

Beyond obfuscation: Signature-based and relocation-resistant vulnerability detection in Uber JARs

Version of May 13, 2024



Dan Plămădeală

Beyond obfuscation: Signature-based and relocation-resistant vulnerability detection in Uber JARs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dan Plămădeală
born in Chişinău, Moldova



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2024 Dan Plămădeală.

Cover picture: An abstract image based on the concept of this thesis project generated using DALL-E 3.

The content throughout parts of this report has been proofread and enhanced using Grammarly and ChatGPT.

Beyond obfuscation: Signature-based and relocation-resistant vulnerability detection in Uber JARs

Author: Dan Plămădeală
Student ID: 5503590

Abstract

Software development often relies on dependencies managed by package managers to simplify the integration of external libraries and frameworks, reducing development time. However, developers sometimes choose to bundle dependencies directly within their software packages. Bundling dependencies means including all necessary third-party frameworks directly within the application's distributable archive, such as a JAR file, to ensure all components are present without needing external installations. This practice, resulting in Uber JARs (or fat JARs), presents both challenges and advantages within the Maven ecosystem. This project examines the prevalence, risks, and impact of Uber JARs by analyzing over 9 million POM files and 12 million JAR artifacts from Maven Central, identifying artifacts with previously undetected vulnerabilities. Notably, 10.48% of the analyzed artifacts, amounting to 915,089.00, fall under the category of Uber JARs, indicating a significant prevalence within the Maven repository. Central to this work, JarSift detects Uber JARs' contents, including the libraries, their versions, and vulnerabilities. JarSift's accuracy is demonstrated with an F1 score ranging from 0.474 to 0.857, depending on the Uber JAR configuration. Analysis reveals about 17.13% Uber JARs in a small-scale dataset contained undisclosed vulnerabilities, and 0.63% of all libraries in our dataset fully completely matched known vulnerable libraries. These findings highlight the need for better detection and mitigation strategies in the Maven ecosystem and inform developers of potential risks, helping them implement more robust security measures.

Thesis Committee:

Chair: Dr. Annibale Panichella, Faculty EEMCS, TU Delft
University supervisor: Dr. Thomas Durieux, Faculty EEMCS, TU Delft
Committee Member: Dr. Jérémie Decouchant, Faculty EEMCS, TU Delft

Preface

This work concludes my one-year journey into the depths of JARs and my 3 years at TU Delft. It was a challenging journey filled with lots of joy, many difficulties, and memories for years to come.

First and foremost, I would like to thank my supervisor, Thomas Durieux, for his amazing and unwavering advice, feedback, and support throughout this project. This work would not be the work it is today without all the valuable insights and multiple brainstorming sessions we have had. Moreover, I would like to express my gratitude to Annibale Panichella, Jérémie Decouchant and Sebastian Proksch

Special thanks to all my friends who have joined me on this journey: Ana, Dan, Ioana, Ion, Konrad, Mariana, Natalia, Radu, Vlad, Wessel, and many others. The Tuesday drink and daily coffee breaks provided the needed atmosphere to recharge and get ready for a new challenge.

Last but not least, I would like to thank my family for their support, love, and encouragement along this path.

Dan Plămădeală
Rotterdam, May 2024

Contents

Preface	iii
Contents	v
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Context	1
1.1.1 Evolution of Software Development and Ecosystems	1
1.1.2 The role of Uber JARs in Dependency Management	1
1.1.3 Motivations and Implications	2
1.1.4 Research Gap and Problem Statement	3
1.2 Contribution	5
1.3 Research objectives	5
1.4 Report structure	5
2 Background	7
2.1 Java ecosystem	7
2.1.1 Maven	7
2.1.2 Gradle	7
2.1.3 Maven Central Repository	8
2.2 Maven Central CVEs	8
2.3 Java Archives	10
2.3.1 JAR file	10
2.3.2 Uber JAR	11
2.3.3 Creation and types of Uber JARs	14
2.3.4 Implications for deployment	15
3 Related work	17

3.1	Context	17
3.2	Vulnerabilities in JARs and on Maven Central	17
3.3	Software Composition Analysis	18
3.4	Code similarity	18
3.5	Software bloat	19
3.6	Software packaging	19
3.7	Existing solutions	19
3.7.1	OSS	19
3.7.2	Commercial	20
3.7.3	Results	21
4	Contribution	25
4.1	Overview and Architecture of JarSift	25
4.1.1	Corpus generation	25
4.1.2	Matching and inference	26
4.2	Signature	27
4.3	Corpus	28
4.4	Matching	29
4.4.1	Preprocessing	30
4.4.2	Matching signature to libraries	30
4.4.3	Candidate selection and ranking	31
4.4.4	Alternative matching	31
4.4.5	Perfect match identification	32
4.4.6	Post-processing and output	32
4.5	Scaling	33
4.6	UI	33
5	Evaluation	35
5.1	RQ1: How efficient is JarSift to detect embedded dependencies?	35
5.1.1	Methodology	35
5.1.2	Dataset	38
5.1.3	Results	38
5.2	RQ2: How many Uber JARs are present in our dataset?	42
5.2.1	Methodology	43
5.2.2	Dataset	43
5.2.3	Results	44
5.3	RQ3: How many Uber JARs on Maven have known undetected vulnerabil- ities?	46
5.3.1	Scanning JARs for vulnerabilities	47
5.3.2	Scanning the corpus for vulnerabilities	48
6	Discussion	51
6.1	Significance of JarSift	51
6.2	Prevalence and risks of Uber JARs	51

6.3	Report detected as new CVE?	52
6.4	Future research directions	52
7	Threats to validity	55
7.1	Internal Threats to Validity	55
7.1.1	False Positives in Signature Detection	55
7.1.2	Potential JarSift Bugs	55
7.1.3	Lack of Obfuscation Support	55
7.1.4	Signature's imperfect entropy	55
7.2	External Threats to Validity	56
7.2.1	Non-determinism in Maven's resolution mechanism	56
7.2.2	Code duplication	56
7.2.3	Incomplete Dataset	56
7.2.4	Uber JAR creation method trends	56
8	Conclusion	59
	Bibliography	61
A	Nomenclature	65
B	Signature information	67

List of Figures

1.1	Cumulative number of packages over the years on Maven Central. Note: The data for 2024 is incomplete.	4
2.1	Apache Commons BeanUtils 1.9.3 vulnerabilities info on mvnrepository.org . .	9
2.2	A minimal example Java source code	12
2.3	Java 21 readable non-verbose bytecode example	12
2.4	Verbose human-readable representation of the compiled bytecode	13
4.1	JarSift pipeline high-level architecture.	26
4.2	Overview of the architecture of the corpus generation pipeline.	27
4.3	Overview of the architecture of the inference phase of the pipeline.	27
4.4	The signature extraction procedure	29
4.5	JarSift persistence layer schema.	30
4.6	The visual representation of the contents of a an Uber JAR containing “net.bytebuddy:bytebuddy-agent:1.12.13” and “org.slf4j:slf4j-api:1.7.2”.	34
4.7	A crafted Uber JAR containing vulnerable libraries with the UI showcasing this information.	34
5.1	Distribution of component artifacts within the “groovy-all” Uber JAR.	42
5.2	Trends in Apache Maven Shade plugin configuration usage over time.	45
5.3	Distribution of non-reported vulnerable artifacts by release date, highlighting the prevalence of recent vulnerabilities.	49

List of Tables

2.1	Java Class file structure	11
3.1	Description of the JARs used to compare existing solutions. Note: The numbered jars (1-13) include <i>com.fasterxml.jackson.core:jackson-databind:2.9.10.6</i> , <i>ch.qos.logback:logback-core:1.2.3</i> , and <i>junit:junit:4.11:test</i> in their “pom.xml” files, all of which are known to be vulnerable.	22
3.2	An overview of the results returned by the tested SCA tools. In total, 15 artifacts have to be detected as having vulnerabilities. Moreover, 15 artifacts contain other libraries embedded in them that had to be detected.	23
5.1	Description of Confusion Matrix Cases	37
5.2	JarSift evaluation results.	39
5.3	Overview of POM files analysis. The percentages in parentheses indicate the proportion of total POM files; percentages in brackets represent the proportion of “maven-shade-plugin” occurrences, directly or with inheritance, respectively.	44

Chapter 1

Introduction

1.1 Context

1.1.1 Evolution of Software Development and Ecosystems

Software development practices have significantly changed in the rapidly evolving information technology landscape. From monolithic architectures to microservices, from isolated development to open-source collaboration, how we create, deploy, and maintain software has changed dramatically. This evolution reflects technological advances and shifts in how developers, companies, and end-users perceive and interact with software systems. Today, software does not exist in isolation; it is a part of a vast and dynamic environment known as a software ecosystem.

Software ecosystems are complex networks of software projects, platforms, and stakeholders. These ecosystems dynamically evolve as stakeholders continuously develop, integrate, and update projects in response to changing needs and technologies. Within these ecosystems, individual projects typically do not exist in isolation but intertwine with one another, relying on shared sources, technologies, and infrastructure [10]. The development contained in these ecosystems often relies on dependencies to reduce costs, improve maintenance, and enhance security [20]. Package managers typically manage these dependencies and handle tasks such as downloading dependencies and building the software. However, reliance on external dependencies introduces challenges, particularly in managing these dependencies to avoid security vulnerabilities, licensing conflicts, and compatibility issues.

1.1.2 The role of Uber JARs in Dependency Management

Despite the advantages of using package managers for dependency management, there are scenarios where developers choose to bundle dependencies directly with their software to ensure compatibility and avoid conflicts between different versions of dependencies Wang et al. [25]. Moreover, developers perform this to ensure that all necessary software artifacts are present for the software to function correctly, especially in environments where the exact configuration of target systems is unknown. Bundling everything into a single, self-contained package simplifies deployment and execution, as there is no need to compile the

software to run it [2, 3]. In the Java and JVM ecosystems, this approach is facilitated by various plugins for popular package managers. Historically, the And project management tool utilized the *One-JAR* plugin¹, while Maven offers the *Maven Shade Plugin*² and the *Maven JAR Plugin*³. For Gradle, the *Gradle Shadow* plugin⁴ is commonly used. Known as Uber JARs or fat JARs, these bundles enhance portability and ensure the application can run in any environment without additional configuration, streamlining the deployment process.

Some software houses and developers are making these Uber JARs available for reuse in publicly available repositories, including Maven Central. They aim to further facilitate development processes by sharing a ready-to-deploy package. However, this practice introduces a challenge: developers lack a reliable and systematic method to determine whether these Uber JARs contain other dependencies or to identify the included ones. This lack of transparency can complicate dependency management and risk the introduction of duplicate or incompatible versions between the bundled and external libraries in projects that use these Uber JARs.

1.1.3 Motivations and Implications

Building upon the concept of Uber JARs, this section explores their usage in software development and the resulting effects. While Uber JARs offer several advantages, carefully assessing their benefits and risks is necessary. Throughout our discussion, we will highlight key aspects that provide significant insight into the implications of using Uber JARs in various development environments.

As previously mentioned, Uber JARs introduce a set of challenges. To begin with, let us consider the challenge of dependency conflicts. Bundling dependencies into a single Uber JAR is efficient but can lead to significant issues. The most important of these is the risk of conflicts between different versions of the same library [2]. This situation occurs when multiple libraries or applications bundled with the Uber JAR depend on various versions of a particular dependency. Such inconsistencies can result in compatibility issues and unexpected behavior, commonly referred to in the industry as “dependency hell” [7]. Resolving these conflicts is challenging and time-consuming, particularly when developers have not clearly documented the bundled dependencies. This complexity increases if developers are unaware of the contents of an Uber JAR or if someone has removed the metadata related to its contents.

Secondly, there is the risk of bundled vulnerabilities. Bundled dependencies in Uber JARs may contain known vulnerabilities that may go unreported or be challenging to detect. Since developers package the software components together, vulnerabilities in any dependencies might go unnoticed, representing a considerable security risk. This situation is especially concerning because it allows undetected vulnerabilities to persist in the application, potentially compromising the entire system’s security.

¹<https://one-jar.sourceforge.net/>

²<https://maven.apache.org/plugins/maven-shade-plugin/>

³<https://maven.apache.org/plugins/maven-jar-plugin/>

⁴<https://imperceptiblethoughts.com/shadow/>

Lastly, for context, a Software Bill of Materials (SBOM) is a comprehensive inventory of all the components, libraries, and modules used in building software. This so-called bill includes both open-source and commercial elements. By maintaining a detailed SBOM, developers, and organizations can better manage security vulnerabilities, compliance, and licensing requirements. However, generating an SBOM has its own set of challenges. While it is possible to create an SBOM when deploying applications or systems using third-party Uber JARs, the accuracy of the results is not always guaranteed. Additionally, some Uber JARs are created in such a way that renders these tools either unusable or unreliable, as noted in [27]. This occurrence may lead to reporting, compliance, and licensing issues, especially for companies doing business with governmental organizations.

1.1.4 Research Gap and Problem Statement

When the bundled software is distributed widely, such as when Uber JARs are deployed on Maven Central for use by other developers, the impact of their use is significantly higher. The lack of transparency and information about the contents of these Uber JARs can lead developers to introduce vulnerabilities into their software systems inadvertently. Some libraries or artifacts bundled in the Uber JAR may contain vulnerabilities. An average developer may find establishing the archive's contents challenging, especially if complex or uncommon methods were used to create it and the process removed metadata. This lack of clarity means that these Uber JARs, when used unknowingly, could create attack vectors that developers are unaware of, which malicious actors may exploit. Such scenarios emphasize the importance of meticulous documentation and careful management of dependencies in software development.

Given these concerns, developing tools that facilitate the detection and analysis of the contents of Uber JARs becomes crucial. While it is indeed valuable to study the prevalence of Uber JARs in the Maven ecosystem and understand their usage patterns, the immediate priority is to give developers the means to identify and assess the components within these bundled packages effectively. Within this context, we propose developing a new tool specifically designed to address these issues. We intend to create a tool that facilitates the detection and analysis of the contents within Uber JARs, providing developers with a robust way to identify, assess, and manage the components bundled in these packages. Moreover, this tool will allow developers to quickly discern potential vulnerabilities or conflicts hidden within Uber JARs, mitigating the risks associated with their use. The substantial growth trends in the Maven Central Repository, as illustrated in fig. 1.1, emphasize the significance of such a tool. This repository's vast and ever-expanding library of components significantly complicates managing and analyzing software dependencies. As the repository grows, the diversity and volume of available libraries, including Uber JARs and other bundled packages, also increase. This expansion amplifies the potential for introducing vulnerabilities and conflicts and elevates the complexity of effectively managing these components. Our argument, therefore, shifts towards the critical need for enhanced tools and methodologies that enable developers to handle the complications introduced by Uber JARs in the software development and deployment process.

There appears to be a significant research gap regarding the content of the libraries

1. INTRODUCTION

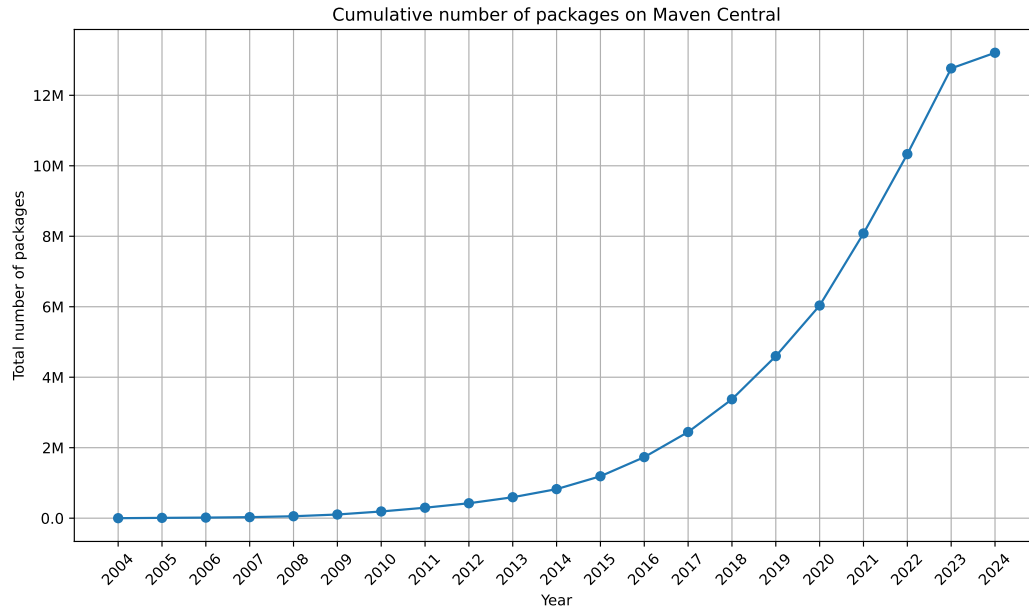


Figure 1.1: Cumulative number of packages over the years on Maven Central. Note: The data for 2024 is incomplete.

published on these repositories, with existing studies limiting themselves to a small-scale analysis of the ecosystem [3]. This research attempts to shed some light on the current state of the Java ecosystem concerning Uber JARs.

