Evaluation of Free and Open Source Tools for Automated Software Composition Analysis

Laura Bottner
Mercedes-Benz Tech Innovation GmbH
Ulm, Germany
laura.bottner@mercedes-benz.com

Artur Hermann
Ulm University, Institute of Distributed Systems
Ulm, Germany
artur.hermann@uni-ulm.de

Jeremias Eppler
Mercedes-Benz Tech Innovation GmbH
Ulm, Germany
jeremias.eppler@mercedes-benz.com

Thomas Thüm
Ulm University, Institute of Software Engineering and Programming Languages
Ulm, Germany
thomas.thuem@uni-ulm.de

Frank Kargl
Ulm University, Institute of Distributed Systems
Ulm, Germany
frank.kargl@uni-ulm.de

ABSTRACT

Vulnerable or malicious third-party components introduce vulnerabilities into the software supply chain. Software Composition Analysis (SCA) is a method to identify direct and transitive dependencies in software projects and assess their security risks and vulnerabilities.

In this paper, we investigate two open source SCA tools, Eclipse Steady (ES) and OWASP Dependency Check (ODC), with respect to vulnerability detection in Java projects. Both tools use different vulnerability detection methods. ES implements a code-centric and ODC a metadata-based approach. Our study reveals that both tools suffer from false positives. Furthermore, we discover that the success of the vulnerability detection depends on the underlying vulnerability database. Especially ES suffered from false negatives because of the insufficient vulnerability information in the database.

While code-centric and metadata-based approaches offer significant potential, they also come with their respective downsides. We propose a hybrid approach assuming that combining both detection methods will lead to less false negatives and false positives.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Abstraction, modeling and modularity*; Risk management; **Empirical software validation**; Software testing and debugging.

CSCS '23, December 05, 2023, Darmstadt, Germany

ACM ISBN 979-8-4007-0454-3/23/12…$15.00
https://doi.org/10.1145/3631204.3631862

KEYWORDS

Software Composition Analysis, Vulnerable Dependency Identification, Software Supply Chain Security, Secure Software Development Life Cycle

ACM Reference Format:
Laura Bottner, Artur Hermann, Jeremias Eppler, Thomas Thüm, and Frank Kargl. 2023. Evaluation of Free and Open Source Tools for Automated Software Composition Analysis. In *Computer Science in Cars Symposium (CSCS '23), December 05, 2023, Darmstadt, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3631204.3631862

1 INTRODUCTION

The automotive industry is an example of a domain where there is an ever-increasing amount of (safety-)critical functionality controlled by software. Connectivity is creating a larger surface for remote attacks, and software complexity makes it tremendously difficult to manage and eliminate vulnerabilities. While this is also true for many other domains such as IoT, smart homes, and Industry 4.0, we focus our work on the automotive domain as it was our motivation to investigate Software Composition Analysis (SCA).

The combination of safety-critical functionality controlled by software together with remote access and vulnerabilities can lead to disastrous consequences for human lives, as the experience of the journalist Andy Greenberg attests to. He experienced first hand a remote take over of the car he was sitting in [13]. The security researchers Charlie Miller and Chris Valasek, demonstrating an attack to his car, were able to control the engine speed remotely while he was driving on a highway at speed. They gained access to the internal car control through the infotainment system [36] which was connected to a mobile network that allowed bi-directional communication to the car. The software of the infotainment system was developed by a third party. This example demonstrates three main points. First, software has become an integral part of cars and can be vulnerable to attacks. Second, it is important to secure the IT infrastructure the car is connected to. Third, software developed by third-parties can introduce vulnerabilities enabling attacks. Next, we elaborate on each point in more detail.

The previous attack demonstrates how connecting our vehicles to surrounding infrastructure and back-end systems increases the

attack surface and makes them more vulnerable as attackers can mount remote attacks [8, 34, 37, 40]. In another instance, a team of security researchers was able to attack connected systems of major automotive companies and in some instances gain remote access to the cars itself [30]. In the case of Mercedes-Benz, the car connected back-end systems were running software written in Java.

This remote exposure is critical as automotive software is highly complex and often consists of 100 millions lines of code or more, running on the electronic control units (ECUs) and telematics and infotainment systems inside the car [3, 29]. ECUs can range from small microcontrollers to embedded devices able to run an operating system [6, 31]. From windshield wipers to the electric steering, ECUs are used to control a broad variety of functions [2, 19, 22]. This software complexity introduces a very high risk for software vulnerabilities, which attackers can exploit once they can access a car [9, 20, 33, 35].

In many cases, these vulnerabilities originate from third-party software that is not even developed by the OEM. These third-party software either comes from suppliers or, to an increasing percentage, consists of open source components [15]. For example, Automotive Grade Linux for the infotainment system [21].

These third-party software components pose the challenge of securing the software supply chain in and around the car [32]. In one example, security researchers were able to attack an electric vehicle charging station by exploiting the Log4Shell vulnerability in the logging library Log4j, which was used by the charging station software written in Java. [7].

Software composition analysis (SCA) is one approach to secure the software supply chain. SCA analyses third-party components contained in a software project to identify vulnerabilities in the components. For this purpose, information about dependencies and versions of a software project are determined and compared with a vulnerability database.

This paper investigates and compares the two Free and Open Source (FOSS) SCA tools, namely Eclipse Steady (ES) and OWASP Dependency Check (ODC). We analyze a variety of proof-of-concept and real-world Java projects each of which has at least one known vulnerable dependency. The focus is on Java, because Java is used in back-end systems of connected car software and therefor part of the attack surface of vehicle-to-everything (V2X) applications.

We test both tools against the data set (projects) with known vulnerabilities and automate the workflow to improve the reproducability of the analysis. Additionally, we examine the results of the metadata and code-centric approaches with the help of a high effort manual code review.

2 STATE OF THE ART

Modern software projects often use many third-party libraries as dependencies. However, vulnerabilities in these dependencies can lead to potential security risks in these projects. To secure software projects and the corresponding software supply chain against malicious software components, there exists several standards, such as the *Supply Chain Levels for Software standard* (SLSA) [12] and the *OWASP Software Component Verification Standard* (SCVS) [11]. In particular, SCVS proposes a component analysis to identify potential risk areas in the use of third-party software components, which is exactly what SCA does. With the increasing popularity of open source libraries over the years, SCA has already become a necessary process for application security programs [24], making it highly relevant to the development process of new software products.

There exist already several approaches of SCA, which can be categorised into two groups. In the first group of SCA tools, a Software Bill of Materials (SBOM) already exists for a software project. Such an SBOM shows the entire dependencies of an application and its versions. These dependencies are then checked for vulnerabilities. An example of an SCA tool is Dependency Track, which can be used to monitor SBOMs [1]. However, the disadvantage of such SCA tools is that an SBOM is required for the corresponding software component, which is not always the case. For example, the study of Boming et al. found that most existing third-party software products or components do not come with SBOMs [39]. Furthermore, there is no consensus on the exact information that should be included in SBOMs.

In contrast, the second group of SCA tools does not rely on already existing SBOMs, as it can scan the software automatically. Based on the source code, the automated SCA tool independently creates an SBOM and analyzes it for vulnerabilities in the dependencies. Therefore, in this case, it does not matter whether an SBOM was provided with the software component or not. For this reason, we focus on this second group of SCA tools in our work.

