# ON THE SECURITY BLIND SPOTS OF SOFTWARE COMPOSITION ANALYSIS CAUSED BY CLONING AND SHADING

A PREPRINT

**Jens Dietrich**
Victoria University of Wellington
Wellington
New Zealand
jens.dietrich@wgtn.ac.nz

**Shawn Rasheed**
UCOL — Te Pūkenga
Palmerston North
New Zealand
unshorn@gmail.com

**Alexander Jordan**
Oracle Labs
Vienna
alexander.jordan@oracle.com

June 7, 2023

**ABSTRACT**

Modern software heavily relies on the use of components. Those components are usually published in central repositories, and managed by build systems via dependencies. Due to issues around vulnerabilities, licenses and the propagation of bugs, the study of those dependencies is of utmost importance, and numerous software composition analysis tools have emerged to address those issues.

A particular challenge are hidden dependencies that are the result of cloning or shading where code from a component is "inlined", and, in the case of shading, moved to different namespaces.

We present an approach to detect cloned and shaded artifacts in the Maven repository. Our approach is lightweight in that it does not require the creation and maintenance of an index, and uses a custom AST-based clone detection.

Our analysis focuses on the detection of vulnerabilities in artifacts which use cloning or shading. Starting with eight vulnerabilities with assigned CVEs (four of those classified as critical) and proof-of-vulnerability projects demonstrating the presence of a vulnerability in an artifact, we query the Maven repository and retrieve over 16k potential clones of the vulnerable artifacts. After running our analysis on this set, we detect 554 artifacts with the respective vulnerabilities (49 if versions are ignored). We synthesize a testable proof-of-vulnerability project for each of those. We demonstrate that existing SCA tools often miss these exposures.

***Keywords*** vulnerability detection, clone detection, shading, sofwtare composition analysis, Java, Maven

## 1 Introduction and Background

Modern software systems often use components in order to achieve economy of scale. The process is recursive – components also use other components, resulting in deep and complex component ecosystems [44, 21, 10]. This has in turn created new challenges. The prime example is vulnerability propagation, infamous examples include the *equifax* [40, 24] and *log4shell* [41, 18] incidents, with vulnerable and outdated components now being acknowledged as being a major security risk [1]. Other related issues include license compliance [34], typo-squatting [39], and lifecycle issues of components as demonstrated by the *leftpad* incident [6].

In response to those challenges, software composition analysis (SCA) tools have emerged that scan the dependency networks, and cross-reference them with known vulnerabilities catalogued in databases such as the National Vulnerability Database (NVD) [1] and the GitHub Advisory Database [2]. If a vulnerable dependency is found, developers are

[1] https://nvd.nist.gov/vuln
[2] https://github.com/advisories

notified (for instance, via pull requests) and can upgrade dependencies to a newer version. Examples of such tools include GitHub's *dependabot* [3], *snyk* [4], *OWASP dependency check* [5], tooling integrated into development environments such as *IntelliJ's dependency analysis* backed by *checkmarx*, and features or plugins of build tools like *npm audit* (for JavaScript) and Sonatype's *oss index* Maven plugin [6].

At a high-level an SCA tool combines two components, a scanning component to find dependencies, and a vulnerability database (*vulnerability DB*) to decide whether a dependency has a known vulnerability or not. Depending on the implementation, a bit of matching logic is required to combine the two components providing a bridge between the low-level packages used by build systems (e.g., Maven artifacts in the Java/Maven ecosystem) and a more coarse-grained way of identifying software at the product-level, like NVD does by using the CPE (Common Platform Enumeration) naming standard. Because a direct mapping between software products and their artifacts (source code or binary packages), this matching is not always straight-forward and can cause inaccuracies. Versions are another source of inaccuracy for SCA tools, often due to the fact that it is hard to pinpoint when a vulnerability was introduced, in which case (conservative) assumptions have to be made.

With the exception of *Eclipse Steady*, which uses program analysis to determine reachability of vulnerable code, SCA tools generally do not assert whether a vulnerable dependency makes an application unsafe (e.g. because it is exploitable by an attacker) or safe (e.g. because the dependency is unused).

Open source SCA tools rely on public information for their vulnerability DBs and depend on wider community efforts to update and correct this information. Commercial tools (e.g. *snyk*) may provide their own vulnerability DB, which may refine or extend the information that is available in public. Mismatch between information in vulnerability DBs is possible, often due to timing issues where one DB is updated sooner than another. It is however in the interest of commercial vendors to eventually have their DBs aligned with public knowledge to avoid confusion among customers.

Like all program analyses, SCA tools suffer from precision problems, i.e. false positives. They may for instance detect dependencies to vulnerable code in a library that is not actually reachable [25]. This could in principle be tackled by employing more fine-grained analyses like call graph construction, although the price (in terms of computational resources needed) could be significant.

What is more relevant for our discussion is that those analyses are not sound either, i.e. they miss dependencies and therefore problems such as vulnerabilities associated with those dependencies [8].

The first source of unsoundness is late binding, i.e. applications that "discover" capabilities at runtime, leading to dependencies that are not visible in the build configurations or code SCA tools analyse. For Java, plugin-based application frameworks like *OSGi* widely used in application servers and programming tools (Eclipse) facilitate this. This is outside the scope of our study.

A second cause of unsoundness is *cloning*. With cloning, code is copied into the project, and these copies can carry vulnerabilities which are then obfuscated by the process. This can take place when an application directly clones code, or cloning is used by libraries which are then used as dependencies by downstream clients. Cloning can take place on multiple levels, from code snippets, functions, classes to entire components. Code sharing, discussion and tutorial web sites like *stackoverflow* [31, 3] and more lately AI-based tools like *Copilot* promote cloning of some kind [27]. With cloning, basic engineering principles like *DRY (do not repeat yourself)* are violated, and in the long term the (lack of) maintenance of cloned code is highly problematic. For vulnerability detection, a particular problem is that many clones are not perfect, i.e. they are often somehow transformed, and generally lack provenance. I.e., the original source of the clone is often opaque, and tools cannot reason about it. *Copilot* is the most extreme example where some abstraction and aggregation is used to produce code from potentially large amounts of input sources.

However, there are also advantages to cloning, and cloning may even be used in order to make code more secure and reliable. For instance, if a dependency is only used for the purpose of using a rather small and trivial piece of functionality from an otherwise large component (perhaps with additional dependencies), then cloning can be a sensible strategy as it may reduce the size of a product to be deployed, and may also reduce its attack surface by removing now redundant functionalities.