Our investigation reveals that existing tools and the current state-of-the-art can only correctly recover and detect embedded artifacts in Uber JARs under limited conditions. This finding raises concerns due to using Uber JARs in software pipelines. In response, we performed a comprehensive review of open-source and commercial tools aimed at scanning Uber JARs for vulnerabilities. Initial results indicate a notable disparity between the claimed capabilities of these tools and their actual performance in diverse conditions, highlighting a critical area for further research.

For example, the *OWASP Dependency-Check*⁵, a tool praised for its ability to detect embedded libraries, demonstrated limitation when faced with Uber JARs where metadata was removed, or shading was applied. Similarly, commercial solutions such as *Snyk*⁶, *Mergebase*⁷, and the *Sonatype Vulnerability Scanner*⁸ showed a significant inability to correctly identify or detect the embedded artifacts within our test suite.

For this reason, we argue that it is essential to introduce and develop a methodology and a tool that would provide a straightforward way to extrapolate the libraries used. The

⁵<https://owasp.org/www-project-dependency-check/>

⁶<https://docs.snyk.io/scan-with-snyk/snyk-open-source>

⁷<https://mergebase.com/java/>

⁸<https://www.sonatype.com/products/vulnerability-scanner>

relevant stakeholders could use this information to solve the issues outlined earlier. Put differently, this tool is expected to enhance the efficiency and speed of the software validation process and deployment to a production environment significantly. Moreover, the tool would also allow studying the presence of Uber JARs in the Maven ecosystem.

1.2 Contribution

In this work, we propose JarSift, a tool designed to infer embedded libraries within any JAR file and identify potential vulnerabilities through integration with a CVE database. Additionally, we present a dataset containing a list of Uber JARs available on the Maven Central repository. Furthermore, we showcase an empirical analysis of the Maven ecosystem, offering crucial insights into the prevalence of Uber JARs and identifying potential vulnerabilities.

1.3 Research objectives

To accomplish the goals mentioned above, we set out to answer the following research questions:

RQ1 How efficient is JarSift to detect embedded dependencies? In this research question, we analyze the precision and recall of JarSift to detect embedded dependencies. The goal of this research question is to validate the JarSift and to ensure that our observations for the other research questions are valid.

RQ2 How many Uber JARs are present in our set of artifacts from the Maven Central Repository? In this second research question, we analyze our dataset's frequency of Uber JARs. This research question measures the impact of Uber JARs on the Maven ecosystem.

RQ3 How many Uber JARs on Maven have known vulnerabilities that remain undetected? In our final research question, which also represents one of the applications of JarSift, we look for existing Uber JARs that contain vulnerable bundled artifacts but are not marked as such. Additionally, we attempt to quantify their impact on the Maven ecosystem.

By answering these research questions, we highlight the current state of Uber JARs in the Maven ecosystem and illustrate how their usage can be problematic. This work also opens new research directions, such as analyzing the conflicts that Uber JARs creates in projects that use them.

1.4 Report structure

The rest of this report is structured as follows: chapter 2 gives a detailed overview of the background information and context required to understand this research. Chapter 3 delves

1. INTRODUCTION

into the related work done in academia on the subject. Chapter 4 goes into detail regarding this research project's contributions with chapter 5 describing the methodology used to answer the research questions and their respective results. Following that, chapter 6 is used to discuss the results and future work required. To finish it all, chapters chapter 7 and chapter 8 identify possible factors or issues that may have influenced the results of this research and present concluding remarks, respectively.

Chapter 2

Background

In this chapter, we provide additional background to the concepts we use in this project. We start by explaining what Maven is, what CVEs are, and how these are relevant in the Maven ecosystem. Then, we describe in more detail what shading is and the causes that lead to its usage.

2.1 Java ecosystem

2.1.1 Maven

Maven¹ is a build automation tool that is generally language agnostic and primarily used for Java, Scala, and other Java platform (JVM) languages such as Kotlin. It mainly takes care of building all the components and modules required for the final runnable software artifact, dependency management, testing, deployment, and publishing. Given a declarative set of dependencies, it will download all the necessary direct dependencies and their respective transitive dependencies, subsequently integrating them and making them accessible for the application's runtime environment. Additionally, it has an exhaustive set of plugins, which allows for a vast extension of its functionality.

To resolve the necessary dependencies for a project, Maven primarily uses a default package repository, currently the Maven 2 Central² repository. However, users can also easily configure it to consume packages from other repositories, including the Atlassian, Sonatype, and Spring Plugins repositories, each with millions of indexed packages.

2.1.2 Gradle

Gradle is another modern build automation tool designed with the same end goals as Maven. The most important differences lie in its emphasis on multi-project builds and tasks, with incremental builds supported by default. The main overlap of interest in this research project's context is that Gradle uses the same package repositories as Maven. For this reason, the

¹<https://maven.apache.org/>

²<https://repo1.maven.org/maven2/>

2. BACKGROUND

problem we are trying to solve and its respective solution may be applied to the Gradle ecosystem.

2.1.3 Maven Central Repository

The Maven Central Repository, the primary repository for Maven, has shown remarkable growth since its creation. At the time of writing this, it contains a vast collection of around 12 million packages, collectively occupying more than 35 terabytes of data. Illustrated in fig. 1.1, the repository's size has steadily increased since its inception in 2005, with the release of Maven 2.0.

In the Java ecosystem, including on the Maven Central repository, each artifact has a so-called GAV coordinate, which refers to the three essential components of identifying it: the Group ID, the Artefact ID and its version. Each artifact contains at least a POM file, which normally describes the GAV coordinate of said artifact. Most artifacts also contain a JAR file containing the compiled Java bytecode. Some artifacts also have a Javadoc bundle, which includes all of the artifact documentation in HTML. Moreover, a “sources” archive may also be present, essentially containing all the Java source code used to obtain the compiled JAR file. Additionally, sometimes a “tests” and “test-sources” archive may be present. However, we are mainly interested in the standalone JAR archive.

The repository is a remote directory listing all the artifacts at their respective URLs, which Maven resolves given the GAV coordinates of the dependencies necessary for a given project. Maven retrieves the transitive dependencies required for building/running the project from this source.

2.2 Maven Central CVEs


As we go deeper to understand the potential vulnerabilities of Uber JARs, it is imperative to look at the role of Common Vulnerabilities and Exposures (CVEs), particularly those associated with Java Maven Libraries. CVEs track a multitude of ecosystems, which include the Maven ecosystem as well. Since most modern Java applications are built on top of countless open-source libraries, mitigating any vulnerability-related risks is of utmost importance. These must be reported by developers and security advisories must be created for the relevant parties.

One of Maven's main features is that it resolves all the dependencies and their respective dependencies, improves the overall development experience, and makes deploying and running an application much more straightforward. However, this convenience comes with the responsibility of ensuring these libraries are secure and free of vulnerabilities, which is where CVEs come in. In the context of Java Maven libraries, these CVEs give essential information about known vulnerabilities in libraries published on Maven. CVE IDs identify direct vulnerabilities and specifically target a defined set of artifacts, usually encompassing a Group ID, an artifact ID, and a version range, or, in other words, a list of entries, each containing an identification number, a description, and at least one public reference. In contrast, vulnerabilities from dependencies commonly refer to vulnerabilities related to dependencies that an artifact transitively requires. For instance, consider an artifact *A* that

depends on another artifact *B*. If *B* contains a direct vulnerability, integrating *A* into a project will lead to *B* being added to the classpath as well, thus introducing the vulnerability. In our study of vulnerabilities in Uber JARs, considering these Maven library-related CVEs is critical. By doing this, we can learn about the scope and nature of potential security risks linked to using Uber JARs.

Maven users can refer to CVEs to understand risks related to the libraries they use. However, tracking and managing these vulnerabilities can be difficult, especially for more substantial projects with many dependencies, and even more so when these dependencies are included together in an Uber JAR. The MVN Repository³, for example, shows for each artifact all the known direct vulnerabilities, and if there are any dependencies in the POM file related to an artifact, shows vulnerabilities from dependencies as well as seen in figure fig. 2.1. The Maven Central Repository Search⁴ is gradually being phased out in favor of the Maven Central⁵. Both platforms display only the direct vulnerabilities, not the vulnerabilities contained in dependencies. However, many tools, plugins for IDEs, CLI apps, and integration middleware, both commercial and open-source, check for vulnerabilities in all the dependencies related to a project.

Home » commons-beanutils » commons-beanutils » 1.9.3

 **Apache Commons BeanUtils » 1.9.3**

Apache Commons BeanUtils provides an easy-to-use but flexible wrapper around reflection and introspection.

License	Apache 2.0
Categories	Reflection Libraries
Tags	commons beans reflection
HomePage	https://commons.apache.org/proper/commons-beanutils/
Date	Sep 21, 2016
Files	pom (14 KB) jar (240 KB) View All
Repositories	Central Liferay Public Redhat EA Velocity
Ranking	#91 in MvnRepository (See Top Artifacts) #2 in Reflection Libraries
Used By	5,488 artifacts
Vulnerabilities	<p>Direct vulnerabilities:</p> <p>CVE-2019-10086</p> <p>Vulnerabilities from dependencies:</p> <p>CVE-2020-15250</p>

Figure 2.1: Apache Commons BeanUtils 1.9.3 vulnerabilities info on mvnrepository.org

³<https://mvnrepository.com/>

⁴<https://search.maven.org>

⁵<https://central.sonatype.org>

2. BACKGROUND

Most of the aforementioned tools rely, in fact, primarily on the National Vulnerability Database (NVD) repository and the databases managed by the MITRE Corporation, commonly referred to as the CVE Program. CVEs, specifically CVE IDs, are created and assigned by a CVE Numbering Authority (CNA). CNAs are organizations authorized to assign CVE IDs within their agreed-upon scope. Usually, these include Software Vendors, Bug Bounty Organizations, and Research and Security Organizations, among others.

To conclude this section, it is important to mention that, for a software artifact being used in a bigger application to be flagged, the vulnerability inside it is well-known in advance. In the context of Maven libraries, this may happen if someone reports this vulnerability beforehand or if a system detects the usage of a given library in a given JAR library file. For the latter to happen, the metadata about the composition of the JAR is contained in the `pom.xml` file should be present. The primary issue here is that this metadata is sometimes wholly removed for various reasons, which will be explored in the following section.

2.3 Java Archives

2.3.1 JAR file

A JAR (Java Archive) file is crucial within the Java ecosystem, packaging one or more Java class files into a single archive. These class files, compiled into JVM bytecode, are essential for the JVM to load and execute. The structure of a compiled class file is carefully organized to contain crucial components derived from the source code. It features a section outlining the class's access modifiers, its superclass, implemented interfaces, and annotations. Each class field is also detailed in a separate section, specifying its access modifiers, name, type, and annotations. Similar precision is applied to each class method, which includes information on access modifiers, names, return types, parameter types, and annotations. These method sections further contain a nested subsection with Java bytecode instructions. A significant part of the class file, the constant pool section, combines all numeric, string, and type constants used in the class, each referenced by their index in the bytecode. This exhaustive assembly of information within a JAR file facilitates the distribution and execution of Java applications in a compact, efficient, and platform-independent manner.

We demonstrate the overall structure of a compiled Java class in table 2.1.

It is important to mention that the compiler generates a separate class file for inner classes, either with a reference to their enclosing method or the main class containing a reference to its inner classes if defined in a class. The compilation process strips all comments, and it eliminates `package` or `import` sections, converting all names to fully qualified names (FQNs). For example, the source code seen in fig. 2.2, when compiled, contains the definitions seen in fig. 2.3, which are in the FQN form. A readable representation can be obtained using the `javap` tool included together with most JDK distributions. A more verbose bytecode representation of the first part of the `main` method of the `Main` example class using the same tool can be seen in fig. 2.4. Although it is not meant for humans, one could easily notice that the source code features can still be easily discernable in the bytecode version. Consider the case of the `"ArrayList<Integer> list = new ArrayList<>();"` line, which can easily be extracted and seen in the first lines of the bytecode: `new #7` and

Table 2.1: Java Class file structure

Modifiers, name, superclass, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field	Modifiers, name, type
	Annotation*
	Attribute*
Method	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled Java bytecode

`invokespecial` #9, where #7 and #9 are entries of the class file’s constant pool, which refer to the `ArrayList` type, and the `<init>` method of the same type of class, which is, in fact, the constructor of said class.

2.3.2 Uber JAR

In the fast-paced world of software development, decisions regarding deployment strategies can significantly impact both the efficacy and security of applications. While the current consensus and trends point towards more granular and managed deployment strategies, such as using package managers, legacy deployment approaches, such as using Uber JARs, still exist.

Uber JARs, from the German “über”, meaning “over” or “super”, also known as a “fat JAR” or a “JAR with dependencies”. This type of JAR file not only contains the bytecode of a Java application but also embeds all its dependencies. Typically, in a JVM environment, each separate dependency contains one or more JAR files containing the Java byte code implementation of the source class files.

The term “Uber JAR” is inherently subjective and can vary according to context and use case. For this work, we define an Uber JAR as a software artifact encompassing at least a portion of its runtime dependencies.

Motivations for using Uber JARs

Developers may create and deploy their Java applications with Uber JARs, for various reasons. One primary motivation is the simplicity of deploying a single JAR, which streamlines the process significantly. This approach also eliminates the potential for library version mismatches when libraries embedded in multiple JAR files are used simultaneously. Additionally, constructing the classpath becomes more straightforward, enhancing ease of

2. BACKGROUND

```
package org.example;

import java.util.ArrayList;
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();

        for (int i = 0; i <= 10; i++) {
            list.add(Math.floorMod(i, 3));
        }

        HashMap<String, Integer> ageMap = new HashMap<>();
        ageMap.put("John", 25);
        ageMap.put("Jane", 30);

        // prints the number of 0s in the list
        System.out.println(list.stream().filter(i -> i == 0).count());
    }
}
```

Figure 2.2: A minimal example Java source code

```
Compiled from "Main.java"
public class org.example.Main {
    public org.example.Main();
    public static void main(java.lang.String[]);
}
```

Figure 2.3: Java 21 readable non-verbose bytecode example

deployment. In scenarios where no further updates to dependencies are planned, using Uber JARs becomes even more enticing as maintainability concerns are minimized. Furthermore, Uber JARs are compatible with the default Java class loader, ensuring broad support and ease of execution across diverse environments. These factors collectively make Uber JARs an attractive option for developers seeking efficiency and simplicity in their deployment pipelines.

Challenges and considerations

Thanks to the widespread adoption of build automation and package managers, software development increasingly faces the challenge of software bloat. While the benefits of build

```

public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=3, args_size=1
    0: new           #7 // class java/util/ArrayList
    3: dup
    4: invokespecial #9 // Method
      ↪ java/util/ArrayList."<init>":()V
    7: astore_1
    [...]
   18: iconst_3
   19: invokestatic  #10 // Method
      ↪ java/lang/Math.floorMod:(II)I
   22: invokestatic  #16 // Method
      ↪ java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   25: invokevirtual #22 // Method
      ↪ java/util/ArrayList.add:(Ljava/lang/Object;)Z
    [...]
   35: new           #26 // class java/util/HashMap
   38: dup
   39: invokespecial #28 // Method
      ↪ java/util/HashMap."<init>":()V
   42: astore_2
   43: aload_2
   44: ldc           #29 // String John
   46: bipush        25
   48: invokestatic  #16 // Method
      ↪ java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   51: invokevirtual #31 // Method java/util/HashMap.put:
      ↪ (Ljava/lang/Object;Ljava/lang/Object;)
      ↪ Ljava/lang/Object;
   54: pop
   55: aload_2
   56: ldc           #35 // String Jane

```

Figure 2.4: Verbose human-readable representation of the compiled bytecode

2. BACKGROUND

automation and package managers are significant, outweighing their drawbacks, they also introduce additional complexities for developers. One notable issue is the so-called “dependency hell”, a scenario in which different packages require the same dependency but in different versions, creating conflicts and complicating project maintenance and leading to issues [22].