There exist already several SCA tools, which are either commercial or free and open source. Examples for such SCA tools are Black Duck from Synopsys [2], Synk [3], Veracode [4], OWASP Dependency Check [5], Eclipse Steady [6], and GitHub Dependabot [7]. An overview of the tools is shown in Table 1. The selected SCA tools used in this paper are described in detail in Section 3.

Automated SCA tools can be classified in two approaches: metadata-based and code-centric approaches. In the metadata-based approach, a package dependency network is created based on metadata and manifest files. This dependency network is used to create a vulnerability report for the corresponding software project [14]. In contrast, the code-centric approach analyzes the code of the software project and compares it with constructs of vulnerability fixes to create vulnerability reports [26].

There exist already some studies in which the different approaches of SCA tools have been used or analyzed. For example, Cadariu et al. used OWASP Dependency Check (ODC) as a vulnerability checker and created a vulnerability alert service based on it. As part of this, they scanned 75 software projects with ODC and determined the number of vulnerabilities and false positives [4]. However, only one SCA tool was used in this work. Therefore, no performance comparison was made with other SCA tools.

Prana et al. analysed 450 software projects implemented in three different programming languages with the SCA tool Veracode. They found that the number of vulnerabilities in a software project correlates with the number of direct and transitive dependencies. This

[1] dependencytrack.org
[2] synopsys.com
[3] snyk.io
[4] veracode.com
[5] owasp.org/www-project-dependency-check
[6] projects.eclipse.org/proposals/eclipse-steady
[7] github.com/dependabot

underlines the importance of careful management of dependencies in software projects [28]. In this work, also only one SCA tool was used, so that no performance comparison with other SCA tools was performed here either.

Pashchenko et al. analyzed the industrial impact of Eclipse Steady (ES), which uses a code-centric approach. For this purpose, 200 of the most popular open source Java libraries were examined. The study found that 20 % of the dependencies affected by a known vulnerability are not deployed and therefore not exploitable. In this work, only one SCA tool was used, so that also here no performance comparison between several SCA tools was conducted [23].

However, there is also some work that has analyzed and compared different SCA tools. For example, the SAP security research team Ponta et al. [26] conducted a study comparing the SCA tools ES and ODC. They scanned 444 software projects from SAP, which were all restricted to Maven projects. The results of their study showed that ODC reported a much higher number of vulnerabilities, but suffers from false positives. ES reported a lower number of vulnerabilities but suffers from false negatives. However, as all scanned projects were created by SAP, it could be the case that these projects have an SAP bias to fulfill some SAP policies. Examples could be a certain coding style or a certain Java version used in all SAP projects. Therefore, the results of ES could be different, when other projects are scanned. This could especially be the case if ES was optimized for SAP software projects, as ES was developed by SAP. Furthermore, only a very limited number of the findings of ES and ODC were reviewed to determine the number of false positives. Therefore, no detailed analysis of the correctness of the tools was conducted in this work.

Imtiaz et al. compared nine SCA tools. These included commercial tools as well as open source tools such as ES and ODC. The objective of the work was, among other things, to analyze the differences in the vulnerability detection and vulnerability reports create by the tools. The analysis showed that the tools report very different vulnerabilities and rarely report overlapping vulnerabilities [16]. In this work, however, only a comparison of the output of the tools was done by comparing the number of vulnerabilities and the created reports. An analysis of whether the output is correct in terms of false positives was not conducted.

The works mentioned above show that an in-depth analysis of different SCA tools based on a heterogeneous selection of software projects and an analysis of the correctness of the output of these tools has not been done in previous works. This analysis of the correctness of different SCA tools is done in this paper to compare their performance for heterogeneous software projects.

3 TOOLS FOR AUTOMATED SOFTWARE COMPOSITION ANALYSIS

In this section of our paper, we describe from a technical perspective the used Software Composition Analysis (SCA) tools in detail. We start by explaining our selection criteria for the tools.

As mentioned in the previous sections, many commercial and open source SCA tools exist. However, in our paper we limit ourselves to Free and Open Source (FOSS) SCA tools. FOSS SCA tools are used for two major reasons. First, understanding existing SCA approaches. Second, the ability to study and modify the tools.

Table 1: Comparison of SCA tools regarding the approach and supported languages

SCA Tool	Approach	Supported Languages
Synopsys BlackDuck	Metadata-based	C/C++, C#, Clojure, Erlang, Golang, Groovy, Java, JavaScript, Kotlin, Node.js, Objective-C, Perl, Python, PHP, R, Ruby, Scala, Swift
Snyk	Unknown	.NET, Apex, Bazel, C/C++, Elixir, Go, Java, JavaScript, Kotlin, Objective-C, PHP, Python, Ruby, Scala, Swift, TypeScript, VB.NET
Veracode	Unknown	.NET, C/C++, C#, Go, Java, JavaScript, Kotlin, Objective C, PHP, Python, Ruby, Swift, Scala
OWASP Dependency Check	Metadata-based	.NET, C/C++, Dart, Go, Java, JavaScript, Node.js, Objective-C, Perl, PHP, Python, Ruby, Swift
Eclipse Steady	Code-centric	Java, Python
GitHub Dependabot	Metadata-based	C#, Go, Java, JavaScript, PHP, Python, Ruby, Scala, Swift, TypeScript

Only a limited number of FOSS SCA tools exist. Of those tools, an even smaller number can be considered mature [25]. The three main selection criteria in our work are: used by a variety of projects or companies, under active development, and developed beyond a proof-of-concept.

Only Eclipse Steady and OWASP Dependency Check meet our selection criteria. Both tools support the analysis of Java well.

3.1 OWASP Dependency Check

OWASP Dependency Check (ODC) is a Software Composition Analysis (SCA) tool, which uses a metadata-based approach to identify vulnerabilities in dependencies. In more detail, ODC has five main components[17]: the engine, the scanner, the analyzer, and the report generator as well as local database.

The local H2 database contains a mirror of the National Institute of Technology (NIST) National Vulnerability Database (NVD). The NVD contains information about software components and their vulnerabilities. Each vulnerability in the NVD can be identified by the MITRE Common Vulnerabilities and Exposures (CVE) number. The CVE is a unique identifier assigned by MITRE. Each CVE in the NVD has one or more software component identifiers in form of the Common Platform Enumeration (CPE) attached to it. The CPE describes a software artifact [5]. It includes vendor, product and version information as well as some other details in the form: `cpe:/[Entry Type]:[Vendor]:[Product]:[Version]:[Revision]:...`. In case, a CPE is attached to a CVE, that means

that the software artifact is vulnerable. For example, CVE-2021-44228[1] or better know under it's popular name Log4Shell[38] has several CPEs attached to it indicating, that each of the attached software artifacts is vulnerable to Log4Shell.

The main functionality of ODC is to extract information from a dependency used in a software artifact and match it against the CPEs associated with a CVE. The ODC engine is the main controller that runs the other components in the order. The engine first executes the scanner. The scanner walks through the files and folders of the given projects and collects any file for which ODC has an analyzer able to process it. Once a list of processable files is created, the engine starts the analyzers. The analyzers try to collect information about the vendor, product and version [18]. ODC calls those three information types evidence. A number of analyzers exists and all of them look for specific indicators to extract the evidence. For example, the JarAnalyzer will collect information from the `Manifest`, `pom.xml`, and the package names within the JAR.

