On the Security Blind Spots of Software Composition Analysis

Jens Dietrich
Victoria University of Wellington
Wellington, New Zealand
jens.dietrich@vuw.ac.nz

Shawn Rasheed
UCOL | Te Pukenga
Palmerston North, New Zealand
unshorn@gmail.com

Alexander Jordan
Oracle Labs
Vienna, Austria
alexander.jordan@oracle.com

Tim White
Victoria University of Wellington
Wellington, New Zealand
tim.white@vuw.ac.nz

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 interest, and numerous software composition analysis (SCA) tools have emerged for this purpose. Most existing tools are based on the analysis of the dependency graph constructed from project metadata (declared dependencies). While this is easy to implement and scales well, there are known issues around the accuracy of the analysis. Recently, improvements have been proposed to address the low precision of this approach.

We explore a different yet related problem: the recall of SCA, i.e., whether existing methods miss dependencies on vulnerable components. We demonstrate that for the Java / Maven ecosystem this is indeed the case as (often somehow obfuscated – *"shaded"*) clones of vulnerable components are deployed in Maven Central, but not marked as vulnerable in vulnerability databases. This can be exploited for subtle package typo-squatting and confusion attacks evading detection by SCA tools.

We demonstrate that such vulnerable clones can be discovered with some rather simple tooling. Our approach is lightweight in that it does not require the creation and maintenance of a custom index, but directly uses Maven Central, and precise by design as it does not introduce new false positives.

We evaluate our approach on 29 vulnerabilities with assigned CVEs. We retrieve over 53k potential vulnerable clones from Maven Central. After running our analysis on this set, we detect 727 confirmed vulnerable clones (86 if versions are aggregated) and synthesize proof-of-vulnerability tests for each of those. We demonstrate that existing software composition analysis tools often miss those exposures. At the time of submission those results have led to changes to the entries for ten CVEs in the GitHub Security Advisory Database (GHSA) via accepted pull requests.

SCORED '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1240-1/24/10
https://doi.org/10.1145/3689944.3696165

CCS Concepts

• **Security and privacy → Software and application security**; **Vulnerability management**; • **Software and its engineering → Software libraries and repositories**.

Keywords

vulnerability detection, clone detection, shading, software composition analysis, Java, Maven

ACM Reference Format:
Jens Dietrich, Shawn Rasheed, Alexander Jordan, and Tim White. 2024. On the Security Blind Spots of Software Composition Analysis. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689944.3696165

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 [11, 26, 64]. This has in turn created new challenges. The prime example is vulnerability propagation: infamous examples include the *equifax* [31, 59] and *log4shell* [22, 60] incidents, with vulnerable and outdated components now being acknowledged as being a major security risk [39]. Other related issues include license compliance [49], typo-squatting [58], and lifecycle issues of components as demonstrated by the *leftpad* incident [7].

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) [36] and the GitHub Advisory Database [19]. If a vulnerable dependency is found, developers are notified and can upgrade dependencies to a newer version. Examples of such tools include GitHub's *Dependabot* [18], *Snyk* [53], *OWASP Dependency-Check* [38], tooling integrated into IDEs such as *IntelliJ* (backed by *checkmarx*), and build plugins like *npm audit* (for JavaScript) and the *OSS Index* Maven plugin [54]. At a high level, an SCA tool combines a scanning component to find dependencies, and a vulnerability database (*vulnerability DB*) to decide whether a dependency has a known vulnerability or not.

To combine the two, some matching logic is required, which provides 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 software identifier at the product-level, like the CPE (Common Platform Enumeration) [1] standard used by NVD. This matching is not always straightforward; it varies across tools, and can introduce 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 not (e.g., because the dependency is unused [55, 56]). 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 false positives. They may for instance detect dependencies to vulnerable code in a library that is not actually reachable [33]. This could in principle be tackled by employing more fine-grained analyses like call graph, data flow or taint analysis, although the price (in terms of computational resources needed) could be significant, and those analyses themselves have to deal with precision issues [57]. On the other hand, SCA analyses are not sound either. In particular, they miss component dependencies and therefore problems such as vulnerabilities associated with those dependencies [9].

A 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. This is an interesting problem but outside the scope of our study.

Another cause of unsoundness is *cloning*. With cloning, code is copied into the project, and these copies can carry vulnerabilities which are then hidden by the process. This can take place when an application directly clones code, or when 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, and classes to entire components. Code sharing, discussion and tutorial web sites like *Stack Overflow* [2, 46] and more lately AI-based tools like *Copilot* promote or facilitate cloning [41]. With cloning, basic engineering principles like *DRY (don't 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.

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, then cloning can be a sensible strategy to reduce the attack surface by removing now redundant dependencies.

[1] https://nvd.nist.gov/products/cpe

In the case of Java [2], there is an additional problem, a relative of the infamous *DLL hell* problem [12]. Large dependency networks may lead to conflicts between different versions of the same class added via multiple dependency paths [62]. Often, the problems resulting from this only manifest at runtime when classes are loaded and linkage related errors caused by binary incompatibility occur. API changes causing this problem are common [24, 45], poorly understood by developers [13], 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 employs 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 [3].