2.3.3 Creation and types of Uber JARs

There is a multitude of plugins and tools that facilitate the creation of Uber JARs. One of the simplest methods involves manually incorporating JAR files into a larger, final JAR. However, industry practice often employs plugins associated with the build automation tool. Notable choices exist for both Gradle and Maven. It is imperative to distinguish among three primary types of Uber JARs: unshaded, shaded, and JAR of JARs.

Unshaded Uber JARs

Uber JARs are JARs comprising the class files of all or some dependencies necessary for running the main application code. This process involves unpacking and repacking all dependency JARs into a single JAR. Unshaded Uber JARs are advantageous because the default classloader supports them. However, a significant limitation arises when multiple dependencies supply the same file, such as `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`. Duplications like this can lead to overwriting, resulting in erroneous behavior. The Maven Assembly Plugin ⁶ is a standard tool for creating unshaded Uber JARs.

Shaded Uber JARs

Uber JARs function similarly to unshaded ones, but they involve renaming the package of some or all dependencies. This technique is used to avoid conflicts arising from different dependency versions included in the shaded Uber JAR. The issue occurs when an application and one of its dependencies rely on distinct versions of the same artifact. This situation could lead to conflicts and runtime errors, as the different versions of the library may have incompatible methods or functionality. Shading addresses this by creating a separate namespace by prepending a package name to the original one. The shading mechanism applies this approach to each library version of the Uber JAR. Doing this makes it possible to include multiple versions of the same library in the Uber JAR or a Java application without causing conflicts. The application and its dependencies can then use the version of the library they were designed to work with, as each version exists in its distinct namespace. The primary issue here is that if a security vulnerability is discovered in a library, it can be more difficult to find and fix the issue in an Uber JAR that uses shading because the library might exist under various names due to shading.

Creating shaded Uber JARs is usually achieved by using the Apache Maven Shade plugin⁷. Its default behavior is similar to that of the Maven Assembly Plugin. However, it only

⁶<http://maven.apache.org/plugins/maven-assembly-plugin/>

⁷<https://maven.apache.org/plugins/maven-shade-plugin/>

allows moving (renaming) a given set of classes, GAV combinations, and package patterns. Notably, certain configuration parameters of this plugin are crucial in correctly determining the contents of an Uber JAR. These include the “useDependencyReducedPomInJar” and “minimizeJar” settings. The former replaces the “pom.xml” file in the resulting Uber JAR with a version lacking dependency information, leading to absent metadata about the archive’s contents. On the other hand, the “minimizeJar” parameter identifies the list of used classes and discards the rest from the resulting Uber JAR. Both parameters make it especially difficult for SCA tools to determine the contents of an Uber JAR accurately.

For Gradle, the Shadow Plugin ⁸ is often preferred, offering functionalities similar to the Maven Shade Plugin. Although it’s possible to construct an Uber JAR containing all the dependencies using Gradle’s build file, this approach does not support shading.

JAR of JARs

JAR of JARs, primarily a legacy approach, is rarely used outside legacy systems. It refers to a JAR file containing other JAR files embedded within. Utilization of this format requires a custom classloader capable of adding all the class files contained in the embedded JARs to the classpath.

2.3.4 Implications for deployment

The primary motive behind using Uber JARs in the industry is their portability and simplicity of deploying such an artifact. The need to manage dependencies separately during the deployment process disappears altogether. Consequently, the application or artifact can be executed in any environment supporting the Java Runtime Environment (JRE) without additional configuration.

⁸<https://imperceptiblethoughts.com/shadow>

Chapter 3

Related work

3.1 Context

Current research on vulnerabilities in Uber JARs and in the Maven ecosystem appears limited, particularly regarding the number of applications deployed that utilize Uber JARs or the prevalence of JARs with dependencies on this platform. The primary focus in recent studies primarily revolves around software debloat, code similarity detection, code deduplication, and vulnerability detection in standalone (non-Uber) JARs.

However, the work by Davies et al. [4, 9] stands out in the context of our research project. The authors propose a method allowing the detection of classes from various libraries within JAR files, leveraging a unique approach to identification. Their approach encompasses the creation of a comprehensive database populated with known class hashes. Once established, this database serves as a reference point, allowing for precise matching of the contents found in a JAR file against the pre-compiled list of class hashes. While this project does not extend their solution per se, it fundamentally uses parts of their concept while significantly expanding its precision and scope. Our methodology significantly enhances the detection capabilities by collecting a broader array of bytecode features. We carefully ensure that our technique is resistant to common obfuscations such as naming changes, shading, relocations, and other modifications that typically alter the hash of a file. Moreover, our approach will be applied to today's considerably evolved Maven ecosystem, which has grown substantially since 2011.

3.2 Vulnerabilities in JARs and on Maven Central

Several studies have investigated vulnerabilities in JAR files and the Maven Central repository. Mir et al. [15] examined the effect of transitivity and granularity on vulnerability propagation in the Maven ecosystem. They explored how vulnerabilities can spread through dependencies and assessed their impact. Düsing and Hermann [6] analyzed the direct and transitive impact of vulnerabilities on different artifact repositories, providing insights into the propagation patterns. Plate et al. [18] focused on impact assessment for vulnerabilities in open-source software libraries, investigating the consequences and severity of such

vulnerabilities.

Additionally, Pashchenko et al. [17] conducted research on vulnerable open-source dependencies, developing a method to count and identify those that pose significant risks. Gkortzis et al. [8] performed a large-scale study on open-source systems, highlighting that larger source code size poses a higher risk of potential vulnerabilities and that 65% of analyzed projects contained at least one known vulnerability.

Moreover, the authors argued that the more dependencies a project depends on, the higher the security risk. Alqahtani et al. [1] argue that present-day vulnerability databases are not used as well as they could, and to solve this, the authors propose a data-linking framework that introduces a linking process enabling the tracing of vulnerabilities across different repository and project boundaries. The most striking finding of this paper is that when considering transitive dependencies, 415,604.00 Maven projects were potentially affected by vulnerabilities.

3.3 Software Composition Analysis

The paper by Imtiaz et al. [12] evaluates the performance of various Software Composition Analysis (SCA) tools in detecting vulnerabilities in software dependencies. The study reveals significant variations in vulnerability reporting by these tools and highlights the necessity of not relying on a single SCA tool due to discrepancies. The authors recommend further research to establish frameworks to identify false positives, a major issue in the effectiveness of SCA tools.

A recent study by Zhao et al. [27] discussed evaluating various SCA tools for Java, focusing on their capability to handle different artifact formats. The authors outlined Uber JARs as a type of Plugin-packed Dependencies (PPD), as being inadequately supported by most SCA tools, which may lead to incomplete vulnerability detection. The study indicates that around 10% of projects contain PPD, suggesting that ignoring them could result in significant false negatives in detection results.

3.4 Code similarity

Code similarity is an important aspect when examining software security. Haq and Caballero [11] conducted a survey on binary code similarity, comparing various tools for measuring code similarity. Their work provides an overview of existing techniques and their effectiveness. Misu et al. [16] as part of their code duplication removal part of the pipeline, used DéjàVu [13], a map of code duplicates on GitHub. The authors employed hashing of files from each project using the SourcererCC's file-level tokenizer [21], which facilitated the identification of exact duplicates and clones with minor differences.

Lopes et al. [13] found that, as part of their study, 60% of the 49 million Java files on GitHub analyzed were distinct. An important aspect of this study is that their file-level feature extraction is slim and compares the hash of all the tokens of a source file.

Another recent study by Dietrich et al. [5] focused on clone detection between Java Maven projects based on the AST tree of the source code. The authors present a new ap-

proach for identifying vulnerable clones in the Maven repository without requiring a custom index. A relevant point expressed in the paper is that existing SCA tools are often inadequate in detecting these vulnerabilities and that these would have to be extended to support vulnerability detection in shaded and cloned artifacts. By focusing on code similarities in open-source projects, the authors aim to shed light on overlooked security risks, enhancing our understanding of software composition analysis's limitations.

3.5 Software bloat

Software bloat, which refers to the presence of unnecessary or excessive code, can introduce security vulnerabilities. Ponta et al. [19] addressed this issue by studying the attack surface reduction in an industrial application. They proposed techniques to minimize software bloat and enhance security. Soto-Valero et al. [23] conducted a comprehensive study of bloated dependencies in the Maven ecosystem. Their research shed light on the prevalence of software bloat and its impact on software security.

3.6 Software packaging

Software packaging plays a crucial role in ensuring the integrity and security of software artifacts. Merlo et al. [14] explored the topic of repackaging on Android, specifically focusing on anti-repackaging techniques. They investigated methods to prevent unauthorized modification and tampering with packaged software. Wang et al. [24] examined the issue of dependency conflicts in software projects, investigating their implications and relevance. In a comparative empirical analysis, Wang et al. [26] proposed solutions for resolving dependency conflicts in Java components, aiming to enhance software stability and security.

These related works contribute to understanding vulnerabilities in JAR files and Maven Central, and code similarity analysis, software bloat, and software packaging.

3.7 Existing solutions

We attempted to gather an exhaustive list of both open-source and commercial tools/services that provide (Uber-)JAR scanning. We created a small test suite containing 16 JAR files, which we created or found in the open-source realm. These files try to exhaustively cover a majority of the parameter combinations and situations that are relevant when it comes to SCA and vulnerability detection of Java artifacts. Below, we outline details regarding the files used in the test:

3.7.1 OSS

We explored existing open-source tools used for vulnerability detection and SCA analysis. Unfortunately, we could only find two such tools - the OWASP Dependency-Check ¹ SCA

¹<https://owasp.org/www-project-dependency-check/>

tool, and Trivy².

OWASP Dependency-Check

This tool detected the embedded libraries when no shading was applied while correctly detecting the versions used. However, when the metadata was removed, and shading was applied, the tool failed to detect any embedded libraries.

For the Uber JARs found on Maven Central (numbered 14 - 16 in table 3.1), the tool successfully managed to detect the contents of the archives or at least a part of them.

In summary, OWASP Dependency-Check is able to detect libraries embedded in Uber JARs only if metadata related to these libraries is not removed from the final Uber JAR.

Trivy

Trivy provides, among others, a filesystem scanner that allows scanning JAR, WAR, PAR, and EAR files. Our experiments with the JAR test suite did not yield anything usable. Based on Trivy's documentation, it parses the "pom.properties" and "MANIFEST.MF" files for information about a given JAR file. If these are not found, it checks its internal database for information about the JAR based on GAV identifiers, exactly like the previous solutions. This situation led to the tool being unable to detect anything at all, thus rendering it unusable for detecting vulnerable contents of Uber JARs.

3.7.2 Commercial

We searched for an exhaustive list of commercial tools that provide (Uber-)JAR scanning. We managed to test the following commercial tools, which provided the possibility to test their functionality without purchasing a subscription:

- Mergebase³
- Snyk⁴
- Sonatype Vulnerability Scanner⁵

Azul Vulnerability Detection

Azul Vulnerability Detection reportedly uses highly granular detection techniques based on hashing, enabling it to find vulnerabilities in shaded, fat, and slim JARs. Unfortunately, they do not provide the possibility to test their claims without acquiring a subscription, and thus, we can not confirm or infirm their statements.

²<https://github.com/aquasecurity/trivy>

³<https://mergebase.com/>

⁴<https://snyk.io/>

⁵<https://www.sonatype.com/products/vulnerability-scanner>

Mergebase

Mergebase developers claim they use a fingerprinting mechanism for binary files to identify and track Java archives' contents accurately. Our experiments failed to yield any usable results or detections using their trial version, which is not limited in functionality. All 16 (Uber-)JARs that we tested with Mergebase were not marked as containing vulnerable libraries. Mergebase could not correctly identify any relevant contents either (identifiers of the artifacts included in the experiment JARs).

Snyk

Snyk Open Source is Snyk's, despite its name, commercial product that allows the scanning and early detection of vulnerabilities in development pipelines. They state limited information about the technical implementation of their scanning algorithm. However, based on information from the technical documentation, the detection mechanism is based on GAV strings and CVE databases that contain vulnerabilities related to these GAV strings. We confirmed this information after running Snyk with our manually created testing suite. Snyk failed to detect any vulnerable artifacts or any vulnerable contents of the testing artifacts. Moreover, even though the metadata contained in our artifacts lists vulnerable libraries, Snyk did not manage to identify these.

Sonatype Vulnerability Scanner

The Sonatype Vulnerability Scanner allows scanning unmanaged JARs. However, based on our experiments, the scanner is limited to recognizing only known GAV strings. Essentially, it can only identify artifacts published on Maven Central and are known to have associated vulnerabilities. Notably, the scanner does not do any advanced archive scanning to determine the contents of the JARs. Consequently, the scanner correctly identified only the three artifacts (numbered 14 - 16 in table 3.1) that are listed on Maven Central. Despite these artifacts containing vulnerable embedded libraries, the absence of reported CVEs rendered them undetectable by the tool.

3.7.3 Results

To conclude this section, out of all the commercial tools we tested, none of them were effective at identifying the contents of the JARs that we have created, regardless of the modifications applied. However, the OWASP Dependency Check tool, which is a widely used and popular open-source tool in the SCA industry, succeeded in detecting some of the intentionally problematic Uber JARs. Despite this, it fell short in cases where when metadata was removed or shading was applied. An overview of the results of our testing can be seen in table 3.2. These limitations in both commercial and open-source tools underscore the rationale for developing JarSift.

3. RELATED WORK

Table 3.1: Description of the JARs used to compare existing solutions. Note: The numbered jars (1-13) include *com.fasterxml.jackson.core:jackson-databind:2.9.10.6*, *ch.qos.logback:logback-core:1.2.3*, and *junit:junit:4.11:test* in their “pom.xml” files, all of which are known to be vulnerable.

No.	Jar File	Description	Vulnerable
1	1.jar	No maven-shade-plugin applied	✗
2	2.jar	Maven-shade-plugin, no additional params	✓
3	3.jar	Like 2, with useDependencyReducedPomInJar	✓
4	4.jar	Like 2, with ch.qos.logback shaded to org.example.shaded	✓
5	5.jar	Like 4, with useDependencyReducedPomInJar	✓
6	6.jar	Like 4, with org.jackson.core shaded to org.example.shaded	✓
7	7.jar	Like 6, with useDependencyReducedPomInJar	✓
8	8.jar	Like 2, without META-INF/maven/**	✓
9	9.jar	Like 3, without META-INF/maven/**	✓
10	10.jar	Like 4, without META-INF/maven/**	✓
11	11.jar	Like 5, without META-INF/maven/**	✓
12	12.jar	Like 6, without META-INF/maven/**	✓
13	13.jar	Like 7, without META-INF/maven/**	✓
14	flink-shaded-jackson.jar	Flink + shaded jackson 2.10.1 with maven folder	✓
15	flink-shaded-guava.jar	Flink + shaded guava 18.0.* with maven folder	✓
16	kyuubi-shaded-zookeeper.jar	Kyuubi + shaded guava 16.0 and curator 2.12.0 with maven folder	✓

Table 3.2: An overview of the results returned by the tested SCA tools. In total, 15 artifacts have to be detected as having vulnerabilities. Moreover, 15 artifacts contain other libraries embedded in them that had to be detected.

SCA Tool	Detects vulnerabilities	Detects contents
OWASP Dependency-Check	4/15 (26.66%)	4/15 (26.66%)
Trivy	0/15 (0%)	0/15 (0%)
Mergebase	0/15 (0%)	0/15 (0%)
Snyk	0/15 (0%)	3/15 (20%)
Sonatype Vulnerability Scanner	0/15 (0%)	0/15 (0%)

Chapter 4

Contribution

4.1 Overview and Architecture of JarSift

Figure 4.1 provides a high-level overview of JarSift’s pipeline, detailing each stage from signature extraction to user interface integration. This section elaborates on each component, describing the architecture and scalability decisions required to accommodate the vast Maven ecosystem.

JarSift begins by extracting signatures from bytecode classes, which is presented in section 4.2, followed by storing these signatures in a database to form a comprehensive corpus, described in section 4.3. It then uses this corpus to identify embedded libraries within Uber JARs, addressing significant challenges and matching as elaborated in section 4.4. The design and implementation required adaptation to manage the scale of data involved, which includes millions of JAR files and billions of classes.

The architecture also integrates a user interface as presented in section 4.6, which enhances developer interaction by allowing easy identification of embedded dependencies and their vulnerabilities. This feature aims to simplify the tool’s operation and promote wider adoption.

In addition to its core functionalities, we connected JarSift to a database of known vulnerable libraries, which we use to identify whether a given Uber JAR contains vulnerable dependencies. The vulnerability database can be updated and, therefore, support future vulnerabilities.

We made our tool publicly available on GitHub¹.