Next, the collected evidence is matched against the locally stored NVD using CPEs. Mismatches can occur. For example, in case the analyzer could not find sufficient information about the dependency.

In the last step, the report generator creates a report containing all the found vulnerabilities.

3.2 Eclipse Steady

Eclipse Steady (ES) is the second SCA FOSS tool evaluated in this paper. It works fundamentally different compared to ODC as it uses a code-centric approach to identify dependencies with vulnerabilities. ES uses static and dynamic code analysis to determine, whether a vulnerability in a dependency affects the analyzed project at all. ES calls this reachability analysis.

ES uses different tasks for the SCA and calls these tasks *goals* [10]. The first task "app" creates a Software Bill of Materials (SBOM) of the application with all direct and transitive dependencies. The SBOM contains the signatures of all methods of the application and it's dependencies. Next, ES matches all method signatures against the Kaybee database [8] with known vulnerability information.

The core functionality of ES is the reachability analysis, which tries to determine whether any of the vulnerable methods of a dependency is called directly or indirectly by the application. ES uses static or dynamic analysis approaches to determine, whether a vulnerable method is called. The app-to-code (a2c) goal analyses the app using static source code analysis and builds a call graph to identify what methods are called and whether any path reaches a vulnerable method. The dynamic analysis goal, called trace-to-code (t2c) first traces all called methods during the runtime of the application. The traces can either be collected by executing JUnit tests or by instrumenting the JVM and running the application. Once, the traces are collected a call graph is created to identify if any vulnerable method is called by the application.

To match the vulnerabilities on a method level, ES requires a different database containing these information. The high-level package information provided by the NVD are not sufficient. As a result, ES uses project Kaybee as it's data source. Kaybee does not only host CVE IDs and affected versions but also fix commits and a list of vulnerable methods [27].

[8] github.com/SAP/project-kb

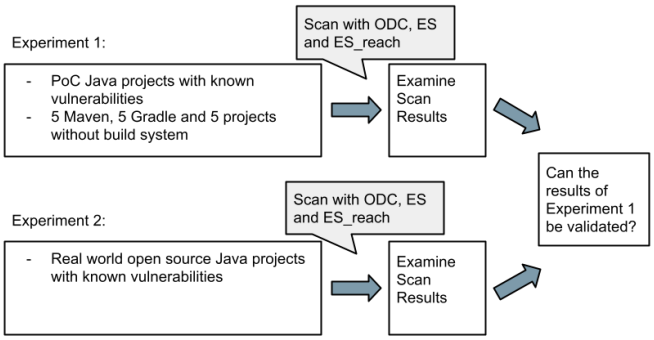

Figure 1: Study design with the first experiment scanning proof-of-concept projects and the second experiment scanning established open source Java projects

4 METHODOLOGY

In the following chapter, we first present the design of the empirical study. Then, a preliminary study is conducted to ensure that the scanning approaches provide comparable reports. After analyzing the results of the preliminary study, we implement the scanning process in Python and Shell scripts to make the results reproducible and automate the scan process. Finally, we present the study with proof-of-concept projects and the study with Java open source projects.

4.1 Study Design

To evaluate the two SCA tools, Eclipse Steady (ES) and OWASP Dependency Check (ODC), we conducted a study with two experiments. The study design is illustrated in Figure 1. In the first experiment, proof-of-concept (PoC) projects were scanned, each implements a proof-of-concept for a specific known vulnerability with an associated CVE ID. For this purpose, five projects without a build tool, five Maven projects, and five Gradle projects were selected. The selected projects are scanned with ES and ODC. Then, the ES reachability analysis (ES_reach) is performed. After that, the ES and ODC reports and the ES and ES_reach reports are analyzed and the results are evaluated.

In the second experiment, established and mature open source Java projects are scanned at a commit, when a known vulnerability existed in the project. For this purpose, five Maven and five Gradle projects are selected, each of which has a known vulnerability. Then, the different scans are performed and the results are evaluated. Finally, the results of the two experiments are reviewed and compared to determine if they are consistent and therefore valid, or if they contradict each other.

4.2 Evaluation Methods

In the following, we present the independent and dependent variables of our experimental study. The independent variable is the scan method. We perform a SCA scan with ODC and a scan with

Table 2: Overview of examined dependent variables per experiment. The dependency analysis and the manual examination were only performed in the first experiment

Scope	Metrics	Exp. 1	Exp. 2
Known Vulnerabilities	TP, FN Rate	×	×
	ES Reachability Analysis	×	×
	Build Tool	×	×
	CVSS Score	×	×
All reported Vulnerabilities	Number of reported Vulnerabilities	×	×
	ES Reachability Analysis	×	×
	Dependency Analysis	×	
	Manual TP, FN examination on Subset	×	

Table 3: Number of vulnerabilities found per project using the CLI, the Maven and Gradle plugins and the CLI together with the plugins

	CLI		plugins		CLI + plugins	
Project ID	ODC	ES	ODC	ES	ODC	ES
CVE-2020-2555 (none)	22	1	0	0	22	1
certPinningVulnerableOkHttp (none)	4	1	0	0	4	1
Struts2Vuln (Maven)	0	0	94	56	94	56
SpringBreakVulnerableApp (Maven)	0	0	129	62	0	0
log4shell-vulnerable-app (Gradle)	0	0	29	48	29	48
cve-2019-14540-exploit (Gradle)	0	0	50	11	0	0

ES. Further we perform the reachability analysis ES_reach. The dependent variables are those that are affected by the manipulation of the scan method. Table 2 provides an overview of all dependent metrics that were examined in the experiments. The vulnerabilities in the projects, which are known in advanced are categorized in TP (identified) and FN (unidentified). We investigate the different build tools Maven and Gradle as they resolve the dependencies and the CVSS Score for possible correlation with vulnerability detection. As it can be seen, we examined the PoC scans more closely than the open source Java projects. The reason for this is that the open source projects are too complex to study their dependencies and perform a manual examination.

4.3 Prestudy

In order to perform scans correctly with the SCA tools ES and ODC and thus obtain meaningful results, both tools were tested in advance. This step is necessary because both tools can scan a project in different ways. There exist a Command Line Interface (CLI) for both tools as well as plugins, which need to be added to the build configuration files. During the preliminary study, we explored different scanning approaches and integrations, including the use of Maven plugins, Gradle plugins, and the CLI. We scanned a total of six projects, including two projects without a build tool, two projects using Maven, and two projects using Gradle. These projects have been chosen randomly out of a subset of proof-of-concept projects containing vulnerable dependencies. The results presented in table 3 show that using the command-line interface alone could not identify vulnerabilities in the Maven and Gradle projects because it cannot resolve dependent libraries. Similarly, the tools that relied solely on plugins were unable to identify vulnerabilities in the projects without a build tool. The combination of plugins and CLI also failed to detect vulnerabilities in one Maven and one Gradle project. We therefore decided to implement scan execution for projects without a build tool using the CLI and for Maven and Gradle projects using the tools' Maven and Gradle plugins.