Maven supports this by providing the *shade plugin* [4]. With this plugin, 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 the dependency metadata remain visible to SCA tools, and they can reason about it. The shade plugin supports both cloning classes as well as "proper" shading (where packages are renamed). However, it is still possible to use copying and shading based on refactoring source code. There are several reasons to do this: (1) *lack of knowledge* – engineers may just not be familiar with the shade plugin; (2) *technical limitations* – build-time shading does not work as expected for complex dependencies as the shade plugin is in essence a static analysis tool, and as such can at best be expected to be *soundy* [30] [5]; (3) *malice* – adversaries may use shading for package typo-squatting and confusion [37, 61] attacks. This is particularly easy in Java/Maven as this only requires a new group name while retaining the original artifact id. For instance, *org.apache.logging.log4j:log4j-core:2.14.1* (with CVE-2021-44228 "log4shell") could be cloned and deployed as *apache-log4j:log4j-core:2.23.1* (the group name has been changed, and the version number falsely suggests that the vulnerability has been removed). Notably, this component would not be associated with CVE-2021-4422 in vulnerability databases, and therefore SCA tools may not be able to alert downstream projects to the presence of the respective vulnerabilities [6].

In this paper, we set out to study the prevalence of this problem, and propose a lightweight solution to address it. For the rest of this paper we refer to shading based on source code refactoring without any declaration of a dependency to the artifact being shaded simply as *shading*. This includes the special case where the original package names are retained (cloning).

The rest of this paper is organised as follows. In Section 2 we describe the analysis pipeline we have developed. The evaluation is split into two parts – we first describe the methodology used

[2] We refer here to Java as the runtime and ecosystem, not the programming language. I.e., this also includes programs that may be written in other languages compiled into Java bytecode, and deployed in the Maven ecosystem, such as *Kotlin*, *Scala* or *Clojure*.
[3] https://github.com/AdoptOpenJDK/openjdk-jdk16/tree/master/src/java.base/share/classes/jdk/internal/org/
[4] https://maven.apache.org/plugins/maven-shade-plugin/
[5] 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 [28, 57].
[6] A similar malicious use of shading is to re-license artifacts in order to avoid license restrictions.

(Section 3), followed by the results (Section 4). We then discuss the limitations of our method (Section 5), the disclosure procedure we followed (Section 6), related work (Section 7) and finish with a short conclusion (Section 8). The details of the repositories containing the source code of our tool and the data sets can be found in Section 5.

2 Vulnerability Detection

2.1 Overview

We describe the method we have developed and used to detect clones and shaded artifacts with known vulnerabilities here. Our method uses an artifact with a vulnerability and a *proof-of-vulnerability (POV)* project with tests as input, and produces a list of artifacts and projects demonstrating the presence of the provided vulnerability in those artifacts.

The design of our method is driven by two objectives: (1) **High Precision** (avoiding false positives) [7]. Precision is known to be an important factor to build analyses acceptable to engineers [4, 14, 51]. (2) **Lightweight**. We do not require the expensive acquisition, construction and maintenance of a separate index. Instead, our method can work with an existing index like Maven Central 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 tools, more vulnerable artifacts can be detected. We do not aim at detecting all those artifacts, and as with all program analyses, precision, recall and performance have to be balanced, as perfect non-trivial analyses are not feasible [16, 48].

Our method consists of three main steps: (1) given a vulnerable component available in Maven Central identified by a GAV (a combination of group id, artifact id and version, sufficient to locate the component) and a CVE, we extract a simple fingerprint that we can use to query Maven Central for similar components to create a pool of clone candidates; (2) we use a custom type-2 clone analysis to find clones amongst those candidates, also accounting for differences in code caused by shading; (3) given a POV project with witness tests for the vulnerability in the original project, we adapt this project for each clone, and run the adapted tests in order to prove the presence of the vulnerability in the clones. We describe each step in more detail below.

2.2 Clone Candidate Acquisition

We use unqualified (i.e., the package name is not considered) class names as fingerprints to identify potential clones. There are two reasons for this: (1) the Maven REST API [8] supports artifact queries by unqualified class names; (2) unqualified class names are not changed when relocating classes into other packages during shading. When working with a remote index, using all classes is not a good strategy as common class names produce large result sets with poor precision, wasting network bandwidth. 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 `JdbcDriverManagerFactory` (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 `JdbcDriverManagerFactory` is 4 (`Jdbc`, `Driver`, `Manager`, `Factory`), whereas the count for `Utils` is 1. The default strategy we use is to start with all project classes, then sort class names by token length in descending order, and to use the top 10 class names as fingerprint class.

For each fingerprint class name identified, an API query is used to fetch artifacts containing 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. This results in 10 query result sets with up to 1,000 artifacts in each. I.e., we analyse up to 10,000 artifacts for a given vulnerability. 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.

An additional sanitisation step is performed in order to remove artifacts that declare a dependency to the original cloned artifact. Those are less interesting and may even be considered as effective false positives by engineers [51] 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 looks 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.

2.3 Clone Analysis

The clone analysis used is AST-based. I.e., candidate classes are parsed and the ASTs of the original project class and the respective clone candidate class are simultaneously traversed. This requires the acquisition of source code. This is facilitated by a REST API service to retrieve sources from Maven Central. Our method is a type-2 clone detection [50], 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.

[7] Precision here is defined with respect to the presence of vulnerable code, not taking into account whether it is actually exploitable in the context of a particular application. I.e., we want to find clones which are *as vulnerable* as the original component for the respective CVE.

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

```
import org.yaml.snakeyaml.Yaml;
// more imports omitted
public class ConfirmVulnerabilitiesTests {
    @Test public void confirmCVE202238751 () {
        assertThrows(StackOverflowError.class,
            () -> parse("CVE-2022-38751.yml")
        );
    }
    static void parse (String input) throws IOException {
        FileReader reader = new FileReader(new File(input));
        new Yaml().compose(reader);
    }
}
```

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

```
{
  "id": "CVE-2021-44228",
  "artifact": "org.apache.logging.log4j:log4j-core",
  "vulnerableVersions": [
    "2.13.0","2.13.1","2.13.2","2.13.3","2.14.0","2.14.1"
  ],
  "fixVersion": "2.15.0",
  "testSignalWhenVulnerable": "success",
  "references": [
    "https://nvd.nist.gov/vuln/detail/CVE-2021-44228",
    "https://github.com/advisories/GHSA-jfh8-c2jp-5v3q"
  ]
}
```

Listing 2: CVE-2022-38751 POV Metadata

2.4 POV Test Adaptation

To avoid false positives, we rely on existing proof-of-vulnerability (POV) tests embedded in specific POV projects. Consider for instance the test used to demonstrate the presence of CVE-2022-38751, a DOS vulnerability in *snakeyaml*, shown in Listing 1 [9]. The structure of the test is straightforward – parse a malicious payload (*CVE-2022-38751.yml*), and verify that this leads to a stack overflow error.