4.1.1 Corpus generation

At the core of our contributions is a scalable signature extraction method. This initial step was critical, as it laid the basis for our subsequent work. The objective was to create a signature capable of withstanding the various transformations that class files might undergo during the Uber JAR process, thus maintaining the integrity and accuracy of our analysis.

¹<https://github.com/Cornull11/JarSift>

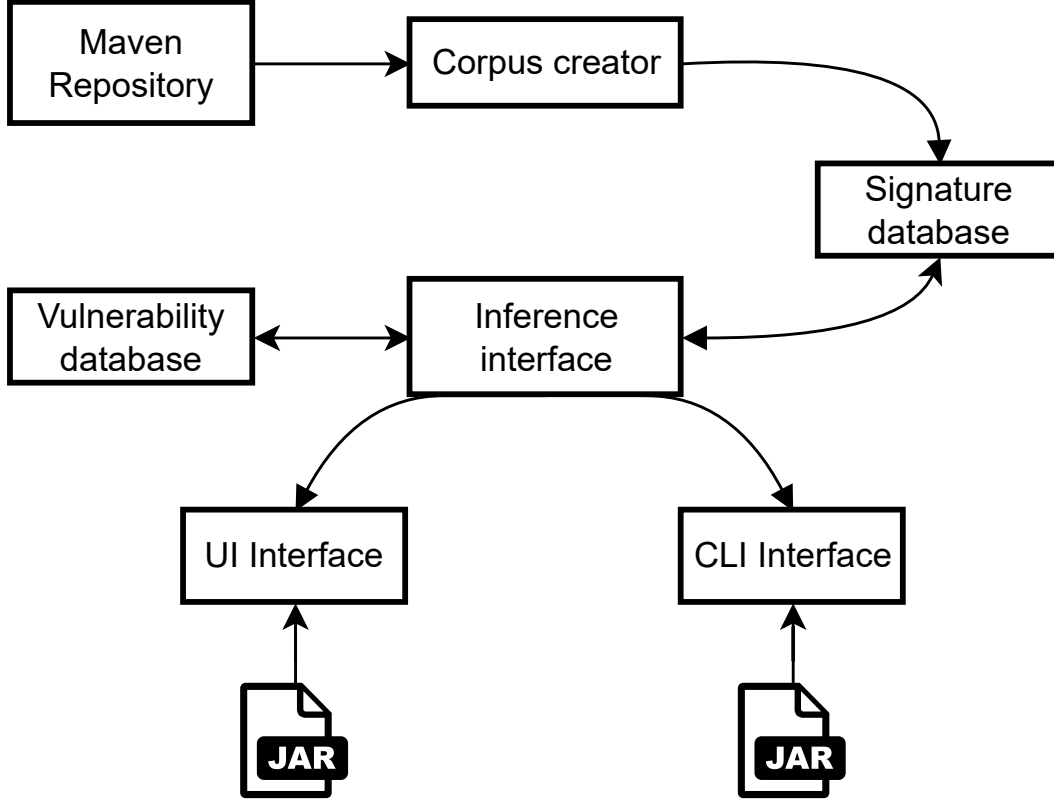


Figure 4.1: JarSift pipeline high-level architecture.

At this stage, the JarSift scans through all JAR files located in the local `.m2` repository, which is a partial copy of the Maven Central Repository. It extracts features from each `.class` file contained in every JAR file. These features are then aggregated into a unique signature through a high-entropy hashing function, and the resulting signatures are committed to a MariaDB database. This database, serving as the corpus, becomes a repository of signature hashes, each linked to a specific library, effectively capturing the characteristics of countless class files across various libraries. An overview of this stage is presented in fig. 4.2.

This corpus serves as an extensive database of class signatures from a large set of JVM dependencies. This database is crucial as it forms the backbone of our tool’s ability to identify whether a class file within Uber JARs originates from a different library. The database and the heuristics we apply to it for matching are elaborated on in section 4.4.

4.1.2 Matching and inference

Next, we have the inference phase of JarSift, where it navigates through the given input Uber JAR, extracting signature hashes in a manner analogous to the corpus creation phase. However, instead of populating the database, these signatures are used to query the pre-

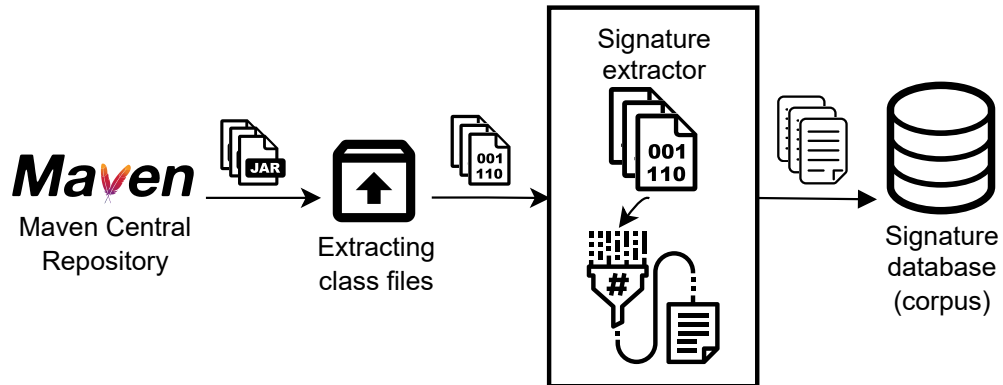


Figure 4.2: Overview of the architecture of the corpus generation pipeline.

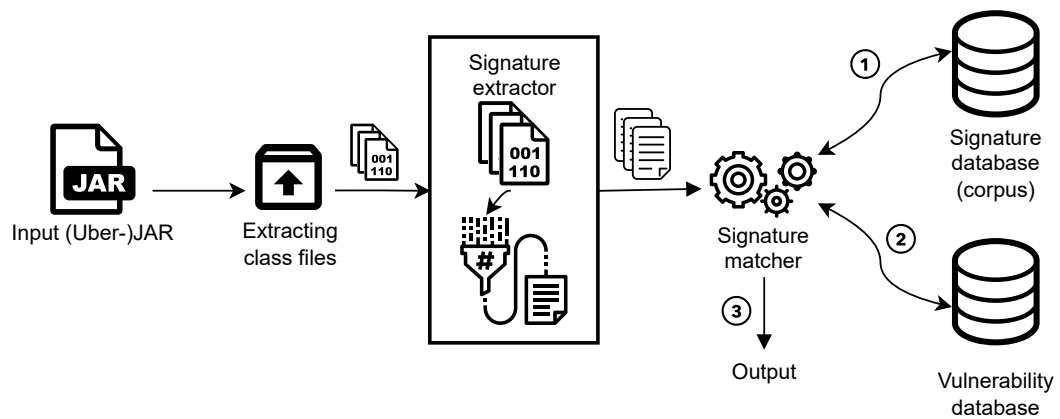


Figure 4.3: Overview of the architecture of the inference phase of the pipeline.

established corpus, allowing the tool to identify all libraries embedded within the Uber JAR. An overview of this stage is presented in fig. 4.3. This phase is critical in addressing the posed research questions.

4.2 Signature

The signature is the cornerstone of JarSift. The signature aims to create a unique and reproducible identifier for each different class while withstanding class relocation which normally moves a class to a different package. Since the signature needs to withstand class relocation, we could not rely on a file hash. Instead, we had to develop a new signature mechanism. Indeed, the signature of a class file is defined by the hash digest of the feature set we extract from the class file. We describe a feature as easily discernible characteristics and aspects that determine the behavior of a class file, which makes it unique. We selected the most distinguishable and extractable features from a compiled class file. These features

include the complete list of the extracted features in appendix B. The feature set consists of the entity’s name without the package name, its access modifier, the type of fields, the return type, the argument type, the exception type, and the opcodes and operands of the inner code of methods. Moreover, we also look at constructors, inner classes, and annotations of every class file that contains these. We present the complete list of the extracted features in appendix B.

Fully qualified names (FQN) are used for class identification within the Java Virtual Machine (JVM). All non-primitive types, classes, and interfaces have their own unique FQN, and it is the primary mechanism used by the Java Virtual Machine to locate definitions. These normally take the form `classpath.Name`, with a real example being `java.util.ArrayList` and it describes the path at which said class can be found. Most classes that start with `java` are part of the standard library. Other classes are found in JAR files linked to the classpath, a conceptual collection of in-memory definitions the JVM utilizes for class lookup. During the Virtual Machine (VM) initialization process, the JVM loads classes based on their locations in the JAR files or the standard library. Afterward, when a reference is found, the JVM looks it up in the classpath to retrieve the corresponding class, which delimits the set of all classes that are loaded in the current JVM runtime.

In fig. 4.4, we present a rough outline of the information we extract from each class file. In order to make our signature extraction resilient to relocations, renaming, and shading, we ignore the path. To be more precise, all references to types, classes, and interfaces are referenced by their fully qualified name (FQN), and since the prefix can always change depending on the modifications and transformations done to a Java project, we retain only the last part.

For the extraction itself, we use the ASM library ² to analyze every given Java class file. The advantage of this library is that it is JVM language-agnostic and can be used for many other software ecosystems, such as Groovy and Kotlin. Moreover, considering that it is a library that analyzes JVM bytecode, it can be used outside of the Java and Maven ecosystem since it can be used to extract features and analyze the bytecode of other JVM languages. ASM is used to extract the features as referenced in appendix B. Following the extraction, these features are aggregated into a dictionary, forming the basis for generating a unique identifier for each unique class, called the “class hash”. To achieve this, we use the XXH3³ hash function, selected for its efficiency and ability to produce high-entropy hash values. The hashes are stored in the corpus database.

4.3 Corpus

The corpus is the persistence layer of JarSift, implemented using the MariaDB database to store the signatures and metadata of known artifacts. This comprehensive collection is needed to identify if a class is already present in a different artifact, with the matching algorithm described in section 4.4. The database is also important for a deeper analysis and understanding of JAR software artifacts, particularly in the context of Uber JARs, allowing

²<https://asm.ow2.io/>

³<https://github.com/Cyan4973/xxHash>

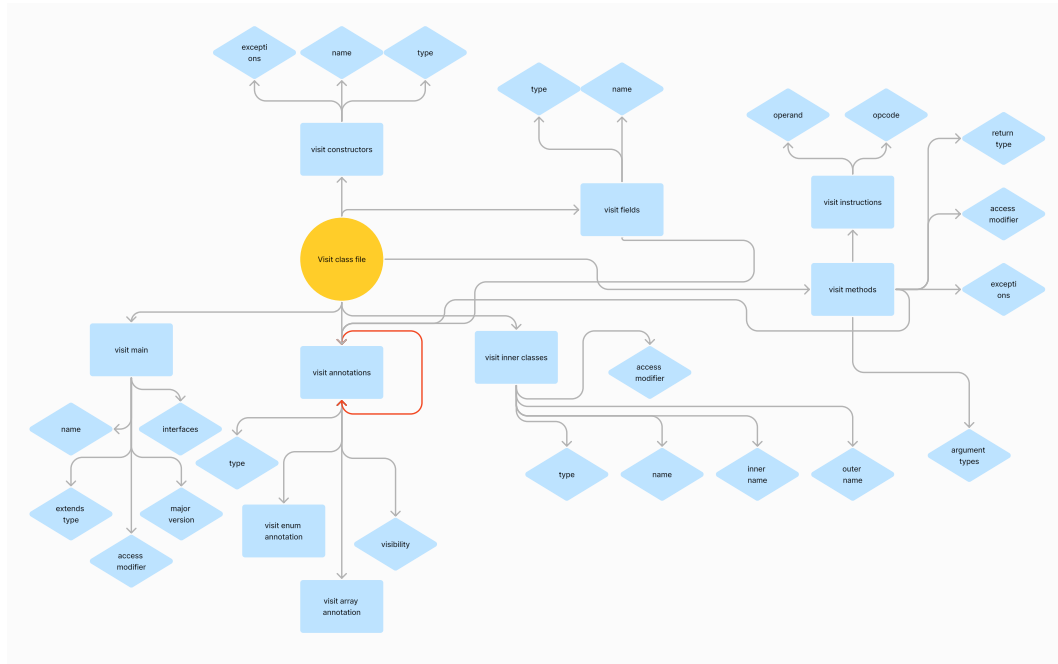


Figure 4.4: The signature extraction procedure

for a multitude of analysis tasks. These include identifying common patterns and trends in library usage, detecting vulnerabilities, and assessing the prevalence of Uber JARs within the Maven ecosystem. Furthermore, the deliberate inclusion of extra fields under the library table allows us to extract insights and generate statistics, seeding our research findings with empirical data and concrete observations. The schema of this database can be viewed in fig. 4.5.

4.4 Matching

The matching phase describes the identification of embedded libraries within a given JAR using JarSift. To accurately determine libraries and their versions contained within an Uber JAR, we perform a series of steps. These steps include decompressing the JAR file, identifying all bytecode classes, and computing their signatures as outlined in section 4.2. The resulting list of signatures, along with the corpus described in section 4.3, serve as inputs for the matching phase. The objective is to match the signatures from the JAR with those in the corpus to identify the corresponding libraries.

This process faces numerous challenges, including the presence of common classes, such as exception classes, that are implemented across multiple projects. Scalability is another significant issue caused by the extensive number of signatures and libraries. Other issues are caused by ecosystem-related issues, such as the high number of duplications in Maven Central and complications due to the Uber JAR process, including class relocation,

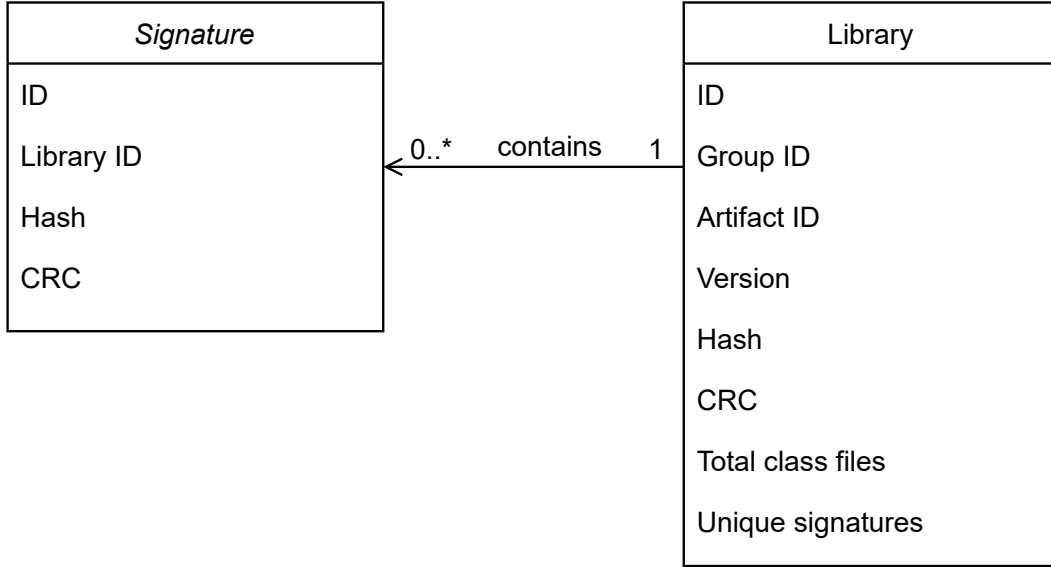


Figure 4.5: JarSift persistence layer schema.

shading, and minimization. It is also quite common for the same file to appear across multiple versions of different libraries. Given this complex context, JarSift faces a non-trivial problem in accurately identifying unique signatures. This section describes the process that JarSift follows to minimize the occurrence of false positives.

4.4.1 Preprocessing

To efficiently query the corpus, the signatures of the input JAR must first be encoded. To achieve this, a temporary table is created in the database to store the signatures of the input JAR. The temporary table is indexed and stored in memory to speed up the corpus query. Then, JarSift identifies the packages in which the classes have been declared based on the path of the class. This information will be used to group signatures together. Another heuristic we apply here is an invariant condition that two classes in the same package should remain together. Finally, JarSift indexes the signatures to packages and the packages to signatures to speed up the matching in later steps.

4.4.2 Matching signature to libraries

The goal of this first matching phase is to identify all the libraries that have at least one class (in this case, a signature) included inside the JAR. This is done by doing an intersection between the temporary table (presented in section 4.4.1) and the corpus section 4.3. The result of this intersection is a list of signatures that are present inside the input JAR and the libraries contained in the corpus. JarSift then computes the proportion of the library that is included in the input JAR, e.g., how many classes from the library are included in the input JAR versus the total number of classes in the library. As a result, we know, for example,

that 110/112 of Library A's classes are included in the input JAR, and 11/11 of Library B's classes are included in the JAR.