For Java, there is an additional problem, a relative of the infamous DLL hell [11]. Large dependency networks may lead to conflicts between different versions of the same class added via multiple dependency paths [42]. Often, the problems resulting from this only manifest at runtime when classes are loaded and linkage related errors caused

[3] `https://github.com/dependabot`

[4] `https://snyk.io/`

[5] `https://owasp.org/www-project-dependency-check/`

[6] `https://sonatype.github.io/ossindex-maven/maven-plugin/`

by binary incompatibility occur. API changes causing this problem are common [30, 20], poorly understood by developers [12], and therefore expensive for projects.

A common solution for this problem is *shading* – a variant of cloning where entire packages are cloned and renamed. Even the Java standard library uses shading, for instance, the OpenJDK version 16 contains shaded versions of *sax* (an XML parser library) and *asm* (a bytecode engineering library) in packages with names starting with `jdk.internal.org.xml.sax` and `jdk.internal.org.objectweb.asm`, respectively [7].

The Java / Maven community acknowledges this issue by providing tools like the *Maven shade plugin* [8]. Here, shading is automated and performed during the build. A dependency to be shaded and the packages to be renamed are declared in the build file (*pom.xml*), and therefore remain visible to SCA tools. We refer to this as *buildtime shading* (*b-shading* for short).

There are cases where other approaches to shading are used - we will provide plenty of examples later in the evaluation section. Here, shading is done by means of refactoring and code organisation tools like IDEs, we therefore refer to this as *designtime shading (d-shading)*. One reason might be the lack of understanding by engineers. However, there might also be more sophisticated reasons to use d-shading. Tools like the shade plugin are static analysis tools, and as such can at best be expected to be *soundy* [23]. The prevalence of dynamic language features in Java is a known challenge for static analysis tools, and leads to a considerable amount of false negatives [38]. In particular, tools like the shade plugin have to rewrite the bytecode of the code to be shaded, and change references (supertypes, method and field descriptors, etc) to the new package names. If such references are missed due to reflective references being present, builds will fail or programs may exhibit unexpected runtime behaviour. In this case, using d-shading might be a sensible choice.

However, d-shading results in blind spots for tools that rely on declared dependencies to infer the presence of vulnerabilities. The question arises whether this is common, and in particular, whether this poses a security risk. This is the question we set out to study.

The rest of this paper is organised as follows. We start with a discussion of related work in Section 2. In order to gauge how commons shading is, we report on a little experiment on the use of the shade plugins in poms found on GitHub (Section 3). These results only concern buildtime shading. In Section 4 we describe the tool pipeline we have developed in order to detect designtime shading of Maven artifacts with a focus on detecting vulnerabilities in the shaded components which are missed by SCA tools. The evaluation is split into two parts – in Section 5 we describe the methodology used, and in Section 6 we present the evaluation results. We discuss the disclosure procedure we followed in Section 8, and finish with a short conclusion.

## 2 Related Work

### 2.1 Detecting and Managing Vulnerable Dependencies

A study by Contrast Security investigated vulnerabilities in Java applications and found that "custom Java applications contain from 5 to 10 security vulnerabilities per 10,000 lines of code." [43]. They point out that it generally has to be assumed that vulnerabilities are present in all applications, but on the other hand, that this does not always render applications as unsafe.

Mir et al [25] point out that "less than 1% of packages have a reachable call path to vulnerable code in their dependencies", alerting to precision problems of dependency-based SCA. However, those results have to be interpreted with caution. The underlying call graph analysis is based on Opal [14], configured to run the rather inaccurate (but fast) class hierarchy analysis (CHA, [16]). This is likely to miss many dynamic call graph edges [38] which are exploited in vulnerabilities. As an example, consider CVE-2015-6420. This vulnerability can be exploited by deserializing objects from an incoming stream, and therefore the call graph path from application classes to vulnerable classes in this library is highly obfuscated, and unlikely to be detected by CHA-based (or any other scalable) call graph construction method. The work by Wu et al [45] is related, with similar limitations.

Kula et al [22] studied how developers respond to vulnerabilities being detected in dependencies they rely on. They found that most of the time outdated dependencies are kept, and developers are unlikely to respond to security advisories [22]. Similar results reporting significant delays to upgrade vulnerable dependencies were also reported for other ecosystems, for instance by Decan et al for NPM [9] and Alfadel et al for Python [2].

[7] `https://github.com/AdoptOpenJDK/openjdk-jdk16/tree/master/src/java.base/share/classes/jdk/internal/org/`

[8] `https://maven.apache.org/plugins/maven-shade-plugin/`

Mirhosseini and Parnin studied whether pull requests (PRs) are effective to speed up upgrades [26]. This mechanism is often deployed by composition analysis tools like *dependabot*. In general, they found that PRs do speed up upgrades, although the merge rate is still surprisingly low at around a third of all PRs. Alfadel et al studied particular PRs made by a popular SCA tool, GitHub's *dependabot*, and found a significantly higher acceptance (merge) rate of about two thirds. This study considered only NPM projects.

Dann et al [8] studied several OSS vulnerability scanners (*OWASP dependency check, Eclipse steady, snyk, black duck, WhiteSource*) and evaluated their performance on a set of 7,024 projects collected by SAP. They found limitations of the tools to deal with several modifications (re-compilation, re-bundling, metadata-removal and re-packaging) of the original vulnerable projects. Their observations are consistent with ours, and the respective modifications roughly correspond to our notions of cloning and shading.

Bui et al developed *vul4j* [5], a dataset consisting of 79 reproducible vulnerabilities from 51 open-source projects. Reproducibility is achieved via proof-of-vulnerability (POV) tests. This is the same approach we are using to confirm the presence of a vulnerability. Their dataset however requires a dedicated execution environment, where we rely on a lightweight setup of stand-alone projects that can easily be automatically refactored.

Ponta et al [29] propose a hybrid code-centric vulnerability detection that overcomes the limitations (here mainly seen as the low precision) of meta-data based SCA approaches. Their analysis uses code changes introduced by security fixes. The tool resulting from this is *vulas*, later renamed to *steady*. We did use *steady* in the evaluation section (Section 5.2), and the results suggest that this is complementary to our approach.

Our approach depends on the existence of proof-of-vulnerability (POV) projects, and would therefore benefit from the automated generation of exploits. Initial work in this area, based on test case generation using genetic algorithms, has been proposed by Iannone et al [19].