Such tests can be sourced from code snippets or databases (like *ysoserial* [10]) showing how to exploit vulnerabilities, or existing tests in vulnerability patches. We found it generally easy to locate such tests for all vulnerabilities we studied, however, this might be more challenging for new vulnerabilities where patches or independent POV projects are not yet available. Often some additional work was required to get existing tests to compile and run as they required additional helper classes. This was usually a straight-forward iterative process consisting of inspecting compiler error logs, and locating missing classes. We also invested some time to review the acquired POVs, and integrate them into our framework. While there is some moderate manual effort required to create (acquire and check) POVs, there is emerging work to synthesise them [6, 23, 25], which could eventually lead to a fully automated detection pipeline. Those approaches are discussed in more detail in Section 7.

The purpose of such a POV test is not to demonstrate the correctness of the program but to *prove* the presence of a vulnerability. This can be achieved by demonstrating the effects of an exploited vulnerability, such as running a command to create an observable effect like creating a file (for an RCE vulnerability) or triggering a stack overflow, out-of-memory or timeout error (for a DOS vulnerability). Using such exploitability witness tests where these effects are used in oracles facilitates automation. For instance, running such witness tests can be standardised (e.g., run `"mvn test"`). Such POV tests are embedded in a project (which we from now on simply refer to as the POV) that has a minimal setup – a Maven project with a dependency on the project under test declared in `pom.xml`, the actual test, and an additional dependency on *JUnit5*. We found that POV projects are particularly useful to communicate with projects when reporting vulnerabilities as engineers are used to this setup, and can directly use it for diagnosing vulnerable components.

The weak coupling of the test(s) with the project under test achieved by creating a separate POV project (as opposed to adding a test to the actual project via a pull request) has two advantages. Firstly, it means that tests have to rely on public APIs of the program under test, which better reflects real-world usage scenarios than tests that are part of the project and have access to non-public APIs. This is similar to the use or synthesis of tests in client projects [6, 25]. Secondly, this facilitates test adaptation for other projects. I.e., assuming we have constructed a POV t_0 to demonstrate that some vulnerability is present in some component c_0, we can adapt t_0 to t_1 in order to demonstrate that the vulnerability is also present in some similar project c_1. Note that the test adaptation here is not just about adapting the actual test code, but also about the adaptation of the entire test project, as the dependency on c_0 has to be replaced by a dependency on c_1 (in `pom.xml`). The actual test may also need adaptation, as the APIs in c_0 and c_1 might be similar, but not identical. In our case, this adaptation is about changing package names, by replacing the imports in the AST-representation of the test sources to account for the package renaming that may have been performed during shading. The information about which package names to change is recorded during the clone analysis, when ASTs are compared and different package references in otherwise isomorphic ASTs are encountered.

A POV test may be (almost) mechanically constructed from existing project regression tests, in which case success indicates *absence* of the vulnerability; alternatively, it may be designed to succeed when the vulnerability is *present*, as in Listing 1. In order to support both approaches, we add a `pov-project.json` file to each POV project that indicates the vulnerable artifact versions, along with CVE details and the **testSignalWhenVulnerable** property specifying the expected test outcome (*pass*, *fail* or *error*). Note that this format aligns closely with the format used in the GHSA database. An example of this specification for CVE-2022-38751 is shown in Listing 2.

To verify the presence of the vulnerability in a clone, we adapt the POV project for the clone as described above, run *mvn test*, and check whether test signals recorded are identical with the expected signal defined in `pov-project.json`. If this is the case, the presence of the vulnerability is confirmed. This can be done by (mechanically) analysing the *Surefire* reports generated by Maven.

3 Evaluation Methodology

3.1 Dataset

The selection of the set of CVEs used for evaluation was driven by the desire to include:

(1) widely used artifacts, as determined by the number of downstream clients reported by Maven

[9] The code listings are shortened for brevity.
[10] https://github.com/frohoff/ysoserial

(2) CVEs of different types, namely vulnerabilities exploitable for remote code execution (RCE) and denial of service (DOS) attacks
(3) some high-impact vulnerabilities that have been exploited in the wild such as *log4shell*
(4) libraries from different domains
(5) CVEs in libraries we considered good candidates for cloning.

We argue that single-purpose libraries that do not have significant further upstream dependencies and do not use dynamic binding features are better candidates for shading as this makes integration easier. In particular, we expect that complex application frameworks such as *Spring* and *Struts* are not good candidates for shading. 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* [11], which contains a POV for CVE-2015-6420 that we were able to use in a slightly modified, testable form.

We selected 10 CVEs manually to fit those criteria. We then complemented this dataset with CVEs from *vul4j* [5], an independent dataset consisting of vulnerabilities, artifacts and vulnerability patches including regression tests. *Vul4j* consists of 79 CVEs. We found that most are not suitable for our purpose for different reasons: many CVEs in *vul4j* are related to application frameworks, 27 alone are from 3 frameworks (*Spring*, *Struts* and *Jenkins*). Some do not have tests (e.g., CVE-2016-3720 and CVE-2017-5662), the vulnerability cannot be reproduced with the provided test(s) (e.g., CVE-2019-10173, CVE-2018-1000850) or the component flagged as vulnerable is not in Maven Central (e.g., CVE-2018-17202, CVE-2018-17201). In the end we added 19 additional CVEs from *vul4j*. The total dataset of 29 can be found in Table 1. The table shows the wide coverage of our dataset with respect to vulnerability types, years when the CVE was assigned, and vulnerable components. The NVD base scores are extracted from the NVD entry for the respective CVE [12].