Unfortunately, we cannot always rely on 100% matching because some shading options can remove some classes.

4.4.3 Candidate selection and ranking

From the matching process, we generate a comprehensive list of potential libraries that also includes many false positives and noise. In order to reduce the number of false positives, we develop multiple heuristics that reduce the number of false positives, which are described below.

We define two sets for use in the matching phase and for subsequent filtering based on the heuristics described below. The "candidates" set stores potential library candidates identified during the matching process, while the "selfCandidates" set includes libraries that are considered to be identical to, or different versions of, the library containing the input JAR file.

For each class file in the input JAR, we retrieve all libraries containing that file from the database. We then map each class's hash to these libraries in the "libToHash" hash table. The process continues by iterating over each library listed in "libToHash". Libraries containing less than two unique class hashes are excluded from consideration. This exclusion helps avoid the misidentification of generic classes, such as exception classes, which often share the same signatures. On the other hand, if the library includes more than 50% of the unique class hashes from the input JAR file, it is considered a strong candidate and added to both "candidates" and "selfCandidates". If none of these conditions is met, the method iterates through each package associated with the library. A library is made a candidate and added to "candidates" if it contains all class hashes found in a particular package and the package has more than one unique class. The idea is that the two libraries must share a set of classes that have a strong correlation, specifically being located within the same package path, to ensure that the classes are related.

Next, the "candidates" list is sorted based on a combination of factors: inclusion ratio - the ratio of the number of class hashes in the library to the expected number of total classes; number of class hashes - the number of class hashes the library contains; expected number of total classes - the expected number of total classes in the library based on database information; GAV (group ID, artifact ID, version) - lexicographic order of the library's GAV string. The goal is to process the libraries with the highest confidence first.

Finally, the database is queried to retrieve detailed information about each library in the sorted "candidates" list.

4.4.4 Alternative matching

In the process of library identification, it is not uncommon to encounter libraries that are essentially identical across different versions, or that occupy different namespaces altogether. In this stage, we find all the identical alternatives with the exact same signatures. For example, a library might release a new version where the only change is to the test classes, which

do not appear in the JAR file. A specific example is the library “org.apache.lucene:lucene-queryparser”, which shows no bytecode difference between versions 9.9.1 and 9.9.2, resulting in identical signatures.

The procedure begins with an iteration through the sorted “candidates” list. For each library, the presence of class hashes is verified, and if these are not present, the library candidate is skipped. Then, we compare the library with each library in the “selfCandidates” list. Libraries identified as different versions of the same library are marked as self and added to “selfCandidates”.

Lastly, starting from the current position, we iterate through the remaining libraries in the “candidates” list. In instances where two libraries are considered different versions or if one library completely contains the classes of another, the current library is considered an alternative to the other library and vice versa. They are linked accordingly for later usage in the matching process.

4.4.5 Perfect match identification

This stage of the pipeline is to filter and determine the candidates that are the exact same library as given in the input JAR. For example, given a list of package names (e.g., org/apache/lucene/search/join), we try to establish a list of libraries that have this package. Normally, for the given example, only org.apache.lucene:lucene-core of any version should match the query. However, many other artifacts are matched in this case due to duplication and shading on Maven Central.

The process begins with a detailed traversal of each package listed in the “packagesToHashes” map. For every package encountered, we identify all libraries that contain class hashes found in that package. Then, we check if a package is only associated with a single library (excluding self and its marked alternatives). To finish this stage, if a library is the only one associated with a package, it is marked as a “perfect match” and will be further used within the pipeline.

4.4.6 Post-processing and output

This step finalizes and compiles the final results of the inference, which are used for further steps, such as vulnerability matching and presenting the results in the UI.

Initially, we filter the “candidates” list to remove any libraries with no class hashes (these are typically alternatives identified earlier). Then, we return the filtered list of library candidate objects, which contain information about the potential matching libraries, including their GAV, inclusion ratio, number of class hashes, the expected number of total classes, and whether they are self, perfect matches, or have alternatives.

Given the matching results, we can set a varying inclusion threshold, which we find fitting to determine the definite inclusion of a library in an input JAR. After filtering by this parameter, we make use of the vulnerabilities database bundled with JarSift to decide whether the input JAR contains vulnerable classes.

4.5 Scaling

For our study, we gathered versioned packages from Maven Central, which is one of the most popular and widely used repositories of Java artifacts. We collected packages that were released between 2004 and Feb. 2024. The resulting dataset consists of about 12,316,987.00 unique versioned packages of 445,178.00 projects, weighing in at about 8.20 TB. The initial corpus creation took 36 hours on a 20-core CPU with 188 GB of RAM. This successfully extracted more than 400 million signatures, 30 million of which are unique. Considering the vast number of signatures, finding a way to make it scalable and usable was primordial. With no optimizations, the inference stage of JarSift would have taken unusable amounts of time, and thus, a multitude of changes had to be applied. Initial attempts involved denormalizing the database schema. However, the denormalization process would have taken an unfeasible amount of time due to MariaDB's limitations. Moreover, executing the denormalization in a multithreaded manner led to concurrency issues, and the database kernel would reach an unsolvable deadlock.

The next step was to optimize the SQL queries and retrieve only the necessary data. Moreover, creating indices on the most commonly queried columns and the columns used in "JOIN" queries led to a big boost in performance. Still, running inferences on big JARs would lead to very long processing times caused by IO bottlenecks and the limited multithreaded capability of MariaDB's core. For example, a JAR containing 32 thousand class files would take more than 20 minutes on a modern PC just for the inference phase.

The final phase of our optimization process, which led to the version used in production and for the evaluation setup, involved a big change in the persistence backend. Given the availability of vast amounts of memory, loading the main tables into memory was feasible, which allowed for a vast performance improvement, especially for larger JARs. Moreover, it allowed a deeper analysis of the presence of vulnerable JARs in other artifacts.

4.6 UI

The UI is the secondary way of using JarSift, enhancing the user interaction by simplifying how developers can analyze and understand JAR compositions. It provides a visual representation of the JarSift output for a given input JAR, helping identify included dependencies and their vulnerability status. An example of the UI given a JAR containing two shaded libraries can be seen in fig. 4.6. The UI contains a tree-like structure outlining the correspondence of each class file to a certain library. Moreover, a list of alternatives or direct similarities of other versions of the same library is also presented.

In the visual example shown in fig. 4.6, an input JAR containing "net.bytebuddy:bytebuddy-agent:1.12.13" and "org.slf4j:slf4j-api:1.7.2", with the correct detection.

The UI also provides information about the detected libraries that are known to contain vulnerabilities. The same information is provided for the alternative libraries as well. An example of a manually crafted Uber JAR containing "com.fasterxml.jackson.core:jackson-databind:2.9.10.6" and other vulnerable libraries are correctly detected and marked as vulnerable, as seen in fig. 4.7.

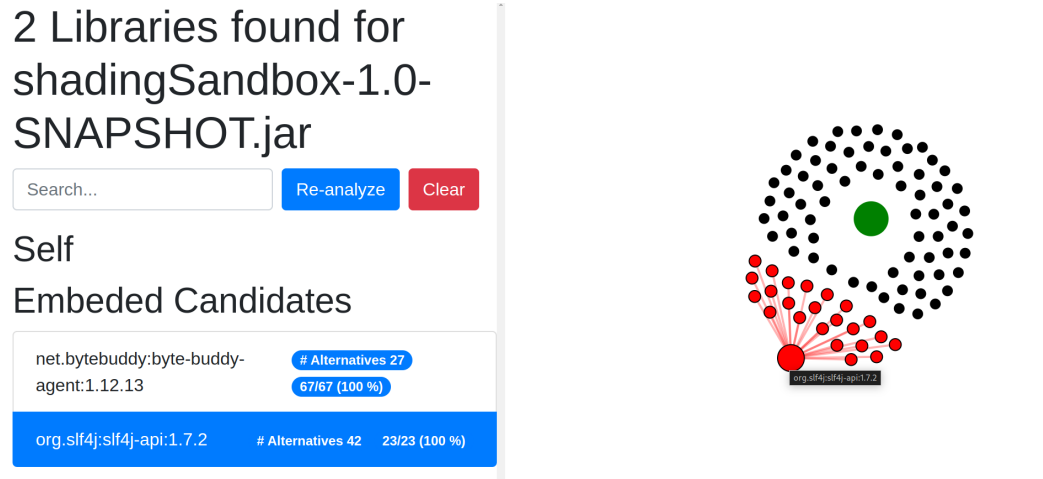


Figure 4.6: The visual representation of the contents of a an Uber JAR containing “net.bytebuddy:byte-buddy-agent:1.12.13” and “org.slf4j:slf4j-api:1.7.2”.

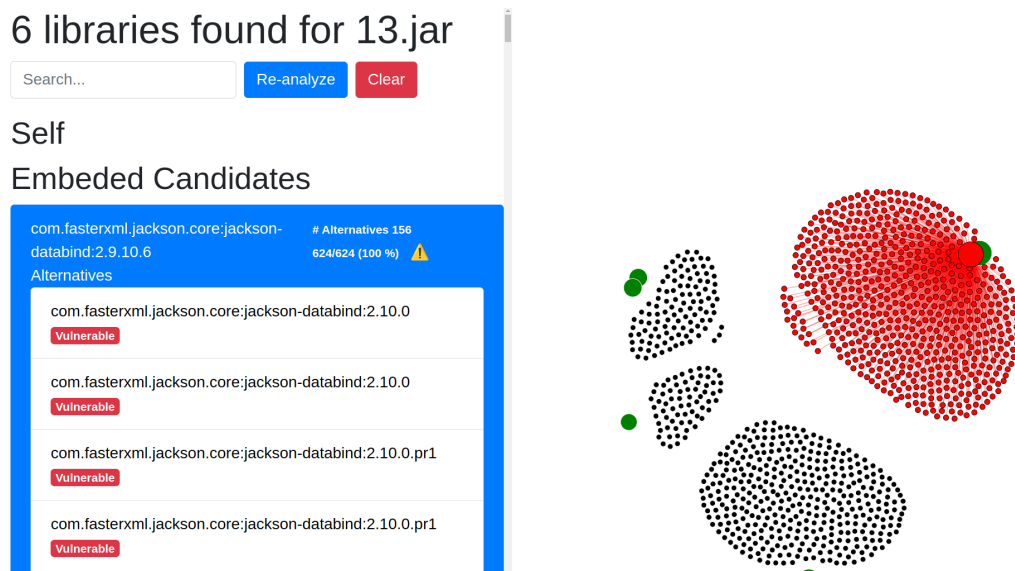


Figure 4.7: A crafted Uber JAR containing vulnerable libraries with the UI showcasing this information.

Chapter 5

Evaluation

This chapter presents the evaluation results of the three research questions introduced in chapter 1. Each section is dedicated to a research question and follows the following structure: evaluation objectives, process methodology, and analysis of the results.

5.1 RQ1: How efficient is JarSift to detect embedded dependencies?

This research question aims to evaluate JarSift’s effectiveness in detecting embedded dependencies in Uber JARs. This evaluation has been performed in a controlled environment to ensure the validity and explainability of the results. JarSift results and limitations are then discussed.

5.1.1 Methodology

The methodology used to evaluate JarSift’s effectiveness was designed to mimic real-world conditions encountered in the Maven ecosystem as closely as possible. For this purpose, we created a controlled environment containing a set of projects with known dependencies, compiled using various shading options to reflect different real-world scenarios. It allows us to perform the evaluation of our methodology, and specifically of JarSift, based on the Uber JAR composition ground truth that we collect.

The complete high-level methodology for JarSift is divided into 3 main steps: creating the dataset of evaluation, executing JarSift on the dataset, and analyzing the results.

Creating the ground truth dataset

The primary goal of creating this ground truth dataset is to have a base of the results that JarSift is expected to detect in the Uber JAR dataset. This way, we can compare how effective JarSift is in detecting the contents of a given Uber JAR under different shading configuration parameters, knowing the embedded libraries beforehand.

To create the ground truth dataset, we define two parameters: m , the maximum number of libraries to be embedded within an Uber JAR, and n , representing the number of library

sets for each plugin configuration. These parameters were selected based on an analysis that showed that these settings provide a comprehensive overview of the typical use cases encountered by developers. Specifically, with m set to 5 and n equal to 200, the process will generate 200 “pom.xml” files, each containing 1 to 5 dependencies. For each of these “pom.xml” files, we mutate the Maven Shade Plugin configuration to establish the effects of Relocation and Minimize JAR settings, both individually and in combination. This step quadruples the count, resulting in a total of 800 “pom.xml” files, each representing a unique configuration scenario.

Given the example input parameters, the output of this procedure is a set of 800 Uber JARs, each created to also contain metadata detailing the embedded content as per its respective configuration. These serve as the basis for the ground truth against which JarSift’s detection accuracy is evaluated.

To ensure proper dependency linkage in the resulting Uber JARs, our ground truth dataset creation process requires classes that can be instantiated. These instantiable classes are needed to generate dummy Java files that simulate real-world applications. We start by randomly selecting a number of libraries, from one to m , from our corpus. For each selected library, we identify and extract the name of a class that can be instantiated.

The instantiable class extraction step is also necessary because, during the Maven Shade Plugin “package” phase, a class must be instantiated for it to be considered a part of the final Uber JAR. If none of the selected libraries contains instantiable classes, the library is disregarded. In the next step, for each initial “pom.xml” file, we configure the Maven Shade Plugin to either activate or deactivate the Relocation and Minimize JAR configuration parameters. This results in four distinct “pom.xml” files per set, each corresponding to one of the configuration permutations.

We use the Maven wrapper’s command, `mvn dependency:list`, to collect the effective versions of all direct and transitive dependencies for project compilation and execution. This information is used as the ground truth for the evaluation. The final step involves executing `mvn package` to package each of the 800 “pom.xml” files into its own Uber JAR, ready for the following evaluation phase. However, we verify whether any class files from the effective libraries are detected within the Uber JAR. In some cases, no class files from the dependencies end up in the Uber JAR, which can occur even when “Minimize JAR” option is not utilized. This absence of class files can happen for several reasons: some dependencies may only contain test artifacts, documentation artifacts, or other non-class file components, thus contributing no executable code to the Uber JAR. In scenarios where the Uber JAR lacks any class files from its dependencies, such Uber JARs are marked as invalid. These invalid Uber JARs are excluded from further analysis and ignored during the evaluation phase.

Computing the evaluation metrics

Following the creation of the ground truth dataset, the next step is the evaluation of JarSift’s performance in detecting embedded libraries within Uber JARs. We describe below the steps required to compute standard evaluation metrics, including the F1-score, recall, and precision, which indicate the JarSift’s effectiveness.

5.1. RQ1: How efficient is JarSift to detect embedded dependencies?

JarSift supports a configurable parameter that we call the inclusion threshold, which describes the minimum proportion of a library’s classes that must be present for JarSift to recognize the library as included within the Uber JAR.

The inputs for this evaluation phase are the n Uber JARs created in the previous stage, each associated with one of four distinct configuration scenarios and their respective meta-data that describes the expected contents. The output of JarSift represents the list of libraries that it detects in each Uber JAR.

The steps we take to evaluate the effectiveness of our proposed tool are:

1. Each Uber JAR, generated according to the methodology described earlier, is processed using JarSift to infer its contents.
2. A range of inclusion threshold levels, from 0.50 to 1.00, are used to observe the effects of varying the requirements of class presence in the detection process.
3. For each threshold level and configuration scenario, we compute a confusion matrix and other relevant evaluation metrics.

The confusion matrix cases can be described in the following way, with the addition of a special case that covers the specificity of our signatures corpus:

Table 5.1: Description of Confusion Matrix Cases

Case	Description
True Positive	A library that is known to be embedded in a Uber JAR by the oracle and is correctly detected by our tool.
True Negative	A library that is known to not be in the Uber JAR and is not detected in the Uber JAR by our tool.
False Positive	A library that is not known beforehand to be contained in the Uber JAR but is detected to be embedded in a given Uber JAR.
False Negative	A library is known to be embedded in a Uber JAR by the Oracle but not detected by our tool.
Unknown	A library that is known to be embedded in a Uber JAR but has not been detected due to our signature corpus missing its signatures.

Although many methods for creating Uber JARs exist, we have opted to omit them from our analysis due to their current lack of use or lack of ongoing maintenance, rendering them irrelevant to the focus of our study. Among these methods are techniques like *One-JAR*¹ and the *Jar Task* from the Ant automation tool², which, despite being available, are not commonly used in current practice. Additionally, manual methods of creating Uber JARs, such as using archive editors or the `jar` command-line utility, can yield artifacts that are less predictable and potentially inconsistent. Given these factors, we have opted to omit them

¹<https://one-jar.sourceforge.net/>

²<https://ant.apache.org/manual/Tasks/jar.html>

from our analysis, and we will focus on more widely used and consistently maintained methods that align with the current trends and practices within the Maven ecosystem.