### 2.2 Clone Detection

Research into code clone detection has established a classification for levels of clone similarity: type-1 clones are identical except for layout (whitespace) and comments; type-2 clones are syntactically equivalent, allowing for renaming of variables, functions, types, etc.; type-3 clones are syntactically similar, additionally allowing for some statements to be added or removed.

Clone detection is used to improve or enforce software development quality standards by detecting unwanted copies of code leading to maintainability or licensing issues, and, in academic settings to detect plagiarism. There is a vast amount of research in this field, covered in surveys such as [35, 32].

We use type-2 clone detection as a proxy to detect compositional clones, i.e. the practice of copying (parts of) existing software libraries into projects.

Binary code similarity [17] can be seen as an instance of clone detection and related to our work. Comparing code at the binary (or bytecode) level comes with the challenge of variations introduced by different (versions of) compilers, different compile-time transformations, and different compile environments.

In particular for Java, clone detection in bytecode has been studied by Dann et al [7]. They address the problem by translating bytecode into an intermediate, soot-based format that can abstract from the particularities of different compilers to some extent. We did consider using a similar approach, however, as source code is readily available in Maven, a traditional AST-based clone detection appears to be the better choice as those problems can be avoided.

Closely related to it, and also targeting the Java open-source ecosystem is SCA-related research focusing on libraries included in released Android applications under the term *third-party library detection* [46]. Note that in this context, research tries to solve the harder problem of creating an analysis that is resilient to hiding and obfuscation of libraries. It does this using similarity search techniques based on features (e.g. class dependency structure, method signatures, control-flow graphs) extracted from bytecode.

## 3 On The Prevalence of Buildtime Shading

To gauge how widespread the practice of shading is, we first focus on b-shading, shading performed at build time. If Maven is used as a build system, this can be achieved by using the Maven shade plugin. To study the prevalence of b-shading, we used a dataset consisting of Maven build files (poms, i.e., `pom.xml`) collected as follows:

1. As a starting point, the *library.io* dataset [9] released on 12 January 2020 was used.

[9] `https://libraries.io/`

2. The projects where filtered out for projects using Maven / Java, and having a github repository.
3. Then `pom.xml` files were extracted from the respective repositories.

This produced an initial set of 103,358 poms. We selected poms using the shade plugin with the following XPath query:

```
//plugin/artifactId[text() = 'maven-shade-plugin']
```

This query yields 3,693 poms (3.57%). We then analysed how often classes are relocated into different packages, by querying those poms with the following XPath query:

```
//plugin/artifactId[text() = 'maven-shade-plugin']/
parent::node()//relocations
```

We found 808 poms (0.78% of all poms) using relocations. The results indicate that build time shading is commonly used, supporting the claim that there are valid use cases for shading in general, and for package renaming in particular.

This raises the questions of how common shading is that does not use plugins (and therefore does not state the dependency), how it can be detected, and what the security implications of this are.

# 4 Blindspot Detection

## 4.1 Overview

We describe the processing pipeline we have developed and used to detect clones and shaded artifacts with known vulnerabilities here. It takes an artifact and a vulnerability as input, and produces a list of artifacts and projects demonstrating the presence of the provided vulnerability in those artifacts. The focus of the tool design is on precision (avoiding false positives), and being lightweight. In particular, it does not require the acquisition, construction and maintenance of a separate index. Instead, it can work with an existing index as long as it makes the information required (source code, poms, artifacts searchable by class names) available through an API. Our aim is to demonstrate that with some fairly simple tooling that goes beyond the metadata-centric approach used by most SCA toolst, more vulnerable artifacts can be detected. We do not aim at detecting all those artifacts, and as with all program analyses, false positives and false negatives (precision and recall) have to be balanced, and achieving both high precision and recall is usually not feasible [33, 15].

The key ideas our tooling is based on are:

1. We do not depend on indexing the Maven repository, instead, we work with the existing repository via the Maven Central REST API [10].
2. We extract a signature of components to be used to identify potential clones using a lightweight method based on unqualified, characteristic class names. This can then be directly used in repository queries.
3. We use a custom AST-based clone detection to identify clones, including those that may have repackaged classes.
4. We use tests to verify the presence of vulnerabilities in clones, and automate the adaptation of those tests for clones, and the evaluation of test results. This leads to a high precision of the vulnerability detection.

## 4.2 Inputs

Our analyses requires the following inputs:

1. An artifact $art_0$ identified by its group-artifact-version (GAV) coordinates $gav_0$ within the Maven repository
2. A known vulnerability $vul$ identified by a CVE
3. A proof-of-vulnerability (POV) Maven project $pov$ that has a direct dependency on $art_0$ and tests demonstrating the presence of $vul$. I.e., these tests succeeds if the exploit of $vul$ is successful.

[10] `https://central.sonatype.org/search/rest-api-guide/`

The project $pov$ is optional in the sense that the tool chain can be run without it. Its purpose is to make the analysis results precise.

### 4.3 Pipeline

Our analysis pipeline consists of the following steps:

1. **Fetch binaries** – Fetch the binary (jar) $art_0.bin$ of $art_0$ from the Maven repository using the REST API.
2. **Fetch sources** – Fetch the source code $art_0.src$ of $art_0$ from the Maven repository using the REST API.
3. **Select classes** – Extract a list of classes $cl.query$ from the $art_0.bin$ and/or $cl.vul$ to be used in queries. Those are non-qualified class names (i.e., package names are omitted).
4. **Fetch class matches** – For each class in $cl \in cl.query$, use the Maven REST API to fetch a set of artifact coordinates (GAVs) $match_{cl}$ of artifacts with the respective class.
5. **Consolidate matches** – Consolidate all sets $match_{cl}$ into a single set $match$ .
6. **Fetch *match* poms** – For each artifact $art \in match$, fetch the pom $art.pom$ using the Maven REST API.
7. **Remove dependents of original artifact** – For each artifact $art \in match$, analyse the pom $art.pom$, and if it contains reference to $art_0$, remove it.
8. **Fetch *match* sources** – For each artifact $art \in match$, fetch the source code $art.src$ using the Maven REST API.
9. **Run clone analysis** – For each artifact $art \in match$, run a clone analysis comparing $art_0.src$ and $art.src$, and if the result is negative, remove $art$ from $match$
10. **Instantiate POV** – For each artifact $art \in match$, instantiate $pov$ by cloning the project and replacing the dependency to $art_0$ by a dependency to $art$, resulting in a project $pov(art)$.
11. **Verifying the Vulnerability** – For each artifact $art \in match$, run `mvn test` on $pov(art)$. If this succeeds, the presence of the vulnerability is confirmed, and $art$ is added to the result.