3.2 SCA Tool Selection

We used a curated set of SCA tools to confirm that: (1) All tool(s) can detect the vulnerability in the original artifacts. (2) Some / all tools fail to detect the vulnerability in some / all clones.

The SCA tools used are listed in Table 2. We selected them in order to provide a variety of detection implementations, aiming to increase the coverage of vulnerability DBs while keeping the effort of running multiple tools manageable. 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 the contents of their DBs to overlap, but not to be equivalent. 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.

[11] https://github.com/frohoff/ysoserial
[12] https://nvd.nist.gov/vuln/detail/<cve>

Table 1: Dataset used in the evaluation, organised by severity (NVD base score as of 10 April 2024) and type. *c-* stands for *"apache commons"*. Vulnerabilities not sourced from the *vul4j* dataset are highlighted bold.

cve	project	type	NVD base score
CVE-2022-25845	fastjon	RCE/XSS	9.8 CRITICAL
CVE-2022-42889	c-text	RCE/XSS	9.8 CRITICAL
CVE-2021-44228	log4j	RCE/XSS	10.0 CRITICAL
CVE-2015-6420	c-collections	RCE/XSS	N/A
CVE-2020-1953	c-config	RCE/XSS	10.0 CRITICAL
CVE-2017-18349	fastjson	RCE/XSS	9.8 CRITICAL
CVE-2016-0779	tomee	RCE/XSS	9.8 CRITICAL
CVE-2015-7501	c-collections	RCE/XSS	9.8 CRITICAL
CVE-2016-6798	sling	other	9.8 CRITICAL
CVE-2016-2510	beanshell	RCE/XSS	8.1 HIGH
CVE-2022-45688	json.org	DOS	7.5 HIGH
CVE-2019-12402	c-compress	other	7.5 HIGH
CVE-2019-0225	jspwiki	other	7.5 HIGH
CVE-2016-6802	shiro	other	7.5 HIGH
CVE-2016-7051	jackson	other	8.6 HIGH
CVE-2017-15717	sling	RCE/XSS	6.1 MEDIUM
CVE-2016-5394	sling	RCE/XSS	6.1 MEDIUM
CVE-2015-6748	jsoup	RCE/XSS	6.1 MEDIUM
CVE-2022-38749	snakeyaml	DOS	6.5 MEDIUM
CVE-2022-38751	snakeyaml	DOS	6.5 MEDIUM
CVE-2018-10237	guava	DOS	5.9 MEDIUM
CVE-2018-11771	c-compress	DOS	5.5 MEDIUM
CVE-2018-1324	c-compress	DOS	5.5 MEDIUM
CVE-2018-8017	tika	DOS	5.5 MEDIUM
CVE-2021-29425	c-io	DOS	4.8 MEDIUM
CVE-2018-1002201	zt-zip	DOS	5.5 MEDIUM
CVE-2014-0050	c-fileupload	DOS	N/A
CVE-2013-2186	c-fileupload	other	N/A
CVE-2013-5960	esapi	other	N/A

Table 2: SCA Tools used

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

4 Evaluation Results

4.1 Pipeline Performance

We implemented a tool pipeline for the method described in Section 2. Using this pipeline, we start with fetching 1,000 potentially matching artifacts for each of up to 10 class names. We record the number of artifacts after each step of processing and filtering. We report a summary of the results in Table 3, both for artifacts, and for artifacts aggregated across versions. This indicates that we find vulnerable clones for 18 / 29 components (column 10 of Table 3). Interestingly, there is one vulnerability where we do not find any matching artifact using any of the initial queries. This is for CVE-2016-6802, a vulnerability in *org.apache.shiro:shiro-all:1.3.1*. This artifact does not define classes itself, but bundles other *shiro* components, acting as a virtual meta component. Our algorithm is currently not able to extract classes to be used as queries here. The

last column shows the number of vulnerable clones where shading was applied and packages have been renamed.

The *clones detected* sets may contain false positives, while the *vulnerability confirmed* sets do not. The ratio between them (727/4,980 = 0.1460) suggests that without the testing synthesis and evaluation steps the analysis might have been too imprecise to be useful.

Figure 1 depicts, for each CVE, the pipeline throughput for each stage of processing as a fraction of the initial set of artifacts acquired via the Maven API. For almost all consolidated artifacts (i.e., artifacts appearing in the result set of more than one query), poms can be acquired. At the no dependency stage, artifacts with a dependency on the original vulnerable artifacts are filtered out as they are likely to represent effective false positives. Our mechanism to acquire sources generally works well, but there are cases where sources cannot be located using the REST API, and may just be missing. We also observed a very few cases where we were able to acquire sources, but unable to unpack the downloaded (potentially corrupted) archives. In the actual clone detection step, a significant number of artifacts is removed. Note that that this step is a simple, very fast static analysis. The last stages to establish whether the instantiated POV project can be built and tested successfully (i.e., the POV signal being confirmed) is a significantly more expensive analysis as this requires a build and multiple interactions with the Maven repository to resolve and fetch the dependencies of the instantiated POV project.