5.1.2 Dataset

The dataset for this research question comprises 12,316,987.00 versioned JAR packages from 445,178.00 projects in the Maven Central repository, which we used to populate the signature corpus. We collected packages and their transitive dependencies released between 2004 and January 2024. The transitive dependencies include artifacts from other repositories and from outside of the specified time range, with some artifacts created before 2004.

The artifact retrieval and collection pipelines are beyond the scope of this research project. However, it was achieved using the Maven Explorer project³, which aims to clone the complete Maven Central repository.

5.1.3 Results

Table 5.2 presents the effectiveness of JarSift to detect embedded dependencies within Uber JARs. The table is divided and presents the results based on the different Uber JAR creation configuration parameters as well as the results with the different inclusion threshold levels. Highlighted in bold are the best-performing results. The evaluation mainly included two settings, the “Minimize JAR” and “Relocation” parameters, significantly influencing how dependencies are packaged and potentially obfuscated in Uber JARs. The presented metrics for each scenario include precision, recall, and the F1 score, providing a complete overview of JarSift’s performance across different conditions.

As expected, the precision and recall are influenced by the configuration of the Uber JAR creation process. The key factors influencing performance are the settings for “Minimize JAR”, “Relocation”, and the variable threshold parameter, which significantly impact how dependencies are packaged, potentially obfuscated within Uber JARs, and respectively detected by JarSift.

It is evident that the threshold setting has a significant effect on dependency detection; specifically, a higher threshold enhances precision but at the expense of recall, pointing towards a clear trade-off between these metrics. At a threshold of 1.00, for instance, while the precision peaks, it is compensated by a decrease in recall, particularly when the “Minimize JAR” and “Relocation” settings are both enabled.

Minimize JAR and Relocation Enabled

With Minimize JAR and relocation enabled, JarSift demonstrates an improvement in precision as the threshold increases, reaching its peak precision of 0.850 at a threshold of 1.00. However, this precision is eclipsed by a low recall value, indicating a trade-off between the tool’s accuracy in identifying true positives and its ability to capture all relevant embedded libraries. Based on our investigation, this is caused by the Minimize JAR configuration

³<https://github.com/cops-lab/maven-explorer/>

5.1. RQ1: How efficient is JarSift to detect embedded dependencies?

Table 5.2: JarSift evaluation results.

Configuration	Threshold	Precision	Recall	F1 Score
Minimize JAR: disabled, Relocation: disabled	0.50	0.312	0.951	0.451
	0.75	0.453	0.950	0.597
	0.90	0.608	0.950	0.727
	0.95	0.684	0.949	0.782
	0.99	0.777	0.948	0.839
	1.00	0.809	0.946	0.857
Minimize JAR: disabled, Relocation: enabled	0.50	0.312	0.950	0.451
	0.75	0.453	0.949	0.596
	0.90	0.608	0.948	0.726
	0.95	0.685	0.947	0.781
	0.99	0.777	0.942	0.837
	1.00	0.808	0.936	0.852
Minimize JAR: enabled, Relocation: disabled	0.50	0.430	0.602	0.337
	0.75	0.564	0.570	0.398
	0.90	0.708	0.554	0.444
	0.95	0.772	0.527	0.453
	0.99	0.834	0.516	0.468
	1.00	0.850	0.515	0.477
Minimize JAR: enabled, Relocation: enabled	0.50	0.430	0.601	0.336
	0.75	0.564	0.569	0.399
	0.90	0.708	0.542	0.444
	0.95	0.772	0.525	0.452
	0.99	0.834	0.512	0.466
	1.00	0.850	0.509	0.474

parameter, which skips all class files in the final Uber JAR that are not used or needed for the Uber JAR to work. Considering the fact that in our Uber JAR generation pipeline, we explicitly use only one class file from a given dependency, the resulting Uber JAR, more often than not, will contain a small percentage of the class files from the original embeddable library. This leads our tool to detect a minimal inclusion ratio for many of the expected embedded libraries, thus preventing them from passing the threshold value.

Minimize JAR Enabled with Relocation Disabled

The results of this configuration are almost identical to the ones from the configuration with both Minimize JAR and Relocation Enabled. Most probably, the minimal improvement in results stems from one of our tool's limitations, designed to support name extraction irrelevant to their package name. However, these heuristics were engineered based on the Java bytecode features and combinations that we encountered during our design and implemen-

5. EVALUATION

tation phase. Although we attempted to cover as many as possible, given the variety of the JVM ecosystem, we could not anticipate all the possible bytecode situations that would require extending our heuristics. The slight improvement in the evaluation results for this configuration stems from the lack of relocation, allowing our tool to almost always correctly extract class names and feature names.

Minimize JAR Disabled with Relocation Enabled

Disabling the Minimize JAR setting significantly impacts the tool’s recall, especially at lower thresholds, indicating that the presence of additional, non-minimized library files in the Uber JAR increases the chances of detecting embedded libraries. Notably, the recall at a 0.50 threshold under these conditions is exceptionally high (over 0.950), pointing at JarSift’s robustness in non-minimized Uber JARs.

Minimize JAR and Relocation Disabled

Disabling both the Minimize JAR setting and not using any relocations yield the best evaluation results for JarSift. This configuration enables our tool to achieve the best precision and recall scores, indicating the ease with which it identifies embedded libraries when the creation of an Uber JAR involves no complicated manipulations and obfuscation techniques. Given the robustness of JarSift irrelevant of relocations, the results are not significantly better than in the case with Minimize JAR disabled and relocation enabled.

Overall

The results underscore a clear influence of JarSift’s effectiveness on the configuration of the Uber JAR creation process, particularly the “Minimize JAR” and “Relocation” options. Higher thresholds tend to favor precision, while lower thresholds benefit recall, highlighting the trade-off between these metrics. This trade-off suggests that optimal use of JarSift requires balancing these metrics based on the specific needs and tolerance for false positives or negatives of the analysis task at hand. Moreover, once the minimization removes classes, JarSift has less information to identify the classes, and therefore, the recall is dropping. The relocation has minimal effect on most metrics since our tool is resilient to such changes. Furthermore, as revealed in the following research question, only 0.22% of all POM files equivalent to 20,485.00 artifacts use the “Minimize JAR” option. This highlights that most artifacts encountered in the ecosystem will not be affected by a low detection accuracy, given their rarity.

In contexts where the objective is to prioritize comprehensive detection coverage, potentially at the expense of encountering false positives, the adoption of a lower threshold is recommended. This strategy would facilitate a more inclusive approach, capturing a broader range of libraries, including those that may require manual verification to rule out inaccuracies. Such an approach is particularly beneficial in scenarios where minimizing the risk of overlooking embedded libraries (false negatives) is essential, and manual oversight is feasible to confirm the findings.

5.1. RQ1: How efficient is JarSift to detect embedded dependencies?

On the other hand, when the analysis requires high confidence in detection accuracy, setting a higher threshold is advisable. This ensures that the tool reports only those embedded libraries it identifies with a high degree of certainty. This approach is suited to situations where the primary concern is to avoid wasting resources on validating false positives, prioritizing precision over an exhaustive detection result.

Several factors contribute to JarSift not achieving 100% precision, with a significant factor being the structure of some artifacts on Maven Central. These artifacts, the so-called “all” identifier, contain a variety of libraries under the same namespace. For example, let us consider a hypothetical “variety-all” package under “com.example”, containing class files from “variety-sql”, “variety-xml”, “variety-yaml”, and “variety-json” among many others. If such a composite library is selected as a dependency in our Uber JAR generation pipeline, our ground truth will contain only the composite dependency, as opposed to its submodules or subdependencies of which it is actually made, especially if these are not explicitly or correctly declared in its “pom.xml” file. As a consequence, JarSift might identify these individual libraries as separate entities rather than detecting them under “variety-all”.

Moreover, our corpus dataset intentionally omits Uber JARs, focusing instead on detecting the contents within these Uber JARs. We chose this approach to ensure the granularity of our analysis, enabling us to detect the composed structure of Uber JARs and identify the specific libraries these contain. This methodology makes sure that we do not just detect the presence of an Uber JAR but rather into its contents, which provides a more detailed and accurate embedded library detection mechanism.

A case of this scenario can be found with the “groovy-all” library within the `org.apache.groovy` group ID. Despite the Uber JAR artifact containing numerous child artifacts from the same namespace, as shown in fig. 5.1, JarSift detects each artifact separately instead of identifying “groovy-all” as one entity. The pie chart in fig. 5.1 depicts the composition of the “groovy-all” Uber JAR, showing that it is mainly composed by the “groovy” core library. However, a substantial “other” category encompasses 48.6%, reflecting the complexity of such Uber JARs on Maven Central.

Despite achieving an F1 score of around 0.850 with Minimize JAR disabled and Relocation either enabled or disabled, this study acknowledges certain limitations. The evaluation focuses on a controlled set of configurations. It does not account for all possible Uber JAR creation methods, such as manual packaging or the use of alternative tools and configuration parameters. Moreover, the specific challenge of detecting libraries with missing signatures (Unknown cases in our confusion matrix) highlights an area for future improvement in signature corpus completeness.

Moreover, this study does not enforce a deterministic inclusion ratio when using “Minimize JAR”, which leads to the number of class files embedded in the artificially created Uber JARs to vary. The study could extend to deterministically ensure that an exact number of class files enter the final Uber JAR.

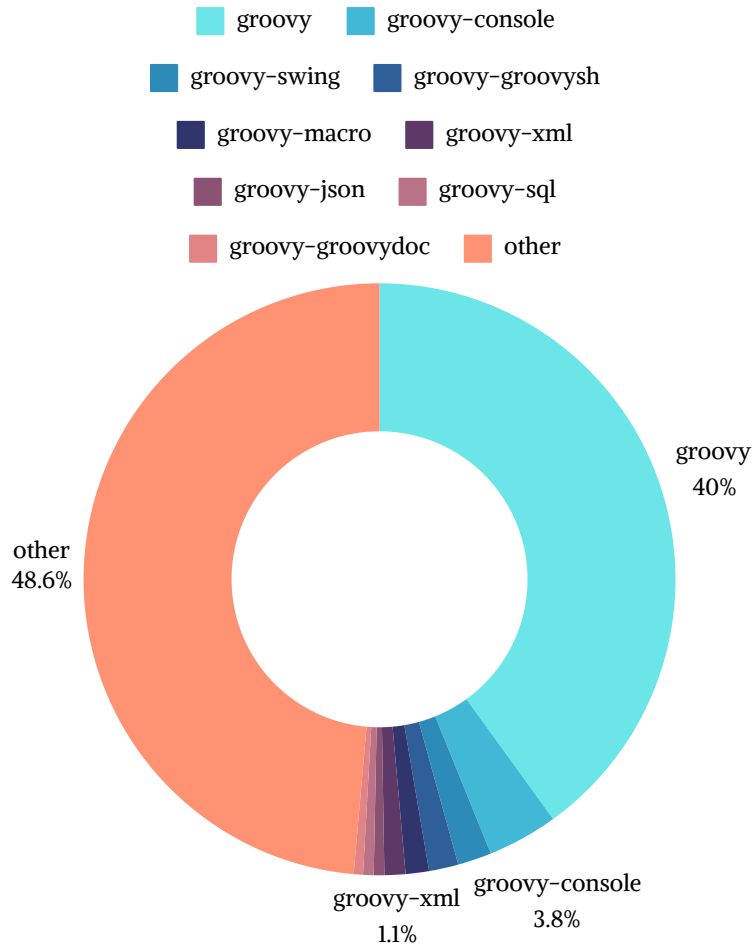


Figure 5.1: Distribution of component artifacts within the “groovy-all” Uber JAR.

Answer to RQ1. How efficient is JarSift to detect embedded dependencies? The effectiveness of JarSift in detecting embedded dependencies has been evaluated using precision, recall, and the F1 score. The F1 score varies between 0.474 and 0.857 depending on the inclusion ratio threshold and Uber JAR generation configuration parameters. We determined that a threshold of 1.00 reaches the best balance between minimizing false positives and maximizing true positive detection, resulting in the best F1 score of 0.857. Overall, we consider that the precision, recall, and F1 scores indicate that JarSift performs reliably for this task. Future potential improvements are presented in the discussion section.

5.2 RQ2: How many Uber JARs are present in our dataset?

Now that we can detect embedded dependencies in JAR files. We want to study the predominance of Uber JARs inside the Maven Ecosystem. The purpose is to understand if deeper

considerations need to be made to manage those specific JARs.

5.2.1 Methodology

To address this research question, our methodology involves the following steps: packaging analysis - we begin by performing a packaging analysis to identify the most commonly used Uber JAR packaging plugins within our dataset. This step enables us to categorize Uber JARs based on the techniques used for their creation, helping us to have a more precise analysis of these files. We specifically analyze the JAR manipulation techniques used during the Uber JAR creation process to understand how these files are changed. This includes examining methods such as embedding libraries and shading, which are essential to understand how Uber JARs are structured; trend analysis - lastly, we look at the adoption trends of Uber JARs within the dataset. This includes analyzing the data over time to determine whether the usage of Uber JARs is increasing, stable, or declining, which is important for predicting future patterns and for various considerations to manage Uber JARs effectively in the future.

The “maven-shade-plugin” and “maven-assembly-plugin” are specifically focused on because they are the most prevalent tools used for creating Uber JARs, as identified by our preliminary dataset analysis. We analyzed all the POM files present in our dataset and extracted the presence of two plugins, the “maven-shade-plugin” and the “maven-assembly-plugin”. Moreover, we extracted the configuration parameters of these two to study the prevalence of parameters and possible configurations encountered on Maven Central and other repositories storing the frequency and contexts of their usage. For POM files with parent POM files, we try to resolve those parameters and apply them to children’s POM files. Still, we take into account the fact that children’s configuration parameters override those of their parents. This data collection pipeline, combined with our MariaDB persistence layer, allows us to infer common usage patterns and trends when it comes to the aforementioned plugins.

5.2.2 Dataset

The dataset for this research question consists of over 9 million POM files. These files were collected using the same methodology as the JAR files in the previous research question, using the Maven Explorer project⁴. Like the JAR files, the POM files are attached to the artifacts from Maven Central, which were included in the earlier research corpus. However, the release dates for many artifacts are unreliable, both in the Maven Central Index database and in the metadata of these POM files. To ensure that the release date we have for each artifact is accurate, during the corpus creation, JarSift also extracts the creation time of either the archive or the META-INF folder within each JAR file during corpus creation and stores this information in the database. The creation date is then used when our analysis scripts cannot reliably determine the release dates of artifacts.

⁴<https://github.com/cops-lab/maven-explorer/>

5.2.3 Results

In addressing our second research question, we performed an in-depth analysis to determine the prevalence of Uber JARs within the Maven ecosystem. By examining over 9 million POM files, our investigation aimed to identify the use of key plugins - “maven-shade-plugin” and “maven-assembly-plugin” - which play an important role in creating Uber JARs. This analysis provides insights into the usage patterns of these plugins and the configurations and techniques most frequently employed for JAR file manipulations. We present the findings of our study in table 5.3.

Table 5.3: Overview of POM files analysis. The percentages in parentheses indicate the proportion of total POM files; percentages in brackets represent the proportion of “maven-shade-plugin” occurrences, directly or with inheritance, respectively.

Metric	Direct inclusion	With inheritance
Total POM Files Analyzed	9,232,456	-
POM files using the Maven Assembly Plugin	307,454 (3.33%)	601,414 (6.51%)
POM files using the Maven Shade Plugin	187,692 (2.03%)	313,675 (3.97%)
POM files with dependency reduced POM	50,782 (0.55%)[27.05% ¹]	69,652 (0.75%)[22.20% ²]
POM files with minimized JAR	20,485 (0.22%)[10.91% ¹]	27,831 (0.30%)[8.87% ²]
POM files with relocations	56,074 (0.60%)[29.87% ¹]	58,216 (0.63%)[18.55% ²]
POM files with filters	51,074 (0.61%)[29.87% ¹]	91,258 (0.98%)[29.09% ²]
POM files with transformers	67,388 (0.72%)[35.90% ¹]	69,654 (0.93%)[27.65% ²]

Figure 5.2 illustrates the usage percentages of different configurations within the shade plugin ecosystem from 2009 to 2023, showing the contribution of each configuration relative to the overall utilization of the shade plugin. The fluctuations and rough transitions are most skewed by the number of packages collected in our dataset.