The insights gained from the pre-study were instrumental in the implementation of Shell and Python scripts to automate the scan process for both tools and the comparison of detected vulnerabilities for each project. The scripts enable us to reproduce scan results and facilitate the evaluation of the scan results.

The scan process consists of five steps. First the projects to be scanned are cloned or downloaded, and necessary preparations like plugins or scan configurations are added. Next the projects are scanned using the CLI, the `./gradlew` or `mvn` command. The reports generated from the projects are combined, resulting in one report for ODC and one for ES. These reports are then compared, and the findings are classified into CVE IDs that were identified by both scan tools or only by ODC or ES. Finally, the reachability analysis is conducted for ES and compared with the previous ES results obtained.

4.4 Experiment 1

The first experiment requires proof-of-concept (PoC) projects that implement a specific known vulnerability, to check if the tools can identify the vulnerability. These projects are typically smaller in scale and are used to showcase these vulnerabilities. Each PoC project implements an exploit for a vulnerability, which is identified by a CVE ID. To find relevant projects on GitHub, we conducted searches using terms like "CVE" and "vulnerable". In total, we chose 15 Java projects for this experiment as shown in Table 4. Among these, five projects did not use a build tool but imported `JAR` files to scan with the command line interface (CLI). The remaining projects used Maven and Gradle as build tools, for each five projects were selected. There were more than five projects available for both Maven and Gradle, therefore the final selection was made randomly. We limit ourselves to 15 projects, as projects without a build tool were hard to find and the effort to examine more results would be beyond the scope.

After selecting the projects, the scan scripts were executed and the results collected and prepared for analysis. In the first experiment, we used additional tools for the evaluation of the ODC and ES

Table 4: Table of scanned proof-of-concept projects with the implemented vulnerability CVE ID and the projects build system

Project ID	CVE ID	Build tool
CVE-2020-2555	CVE-2020-2555	none
certPinningVulnerableOkHttp	CVE-2017-3586	none
Vulnerable-Web-App	CVE-2021-35464	none
CVE-2019-2890	CVE-2019-2890	none
log4j-vul	CVE-2021-44228	none
CVE-2020-5398	CVE-2020-5398	Gradle
log4shell-vulnerable-app	CVE-2021-44228	Gradle
CVE-2022-22965-PoC	CVE-2022-22965	Gradle
spring-cloud-netflix-hystrix-dashboard-cve-2021-22053	CVE-2021-22053	Gradle
Log4J-RCE-Proof-Of-Concept	CVE-2018-1000134	Gradle
text4shell-poc	CVE-2022-42889	Maven
CVE-2022-22980	CVE-2022-22980	Maven
SpringBreakVulnerableApp	CVE-2017-8046	Maven
spring4shell_victim	CVE-2022-22965	Maven
struts2-rce	CVE-2017-5638	Maven

scans. We counted the number of direct and transitive dependencies by creating an SBOM with CycloneDX [9].

4.5 Experiment 2

The second experiment was conducted using mature open source Java projects. To identify suitable projects, the Google search engine was utilized to search for popular Java projects. Once projects with publicly available source code were identified, we searched in the CVE list [10] for a vulnerable version of the projects.

If a vulnerable version was found, the download link was saved, and an attempt was made to build the project locally. Building the projects was challenging, because some versions rely on outdated libraries or external repositories.

The project selection criteria included the ability to build the projects locally, the presence of at least one known security vulnerability, a certain degree of maturity of the projects, and publicly available source code. Based on this criteria, we chose five Maven projects and five Gradle projects as shown in Table 5. Projects without a build tool were excluded from the second experiment, as we were unable to find a project that met the criteria and did not require a build tool.

After selecting the projects, the scan scripts were executed again. Since the Java open source projects are more complex and partly rely older Java versions, we could not execute the scan scripts completely automatically. Afterwards, the reports were collected and processed for analysis.

5 RESULTS

For both experiments, we analyzed the following metrics: the number of identified vulnerabilities that were known in advance, the total number of reported vulnerabilities, any potential correlation

[9] cyclonedx.org
[10] cve.mitre.org

Table 5: Selected open source Java projects for the second experiment with the vulnerable version, the CVE ID of the vulnerability, and the build system

Project ID	Version	CVE ID	Build tool
jackson-databind	2.9.10.7	CVE-2020-35490	Maven
Kafka	2.7.0	CVE-2020-27218	Gradle
OpenSearch	2.4.1	CVE-2023-23612	Gradle
Poi	5.2.0	CVE-2022-26336	Gradle
RxJava	2.2.2	CVE-2020-8908	Gradle
Spring-framework	5.3.18	CVE-2022-22968	Gradle
spark	3.1.2	CVE-2021-38296	Maven
xstream	1.4.15	CVE-2021-21348	Maven
maven-shared-utils	3.3.2	CVE-2022-29599	Maven
commons-text	1.9	CVE-2022-42889	Maven

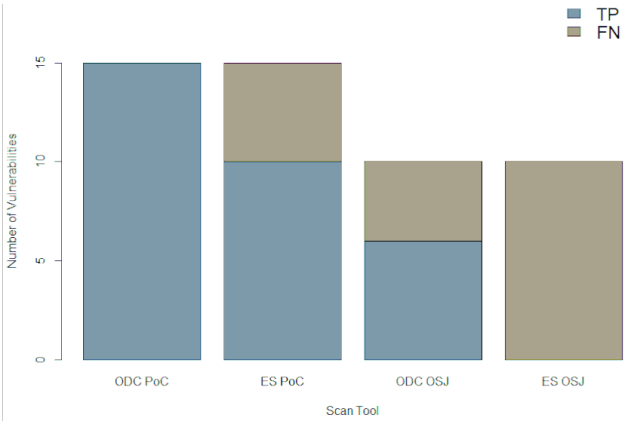

Figure 2: ODC and ES comparison of True Positives (TP) and False Negatives (FN) for the 15 proof-of-concept (PoC) projects and the 10 open source Java (OSJ) projects

with the build tool used and the CVSS Score, as well as the number of vulnerabilities identified as unreachable by ES. Furthermore, in the first experiment, we also examined the project's dependencies, and conducted random manual examination to classify a subset of reported vulnerabilities in true positives (TP) and false positives (FP).

The Figure 2 shows the scan results for the implemented vulnerabilities in the 15 proof-of-concept projects (ODC PoC, ES PoC) and the 10 vulnerabilities existing in the open source Java projects (ODC OSJ, ES OSJ). It can be seen that ODC detected significantly more TP in both the first and the second experiments. Thus, ODC was able to detect all 15 of the 15 implemented vulnerabilities in the first experiment and 6 of the 10 vulnerabilities in the second experiment. ES detected 10 out of the 15 vulnerabilities in the first experiment and no vulnerabilities in the second experiment.

The ES reachability (ES_reach) analysis marked in both experiments all of the TP detected by ES as reachable.