4.2 Vulnerable Artifacts Found

We find vulnerable clones for 18 of the 29 CVEs studied; details are shown in Table 3. The total number of vulnerable artifacts found is 727 across all CVEs. Often those are different versions of the same artifact: after ignoring versions and deduplicating (*"aggregated"*), the number drops to 86. The highest number of vulnerable components is detected for CVE-2022-45688, a DOS vulnerability in *org.json:json:20230227*, with 419 vulnerable clones detected (26 aggregated). JSON parsing and encoding is a popular requirement for data persistence and exchange, and the compact, self-contained nature of *json.org* makes it a good candidate for shading. This is also a library suitable to make data collected by agents persistent, i.e., it is a likely candidate to cause classpath conflicts that can be avoided with shading.

The last column in Table 3 reports the vulnerable clones where shading with renaming of packages has been applied. This is the case for 45.35% of the detected artifacts. This number is high and particularly significant as the embedded code is now less likely to be spotted by developers or tools.

We detected 15 clones for the infamous CVE-2021-44228 log4j vulnerability (3 aggregated across versions); none rename packages.

4.3 SCA Results

We have set out to investigate whether existing SCA tools and analyses find vulnerabilities in clones. To set a baseline, we first had to establish whether the selected SCA tools (see Section 3.2) can detect the vulnerability in the original artifact [13]. Table 4 summarises these results [14]. Note that for *Steady* there are vulnerabilities not included in its custom database. We marked those vulnerabilities as *n/a* in Tables 4 and 5 [15]. We performed a similar check for *Snyk* as it also uses a custom database [16]; it appears that the *Snyk* database contains all CVEs we have studied.

This shows that the SCA tools considered can detect most vulnerabilities, with minor variations between them.

We then used the same tools to check the clones detected by our analysis. The results are shown in Table 5. Since we detect vulnerable clones for only 18 of 29 vulnerabilities, this table has only 18 rows. The *any* column indicates the number of clones that are detected by at least one of these four existing tools: only 20.50% (149/727). However, this is still a very conservative estimate, and the effective detection rate from a practical point of view is lower as projects would typically only use one of these SCA tools.

8 CVEs have at least one clone that *no* standard SCA tool detects.

4.4 Scalability

Experiments were conducted on a server running Linux 6.3.8-arch1-1 with 8 Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz CPUs, and 64GB RAM. The serial run of the analyses for all 29 CVEs took 12 hours 15 minutes. We have invested significant effort in caching of both REST query results and build results, resulting in re-runs being faster by an order of magnitude (50 minutes when running the experiments using 4 parallel tasks).

5 Limitations, Threats to Validity and Reproducibility

5.1 Precision

Our 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 is parser vulnerabilities discovered by fuzzers like *OSS-Fuzz* [52], 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 that 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. But the tests do not check whether actual network lookups happen for those prefixes. 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.

[13] Often, vulnerabilities are reported for entire version ranges. The artifact we consider in this case is the latest within the range. The precise coordinates can be found in the POV repository, in the `pom.xml` dependency settings in the POV project for the respective CVE.

[14] The SCA tool reports used for Tables 4 and 5, including timestamps of when the evaluations were performed, can be found in the POV repository.

[15] The test used for this purpose was whether https://github.com/SAP/project-kb/tree/vulnerability-data/statements/<cve> returned a 404 status code or not, the checks were performed on commit 46b6932

[16] We checked the *Snyk* database through its web interface, using the following URL pattern: https://security.snyk.io/vuln/maven?search=<cve>

[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*.

Table 3: Pipeline throughput statistics – counts after each stage of processing

	query results	consol-idated	valid pom	no depend-ency	sources acquired	clones detected	pov compilable	pov testable	vulnerability confirmed	shaded
artifacts (GAV)										
min	0	0	0	0	0	0	0	0	0	0
max	4,675	2,666	2,664	1,367	1,267	669	574	483	419	190
>0	28	27	27	27	27	24	23	23	18	8
avg	1,840.34	1,029.10	1,025.83	548.14	527.97	171.72	163.07	61.07	25.07	12.72
sum	53,370	29,844	29,749	15,896	15,311	4,980	4,729	1,771	727	369
different versions aggregated (GA)										
max	410	294	293	171	165	44	34	26	26	13
avg	153.86	81.62	81.03	53.24	50.41	11.10	10.17	6.38	2.97	1.00
sum	4,462	2,367	2,350	1,544	1,462	322	295	185	86	29

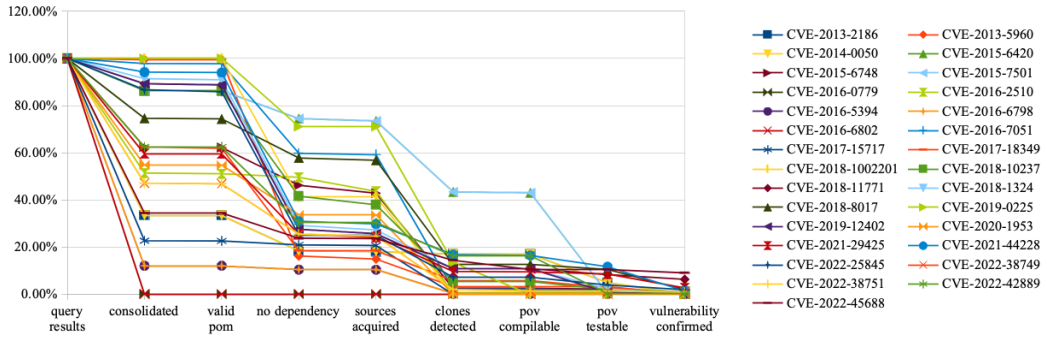

Figure 1: Pipeline throughput relative to the number of artifacts initially fetched