Overview of POM files analysis

Our dataset, containing more than 9 million POM files, serves as a robust basis for this analysis. The analysis reveals that a significant portion of these files use the “maven-assembly-plugin” and “maven-shade-plugin”, with occurrences found in 3.33% and 2.03% of the files, respectively, when considering direct inclusions. When inheritance is accounted for, these figures rise to 6.51% for the “maven-assembly-plugin” and 3.97% for the “maven-shade-plugin”, underscoring the importance of dependency hierarchies within Maven projects and modules.

Dependency-reduced POMs

The practice of utilizing dependency-reduced POMs while streamlining Uber JARs by eliminating extra dependency information poses a significant challenge for SCA tools that rely on POM metadata to deduce the correct dependencies and embedded libraries within a Uber

¹Percentage of total “maven-shade-plugin” direct occurrences using the specific setting.

²Percentage of total “maven-shade-plugin” occurrences with inheritance using the specific setting.

5.2. RQ2: How many Uber JARs are present in our dataset?

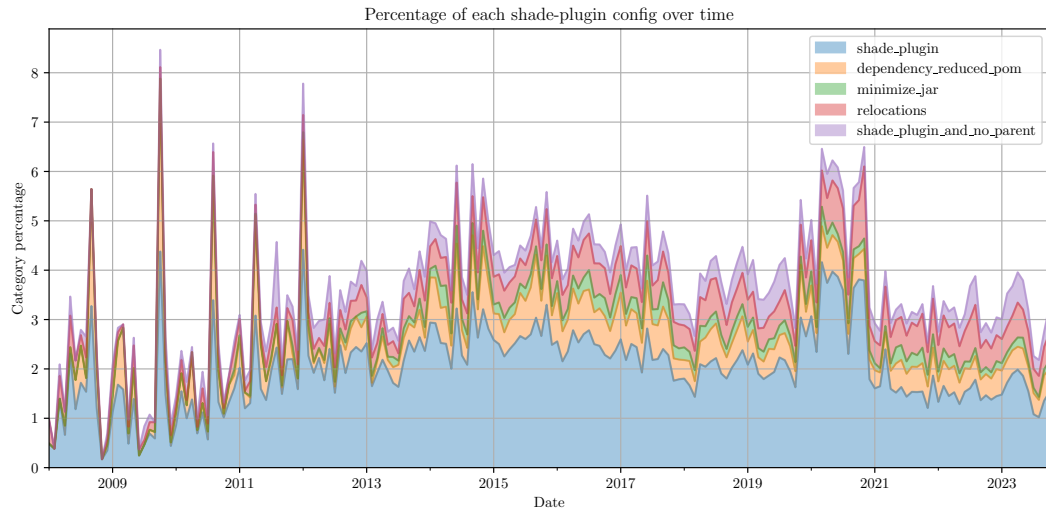


Figure 5.2: Trends in Apache Maven Shade plugin configuration usage over time.

JAR. When this data is absent, such a tool fails to accurately identify the components that constitute the Uber JAR, potentially overlooking embedded libraries or misinterpreting the Uber JAR’s contents.

Minimize JAR impact

Employing the “Minimize JAR” option impacts the detection of class files within Uber JARs, influencing the inference accuracy of SCA tools that do binary analysis, including ours, which operate based on an inclusion threshold ratio. When “Minimize JAR” is applied, the resultant Uber JAR contains a minimal set of class files, directly affecting the tool’s ability to affirm the presence of specific libraries, especially under high inclusion threshold settings. This scenario can lead to underreporting embedded libraries, underscoring the need for careful threshold management.

Effects of relocations

The application of relocations, or shading, entails the renaming and modification of Fully Qualified Names (FQNs), which directly challenges SCA and binary analysis tools that depend on package names and paths to infer Uber JAR contents. However, despite these changes, JarSift is designed to be resilient against such alterations and obfuscations, maintaining its efficacy in detecting embedded libraries.

Implications of filters

The use of filters introduces a high degree of unpredictability into the composition of Uber JARs. Filters can selectively include or exclude a wide range of elements - from entire class paths and specific class files to metadata and license files - rendering the final Uber JAR

a “wildcard” in terms of its contents. This variability significantly complicates the task of accurately analyzing or predicting the components of a Uber JAR, presenting a substantial challenge for both manual inspection and automated tools.

Role of transformers with “maven-shade-plugin”

Transformers play an important role when used in conjunction with the “maven-shade-plugin”, offering a mechanism to manage resource overlaps and merge content from multiple artifacts into a single Uber JAR. They provide a suite of functionalities, from preventing license duplication to aggregating components and merging resources, which are crucial for creating functional Uber JARs. The diverse range of transformers available addresses specific needs, such as merging components, managing MANIFEST entries, and relocating class names in “META-INF/services” resources, among others.

These aspects underscore the complex nature of Uber JAR creation and manipulation within the Maven ecosystem, highlighting the challenges posed to SCA tools and the necessity for advanced tools like JarSift that can overcome these complexities.

Answer to RQ2. How many Uber JARs are present in our dataset? Based on the assumption that all artifacts created with the Maven Assembly and Maven Shade Plugins are Uber JARs, they constitute 6.51% and 3.97% of the JARs in our dataset. These percentages are significant given the total number of POM files, which stands at 9,232,456. This highlights the importance of an in-depth analysis of Uber JARs within the Maven Central ecosystem and their implications. Approximately one-fifth of all the POM files using the Maven Shade Plugin also use configurations like the dependency-reduced POM, minimized JAR, and relocations parameter. These configurations suggest that most Software Composition Analysis (SCA) tools may struggle to effectively and accurately detect the contents of the resulting JAR files. Recommendations for addressing the impact of these complications are discussed in subsequent sections.

5.3 RQ3: How many Uber JARs on Maven have known undetected vulnerabilities?

We have shown in the previous research question that Uber JARs are relatively frequent in the Maven ecosystem. As discussed in the background section, Uber JARs introduce some security considerations, particularly if the embedded dependencies are vulnerable. JarSift allows scanning the JARs in the Maven ecosystem to detect the embedded libraries and, as such, can also see which ones are known to be vulnerable. Moreover, given the fact that we have the signatures of a significant number of libraries, we can also see if the vulnerable libraries are present in other libraries that are not known to be vulnerable by checking if they share the same signatures.

The goal of this research question is to establish how many existing artifacts in our subset of Maven Central contain known vulnerable code. By identifying the prevalence of vulnerabilities within Uber JARs, this research hopes to highlight potential security risks

5.3. RQ3: How many Uber JARs on Maven have known undetected vulnerabilities?

and dormant, unknown attack vectors that could impact a variety of applications depending on these libraries.

This section of our research project is divided into two main parts. The first part focuses on collecting existing (Uber) JARs from the Maven Central repository and scanning them for vulnerabilities. The second part broadens the scope by compiling a list of artifacts in our corpus known to be vulnerable and then determining if these can be found in other libraries within our corpus that are identified as non-vulnerable.

5.3.1 Scanning JARs for vulnerabilities

Methodology

In this part, we collect a set of the most used Maven Central libraries to see whether these contain undetected vulnerabilities. We have targeted probable Uber JARs that contain terms such as “shaded”, “uber”, “fat”, or “all” in their group ID or artifact identifier. Based on our preliminary analysis of the ecosystem, these terms are the ones that are commonly used when developers publish Uber JARs on the Maven Central repository. For each identified Uber JAR, we do the following: First, we use JarSift to infer and compile a list of embedded libraries within each Uber JAR. Then, we examine each of these embedded libraries for vulnerabilities, leveraging a reliable third-party API. If we detect any vulnerabilities in the embedded libraries, the parent Uber JAR is categorized as containing vulnerable code.

Dataset

We have collected the top 100 most used libraries on Maven Central, both the most used known vulnerable version and the most used non-vulnerable version. Moreover, we collected every artifact on Maven Central that has the “shaded”, “uber”, and “fat” in their name, mainly in their artifact ID. We ended up collecting over 1755 artifacts that match these conditions.

Results

After running the detection algorithm on them, we found 337 (17.13%) of them have unknown vulnerabilities. This significant proportion underscores a notable security concern within the Maven ecosystem, highlighting the presence of vulnerabilities even in widely used libraries.

Our discovery shows that a substantial percentage of Uber JARs contain embedded libraries with known vulnerabilities, which points to a critical challenge in dependency management and security within the Maven ecosystem. This situation is aggravated by the practice of shading, which, while offering benefits in namespace management and conflict resolution, may obscure the provenance of embedded libraries, complicating vulnerability detection and remediation efforts.

Our findings raise important considerations for the development and maintenance of Uber JARs. We argue that the presence of vulnerabilities within embedded libraries of Uber JARs poses a significant risk, potentially exposing applications to exploits that compromise

their integrity and the data they manage. The risk is increased by the fact that vulnerabilities within Uber JARs can remain undetected or unaddressed due to the obfuscation of their embedded dependencies. Authors of such Uber JARs may not even be aware of vulnerabilities within their Java artifacts due to the lack of proper communication channels and lack of proper tooling for mitigating such issues. Moreover, the use of shading in Uber JARs, while offering technical advantages, introduces challenges in accurately assessing and tracking the security of these artifacts.

The initial phase of our investigation into the vulnerability ecosystem of Uber JARs on Maven Central reveals a concerning prevalence of known vulnerabilities within these artifacts. This insight not only highlights the security challenges associated with the use of Uber JARs but also emphasizes the need for careful management and remediation of vulnerabilities within the Maven ecosystem. As we proceed to the second part of this research question, our focus will broaden to explore the implications of these findings on the wider corpus of libraries.

5.3.2 Scanning the corpus for vulnerabilities

Methodology

The goal of scanning the corpus for vulnerabilities was to determine how many artifacts in our dataset contain signatures identical to those of known vulnerable artifacts in our signature corpus. The inputs for this stage are the signatures of known vulnerable artifacts, which we filtered based on their presence in our corpus. As output, we provide information about the non-vulnerable artifacts, whose signatures are in the corpus and contain 100% of the signatures of vulnerable artifacts.

Our approach in this segment is based on a three-step process:

1. **Identification of vulnerable artifacts:** The initial step involves selecting artifacts from the Maven ecosystem known to contain vulnerabilities. This selection is based on a comprehensive vulnerability database that contains data from reputable sources, ensuring a broad and accurate representation of known vulnerabilities.
2. **Corpus dataset filtering:** Following the identification of known vulnerable artifacts, we refine our focus to those present within our corpus dataset. This filtration step is needed to limit our analysis to the artifacts most relevant to our research scope.
3. **Signature analysis:** The final analytical step includes examining the signatures of these filtered libraries to identify other libraries within our corpus that contain identical signatures, excluding those with matching group IDs and artifact IDs to avoid different versions of the same input vulnerable library.

Dataset

In this part, other than the signature corpus, we made use of the Open Source Vulnerabilities database, which collects vulnerability data from a multitude of sources, such as the GitHub Advisory Database, the MITRE CVE database, and others. As of October 2023,

5.3. RQ3: How many Uber JARs on Maven have known undetected vulnerabilities?

this database contained 3,935.00 unique known vulnerabilities for the Maven ecosystem. These vulnerabilities cover 1,838.00 unique Maven artifacts, containing a total of 248,407.00 vulnerable versions. We filtered this list to retain only artifacts that are contained in our corpus database, which brought the total count to 189,777.00 unique, vulnerable versions, accounting for 76.39% of all known vulnerable libraries.

The next step was to collect all the signatures of these libraries and see which other libraries contained them. Here, it was important to ignore libraries with the same group ID and artifact ID since it was likely that different versions of the same library would contain the same signatures. However, in some cases, as mentioned in one of the previous sections, there are no significant or discernable bytecode changes between different versions of the same artifacts, which may lead to multiple versions of the same artifact containing vulnerable bytecode of other artifacts.

Results

In our analysis, we found that 31,763.00 libraries in our corpus, or 0.58% of the total, contain signatures matching those of known vulnerable libraries. It is important to note that this figure represents a full match with the signatures of 76.39% of the known vulnerable artifacts that we have identified in our dataset, which totals 5,443,022.00 JAR artifacts and their signatures. The libraries most affected include popular ones such as “org.neo4j:neo4j”, “com.google.protobuf:protobuf-java”, “io.grpc:grpc-protobuf”, “junit:junit”, “io.netty:netty-codec-http”, “com.amazon:aws-java-sdk-s3”, and others, indicating a significant impact across various widely used libraries.

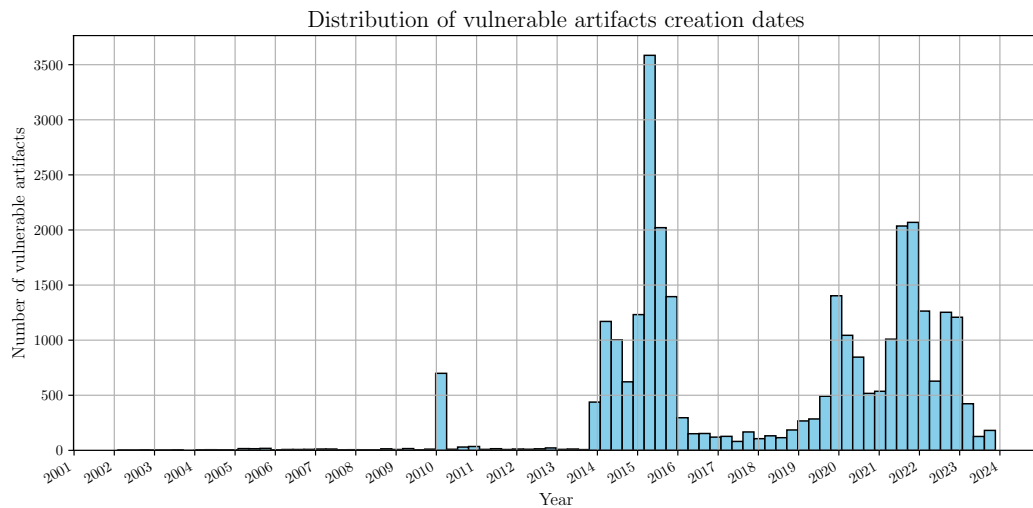


Figure 5.3: Distribution of non-reported vulnerable artifacts by release date, highlighting the prevalence of recent vulnerabilities.

The histogram in fig. 5.3 illustrates the number of non-reported artifacts that contain vulnerable code based on their release date on Maven Central. The concentration of vul-

5. EVALUATION

nerabilities in newer artifacts may suggest that the integration of vulnerable libraries into software projects remains a challenge. The trend points to the importance of incorporating a tool such as JarSift in deployment pipelines to detect these vulnerable artifacts early on.

Our findings reveal a significant challenge within the Maven ecosystem: the widespread presence of known vulnerabilities across numerous Uber JARs. Most of those Uber JARs are not flagged by existing tools; therefore, users of those libraries are unaware that they are using vulnerable software. Future work based on this part of our research should consider varying the inclusion threshold for vulnerable libraries rather than their full inclusion.

This issue underlines the urgent need for better tools to detect vulnerabilities, such as JarSift, which can identify problematic signatures throughout the Maven ecosystem more accurately. These tools should go beyond conventional dependency analysis by considering the deeper, transitive relationships between artifacts. The presence of vulnerabilities across many libraries demonstrates the importance of new approaches that are able to manage and fix those Uber JARs, ensuring the security of artifacts in the Maven ecosystem.

Answer to RQ3. How many Uber JARs on Maven have known undetected vulnerabilities? Our study reveals a notable security concern: approximately 17.13% of the small dataset of probable Uber JARs we analyzed contain undisclosed vulnerabilities. Additionally, our analysis of the signature corpus identified 31,763 (0.63%) libraries that fully contain all the signatures of known vulnerable libraries. These findings point to a significant and concerning issue with the prevalence of vulnerabilities within Uber JARs, indicating problems in managing and detecting secure dependencies in the Maven ecosystem. In the following section, we discuss recommendations for addressing these vulnerabilities and enhancing security practices within Maven.

Chapter 6

Discussion

6.1 Significance of JarSift

The introduction of JarSift is a significant advancement in overcoming the limitation of existing Software Composition Analysis (SCA) tools, particularly in their ability to accurately detect the contents and existing vulnerabilities within Uber JARs. Traditional SCA tools frequently fail when faced with obfuscation techniques such as shading or metadata removal, which obscure the true contents of Uber JARs. JarSift’s approach, based on analyzing bytecode signatures, offers a robust solution to this challenge. By tackling these obfuscation layers accordingly, JarSift provides an accurate assessment of Uber JAR contents, significantly improving the detection of embedded vulnerabilities. This capability is crucial for improving software deployment processes, ensuring that developers and security analysts have a clear understanding of the security state of the components they integrate and deploy in their projects.