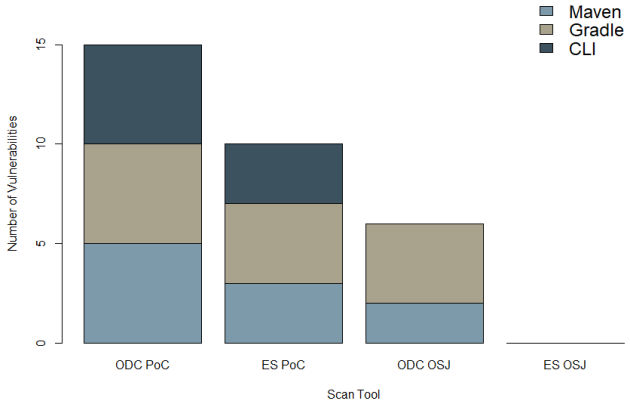

Figure 3: ODC and ES comparison of True Positives (TP) and False Negatives (FN) for the 15 proof-of-concept (PoC) projects and the 10 open source Java (OSJ) projects with related build tools

The results of the scans of the PoC projects and the Java open source projects were also examined concerning a possible correlation with the build tool. In Figure 3 the reported known vulnerabilities from the respective projects are classified by build tool. For the PoC projects, ES (ES PoC) reported four vulnerabilities from Gradle projects, and three correctly from each of the others. For ODC (ODC PoC), no differences between the build tools can be detected. We observed differences in the open source Java projects for ODC (ODC OSJ). Four vulnerabilities were correctly identified from the Gradle projects, but only two from the Maven projects. No observations could be made for ES (ES OSJ) since no vulnerabilities were identified.

In addition, the CVSS scores of the known vulnerabilities from the PoC projects and the Java open source projects were examined in more detail. Table 6 shows which CVSS scores had a vulnerability that resulted in a TP or False Negative (FN). For the 15 PoC projects, no differences can be observed for ODC because all CVSS scores were equally well detected. Looking at the ES report results, all four critical vulnerabilities were detected with a score of 10, but only half of the vulnerabilities were detected with a CVSS score of 9.8. In addition, one vulnerability was not detected with a score of 7.5 and one was detected with a score of 7.2. The Man-Withney U test did not show significant results for ES (p=0.4435), so there is no relationship between the CVSS score and the vulnerability detection.

For the Java open source projects, no observations could be made regarding the CVSS score and ES scans, since no vulnerability was detected. However, looking at the results of ODC, it can be seen that all vulnerabilities with CVSS scores from 3.3 up to and including 7.5 were detected, but all those with a higher CVSS score were not. The Man-Withney U test showed significantly higher detection for vulnerabilities with lower CVSS scores (p=0.01363), than with higher ones, for ODC.

Figure 4 shows the total number of reported vulnerabilities found by ODC or ES, and the ES reachability analysis (ES_reach). The left figure shows the results of the scans of the proof-of-concept (PoC) projects, and the right figure the results of the open source Java projects. Comparing the ellipses of ODC and ES and their intersections, it can be seen that in both cases ODC reported significantly more vulnerabilities compared to ES. Further both figures show, that the intersections in both experiments were relatively small.

Table 6: True Positives (TP) and False Negatives (FN) according to the CVSS of the 15 PoC vulnerabilities

PoC					*OSJ*				
	ODC		*ES*			*ODC*		*ES*	
CVSS	*TP*	*FN*	*TP*	*FN*	CVSS	*TP*	*FN*	*TP*	*FN*
5.9	1	0	1	0	3.3	1	0	0	1
6.4	1	0	1	0	4.8	1	0	0	1
7.2	1	0	0	1	5.3	1	0	0	1
7.5	1	0	1	0	5.5	1	0	0	1
8.8	1	0	0	1	7.5	2	0	0	1
9.8	6	0	3	3	8.1	0	1	0	1
10	4	0	4	0	8.8	0	1	0	1
					9.8	0	2	0	2

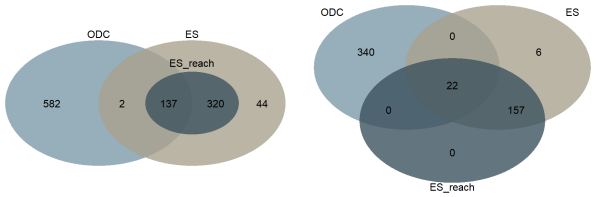

Figure 4: ES reachability analysis for the PoC projects on the left and for the open source Java projects on the right

In the PoC projects, a total of 1084 vulnerabilities were reported. Of these, 139 (12.84 %) vulnerabilities were detected by both tools, 582 (53.69 %) by ODC only, and 364 (33.58 %) by ES only.

A total of 525 vulnerabilities were reported in the Java open source projects. Of these, 22 (4.19 %) vulnerabilities were detected by both tools, 340 (67.76 %) by ODC only, and 163 (31.05 %) by ES only.

In addition to vulnerability scans, ES_reach analysis, was performed for Eclipse Steady using app-to-code (a2c) and trace-to-code (t2c) methods. ES_reach marked 46 (9.15 %) of the 503 vulnerabilities of the PoC projects and 6 (3.24 %) of the 185 vulnerabilities of the Java open source projects as unreachable. Less than 10 % of the vulnerabilities were marked as unreachable in both experiments. In the experiment with the Java open source projects, even less than 5 % of the vulnerabilities were marked as unreachable.

In Figure 4 it can also be observed that ES_reach identified only two out of the 139 vulnerabilities in the intersection as unreachable in the first experiment. In the second experiment, ES_reach did not mark any of the 22 vulnerabilities in the intersection as unreachable.

The vulnerabilities found in the first experiment were also examined in terms of their direct or transitive dependencies. For this purpose, all dependencies found were first examined in more detail. To validate these, we chose CycloneDX to generate a Software Bill of Material (SBOM) in CycloneDX format. ES, ODC, and CycloneDX

Table 7: Number of vulnerabilities found per scan tool in direct and transitive dependencies

	ODC only	ES only	ODC and ES
direct vulnerabilities	225 (38.66 %)	53 (14.56 %)	29 (20.86 %)
transitive vulnerabilities	357 (61.34 %)	311 (85.44 %)	110 (79,14 %)
total (100%)	582	364	139

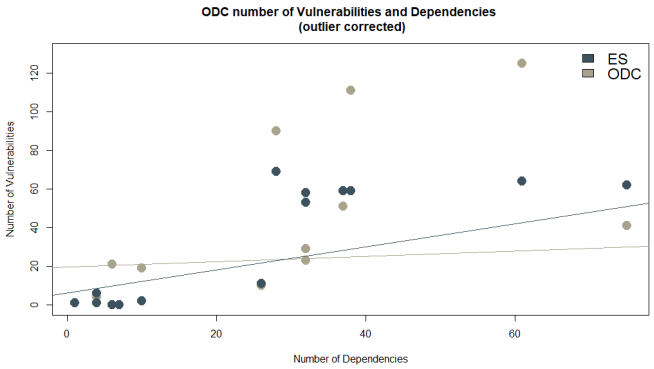

Figure 5: Positive Pearson correlation between number of vulnerabilities and the number of dependencies for ODC (outlier corrected) and ES

found the same direct and transitive dependencies per project. The results show that of the total 393 dependencies, 58 (14.76 %) are direct dependencies and 335 (85.24 %) are transitive dependencies.