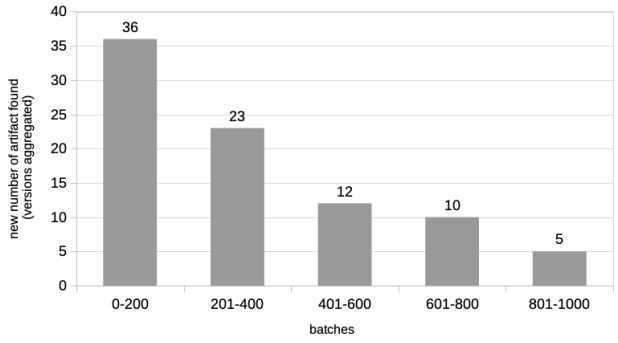

Figure 2: New vulnerable clones retrieved by batch, aggregated (versions ignored)

5.2 Soundness

Our analysis is unsound in the sense that it does not find all vulnerable components that are clones of already known vulnerable components. As with all program analysis, we must strike a balance between precision, scalability and recall, with theoretical and practical limitations implying that a non-trivial analysis that is precise, sound and fast is not possible. Priority was given to precision in line with industry best practices, driven by developer acceptance [4, 14, 51]. 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 decided to limit interactions with 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 caches extensively), sometimes those restrictions (number of classes used to detect clone

Table 4: CVEs detected by various SCA tools in the original artifact associated with the CVE

CVE	grype	owasp	snyk	steady	any	all
CVE-2013-2186	✓	✓	✓	n/a	✓	×
CVE-2013-5960	✓	×	✓	×	✓	×
CVE-2014-0050	✓	✓	✓	×	✓	×
CVE-2015-6420	✓	✓	✓	n/a	✓	×
CVE-2015-6748	✓	✓	✓	✓	✓	✓
CVE-2015-7501	✓	×	✓	n/a	✓	×
CVE-2016-0779	×	✓	×	✓	✓	×
CVE-2016-2510	✓	✓	✓	✓	✓	✓
CVE-2016-5394	✓	✓	✓	n/a	✓	×
CVE-2016-6798	✓	×	×	n/a	✓	×
CVE-2016-6802	✓	✓	✓	✓	✓	✓
CVE-2016-7051	✓	✓	✓	✓	✓	✓
CVE-2017-15717	✓	✓	✓	n/a	✓	×
CVE-2017-18349	✓	✓	✓	✓	✓	✓
CVE-2018-1002201	✓	✓	✓	✓	✓	✓
CVE-2018-10237	✓	✓	✓	✓	✓	✓
CVE-2018-11771	✓	✓	✓	✓	✓	✓
CVE-2018-1324	✓	✓	✓	✓	✓	✓
CVE-2018-8017	✓	✓	×	✓	✓	×
CVE-2019-0225	×	✓	×	✓	✓	×
CVE-2019-12402	✓	✓	✓	✓	✓	✓
CVE-2020-1953	✓	✓	✓	✓	✓	✓
CVE-2021-29425	✓	✓	✓	✓	✓	✓
CVE-2021-44228	✓	✓	✓	✓	✓	✓
CVE-2022-25845	✓	✓	✓	n/a	✓	×
CVE-2022-38749	✓	✓	✓	n/a	✓	×
CVE-2022-38751	✓	✓	✓	n/a	✓	×
CVE-2022-42889	✓	✓	✓	×	✓	×
CVE-2022-45688	✓	✓	✓	n/a	✓	×
sum	27	26	25	16	29	13

Table 5: CVEs detected by various SCA tools in clones found by our approach

CVE	total	grype	owasp	snyk	steady	any
CVE-2015-6420	3	0	0	0	n/a	0
CVE-2015-7501	3	0	0	0	n/a	0
CVE-2016-2510	3	0	1	1	3	3
CVE-2016-5394	1	1	1	0	n/a	1
CVE-2016-6798	1	1	0	0	n/a	1
CVE-2016-7051	7	0	7	0	1	7
CVE-2018-10237	31	8	21	0	17	26
CVE-2018-11771	92	0	3	0	1	3
CVE-2018-1324	2	0	2	0	0	2
CVE-2018-8017	17	0	0	0	3	3
CVE-2019-12402	1	0	1	0	0	1
CVE-2021-29425	56	0	5	1	1	5
CVE-2021-44228	15	0	14	0	15	15
CVE-2022-25845	30	0	2	0	n/a	2
CVE-2022-38749	21	0	21	1	n/a	21
CVE-2022-38751	21	0	21	1	n/a	21
CVE-2022-42889	4	0	4	0	0	4
CVE-2022-45688	419	0	34	1	n/a	34
all	727	10	137	5	41	149

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 source code modifications beyond package renaming and altering or removing comments. Whether lowering the clone detection threshold would detect more vulnerable artifacts is an interesting topic for future research. We expect that the law of diminishing returns will apply.

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 for the clone detection step when language-specific ASTs are constructed. Components written in other JVM-targeting languages are thus not covered, decreasing the detection rate of our tool. While the pipeline analysis (Table 3) suggests that this is not a big problem, future work can address it by writing source-code based clone analyses for alternative languages like Kotlin, or by switching to a bytecode-based method that can abstract from compiler specifics [8] and deal with the effects of non-deterministic compilation [66].

We make no claim that the parameters used in our analysis are optimal. Clearly, fetching more pages of query result data would improve recall, though, as Figure 2 shows, there is some evidence of diminishing returns. The initial query size of 1,000 was chosen as a good trade-off between performance and recall.

5.3 Reproducibility

Both the repository and the vulnerability database constantly evolve, inherently limiting reproducibility. We expect that, as we release results as described in Section 6 and the GHSA database gets updated and synchronised with other databases, many of the vulnerable clones we detect will be reported by other SCA tools, which use those databases. This will affect the results reported in Table 5.

