergiorgio, Plate, Henrik, Martinez, Matias, Barais, Olivier, 2023. SoK: Taxonomy of attacks on open-source software supply chains. In: 2023 IEEE Symposium on Security and Privacy. SP, pp. 1509–1526, https://doi.org/10.1109/sp46215.2023.10179304.

Li, Rongye, Zhao, Erfan, Chen, Xinglong, Hao, Kegang, Hou, Anran, Gong, Jiacheng, 2024. Pre-built dependency graph-based TPLs analysis. In: 2024 4th International Symposium on Computer Technology and Information Science. ISCTIS, pp. 666–671, https://doi.org/10.1109/isctis63324.2024.10698860.

Ma, Yuxing, 2018. Constructing supply chains in open source software. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion. ICSE-Companion, pp. 458–459, https://doi.org/10.1145/3183440.3183454.

Martínez, Jeferson, Durán, Javier M., 2021. Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. Int. J. Saf. Secur. Eng. 11 (5), 537–545, https://doi.org/10.18280/ijsse.110505.

Mayer, Philip, Kirsch, Michael, Le, Minh Anh, 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. J. Softw. Eng. Res. Dev. 5, 1–33, https://doi.org/10.1186/s40411-017-0035-z.

Microsoft, 2025. Software composition analysis. https://learn.microsoft.com/en-us/devsecops/playbook/capabilities/security/software-composition-analysis. (Accessed 15 March 2025).

O'Donoghue, Eric, Reinhold, Ann Marie, Izurieta, Clemente, 2024. Assessing security risks of software supply chains using software bill of materials. In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion. SANER-C, pp. 134–140, https://doi.org/10.1109/saner-c62648.2024.00023.

Ohm, Marc, Plate, Henrik, Sykosch, Arnold, Meier, Michael, 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17. pp. 23–43, https://doi.org/10.1007/978-3-030-52683-2_2.

Ombredanne, Philippe, 2020. Free and open source software license compliance: Tools for software composition analysis. Computer 53 (10), 105–109, https://doi.org/10.1109/mc.2020.3011082.

Open Source Initiative, 2025. The open source definition. https://opensource.org/osd. (Accessed 17 March 2025).

Pei, Kexin, She, Dongdong, Wang, Michael, Geng, Scott, Xuan, Zhou, David, Yaniv, Yang, Junfeng, Jana, Suman, Ray, Baishakhi, 2022. NeuDep: Neural binary memory dependence analysis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 747–759, https://doi.org/10.1145/3540250.3549147.

Ponta, Serena Elisa, Plate, Henrik, Sabetta, Antonino, 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 449–460, https://doi.org/10.1109/icsme.2018.00054.

Prana, Gede Artha Azriadi, Sharma, Abhishek, Shar, Lwin Khin, Foo, Darius, Santosa, Andrew E, Sharma, Asankhaya, Lo, David, 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. Empir. Softw. Eng. 26, 1–34, https://doi.org/10.1007/s10664-021-09959-3.

Sharma, Pranet, Shi, Zhenpeng, Şimşek, Şevval, Starobinski, David, Medina, David Sastre, 2024. Understanding similarities and differences between software composition analysis tools. IEEE Secur. Priv. https://doi.org/10.1109/msec.2024.3410957.

Snyk CLI Developers, 2025. Snyk CLI releases. https://github.com/snyk/cli/releases. (Accessed 15 June 2025).

Snyk Documentation, 2025. What is snyk. https://docs.snyk.io/what-is-snyk. (Accessed 15 June 2025).

Sorocean, Oleg, Ayala-Rivera, Vanessa, Portillo-Dominguez, A. Omar, 2024. Enhancing visibility of components and dependencies across diverse it environments with open-source software-bill-of-materials generation tools. In: 2024 12th International Conference in Software Engineering Research and Innovation. CONISOFT, pp. 165–174, https://doi.org/10.1109/conisoft63288.2024.00030.

SourceClear, 2025. Evaluation framework for dependency analysis. https://github.com/srcclr/efda. (Accessed 9 March 2025).

SPDX, 2025. SPDX license list data. https://github.com/spdx/license-list-data. (Accessed 1 March 2025).