We argue that a potential and probable reason that Uber JARs are not properly managed, supported, and detected by most existing SCA tools is the lack of academic and industry focus on the predominance of Uber JARs in the ecosystem. Consequently, their importance has never been considered in developing features and tools supporting them. Moreover, our analysis reveals that Uber JARs are quite common, suggesting that the findings of this study could serve as a motivation to support the development of improved methods for the analysis and detection of embedded dependencies.

6.2 Prevalence and risks of Uber JARs

Our investigation of the Maven Central Repository has shown an increasing trend in the use of Uber JARs, raising potential security concerns due to the lack of transparency regarding their contents. The findings reveal that many of these artifacts contain embedded libraries with known vulnerabilities, creating potentially hidden attack vectors. This situation is worsened when Uber JARs do not use shading and conflict with a user’s primary dependencies. In such cases, vulnerabilities persist even if the user declares a more up-to-date dependency as a requirement, leading to security risks. The discovery of a significant

level of duplication within the Maven ecosystem further complicates this issue. Numerous instances of libraries, unique by group ID and artifact ID, show no substantive changes across versions. Furthermore, the presence of libraries that are essentially clones of popular libraries, with only minor modifications such as group ID and artifact ID changes, complicates dependency management. Such practices not only introduce unnecessary complexity but also complicate the effective management of dependencies, particularly for large-scale projects with complex dependency relations.

6.3 Report detected as new CVE?

During our study phase, we attempted to contact CVE Name Authorities (CNA) to obtain more details about the correct procedure for reporting vulnerabilities when an existing artifact contains vulnerable code directly rather than depending on it via a resolution mechanism such as Maven. Unfortunately, none of our inquiries were responded to, leading to a dead end. According to the official CNA rules ¹, if a CVE is assigned to a software artifact, the CNA must notify the maintainer of that code about the vulnerability, provided that the vulnerable embedded dependency has not been modified. If it has been modified, a new vulnerability must be reported. This raises a question: Is shading considered a modification? Shading can be described as simply reusing the same library without altering its functionality or code. Thus, under some circumstances, shaded libraries can be treated as unmodified, complicating the reporting and management of vulnerabilities related to them.

JarSift would be the perfect tool to identify vulnerabilities contained in Uber JARs, which, paired with a manual check, could be a straightforward mechanism for collecting evidence and relevant information about unknown vulnerabilities detected in existing software artifacts.

6.4 Future research directions

Considering the discoveries and challenges identified through our study, multiple directions for future research have become apparent:

Dynamic analysis integration

Extending JarSift's static analysis capabilities with dynamic analysis techniques could offer a more comprehensive understanding of vulnerabilities within Uber JARs. Specifically, this extension would determine whether vulnerable code paths are actively utilized, providing a deeper evaluation of the true security risks involved.

Broadening the research scope

Expanding the scope to include other JVM languages and software ecosystems beyond Maven Central could yield valuable insights into the challenges and practices associated

¹<http://cve.mitre.org/cve/cna/rules.html>

with Uber JARs in various development contexts. Moreover, our signature extraction methodology could be used for other compiled languages for binary fingerprinting and matching.

Corpus expansion

Extending the corpus with a broader selection of artifacts from Maven Central and other repositories such as Atlassian² and Hortonworks³ would enhance the applicability of our study's findings. By doing so, JarSift would be able to detect libraries that are exclusively released on those repositories.

Refinement of false positive heuristics

Improving the heuristics for filtering false positives would increase JarSift's precision, making it an indispensable tool for developers and security analysts. Enhancements could include better detection of packages that contain many artifacts by collecting their signatures, as well as a smarter evaluation of whether two different package names are related or originate from distinct packages. Considering how important minimizing false positives is when it comes to such sensitive aspects of software development and deployment.

Code duplication and license misuse detection

The core functionality provided by JarSift allows finding code duplication among JAR files. The extensive scale of this study allows for a comprehensive code duplication analysis, providing deeper insights into the ecosystem's condition. By applying appropriate heuristics and matching algorithms to the corpus database, JarSift could accurately identify both potentially insecure bytecode snippets and instances of code duplication.

The same principle can be utilized to detect misuse of existing open-source code in other projects, ensuring compliance with licensing terms and avoiding license mix-ups.

CI/CD integration

Integrating JarSift within Continuous Integration/Continuous Deployment (CI/CD) pipelines could significantly enhance the security of software delivery processes. Connecting JarSift's analysis capabilities directly into these pipelines can detect and address vulnerabilities early in the development process, minimizing the risk of deploying compromised software. JarSift could continuously scan all downloaded JARs within the development pipeline, flagging dependencies and notifying the relevant parties about issues.

Future research could explore further optimizations of JarSift's heuristics and analysis pipeline to minimize the impact on deployment speed while maximizing true positives and accuracy, thus facilitating a quicker and fitting resolution of security-related issues.

²<https://packages.atlassian.com/mvn/maven-external/>

³<https://repo.hortonworks.com/content/repositories/releases/>

Automatic scan for new libraries and new CVEs

Extending JarSift to support a continuously automatic extension of the signatures database and new Common Vulnerabilities and Exposures (CVEs) would be a great addition to the toolset of a developer or DevOps Engineer. This would provide instant alerts to developers about the need to update or patch their dependencies in case they are using Uber JARs or similar deployment methods. This could be especially important for projects where dependencies might not be frequently reviewed or scrutinized within a timely manner, especially for long-term support projects.

SBOM based on JarSift

Given the functionality that JarSift provides, it could easily be adapted into a tool that creates Software Bill of Materials (SBOM) for Java-based projects. It would only necessitate implementing the functionality that exports the detected libraries in a project or at a given path. This future development relates to the CI/CD pipeline integration because the automatic creation and updating of the SBOM of a project can be set up together with the vulnerability notification system. This would once again optimize the development process and facilitate better visibility and ease of use for engineers and security analysts involved in a project.

Chapter 7

Threats to validity

In this chapter, we discuss potential threats to the validity of our research findings.

7.1 Internal Threats to Validity

7.1.1 False Positives in Signature Detection

The signature used by JarSift for vulnerability detection may introduce false positives. While our results demonstrate that the incidence of false positives is minimal, it is still possible that specific signatures may lead to inaccurate detections. Further refinement and evaluation of the signature set are necessary to minimize false positives and enhance the tool's precision.

7.1.2 Potential JarSift Bugs

Despite our efforts to develop a reliable tool, we cannot completely rule out the possibility of bugs or errors in JarSift. We plan on making the JarSift open-source, allowing other researchers to inspect and verify its implementation. Collaboration and community involvement are encouraged to identify and rectify any potential issues.

7.1.3 Lack of Obfuscation Support

One threat to the validity of our study is the lack of support for obfuscated code. Our tool, JarSift, does not currently handle obfuscation techniques, which may limit the generalisability of our findings to software systems that employ obfuscation. Future work should consider extending the tool's capabilities to address this limitation.

7.1.4 Signature's imperfect entropy

Our signature creation workflow has certain limitations that prevent us from retaining minor changes in every class file. Specifically, our tool fails to accurately capture features of the Java language that are uncommonly used or obscure, resulting in inaccuracies in a

class's Java bytecode. However, we are constantly improving and extending our signature extraction to account for these features.

7.2 External Threats to Validity

7.2.1 Non-determinism in Maven's resolution mechanism

During our experiments, we noticed an inconsistency in Maven's dependency management. Although Maven reports a list of dependencies with their respective versions required to compile a JAR, it sometimes embeds a different version in the final Uber JAR. We had to account for this, and during the evaluation of JarSift, we had to check the end Uber JAR for the real embedded versions.

7.2.2 Code duplication

Based on our data exploration, we noticed that there are hundreds of thousands of duplicated versions in the Maven Central Repository realm. There are two main reasons why this usually occurs. First, some of the releases are related to external changes, such as test classes or configuration files, which lead to a new release but no discernible change in the class files included in a JAR file. Next, for big projects, whenever a new version of the parent package is released, all the subsequent child packages get a new release with the same new version, leading to an enormous amount of duplication on Maven Central. In some cases, there may be dozens of versions in a row with no change in the signature of the JAR file.

This results in our tool's inability to differentiate between versions as it selects the first version in lexicographic sorting. However, we retain alternative versions in our matching sequence. To put it simply, we compare the class file signatures of different versions of a library. This way, we can identify true positives in the evaluation of our tool by marking a result as such if one of the inferred versions matches the expected version. By retaining this information, we can evaluate our tool's performance accurately.

7.2.3 Incomplete Dataset

Our dataset does not encompass the complete Maven Central repository. Although it covers a substantial portion of it and includes the most recent releases, the exclusion of certain projects or versions may introduce limitations to the generalizability of our findings. Future work should consider expanding the dataset to include a more comprehensive representation of software projects.

7.2.4 Uber JAR creation method trends

Given the fact that we are only studying the trends regarding the usage of the `maven-shade-plugin` method for creating Uber JARs, we are most certainly missing other more obscure or less popular methods of creating these. In the past, before the Maven Shade

Plugin became the most commonly used one, manual creation, Eclipse IDE tools, and Ant scripts, among others, were also used to create Uber JARs.

Chapter 8

Conclusion

In this chapter, we reflect on the results and draw conclusions from our research.

This project had the goal of improving the understanding and management of Uber JARs within the Maven ecosystem by developing JarSift, a tool designed to detect embedded libraries and identify potential vulnerabilities. Our investigation has not only underlined the prevalence of Uber JARs but also revealed their often missed vulnerabilities, which pose significant security risks.

The evaluation of JarSift demonstrates its efficiency in detecting embedded dependencies, which is supported by precision, recall, and F1 scores, with the latter ranging from 0.474 to 0.857, depending on the Uber JAR creation configuration. This highlights JarSift's potential as a tool for dependency management and detection within the Maven ecosystem, capable of addressing issues and security risks associated with Uber JARs. Our findings show that Uber JARs are not only prevalent but also frequently contain undetected vulnerabilities, with approximately 17.13% of the Uber JARs in our small dataset of 1,755.00 artifacts containing undisclosed security vulnerabilities.

Furthermore, 6.51% and 3.97% of the JARs created were identified to be using the Maven Assembly and Maven Shade Plugins. Given the total of 9,232,456.00 POM files analyzed, these percentages underline the extensive impact Uber JARs have on dependency management and artifacts released on the Maven Central repository. Approximately one-fifth of the POM files using the Maven Shade Plugin also use configurations such as dependency-reduced POM, minimizing JAR, and relocations, which often hinder the effectiveness of existing Software Composition Analysis (SCA) tools. These configurations are precisely what JarSift is designed to mitigate.

This research underlines the need for improved practices in analyzing Uber JARs to prevent security vulnerabilities and enhance traceability within the Maven ecosystem. We recommend the adoption of tools like JarSift, which can significantly improve dependency management and vulnerability detection in Maven projects by carefully examining the contents of used JARs. It is crucial to implement best practices in managing dependencies and safeguarding against security threats. Furthermore, we encourage ongoing research and the development of tools and methodologies aimed at addressing the challenges posed by Uber JARs.

Finally, the findings and insights gathered from this study not only validate the effec-

8. CONCLUSION

tiveness of JarSift and our methodology but also point towards an expected path for continuous improvement in dependency management and security assessment within the Maven ecosystem.

Bibliography

- [1] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. Sv-af — a security vulnerability analysis framework. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 219–229, 2016. doi: 10.1109/ISSRE.2016.12.
- [2] Binildas Christudas. *Microservices in Depth*, page 35–53. Apress, 2019. ISBN 9781484245019. doi: 10.1007/978-1-4842-4501-9_3. URL http://dx.doi.org/10.1007/978-1-4842-4501-9_3.
- [3] 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, 2022. doi: 10.1109/TSE.2021.3101739.
- [4] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage: Finding the provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11*, page 183–192, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305747. doi: 10.1145/1985441.1985468. URL <https://doi.org/10.1145/1985441.1985468>.
- [5] Jens Dietrich, Shawn Rasheed, Alexander Jordan, and Tim White. On the security blind spots of software composition analysis, 2023.
- [6] Johannes Düsing and Ben Hermann. Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. *Digital Threats: Research and Practice*, 3(4):1–25, 2022.
- [7] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: Finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020*, page 463–474, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397388. URL <https://doi.org/10.1145/3395363.3397388>.

- [8] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. A double-edged sword? software reuse and potential security vulnerabilities. In Xin Peng, Apostolos Ampatzoglou, and Tanmay Bhowmik, editors, *Reuse in the Big Data Era*, pages 187–203, Cham, 2019. Springer International Publishing. ISBN 978-3-030-22888-0.
- [9] Michael W. Godfrey. Understanding software artifact provenance. *Science of Computer Programming*, 97:86–90, 2015. ISSN 0167-6423. doi: <https://doi.org/10.1016/j.scico.2013.11.021>. URL <https://www.sciencedirect.com/science/article/pii/S0167642313003055>. Special Issue on New Ideas and Emerging Results in Understanding Software.
- [10] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 International Workshop on Ecosystem Architectures*, WEA 2013, page 1–5, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323147. doi: 10.1145/2501585.2501586. URL <https://doi.org/10.1145/2501585.2501586>.
- [11] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity, 2019.
- [12] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM ’21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386654. doi: 10.1145/3475716.3475769. URL <https://doi-org.proxy-ub.rug.nl/10.1145/3475716.3475769>.
- [13] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133908. URL <https://doi.org/10.1145/3133908>.
- [14] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security*, 103: 102181, 2021.
- [15] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211. IEEE, 2023.
- [16] Md Rakib Hossain Misu, Rohan Achar, and Cristina V. Lopes. Sourcererjbf: A java build framework for large-scale compilation. *ACM Trans. Softw. Eng. Methodol.*, dec 2023. ISSN 1049-331X. doi: 10.1145/3635710. URL <https://doi.org/10.1145/3635710>. Just Accepted.

-
- [17] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Masci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.
- [18] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE, 2015.
- [19] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 555–558, 2021. doi: 10.1109/ICSME52107.2021.00056.
- [20] Kristiina Rahkema, Dietmar Pfahl, and Rudolf Ramler. Analysis of library dependency networks of package managers used in ios development. In *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, May 2023. doi: 10.1109/mobilsoft59058.2023.00010. URL <http://dx.doi.org/10.1109/MOBILSoft59058.2023.00010>.
- [21] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerccc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE ’16*, page 1157–1168, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884877. URL <https://doi.org/10.1145/2884781.2884877>.
- [22] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers’ build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, page 724–734, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568255. URL <https://doi.org/10.1145/2568225.2568255>.
- [23] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):45, 2021.
- [24] 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*, pages 319–330, 2018.
- [25] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. Will dependency conflicts affect my program’s

- semantics? *IEEE Transactions on Software Engineering*, 48(7):2295–2316, 2022. doi: 10.1109/TSE.2021.3057767.
- [26] Yongzhi Wang, Chengli Xing, Jinan Sun, Shikun Zhang, Sisi Xuanyuan, and Long Zhang. Solving the dependency conflict of java components: A comparative empirical analysis. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 109–114. IEEE, 2020.
- [27] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, page 960–972, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616299. URL <https://doi.org/10.1145/3611643.3616299>.

Appendix A

Nomenclature

In this appendix we give an overview of frequently used terms and abbreviations.

GAV: groupId, artifactId, version

JAR: Java Archive

WAR: Web Application Resource or Web application Archive

CVE: Common Vulnerabilities and Exposures

SCA: Software Composition Analysis

SBOM: Software Bill of Materials

JVM: Java Virtual Machine

POM: Project Object Model

CNA: CVE Naming Authority

FQN: Fully Qualified Name

Appendix B

Signature information

Caption: List of features that are part of the signature. Elements in this list marked with a [] at the end imply that there can be 0 or more elements of this sort. Moreover, elements marked in bold may and typically do contain a fully qualified name.

- major version
- access modifier
- **name**
- **extends type**
- **interfaces[]**
- **fields[]**
 - name
 - **type**
- **methods[]**
 - access modifier
 - name
 - **type**
 - **exceptions[]**
 - **instructions[]**
 - * opcode
 - * **operand**
 - **argument types**
 - **return type**
- **constructors[]**

B. SIGNATURE INFORMATION

- name
 - **type**
 - **exceptions[]**
- inner classes[]
 - name
 - outer name
 - inner name
 - access modifier
 - **type**
- **annotations[]**
 - **annotation arguments[]**
 - * **type**
 - **array arguments[]**
 - **arguments[]**