The source code of the tool, the input dataset (POV for the various CVEs) and the release repository are available here:

(1) https://github.com/jensdietrich/shadedetector/ – tool source
(2) https://github.com/jensdietrich/xshady/ – POVs for original vulnerable projects (see Section 3.1), including the reports generated by SCA tools (in `<project>/scan-results`)
(3) https://github.com/jensdietrich/xshady-release– released POVs for shaded components found (see Section 6.1), including the reports generated by SCA tools (in `<project>/scan-results`)

6 Disclosure

6.1 Disclosure Process

We describe the process we are using to disclose our findings. This is not straightforward 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 detected vulnerability, 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 as having a high number of contributors to the associated repository, or 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.

For projects that are full clones, not critical or remediated, we performed *database disclosure*: We released the instantiated POV projects into a GitHub repository, and published results by modifying the entries in the GitHub advisory database via pull requests. We disclosed all other projects to the vendor.

6.2 Accepted Disclosures

At the time of submission, our work had resulted in 10 changes to the GitHub security advisory via accepted pull requests [18] (note that some pull requests were manually merged in – these are marked with an asterisk): CVE-2022-38749 (PR: 2258*), CVE-2022-42889 (PR: 2273*), CVE-2015-6420 (PR: 2326), CVE-2018-10237 (PR: 2444*), CVE-2021-44228 (PR: 2445*), CVE-2019-12402 (PR: 2823*), CVE-2016-5394 (PR: 2826), CVE-2016-6798 (PR: 2827), CVE-2015-7501 (PR: 2841), CVE-2018-1324 (PR: 2855).

We found CVE-2022-45688 in a shaded version of *json.org* in several components in the *org.graalvm.tools* group. Those were disclosed to the vendor, and a patch was announced in the Oracle Critical Patch Update Advisory July 2023 [19]. Those vulnerabilities were classified as non-exploitable.

7 Related Work

A study by Contrast Security investigated vulnerabilities in Java applications and found that "custom Java applications contain from

[18]The URL pattern for the respective pull request is https://github.com/github/advisory-database/pull/<PR>
[19]https://www.oracle.com/security-alerts/cpujul2023.html

5 to 10 security vulnerabilities per 10,000 lines of code." [63]. 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 unsafe. Mir et al. [33] 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 [15], configured to run the rather inaccurate (but fast) class hierarchy analysis (CHA, [20]). This is likely to miss many dynamic call graph edges [57] 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. [65] is related, with similar limitations. Kula et al. [27] 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 [27]. Similar results, reporting significant delays to upgrade vulnerable dependencies, were also observed for other ecosystems, for instance by Decan et al. for NPM [10] and Alfadel et al. for Python [1]. Mirhosseini and Parnin studied whether automated pull requests (PRs) are effective to speed up upgrades [34]. This mechanism is often deployed by composition analysis tools like GitHub's popular *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 *Dependabot*, and found a significantly higher acceptance (merge) rate of about two thirds. This study considered only NPM projects. Dann et al. [9] 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. We use some suitable parts of this dataset for our evaluation – details will be discussed in Section 3.1.

Ponta et al. [42] propose a hybrid, code-centric vulnerability detection method that overcomes the limitations (here mainly seen as the low precision) of metadata-based SCA approaches. Their analysis uses code changes introduced by security fixes. The tool resulting from this is *vulas*, later renamed to *Steady*. Ponta et al. [43] then compare the performance of *Steady* with *OWASP Dependency-Check*, and find that the sampled *Steady* results are all true positives, while *OWASP Dependency-Check* produces a significant number of false positives. Their evaluation includes the detection of re-bundled packages; this notion encompasses shading. We have used both *Steady* and *OWASP Dependency-Check* in our evaluation, and found that our tool finds vulnerabilities both tools miss (see Section 4.3). Possible reasons for *Steady* failing to detect certain vulnerabilities are (1) gaps in the knowledge base it relies on, and (2) the fact that it does not use a specific driver for the dynamic analysis part. This affects the reachability analysis, and may therefore lead to false negatives. In our approach, this problem is avoided by the use of targeted test cases. Our approach does not rely on the creation and maintenance of a knowledge base, whereas *Steady* does. In Tables 4 and 5 we report CVEs not supported by *Project KB* to gauge the impact the first reason has on *Steady* results.

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 [47, 50]. 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 [21] 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. [8]. 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* [67]. 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.

A recent study looked into a problem closely related to ours: Rack and Staicu [44] studied the implications of using JavaScript bundlers like *webpack* or *rollup* to merge library code into applications. They found that this practice is common, partially motivated by the desire to avoid name clashes [40], a problem similar to the classpath hell that motivates developers to shade Java libraries. They found numerous web sites with vulnerabilities from bundled libraries, although it is unclear how many of those vulnerabilities are exploitable. The difference to our study is that they studied applications (web sites), while we studied libraries. Studying web sites creates a much larger dataset that can be acquired by crawling, whereas our study focuses on deployed components in a component ecosystem, i.e., software upstream in the software supply chain,

each therefore with greater impact. Due to the use of proof-of-vulnerability tests our analysis is also more precise. Finally, our analysis is based on another ecosystem, Java.