Examining the 1084 total vulnerabilities and classifying them in direct and transitive dependencies, 778 (71,77%) were transitive and 306 (28.23%) direct. The results are separated into three classes, only reported by ODC, only reported by ES, and reported by ES and ODC and shown in Table 7. Of the 582 vulnerabilities reported only by ODC, 357 (61.34 %) were transitive vulnerabilities, and 224 (38.49 %) were direct. ES reported 311 (85.44 %) transitive and 53 (14.56 %) direct vulnerabilities of the 364 detected vulnerabilities. The 139 vulnerabilities reported by both tools were 110 (79,14 %) transitive and 29 (20.86 %) direct.

Regarding the correlation between the number of vulnerabilities and the number of dependencies, no significant results were obtained when counting all projects for ODC. Excluding the outlier project CVE-2019-2890 with over 100 findings for one direct dependency, a significant positive Pearson correlation ($p = 0.01815$) could be observed for ODC. For ES no outliers were detected, and the Pearson correlation revealed a high significant value ($p = 0.0001676$). Therefore the number of dependencies is positive correlated to the number of vulnerabilities for both tools. Figure 5 models the positive correlation between the number of vulnerabilities and the number of dependencies for ODC and ES.

Apart from the 15 implemented known vulnerabilities of the PoC projects we want to get an overview of the reported vulnerabilities

Table 8: Number of examined vulnerabilities per class

Detected by	Transitive vulnerabilities	Direct vulnerabilities
ODC	6	6
ES	6	6
ODC and ES	6	6

by ODC and ES by classifying them into TP and FP. Therefore a randomized sample of a total of 48 vulnerabilities was examined with the help of a manual code review. It was not possible to investigate a representative set of vulnerabilities, because of the extremely high effort of the manual analysis, therefore classes were formed from which six vulnerabilities were randomly drawn. For the first manual investigation, a total number of 36 vulnerabilities were drawn from the six classes. The classes consisted of vulnerabilities found only by ES, found only by ODC, and found by both paired with transitive and direct dependencies, as shown in Table 8. For each class, six vulnerabilities were randomly selected and examined.

Manual examination revealed that five of the 36 vulnerabilities were TP and the other 31 were FP and is presented in Figure 6. Two of the TP were found by both tools in direct dependencies. One TP was found by both tools in a transitive dependency. The other two TP were found only by ODC and belong to direct dependencies. It must be mentioned here that ES correctly detected the dependencies, but all 12 of the investigated vulnerabilities found only by ES concern older versions of the identified dependencies and are therefore FP. We also examine the vulnerabilities found by ES and the ES reachability analysis. For this purpose, a sample of 12 vulnerabilities was examined with a code review, six vulnerabilities of the unreachable and six vulnerabilities of the reachable. The investigation revealed that all 12 vulnerabilities were FP. Again ES and ES_reach were able to correctly detect the dependencies, but for 10 of the findings examined, the vulnerability involves an older version of the dependency. The two vulnerabilities that could potentially impact the project were FP due to application configuration and setup. These two vulnerabilities were also detected by ODC.

6 DISCUSSION

In this section limitations of this work are presented and the results are discussed.

6.1 Limitations

In the following paragraph we address limitations that might affect the results of our study. One important limitation that applies to both experiments is the selection and number of projects. We had to limit our project selection to projects that can be scanned, which means that they can be built locally and that they are open source. Furthermore, we had to reduce the set of projects to a number that we can investigate in the given amount of time. We recognize that selecting other projects may yield different results and further research is needed to validate our results.

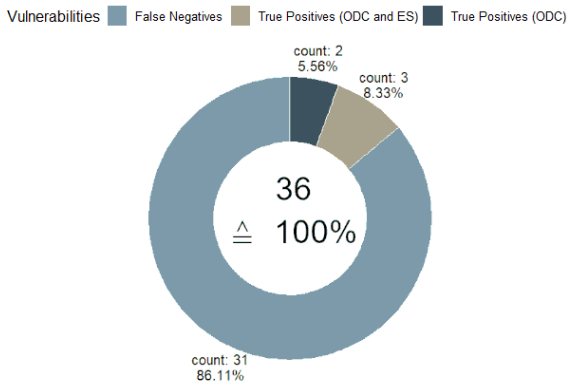

Figure 6: False Negatives (FN) and True Positives (TP) of the manual examined subset. The TP are classified in found by both tools and found only by ODC

Another limitation is that some investigations could only be performed for the scan results of the proof-of-concept (PoC) projects. The dependencies could not be examined for the second experiment, because ES cannot return a list of all found dependencies. Thus, it would have been a huge effort to extract them manually from the Eclipse Steady WebUI frontend. Also, the CycloneDX tool could not resolve the dependencies for all projects. A comparison, as it was done with the PoC projects, was therefore not possible.

When manually reviewing vulnerabilities, we used code review to identify vulnerabilities as TP or FP. Code review cannot determine with 100 % certainty whether a vulnerability is a TP or an FP. Only exploiting the vulnerability can confirm that it is a vulnerability. Writing an exploit and applying it to the application is not possible within the scope of this paper. Furthermore, there is a lack of documentation on how individual vulnerabilities can be exploited.

In addition, due to the time available and the high effort required for manual code reviews, only a set of 48 vulnerabilities in total were investigated. Of these, 36 came from the ODC and ES reports and 12 from the ES and ES reachability analysis (ES_reach) comparison. To obtain a representative sample with a confidence level of 95 % and a margin of error of 5 %, 284 out of the total 1084 reported vulnerabilities would need to be examined. For the comparison of ES and ES_reach, 218 vulnerabilities would need to be examined.

Only the two free and open source tools ODC and ES were examined. However, the market for commercial and open source SCA tools offers a wide range of tools. Other SCA tools were not considered in this study because we focused on open source tools, different scan approaches and a certain maturity level. Furthermore, only Java projects were examined in our study. ODC implements analyzers for various programming languages and both tools, ODC and ES implement experimental vulnerability detection for Python. In addition, SCA Tools which can detect C and C++ dependencies should be considered to meet the requirements of the automotive context. More research is needed to make a generally valid statement about SCA methods and tools.

6.2 Results Discussion

In both the first and second experiments, ODC consistently detected a higher number of TP for known vulnerabilities compared to ES. When examining the overall count of vulnerabilities, it becomes evident that ODC reports a greater number of vulnerabilities. Additionally, it is worth noting that ODC and ES identify distinct vulnerabilities within the same projects, resulting in a relatively small intersection of vulnerabilities discovered by both tools. Similar results were reported in [16, 26].

After conducting a manual examination of certain vulnerabilities, it was discovered that both tools exhibit a notable number of false positives (FP). ODC tends to produce FP reports when the vulnerability solely affects specific methods that the project does not utilize or when specific configurations are required to exploit the vulnerability. In contrast, ES generates FP reports through a bug, by identifying vulnerabilities for outdated versions of dependencies that no longer impact the current version. The majority of TP are identified within the intersection of vulnerabilities detected by both tools, with the second-highest number of TP being exclusively detected by ODC.

Based on these findings, it can be concluded that ODC provides better quality SCA scan results compared to ES in this study. ODC is less affected by false negatives and was able to detect more true positives.