We describe the more interesting steps briefly in the rest of this section, and state the settings we used successfully in the evaluation section for the steps that are configurable. We do not claim that those settings are optimal. But the configuration used produces a reasonable yield in terms of artifacts with vulnerabilities discovered with modest computational resources.

### 4.4 Class Selection

We use unqualified class names as fingerprints to identify potential clones. There are two reasons for this: (1) the Maven REST API supports queries by unqualified class names (2) unqualified class names are not changed when repackaging (relocating) code during shading.

When working with a remote index, using all classes is not necessarily the best strategy as each class name is then used in a query, i.e. each class will result in one or multiple network calls. We have used a simple approach to look for signature classes with names likely to be unique. For instance, a short name like `Utils` is likely to be used by many components. However, something like `JSONDriverManagerFactory` (hypothetical) is more likely to be unique. The heuristic used is to count camel case tokens in class names, and look for classes with a high count. In the example above, the count for `JSONDriverManagerFactory` is 4, whereas the count for `Utils` is 1.

The default strategy we have employed is to sort class names by token length, and to use the top 10 class names.

### 4.5 Fetch Class Matches

For each class name identified in step 3, an API query is used to fetch artifacts containing one or more classes with this name. The API uses paging, and limits the number of results returned by each query to 200. We use 5 pages of 200 results each, i.e. a maximum of 1,000 artifacts per class is analysed.

### 4.6 Query Consolidation

The process described above results in 10 query result sets with up to 1,000 artifacts in each. A consolidation strategy identifies the artifacts likely to represent clones. Strategies like intersection or union of result sets are possible, the union is likely to contain many accidental matches that contain only a single matching class. The other extreme, the

intersection, may exclude many artifacts that only partially clone the original artifact, but could still contain all classes necessary to exploit a vulnerability. The strategy we have used is that an artifact must occur in at least two result sets, i.e. it must contain at least two classes with names matching classes in the original artifact selected for querying.

### 4.7 Remove Dependents of Original Artifact

This step is performed in order to remove artifacts that declare a dependency to the original artifact. Those are less interesting and may even be considered as effective false positives by engineers [36] as SCA tools usually detect vulnerabilities propagated through such dependencies. For this purpose we acquire and analyse the pom of the artifact. The pom analysis is looking for three patterns:

1. There is no reference in the dependency section to the original artifact.
2. There is no reference to the original artifact within the shade plugin.
3. The group id and artifact id of the clone candidate are different from the group and artifact ids of the original artifact.

The last rule ensures that the tool does not produce results representing different versions of the original artifact. Our analysis also includes references in parent poms for artifacts generated by multi-module projects.

### 4.8 Clone Analysis

The clone analysis used is AST-based. I.e., candidates classes are parsed and the two ASTs are simultaneously traversed. Our method is a type-2 clone detection [35], i.e., we are looking for isomorphic structures but allow some variations in types and comments.

Nodes corresponding to comments are ignored as authors may change comments (for instance, to alter copyright or authorship notices, or to add comments about the origin of the code). For nodes corresponding to type names, the scopes (package names) are ignored.

### 4.9 Instantiating the POV Project

For each artifact $art \in match$, we instantiate the POV project $pov$ by cloning it, replacing the dependency in the pom to $art_0$ by a dependency to $art$, and replacing references to fully qualified class names in classes defined in $pov$ (in particular, tests) if classes are re-packaged by the clone. A map of classes relocated into new packages is provided by the clone analysis, this is used here to replace the class names (for instance, in import statements).

Consider for instance the test used to demonstrate the presence of CVE-2022-38751, a DOS vulnerability in *snakeyaml*, shown in Listing 1 [11]. The structure of the test is straight-forward – parse a malicious payload (*CVE-2022-38751.yml*), and check that this leads to a stackoverflow error. If this leads to some other error or exception (such as an *IllegalArgumentException*), the test fails, indicating that the vulnerability is not present.

```
1  import org.yaml.snakeyaml.Yaml;
2  // more imports ommitted
3  public class ConfirmVulnerabilitiesTests {
4      @Test public void confirmCVE202238751 () {
5          assertThrows(
6              StackOverflowError.class,
7              () -> parse("CVE-2022-38751.yml")
8          );
9      }
10     static void parse (String input) throws IOException {
11         FileReader reader = new FileReader(new File(input));
12         new Yaml().compose(reader);
13     }
14 }
```

Listing 1: Testing CVE-2022-38751 (snakeyaml)

Also note the import statement in line 1. If we find a clone, the original test can be copied, and instantiating the POV project is merely a matter of replacing the dependency on *snakeyaml* by a dependency on the respective clone. If during the clone detection phase clones are detected in different packages, then the import statement needs to be changed as well. This is done by manipulating the ASTs of the respective source files.

[11] The code listings are shortened for brevity

### 4.10 Verifying The Presence of a Vulnerability in a Cloned or Shaded Project

For each artifact, $art \in match$, we run `mvn test` on $pov(art)$. If tests succeed, the presence of the vulnerability is confirmed. This is done by analysing generated *surefire* reports in XML format.

A particular issue that needs to be taken into account is that tests may result in four states – *success*, *fail*, *error* and *skip*. Builds with tests succeed if all tests are in a *success* or *skip* state. This is an optimistic "did not fail assumption". However, we found that it is often practical or even necessary to use assumptions in tests confirming vulnerabilities.

For instance, consider the test confirming CVE-2022-25845 in *fastjson*, shown in Listing 2. The test confirms the execution of an OS command triggered by parsing a document, the command used here is "`touch foo`". This command is defined in the JSON document to be parsed (CVE-2022-25845.json), the *@BeforeEach* fixture is used to erase the file if present. This OS command is only available on unix-like operating systems, and the vulnerability can only be exploited for certain JRE versions. This is encoded using JUnit precondition (assumption) annotations (lines 10-11), and tests are skipped (instead of failed) if those conditions are not satisfied. Therefore, the analysis needs to confirm that all tests have succeeded, which is a stricter requirement (i.e. stricter than the default *surefire* behaviour).

```
1  import com.alibaba.fastjson.JSON;
2  // more imports ommitted
3  public class ConfirmVulnerabilitiesTests {
4      @BeforeEach public void clearGeneratedFile() {
5          File file = new File("foo");
6          if (file.exists()) {
7              Assumptions.assumeTrue(file.delete());
8          }
9      }
10     @Test @EnabledOnOs({OS.MAC,OS.LINUX})
11     @EnabledForJreRange(min=JRE.JAVA_8,max=JRE.JAVA_11)
12     public void confirmCVE202225845 () throws Exception {
13         Path generatedFile = Path.of("foo");
14         Assumptions.assumeFalse(Files.exists(generatedFile));
15         Path payload = Path.of("CVE-2022-25845.json");
16         Assumptions.assumeTrue(Files.exists(payload));
17         String json = Files.readString(payload);
18         JSON.parse(json);
19         Thread.sleep(1000);
20         assertTrue(Files.exists(generatedFile));
21     }
22 }
```