Our approach leverages proof-of-vulnerability (POV) projects with tests, and would therefore benefit from the automated generation of such projects and tests in order to scale up the analysis and completely automate the detection of vulnerable clones. Currently, the POVs we use are crafted from existing exploit code or regression tests found in patches. There is some initial work to automate this. Iannone et al. [23] proposed *SIEGE*, a tool to automatically generate test cases demonstrating the exploitability of vulnerabilities. *SIEGE* is built on top of *EvoSuite* [17], and customises its algorithm by using fitness functions built from known details of a given vulnerability (i.e., class and method names, and line number). *SIEGE* does not use specific oracles to test for the effects of exploits. Kang et al. [25] proposed *test mimicry* and a tool *TRANSFER* implementing this approach. The aim is to construct a test case exploiting a library vulnerability in a library client. Like *SIEGE*, *TRANSFER* also uses *EvoSuite*, but uses an existing library witness test case to guide the genetic algorithm. This is similar to our use of regression tests included in the *vul4j* dataset. Chen et al. [6] propose *Vesta*, an approach that is conceptually similar to *TRANSFER*, but does not rely on library tests. Instead, (less structured) exploit code can be used to guide test generation. This approach might be suitable to partially automate the generation of POV projects and tests from exploit databases and snippets like *ysoserial*. As with *SIEGE*, It remains an open problem how to generate oracles that facilitate testing for the specific effects of an exploit.

Our use of POV tests can be considered a special case of test adaptation. Mirzaaghaei et al. [35] proposed test adaptation to repair existing and generate new test cases from existing ones during software evolution. Test adaptation across *different* applications (i.e., not just different versions of the same application) has been used successfully to utilise GUI tests in mobile apps (*"source"* apps) to generate tests in other apps (*"target"* apps) [3, 29, 32]. This requires some semantic mapping between apps. This is conceptually similar to our use of POV projects testing a vulnerable component (source) to synthesise POVs for vulnerable clones (targets). In our case, the semantic mapping is straightforward as we can assume that the clones are semantically very similar to the original component by design [20], and mapping is merely a matter of adjusting package references changed due to shading, and component references. Another difference is that our tests are not part of the component under test, but embedded in an independent dedicated test project to facilitate use cases like vulnerability reporting. The work of Kang et al. [25] discussed above can also be considered as test adaptation, with semantic matching based on existing tests providing fitness functions for test generation.

8 Conclusion

We have presented a novel lightweight 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 vulnerable clones for more than half of the 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 rely only on project metadata. Several accepted GHSA pull requests emphasise the practical relevance of our findings.

[20]We are careful here not to claim equivalence, as even renaming packages can change the semantics of a component, e.g., when reflection is used.

9 Acknowledgements

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

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.
[2] Sebastian Baltes and Christoph Treude. 2020. Code duplication on stack overflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 13–16.
[3] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 54–65.
[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
[5] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. 2022. 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*. 464–468.
[6] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2023. Exploiting Library Vulnerability via Migration Based Automating Test Generation. *arXiv preprint arXiv:2312.09564* (2023).
[7] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2021. On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2695–2708.
[8] Andreas Dann, Ben Hermann, and Eric Bodden. 2019. Sootdiff: Bytecode comparison across different java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. 14–19.
[9] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. 2021. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3613–3625.
[10] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.
[11] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24 (2019), 381–416.
[12] Stephanie Dick and Daniel Volmar. 2018. DLL hell: Software dependencies, failure, and the maintenance of Microsoft Windows. *IEEE Annals of the History of Computing* 40, 4 (2018), 28–51.
[13] Jens Dietrich, Kamil Jezek, and Premek Brada. 2016. What Java developers know about compatibility, and why this matters. *Empirical Software Engineering* 21 (2016), 1371–1396.
[14] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.
[15] Michael Eichberg and Ben Hermann. 2014. 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*. 1–6.
[16] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. 24–27.
[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[18] GitHub, Inc. 2020. Dependabot – Automated dependency updates built into GitHub. https://github.com/dependabot.
[19] GitHub, Inc. 2024. GitHub Advisory Database. https://github.com/advisories.

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

[21] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *Comput. Surveys* 54 (6 2021), 1–38. Issue 3. https://doi.org/10.1145/3446371

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

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

[24] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break–an empirical study. *Information and Software Technology* 65 (2015), 129–146.

[25] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 276–288.

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

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

[28] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 507–518.

[29] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[30] 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. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

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

[32] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of gui events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–190.

[33] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. *arXiv preprint arXiv:2301.07972* (2023).

[34] Samim Mirhosseini and Chris Parnin. 2017. 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)*. IEEE, 84–94.

[35] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 231–240.

[36] National Institute of Standards and Technology, U.S. Department of Commerce. 2022. National Vulnerability Database. https://nvd.nist.gov/vuln.

[37] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond typosquatting: an in-depth look at package confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3439–3456.

[38] OWASP Foundation, Inc. 2013. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/.

[39] OWASP Top 10 team. 2021. A06:2021 – Vulnerable and Outdated Components. https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/.

[40] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751.

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

[42] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–460.

[43] 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.

[44] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3198–3212.

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

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

[47] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. https://doi.org/10.1016/j.infsof.2013.01.008

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

[49] Dirk Riehle and Nikolay Harutyunyan. 2019. 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*. Springer, 83–95.

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

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

[52] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. In *USENIX Security symposium*. USENIX Association.

[53] Snyk Limited. 2015. snyk. https://snyk.io/.

[54] Sonatype Inc. 2015. Apache Maven plugin for Sonatype OSS Index. https://sonatype.github.io/ossindex-maven/maven-plugin/.

[55] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1021–1031.

[56] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

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

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

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

[60] The MITRE Corporation. 2021. Apache Log4j2 Vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228.

[61] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE, 509–514.

[62] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. 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)*. 319–330.

[63] Jeff Williams and Arshan Dabirsiaghi. 2014. 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.

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

[65] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem. (2023).

[66] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. 2022. Towards build verifiability for Java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 297–306.

[67] Xian Zhan, Tianming Liu, Yepang Liu, Yang Liu, Li Li, Haoyu Wang, and Xiapu Luo. 2021. A Systematic Assessment on Android Third-party Library Detection Tools. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3115506