The vulnerability findings between the two SCA tools diverge from each other. Both tools implement different approaches for vulnerability detection and rely on distinct vulnerability databases as sources of information. An SCA tool comparison study conducted by Imtiaz et al. [16] yielded similar results and attributed the disparities to differences in the vulnerability databases used. ODC utilizes the NVD, while ES has its own database, the Project Kaybee. Upon closer examination of Project Kaybee, we observed that it contains fewer entries compared to the NVD. The existing entries predominantly comprise older vulnerabilities spanning from 2015 to 2019, which accounts for the variations in detected vulnerabilities. Additionally, ES had difficulties in accurately identifying vulnerable versions of libraries in our experiment. While it correctly identified the dependencies, it reported vulnerabilities that did not affect the current versions of those dependencies. This discrepancy constitutes another reason for the limited overlap in the vulnerabilities detected. We also consider the scan methodology as a potential factor affecting vulnerability detection. For instance, considering the *commons-text* vulnerability CVE-2022-42889, which is present in both databases. ODC successfully identified several vulnerable versions, including *commons-text-1.8*, whereas ES did not. ES, on the other hand, reported the vulnerability for commons-text-1.9 but not for version 1.8, even though neither version was explicitly listed in the vulnerability database.

Our results further suggest that existing vulnerabilities in Gradle projects are more likely to be discovered by the tools than in Maven or CLI projects. However, this statement cannot be validated unambiguously, as this effect only occurred in one experiment for one tool at a time. The reason for this effect could also depend on the selected projects and not the build tool. Further experiments and investigations should be carried out to be able to make a meaningful statement.

In our study, ES reachability analysis (ES_reach) provided the same results as ES for the known vulnerabilities, and none of the TP were marked as FP. In addition, ES_reach marked almost all vulnerabilities from the intersection of the two SCA tools as reachable. Since most of the manually examined TP are from this class, we suspect that ES_reach indeed helps to detect and eliminate FP. However, no definitive conclusions can be drawn and further research is needed.

The key findings of our study indicate that ODC outperformed ES in terms of delivering better scan results, primarily due to its ability to detect more true positives. The higher number of false negatives in ES can be attributed to missing entries in the vulnerability database rather than the vulnerability detection approach itself. Both SCA tools, however, reported a significant number of false positives. ES reported FP by identifying vulnerabilities in outdated versions of correctly detected libraries, while ODC had FP due to its inability to verify if vulnerabilities are truly reachable and potentially exploitable.

Based on previous research and on these findings, we conclude that both metadata-based and code-level SCA methods have their own potential strengths and limitations. To overcome these limitations and take advantage of the benefits of each approach, we propose a hybrid SCA method as shown in Figure 7. By combining a metadata-based method with a code-level-based approach, we aim to address the limitations of each method and improve the accuracy of vulnerability detection. The metadata-based method serves as a preliminary step to reduce false negatives, ensuring that existing and exploitable vulnerabilities are not missed. False negative results are the most critical from a security standpoint since they fail to report existing and exploitable vulnerabilities, leaving developers unaware of them. Since the metadata-based method cannot perform a reachability analysis and tends to report many false positives and lacks information about exploitability, it is beneficial to utilize a code-level-based approach in the subsequent step. This approach leverages code constructs to identify vulnerable methods, classes, and libraries. Furthermore, employing dynamic and static code analysis allows for the evaluation of the potential exploitability of the identified vulnerable code, thereby reducing the occurrence of false positives. This integration of different scanning methods embodies the principle of security in depth, enhancing the capability to identify vulnerabilities effectively.

We assume that by combining the advantages of metadata-based and code-level SCA, the proposed hybrid method aims to enhance the accuracy and effectiveness of vulnerability detection, providing more reliable results in identifying actual vulnerabilities while reducing false positives and false negatives. Whether this is the case should be investigated in a further study.

Ensuring the quality of the vulnerability database holds significant importance for the success of both scanning methods. The data must be accurate, consistent, and comprehensive to facilitate reliable SCA.

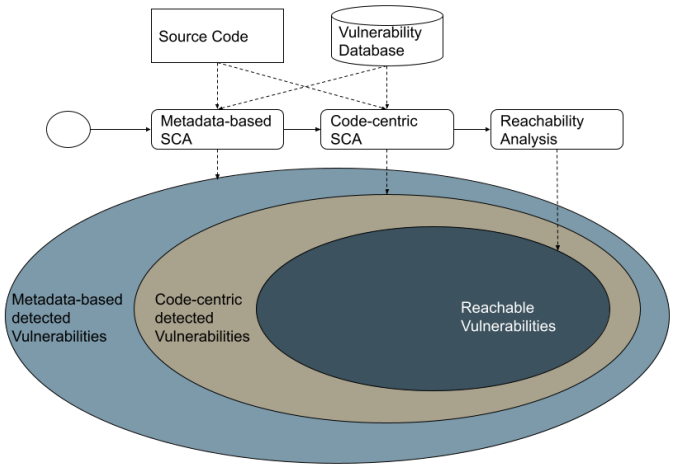

Figure 7: Combination of metadata-based SCA and code-centric SCA

7 CONCLUSION AND FUTURE WORK

Software composition analysis (SCA) aims to detect vulnerabilities in the software supply chain. While this work is motivated by attacks of vehicle-to-everything (V2X) back-end applications, the use of open source libraries is prevalent in many domains. In our study, we found that metadata-based SCA with OWASP Dependency Check (ODC) detected significantly more true positives than the code-level SCA with Eclipse Steady (ES) for Java projects. This discrepancy can be primarily attributed to the disparity in the relying databases and a bug in the ES vulnerability detection. ODC utilizes a local mirror of the NVD, which has a considerably larger database compared to ES's Kaybee project. Kaybee contains only a limited set of vulnerabilities, manually maintained, and this limitation leads to ES missing a significant portion of vulnerabilities not included in its database. Further investigation is needed to determine whether the different approaches also contribute to the different results. We suspect this is the case because, for instance, the *commons-text* vulnerability CVE-2022-42889 exists in both databases. However, for *commons-text-1.8* it was reported by the metadata-based SCA but not by the code-level SCA, while for *commons-text-1.9* both SCA tools reported the vulnerability.

While both metadata-based and code-level SCA tools provide valuable insights into software supply chain security, they come with some inherent limitations. Vulnerability detection associated with CVE IDs depends on database completeness and may miss zero-day vulnerabilities. Data quality is variable and standards for vulnerability identification are lacking. The process of upgrading vulnerable library versions to non-vulnerable versions can be complex, so automatic remediation by an SCA tool is not always possible. SCA tools cannot uniquely classify true positives without validation. While they can report contextual vulnerabilities and potentially exploitable issues, they have difficulty identifying false positives. User-related factors or specific configurations may also escape detection. These limitations underscore the need to use SCA results as a starting point. Human expertise and additional tools are often essential for accurate assessment, taking into account the specific context and configurations of the system to maintain a secure software supply chain. Furthermore, SCA tools for languages such as Rust, Go, C and C++ should also be investigated, as these are particularly important in the automotive sector.