Listing 2: Testing CVE-2022-25845 (fastjson)

## 5 Evaluation Methodology

### 5.1 Dataset

Our evaluation dataset consists of vulnerabilities and the associated artifacts in the Maven repository for which those vulnerabilities have been reported. The selection was driven by the following considerations: (1) to select widely used artifacts, as determined by the number of downstream clients reported by Maven (2) to select CVEs of different types, namely vulnerabilities exploitable for remote code execution (RCE) and denial of service (DOS) attacks (3) to include some high-impact vulnerabilities that have been exploited in the wild such as *log4shell* (4) to select libraries from different domains.

Since our aim was to make CVEs testable in order to design a precise analysis, we furthermore gave preference to CVEs with available proof-of-vulnerability projects we could then reuse (usually with some modifications). In particular for vulnerabilities that have a high severity, such projects often exist. Sometimes projects covering entire classes of vulnerabilities can be used for this purpose, a good example is *ysoserial* [12] that also contains a POV for CVE-2015-6420 which we used in a slightly modified, testable form.

Sometimes other CVEs exist for the same artifact which are closely related to the ones we selected. Examples are CVE-2015-7501 (closely related to CVE-2015-6420) and several DOS vulnerabilities in *snakeyaml* all closely related to CVE-2022-38749, including CVE-2022-38752, CVE-2022-38751 and CVE-2022-38750. Including them would have resulted in more results. We opted here to experiment on combinations of artifacts and vulnerabilities that are distinctive.

[12] `https://github.com/frohoff/ysoserial`

| artifact (gav) | short name | description | usage |
|---|---|---|---|
| commons-collections:commons-collections:3.2.1 | collect | data structure library | 6,508 |
| org.apache-extras.beanshell:bsh:2.0b5 | beanshell | scripting DSL | 71 |
| com.google.guava:guava:11.0.1 | guava | utilities and data structures | 34,399 |
| com.alibaba:fastjson:1.2.80 | fastjson | JSON parser | 5,986 |
| org.yaml:snakeyaml:1.25 | snakeyaml | YAML parser | 3,873 |
| org.apache.logging.log4j:log4j-core:2.14.1 | log4j | logging | 9,515 |
| org.apache.commons:commons-text:1.9 | c-text | string utilities | 2,947 |
| org.json:json:20220924 | json.org | JSON parser | 5,228 |

Table 1: Artifacts used in the evaluation

| vulnerability | artifact short name | CWE(s) | severity | description |
|---|---|---|---|---|
| CVE-2015-6420 | collect | 502 | 9.8 critical | remote code execution using a serialized Java object |
| CVE-2016-2510 | beanshell | 19 | 8.1 high | remote code execution using a serialized Java object |
| CVE-2018-10237 | guava | 770 | 5.9 medium | denial of service using unbounded memory allocation |
| CVE-2022-25845 | fastjson | 502 | 9.8 critical | remote code execution using crafted JSON input |
| CVE-2022-38749 | snakeyaml | 121, 787 | 6.5 medium | denial of service using crafted YAML input |
| CVE-2021-44228 | log4j | 20, 400, 502, 17 | 10 critical | remote code execution using logged strings |
| CVE-2022-42889 | c-text | 94 | 9.8 critical | remote code execution through string interpolation |
| CVE-2022-45688 | json.org | 787 | 7.5 high | denial of service using crafted XML input |

Table 2: CVEs used in the evaluation

Table 1 lists the artifacts we studied. The last column contains the usage count of the respective artifact (all versions) as calculated by `https://mvnrepository.com/artifact/<groupId>/<artifactId>`, accessed on 3 May 2023, this provides some indication of the popularity of the respective package. All packages used are widely used with the exception of *beanshell* that has only moderate uptake.

We selected an artifact version by picking the latest versions tagged with the CVE using `https://mvnrepository.com/artifact/<groupId>/<artifactId>`, accessed on 3 May 2023.

All those artifacts have known vulnerabilities with assigned CVE identifiers listed in Maven Central. We selected vulnerabilities based on severity (at least medium, preferably high), and their reproducibility through tests. Table 2 lists the vulnerabilities we studied, cross-referenced with artifacts using the short names defined in Table 1. Vulnerability meta data (CWEs and severities are sourced from the National Vulnerability Database (NVD), obtained from `https://nvd.nist.gov/vuln/detail/<CVE>` accessed on 3 May 2023.

### 5.2 SCA Tool Selection

There are numerous tools available to detect the presence of vulnerable dependencies in software projects. During evaluation we use a curated set of SCA tools to do the following:

1. To confirm that the tool(s) can detect the vulnerability in the original artifacts.
2. To confirm that some / all tools fail to detect the vulnerability in some / all clones.

For this purpose, we use the set of SCA tools below. We selected them in order to provide a variety of detection implementations, while aiming to increase the coverage of vulnerability DBs and keeping the effort of running multiple tools manageable.

| tool | mode | databases (java) |
|---|---|---|
| OWASP Dependency Check (owasp) | plugin | NVD, OSS Index |
| snyk | cli | proprietary |
| grype | cli | NVD, GHSA |
| Eclipse Steady (steady) | plugin | Project KB |

Some tools have the option of either invoking them from the command line (cli) or integrating scanning with the build process (plugin), thus we perform evaluation with tools in both categories.