Sushma, D, Nalini, MK, Kumar, R Ashok, Nidugala, MuraliKrishna, 2023. To detect and mitigate the risk in continuous integration and continues deployments (CI/CD) pipelines in supply chain using snyk tool. In: 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions. CSITSS, pp. 1–10, https://doi.org/10.1109/csitss60515.2023.10334136.

Tang, Wei, Wang, Yanlin, Zhang, Hongyu, Han, Shi, Luo, Ping, Zhang, Dongmei, 2022. LibDB: An effective and efficient framework for detecting third-party libraries in binaries. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 423–434, https://doi.org/10.1145/3524842.3528442.

Wang, Xueqiang, Zhang, Yifan, Wang, XiaoFeng, Jia, Yan, Xing, Luyi, 2023. Union under duress: Understanding hazards of duplicate resource mismediation in android software supply chain. In: 32nd USENIX Security Symposium. USENIX Security 23, pp. 3403–3420, https://dlnext.acm.org/doi/10.5555/3620237.3620428.

Wermke, Dominik, Klemmer, Jan H., Wöhler, Noah, Schmüser, Juliane, Ramulu, Harshini Sri, Acar, Yasemin, Fahl, Sascha, 2023. "Always contribute back": A qualitative study on security challenges of the open source supply chain. In: 2023 IEEE Symposium on Security and Privacy. SP, pp. 1545–1560, https://doi.org/10.1109/sp46215.2023.10179378.

Wetter, James, Ringland, Nicky, 2021. Understanding the impact of apache log4j vulnerability. Retrieved January 11:2022. https://doi.org/10.1007/978-1-4302-0034-5_2.

Wikipedia contributors, 2025a. Npm left-pad incident. https://en.wikipedia.org/wiki/Npm_left-pad_incident. (Accessed 11 March 2025).

Wikipedia contributors, 2025b. Software composition analysis. https://en.wikipedia.org/wiki/Software_composition_analysis. (Accessed 15 March 2025).

Wu, Yulun, Yu, Zeliang, Wen, Ming, Li, Qiang, Zou, Deqing, Jin, Hai, 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. pp. 1046–1058, https://doi.org/10.1109/icse48619.2023.00095.

Yang, Haoran, Nong, Yu, Wang, Shaowei, Cai, Haipeng, 2024. Multi-language software development: Issues, challenges, and solutions. IEEE Trans. Softw. Eng. 50 (3), 512–533, https://doi.org/10.1109/tse.2024.3358258.

Zahan, Nusrat, 2023. Software supply chain risk assessment framework. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, pp. 251–255, https://doi.org/10.1109/icse-companion58688.2023.00068.

Zhan, Xian, Fan, Lingling, Liu, Tianming, Chen, Sen, Li, Li, Wang, Haoyu, Xu, Yifei, Luo, Xiapu, Liu, Yang, 2020. Automated third-party library detection for android applications: Are we there yet? In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 919–930, https://doi.org/10.1145/3324884.3416582.

Zhang, Yuan, Dai, Jiarun, Zhang, Xiaohan, Huang, Sirong, Yang, Zhemin, Yang, Min, Chen, Hao, 2018. Detecting third-party libraries in android applications with high precision and recall. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 141–152, https://doi.org/10.1109/saner.2018.8330204.

Zhao, Lida, Chen, Sen, Xu, Zhengzi, Liu, Chengwei, Zhang, Lyuye, Wu, Jiahui, Sun, Jun, Liu, Yang, 2023b. Software composition analysis for vulnerability detection: An empirical study on java projects. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 960–972, https://doi.org/10.1145/3611643.3616299.

Zhao, Yuhang, Liang, Ruigang, Chen, Xiang, Zou, Jing, 2021. Evaluation indicators for open-source software: a review. Cybersecurity 4, 1–24, https://doi.org/10.1186/s42400-021-00084-8.

Zhao, Jun, Ren, Yi, Li, Bao, Zhao, Xin, Zhang, Jianfeng, 2023a. FOSSLT: An efficient model for automatic finding open source software license texts. In: 2023 2nd International Conference on Cloud Computing, Big Data Application and Software Engineering. CBASE, pp. 269–275, https://doi.org/10.1109/cbase60015.2023.10439089.