REFERENCES

[1] [n. d.]. NVD - CVE-2021-44228. https://nvd.nist.gov/vuln/detail/CVE-2021-44228#match-9066512

[2] 2017. Clemson Vehicular Electronics Laboratory: Automotive Electronic Systems. http://web.archive.org/web/20171120173150/http://www.cvel.clemson.edu/auto/systems/auto-systems.html

[3] Vard Antinyan. 2020. Revealing the complexity of automotive software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1525–1528. https://doi.org/10.1145/3368089.3417038

[4] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 516–519.

[5] Brant A Cheikes, David Waltermire, and Karen Scarfone. 2011. *Common platform enumeration :: naming specification version 2.3.* Technical Report NIST IR 7695. National Institute of Standards and Technology, Gaithersburg, MD. NIST IR 7695 pages. https://doi.org/10.6028/NIST.IR.7695 Edition: 0.

[6] Tamal Das. 2022. What CPU Does a Car ECU Run On? https://www.makeuseof.com/cpu-for-car-ecu/ Section: Technology Explained.

[7] Sébastien Dudek. 2021. Examining Log4j Vulnerabilities in Connected Cars and Charging Stations. https://www.trendmicro.com/en_us/research/21/l/examining-log4j-vulnerabilities-in-connected-cars.html Section: research.

[8] Christof Ebert and John Favaro. 2017. Automotive Software. *IEEE Software* 34, 3 (May 2017), 33–39. https://doi.org/10.1109/MS.2017.82 Conference Name: IEEE Software.

[9] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. 2015. Fast and vulnerable: a story of telematic failures. In *Proceedings of the 9th USENIX Conference on Offensive Technologies (WOOT'15)*. USENIX Association, USA, 15.

[10] Eclipse Foundation. [n. d.]. *Eclipse Steady Analysis Manual*. https://eclipse.github.io/steady/user/manuals/analysis/

[11] OWASP Foundation. 2023. *Software Component Verification Standard Measure and Improve Software Supply Chain Assurance*. https://scvs.owasp.org/

[12] The Linux Foundation. 2023. *Safeguarding artifact integrity across any software supply chain*. https://slsa.dev/spec/v0.1/threats

[13] Andy Greenberg. 2015. Hackers Remotely Kill a Jeep on the Highway—With Me in It. *Wired* (July 2015). https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[14] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. 2021. Präzi: From Package-based to Call-based Dependency Networks. *CoRR* abs/2101.09563 (2021). arXiv:2101.09563 https://arxiv.org/abs/2101.09563

[15] Freddie Holmes. 2018. Auto industry's thirst for software is quenched by open source. https://www.automotiveworld.com/articles/auto-industrys-thirst-for-software-is-quenched-by-open-source/

[16] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 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)*. 1–11.

[17] Jeremy Long and Seth Jackson. 2017. Architecture. https://github.com/jeremylong/DependencyCheck/wiki/Architecture

[18] Jeremy Long and JoyChou. 2017. How does it work? https://github.com/jeremylong/DependencyCheck/wiki/How-does-it-work%3F

[19] Jürgen Mössinger. 2010. Software in Automotive Systems. *IEEE Software* 27, 2 (March 2010), 92–94. https://doi.org/10.1109/MS.2010.55

[20] Sen Nie, Ling Liu, Yuefeng Du, and Wenkai Zhang. 2018. OVER-THE-AIR: HOW WE REMOTELY COMPROMISED THE GATEWAY, BCM, AND AUTOPILOT ECUS OF TESLA CARS. (Aug. 2018). http://i.blackhat.com/us-18/Thu-August-9/us-18-Liu-Over-The-Air-How-We-Remotely-Compromised-The-Gateway-Bcm-And-Autopilot-Ecus-Of-Tesla-Cars-wp.pdf

[21] Emily Olin. 2020. Subaru Adopts AGL Software for Infotainment on New 2020 Subaru Outback and Subaru Legacy. https://www.automotivelinux.org/announcements/subaru-outback/

[22] Paul O'shea. 2017. Automotive electronics: What are they, and how do they differ from "normal" electronics? https://www.powerelectronicsnews.com/automotive-electronics-what-are-they-and-how-do-they-differ-from-normal-electronics/

[23] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[24] Amrita Pathak. [n. d.]. *Software Composition Analysis (SCA): Everything You Need to Know in 2022*. https://geekflare.com/software-composition-analysis/

[25] Etiel Petrinja, Ranga Nambakam, and Alberto Sillitti. 2009. Introducing the OpenSource Maturity Model. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. 37–41. https://doi.org/10.1109/FLOSS.2009.5071358

[26] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[27] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 383–387. https://doi.org/10.1109/MSR.2019.00064 ISSN: 2574-3864.

[28] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* 26, 4 (2021), 1–34.

[29] Robert N. Charette. 2009. This Car Runs on Code - IEEE Spectrum. https://spectrum.ieee.org/this-car-runs-on-code

[30] Sam Curry. 2023. Web Hackers vs. The Auto Industry: Critical Vulnerabilities in Ferrari, BMW, Rolls Royce, Porsche, and More. https://samcurry.net/web-hackers-vs-the-auto-industry/

[31] Karthik Shanmugam. 2019. Securing Inter-Processor Communication in Automotive ECUs. 2019–26–0363. https://doi.org/10.4271/2019-26-0363

[32] Synopsys. 2023. *Open Source Security and Risk Analysis*. Technical Report. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2023.pdf

[33] Tencent Keen Security Keen Lab. 2021. Tencent Security Keen Lab: Experimental Security Assessment of Mercedes-Benz Cars. http://keenlab.tencent.com/2021/05/12/Tencent-Security-Keen-Lab-Experimental-Security-Assessment-on-Mercedes-Benz-Cars/index.html

[34] Upstream. 2022. *Global Automotive Cybersecurity Report 2022*. Technical Report. https://info.upstream.auto/hubfs/Security_Report/Security_Report_2022/Upstream_Security-Global_Automotive_Cybersecurity_Report_2022.pdf

[35] Chris Valasek and Charlie Miller. 2014. A Survey of Remote Automotive Attack Surfaces. (July 2014). https://ioactive.com/wp-content/uploads/2018/05/IOActive_Remote_Attack_Surfaces.pdf

[36] Chris Valasek and Charlie Miller. 2015. Remote Exploitation of an Unaltered Passenger Vehicle. (Aug. 2015). https://ioactive.com/wp-content/uploads/2018/05/IOActive_Remote_Car_Hacking-1.pdf

[37] VicOne. 2022. *Steering Clear: VicOne 2022 Automotive Cybersecurity Report*. Technical Report. https://vicone.com/files/rpt-automotive-cybersecurity-in-2022.pdf

[38] Free Wortley, Forrest Allison, and Chris Thompson. 2021. Log4Shell: RCE 0-day exploit found in log4j, a popular Java logging package | LunaTrace. https://www.lunasec.io/docs/blog/log4j-zero-day/

[39] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. *arXiv preprint arXiv:2301.05362* (2023).

[40] Maria Zhdanova, Julian Urbansky, Anne Hagemeier, Daniel Zelle, Isabelle Herrmann, and Dorian Höffner. 2022. Local Power Grids at Risk – An Experimental and Simulation-based Analysis of Attacks on Vehicle-To-Grid Communication. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 42–55. https://doi.org/10.1145/3564625.3568136