We expect both the functionality of these SCA tools and contents of their DBs to overlap, but not to be equivalent either. Reasons for this are discussed in Section 1. As an example, adding GitHub's *dependabot* would not have increased DB coverage of our evaluation because its vulnerability DB, GHSA, is already covered by our selection.

| vulnerability | query results | consolidated | no dependency | clones detected | pov compiled | pov tested |
|---|---|---|---|---|---|---|
| CVE-2015-6420 | 1,250 (87) | 1,196 (82) | 982 (66) | 553 (20) | 549 (20) | 3 (3) |
| CVE-2016-2510 | 408 (83) | 401 (80) | 390 (79) | 12 (4) | 12 (4) | 3 (2) |
| CVE-2018-10237 | 3,285 (384) | 2,077 (285) | 1,009 (181) | 177 (36) | 177 (36) | 22 (6) |
| CVE-2021-44228 | 1,137 (92) | 1,064 (79) | 339 (33) | 185 (7) | 182 (7) | 15 (3) |
| CVE-2022-25845 | 1,532 (151) | 1,358 (130) | 419 (58) | 116 (17) | 115 (17) | 30 (6) |
| CVE-2022-38749 | 2,217 (185) | 1,076 (115) | 585 (79) | 39 (10) | 29 (8) | 20 (5) |
| CVE-2022-42889 | 2,256 (152) | 1,236 (80) | 339 (34) | 92 (8) | 92 (8) | 4 (1) |
| CVE-2022-45688 | 4,160 (347) | 1,802 (117) | 1,333 (94) | 648 (41) | 579 (24) | 457 (23) |
| (sum) | 16,245 (1,481) | 10,210 (968) | 5,396 (624) | 1,822 (143) | 1,735 (124) | 554 (49) |

Table 3: Processed Artifacts at each stage, numbers in brackets are classes of artifacts with the same group and artifact id (i.e., versions are ignored)

# 6 Evaluation Results

## 6.1 Pipeline Performance

As described earlier in Section 4.5, we start with fetching 1,000 potentially matching artifacts for each class name, up to 10,000 artifacts in total (for 10 classes). Table 3 summarises the number of artifacts after each stage of the processing pipeline. We report both the number of artifacts (as identified by their Maven coordinates, i.e. GAVs[13]) and the number of artifact equivalence classes we obtain by identifying artifacts that have the same group and artifact id, but different versions (i.e. GAs).

Some stages are aggregated. No dependency means that the respective instantiated POV project does not depend on the original artifact, but this may include some (rare) cases where the pom cannot be acquired. Similarly, if the clone analysis fails, this could be caused by sources not being available, or unparsable. We observed a very few cases where this is the case. An example where this was the case is *org.jruby.extras:jvyamlb*. This particular issue occurred when analysing CVE-2022-38749.

## 6.2 Scalability

We do not report analysis times but profiling reveals that this is dominated by performing the REST API queries, and building the instantiated POV projects.

To optimise the REST queries, we use a cache that is similar to the standard Maven cache (in $\sim /.m2$). This has a dramatic effect on performance that mirrors the standard user experience with building large projects.

The major performance bottleneck when building the instantiated POV projects is when tests need to interact with the OS or external services. For instance, the test in the CVE-2021-44225 POV project needs to start an *ldap* server in the fixture. The code execution checked by the test creates a new file by running *touch*. This interaction between Java and the OS is a potential source of flakiness, which is mitigated by pausing the test before checking whether the file exists [14]. This pause naturally delays the execution.

We found that for all analyses we could run the entire pipeline in under 3h without a cache, and in under 1h with the cache populated (i.e., consecutive runs), on standard commodity hardware [15].

## 6.3 Vulnerable Artifacts Found

Table 4 lists the artifacts found for the respective vulnerabilities. For brevity we omit version numbers of the respective artifacts.

We have found some instances for each of the vulnerabilities we studied. The different counts may reflect how attractive a library is for shading. For instance, *org.json:json* (CVE-2022-45688) is very attractive for projects to shade: it is small, has a well-defined purpose (read and write JSON), and it often needed for projects that want to export or import data.

It is remarkable that there are still undetected occurrences of high profile vulnerabilities like CVE-2021-44228 and CVE-2015-6420 months or even years after those had been reported.

[13] GAV = groupId + artifactId + version

[14] See for instance Listing 2, line 19

[15] MacBook Pro Apple M1 Pro, 32 GB of memory, MacOS Monterey 12.6.1, running a Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing) JVM

| groupId:versionId | versions | shaded |
|---|---|---|
| CVE-2015-6420 | | |
| net.sourceforge.collections:collections-generic | 1 | yes |
| org.apache.servicemix.bundles:org.apache.servicemix.bundles.collections-generic | 1 | yes |
| org.apache.servicemix.bundles:org.apache.servicemix.bundles.commons-collections | 1 | no |
| CVE-2016-2510 | | |
| ¡masked-artifact-1¿ | 2 | no |
| org.beanshell:bsh | 1 | no |
| CVE-2018-10237 | | |
| com.google.guava:guava-jdk5 | 8 | no |
| com.googlecode.guava-osgi:guava-osgi | 2 | no |
| de.mhus.ports:vaadin-shared-deps | 5 | yes |
| org.apache.servicemix.bundles:org.apache.servicemix.bundles.guava | 4 | no |
| org.hudsonci.lib.guava:guava | 2 | no |
| org.sonatype.sisu:sisu-guava | 1 | no |
| CVE-2021-44228 | | |
| com.guicedee.services:log4j-core | 13 | no |
| org.xbib.elasticsearch:log4j | 1 | no |
| uk.co.nichesolutions.logging.log4j:log4j-core | 1 | no |
| CVE-2022-25845 | | |
| com.rover12421:fastjson | 1 | no |
| com.taobao.arthas:fastjson | 1 | no |
| com.weicoder.extend:fastjson-extend | 1 | no |
| com.weicoder.extend:json-extend | 2 | no |
| com.weicoder.fork:fastjson-jdk11 | 1 | no |
| org.apache.servicemix.bundles:org.apache.servicemix.bundles.fastjson | 24 | no |
| CVE-2022-38749 | | |
| be.cylab:snakeyaml | 1 | no |
| ¡masked-artifact-2¿ | 1 | no |
| ¡masked-artifact-3¿ | 4 | no |
| org.testifyproject.external:external-snakeyaml | 11 | yes |
| pl.droidsonroids.yaml:snakeyaml | 3 | no |
| CVE-2022-42889 | | |
| com.guicedee.services:commons-text | 4 | no |
| CVE-2022-45688 | | |
| com.dimajix.flowman:flowman-plugin-json | 29 | yes |
| com.github.anandvarkeyphilips:json-java-ordered | 1 | no |
| com.github.fangjinuo.easyjson:orgjson-to-easyjson | 23 | no |
| com.github.jinahya:json-retrotranslated | 1 | no |
| com.github.tsohr:json | 1 | yes |
| com.guicedee.services:json | 241 | no |
| com.jeramtough:jtlog | 6 | yes |
| com.jwebmp.jre11:json | 3 | no |
| com.jwebmp.thirdparty:json | 6 | no |
| com.jwebmp:json | 9 | no |
| com.xliic:json-java | 1 | no |
| in.co.s13:common-json | 2 | no |
| io.github.bes2008.solution.easyjson:orgjson-to-easyjson | 5 | no |
| ¡masked-artifact-4¿ | 7 | yes |
| io.gravitee.policy:gravitee-policy-mock | 5 | yes |
| io.gravitee.policy:gravitee-policy-xml-json | 5 | yes |
| ¡masked-artifact-5¿ | 18 | no |
| org.codeartisans:org.json | 1 | no |
| ¡masked-artifact-6¿ | 22 | yes |
| ¡masked-artifact-7¿ | 22 | yes |
| ¡masked-artifact-8¿ | 22 | yes |
| ¡masked-artifact-9¿ | 22 | yes |
| ¡masked-artifact-10¿ | 5 | yes |

Table 4: Vulnerable Artifacts Detected

| vulnerability | grype | owasp | snyk | steady |
|---|---|---|---|---|
| CVE-2015-6420 | ✓ | ✓ | ✓ | × |
| CVE-2016-2510 | ✓ | ✓ | ✓ | ✓ |
| CVE-2018-10237 | ✓ | ✓ | ✓ | ✓ |
| CVE-2021-44228 | ✓ | ✓ | ✓ | ✓ |
| CVE-2022-25845 | ✓ | ✓ | ✓ | × |
| CVE-2022-38749 | ✓ | ✓ | ✓ | × |
| CVE-2022-42889 | ✓ | ✓ | ✓ | × |
| CVE-2022-45688 | ✓ | ✓ | ✓ | × |

Table 5: Vulnerability detection by SCA tool for evaluation artifacts

| vulnerability | (sum) | grype | owasp | snyk | steady |
|---|---|---|---|---|---|
| CVE-2015-6420 | 3 | 0 | 0 | 0 | 0 |
| CVE-2016-2510 | 3 | 0 | 1 | 1 | 3 |
| CVE-2018-10237 | 22 | 8 | 12 | 0 | 17 |
| CVE-2021-44228 | 15 | 0 | 14 | 0 | 15 |
| CVE-2022-25845 | 30 | 0 | 2 | 0 | 0 |
| CVE-2022-38749 | 20 | 0 | 20 | 0 | 0 |
| CVE-2022-42889 | 4 | 0 | 4 | 0 | 0 |
| CVE-2022-45688 | 454 | 0 | 33 | 0 | 0 |
| **(sum)** | **554** | **8** | **86** | **1** | **35** |

Table 6: Total of artifacts reported by each SCA tool from the set of vulnerable artifacts by CVE

Note that some artifact names are masked due to the disclosure process we follow, this will be described in detail in Section 8.

### 6.4 SCA Results

We set out to identify vulnerabilities in artifacts missed by existing tools. Table 5 lists if each CVE is detected by the SCA tools. Table 6 lists the proportion of artifacts that SCA tools report as vulnerable.

Not surprisingly, *steady* performs better than purely dependency-based tools as it employs program analysis. However, this is not consistent. The fact that is performs poorly on newer vulnerabilities suggests that updates of the database it relies on (project KB [16] [28]) are lagging behind.

More surprisingly, *owasp* also outperforms other tools for several CVEs. This is interesting as it relies on metadata, and (unlike *steady*) does not perform code analysis. However, it also analyses runtime metadata (as opposed to buildtime metadata, i.e. poms) located in the deployed artifacts (jars), usually in `META-INF` and its subfolders. This enables it to infer dependencies tools only relying on poms might miss.

## 7 Limitations and Threats to Validity

### 7.1 Precision

The analysis is designed to be precise. This is ensured by making vulnerabilities testable through POVs. However, there is a possibility that those tests do not correctly reflect the vulnerability. Sometimes vulnerabilities are reported in great detail. An example are parser vulnerabilities discovered by fuzzers like *oss-fuzz* [37], which discovers and reports payloads [17]. Sometimes, reports are vague (and sometimes this is on purpose as part of the disclosure process), and POVs are constructed from the understanding of an individual programmer of the vulnerability. Sometimes, those tests may miss some additional security measures clones may introduce - for instance, the tests for CVE-2022-42889 in *commons-text:1.9* check whether interpolator lookup provides entries for the *script*, *dns* and *url* prefixes, and test the execution of an OS command using the script prefix. Tests do not check whether actual network lookups happen with string prefixed with *dns* and *url*. This is an engineering compromise – additional network connectivity makes tests flaky , and slows down the pipeline, and we deem the overall risk that this introduces false positives very low.

### 7.2 Soundness

Our analysis is unsound. As with all program analysis, we have to strike a reasonable balance between precision, scalability and recall, with theoretical and practical limitations implying that a non-trivial analysis that is precise,

[16] `https://sap.github.io/project-kb/`

[17] For instance, see `https://www.cvedetails.com/cve/CVE-2022-38750/`, `https://bitbucket.org/snakeyaml/snakeyaml/issues/526/stackoverflow-oss-fuzz-47027` for a CVE reported by *oss-fuzz*

sound and fast is not possible. Priority was given to precision in line with industry best practices, driven by developer acceptance [4, 36, 13]. Scalability considerations had to be taken into account as repositories are very large and evolving, and maintaining a copy is not feasible for economic reasons. Therefore, we have made decisions to limit interactions with the Maven repositories via the REST API by limiting the number of queries. While some of this can be achieved by engineering (in particular, our tool extensively uses caching, similar to what other Maven clients do) , sometimes those restrictions (number of classes used to detect clone candidates, number of results and pages fetched for each query) imply that results are missed.

Our analysis will also miss clones that are on the subclass level (e.g., single functions), or clones that have custom modifications of source code beyond package renaming and altering or removing comments. Lowering the threshold for clone detection would be interesting, the question being whether this still would lead to the detection of vulnerable artifacts. This is an area for future research. We expect that the law of diminishing returns will apply here.

We think that the proposed simple analysis is still useful as its purpose is not to measure the number of artifacts associated with vulnerabilities, but to demonstrate that this is a significant problem that deserves attention.

Another limitation of our analysis is that it relies on source code. This means that components written in other languages that can be compiled into Java bytecode and deployed in the Maven repository are not covered, and this decreases the detection rate of our tool. This problem can be addressed in future work by writing a source-code based clone analysis for the respective languages, or by switching to a bytecode-based method that can abstract from compiler specifics [7].

### 7.3 Limitations of Reproducibility

By design, there are certain limitations on reproducibility. Both the repository and the vulnerability database (and therefore the SCA tools) permanently evolve and we expect that many of the vulnerable components we detect will be marked as such eventually as we release results as described in Section 8.

We make the cache containing query results, poms and source available, and report the results of running the SCA tools at the time when the experiments were conducted on the artifacts in a release repository.

Repository to be made public after disclosure to vendor delay

## 8 Disclosure

We describe the process we are using to disclose our findings. This is not straight-forward as we are not finding new vulnerabilities, so the standard vulnerability disclosure process does not necessarily apply. Instead, we detect new propagation pathways along which vulnerabilities spread, i.e. hidden dependencies not being detected by existing SCA tools due to their current limitations.

However, there is a grey zone between cloning or shading a library, and inlining some code that becomes part of a unique new product, with its own unique vulnerabilities. To decide how to disclose the presence of a vulnerability detected, we took the following criteria into account:

1. Whether the project is designed to be a full clone of the original artifact. This is determined by the artifact name being the same or very similar to the name of the original artifact. This can still be the case if the artifact uses shading.
2. Whether the project is critical. This is defined by having a low number of contributors to the associated repository, and no external dependents on Maven central outside the group of the artifact.
3. Whether the project has been remediated, interpreted as whether there was a newer version available in the repository at the time of the analysis, and the analysis did not detect the vulnerability in this version.

Based on this we used a disclosure procedure that has two possible outcomes: database disclosure, or disclosure to vendor.

For database disclosure, we use a release repository on GitHub [18] where we release the instantiated POV projects, and publish results by modifying the entries in the GitHub advisory database via pull requests.

Our disclosure process is depicted in Figure 1.

[18] The release repository is `https://github.com/jensdietrich/xshady-release`, the original POV projects used as input can be found in `https://github.com/jensdietrich/xshady`

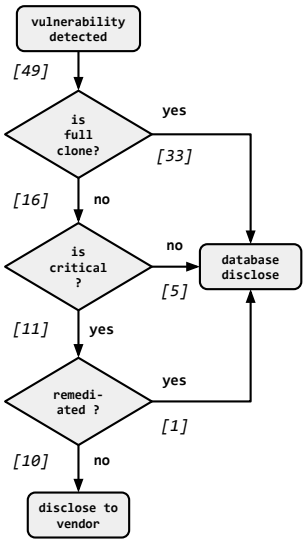


Figure 1: Disclosure procedure

Based on this process, we found ten artifacts (GAs) with several versions that require disclosure to the vendor; they relate to the following CVEs: CVE-2016-2510 (1), CVE-2022-38749 (2), CVE-2022-45688 (7). The numbers in square brackets in Figure 1 represent the number of artifact versions (GAVs) at each step of the process.

## 9 Conclusion

We have presented a novel approach to detect the presence of vulnerabilities in components that use cloning and shading. We demonstrated that this reveals blind spots in vulnerability databases and tools relying on those. This is a common problem – we detected artifacts for all vulnerabilities studied, including vulnerabilities that are critical, and have been known for years.

Our results indicate that we need to design software composition analysis tools that perform deeper analyses that do not only rely on project meta-data.

## 10 Acknowledgements

The authors would like to thank Dhanushka Jayasuriya and Emanuel Evans. The work of the first author was supported by a gift by Oracle Labs Australia, and by the Veracity project funded by the New Zealand National Science Challenge for Technological Innovation (Sfti).

## References

[1] A06:2021 – vulnerable and outdated components, 2021. `https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/`.

[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 28(3):59, 2023.

[3] Sebastian Baltes and Christoph Treude. Code duplication on stack overflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 13–16, 2020.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[5] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 464–468, 2022.

[6] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering*, 48(8):2695–2708, 2021.

[7] Andreas Dann, Ben Hermann, and Eric Bodden. Sootdiff: Bytecode comparison across different java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 14–19, 2019.

[8] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2021.

[9] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018.

[10] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24:381–416, 2019.

[11] Stephanie Dick and Daniel Volmar. Dll hell: Software dependencies, failure, and the maintenance of microsoft windows. *IEEE Annals of the History of Computing*, 40(4):28–51, 2018.

[12] Jens Dietrich, Kamil Jezek, and Premek Brada. What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, 21:1371–1396, 2016.

[13] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[14] Michael Eichberg and Ben Hermann. A software product line for static analyses: the opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[15] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[16] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, 1997.

[17] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Computing Surveys*, 54:1–38, 6 2021.

[18] Raphael Hiesgen, Marcin Nawrocki, Thomas C Schmidt, and Matthias Wählisch. The race to the vulnerable: Measuring the log4j shell incident. *arXiv preprint arXiv:2205.02544*, 2022.

[19] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400. IEEE, 2021.

[20] Kamil Jezek, Jens Dietrich, and Premek Brada. How java apis break–an empirical study. *Information and Software Technology*, 65:129–146, 2015.

[21] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.

[22] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:384–417, 2018.

[23] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[24] Jeff Luszcz. Apache struts 2: how technical and development gaps caused the equifax breach. *Network Security*, 2018(1):5–8, 2018.

[25] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. *arXiv preprint arXiv:2301.07972*, 2023.

[26] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, pages 84–94. IEEE, 2017.

[27] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

[28] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019.

[29] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 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)*, pages 449–460. IEEE, 2018.

[30] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.

[31] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 47(3):560–581, 2019.

[32] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[33] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[34] Dirk Riehle and Nikolay Harutyunyan. Open-source license compliance in software supply chains. In *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability: Communications of NII Shonan Meetings*, pages 83–95. Springer, 2019.

[35] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[36] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[37] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. In *USENIX Security symposium*. USENIX Association, 2017.

[38] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*, pages 1049–1060, 2020.

[39] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending against package typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*, pages 112–131. Springer, 2020.

[40] The MITRE Corporation. Apache struts 2 vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5638.

[41] The MITRE Corporation. Apache log4j2 vulnerability, 2021. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228`.

[42] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE'18)*, pages 319–330, 2018.

[43] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, 2014. `https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/Contrast_-_Insecure_Libraries_2014.pdf`.

[44] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361, 2016.

[45] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. 2023.

[46] Xian Zhan, Tianming Liu, Yepang Liu, Yang Liu, Li Li, Haoyu Wang, and Xiapu Luo. A systematic assessment on android third-party library detection tools. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.