Uncovering Secrets of the Maven Repository

Maven packaging

Priyam Rungta[1]

Supervisor(s): Sebastian Proksch[1]**, Mehdi Keshani**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Priyam Rungta
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Mehdi Keshani, Soham chakraborty

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Maven, a widely adopted software ecosystem for Java libraries, plays a critical role in the development and deployment of software applications. However, there exists a limited understanding of the composition and characteristics of the Maven repository, leaving users and contributors unaware of the contents they interact with. This research aims to address this knowledge gap by conducting a comprehensive analysis of Maven packaging and informing developers, library maintainers, security analysts, and the open-source community about Maven library practices. The research investigates the secrets of the Maven repository, focusing on Maven packaging. Using data from the POM file, Maven index file, and Maven repository, we analyze the distribution of packaging types, checksums, qualifiers, and file types within Maven libraries. The experiment involves examining 479,915 packages from the Maven repository, utilizing the POM file, the Maven index, the Maven repository and manual requests to the Maven repository. The results reveal that JAR is the packaging type in more than 75% packages across all sources, and inconsistencies are found among different data sources, highlighting the need for improved data consistency and reliability within the Maven ecosystem. Furthermore, the adoption of the sha256 and sha512 checksum algorithms remains limited, with only 1.4% of packages utilizing these secure hash functions. In terms of qualifiers, sources and Javadoc exhibit the highest prevalence, with adoption rates of 82% and 76% respectively. Moreover, class files and XML are identified as the most frequently packaged file types, encompassing 71% and 61% of the packages, respectively among a very diverse classification. These findings provide insights into Maven library characteristics and inform optimization of library usage.

Keywords: Maven, Packaging type, Checksum, Qualifier, Type of files

1 Introduction

Maven is a software project management tool. Based on the concept of a project object model, Maven can manage a project's build, reporting and documentation from a central piece of information [1]. The Maven repository [1] serves as a centralized hub for developers to store, share, and manage projects that are based on the Java Virtual Machine (JVM).

Our research aims to provide valuable insights to the software engineering community by comprehensively examining the features and characteristics of Maven libraries. By focusing on Maven packaging and its surrounding ecosystem, we seek to gain a deeper understanding of Maven as a central place where developers ship their final products. This knowledge will enable us to optimize the usage of Maven libraries and contribute to the advancement of software engineering practices.

The structure of the Maven ecosystem provides a valuable source of data and a unique opportunity to study and analyze the distribution of Java libraries. By examining their characteristics and attributes, we gain valuable insights into the ecosystem and its practices, which is of great benefit to both library maintainers and the ecosystem. Additionally, having a deeper understanding of the maven packaging would help in understanding how the lifecycle of each package differs from others. This would further help to build a custom package type and customize the default build lifecycle.

This paper aims to comprehensively analyze packaging types, checksums, qualifiers, and executable contents within the Maven repository. The research questions focus on determining the prevalence of packaging types, identifying common bytecode checksum algorithms and their trends, investigating commonly used qualifiers, and identifying file types packaged in libraries' executables.

To achieve this goal, the paper proposes an experimental setup comprising a core infrastructure and a setup designed to address specific research questions. The core infrastructure is responsible to download the Maven index file, process the packages and store the result in a database after applying the setup for the specific research questions. To address the research questions, the packaging type of each package is acquired by consulting three sources: the package's index, the POM file, and the Maven repository. These sources collectively contribute to determining the packaging type. Similarly, checksum types and qualifier artifacts are obtained via manual requests that involve parsing artifact names. Furthermore, the executable artifact is resolved to extract the information, specifically the file extensions of all entries contained within the artifact. This methodology leveraging multiple sources and employing manual requests enables the effective addressing of the research questions.

The experimental setup described above provided a comprehensive approach to address the research questions. After completing the experiments, 1.4% of the packages could not be analyzed due to unavailable POM files. However, we found that the prevalent packaging type across all three sources is *jar*, accounting for more than 75% packages. While around 3% of packages had multiple packaging types, the rest consisted of a single type. During the analysis, we discovered discrepancies between POM files, the index file, and the Maven repository in certain packages, suggesting inconsistencies within the system. Specifically, 9% packages showed inconsistencies between the POM file and the index file, 4% packages had discrepancies between the index file and the repository, and 12% packages exhibited inconsistencies between the POM file and the repository. Additionally, we came across some unconventional packages that deviated from the expected norms.

Regarding checksum algorithms, the analysis revealed that 99.9% packages utilized md5 and sha1 algorithms, while only 1.4% packages adopted sha256 and sha512 algorithms. Interestingly, 0.08% packages lacked any checksum algorithm as Maven only recommends the usage of a checksum algorithm but does not enforce its implementation.

When it comes to qualifiers, the analysis of qualifiers per package in the Maven repository showed that sources were the predominant qualifier, found in 82% of the packages. Javadoc qualifiers were also prominent, appearing in 76% of the packages.

Upon examining the primary executable file types, we observed that class files comprised the majority, being present in 71% of the analyzed packages. Additionally, XML files were prevalent, appearing in 61% of the packages. It is worth noting that 80% of packages encompassed file types beyond the top 10 categories, reflecting a diverse array of file types within the dataset.

Our research bridges the gap for multiple stakeholders. For developers and contributors, it will help them to make informed decisions about selecting appropriate packaging types for their projects and understanding the packaging choices made by popular libraries. Library maintainers can benefit

[1] https://mvnrepository.com/

as it can help them make informed decisions when releasing new versions, considering packaging type compatibility, and aligning with industry practices. Additionally, security analysts and the open-source community can leverage this research to analyze the usage of different checksum algorithms for bytecode. They can assess the adoption rates of more secure algorithms like SHA-256 and track the evolution of checksum practices over time. Lastly, this research can help all the stakeholders assess the security of their dependencies and make informed decisions when selecting libraries and the knowledge of common qualifiers can assist in accessing and utilizing source code or documentation for libraries, facilitating better understanding and usage of third-party code.

2 Background

In this section, we discuss the glossary of the terminology discussed in the report and highlight the findings from the related work on the Maven ecosystem.

2.1 Glossary

This section provides definitions of key terms and acronyms used throughout the paper.

Maven Index File: A file generated by Maven during the indexing process, containing metadata information about the artifacts stored in a Maven repository, enabling efficient search and retrieval of dependencies during the build process.

.m2: The ".m2" folder is a directory that is automatically created in the user's home directory. It serves as the default location for storing Maven-related files and artifacts. The ".m2" folder is used as the local repository by Maven to cache downloaded dependencies, plugins, and other artifacts required for building Java projects.

Packaging type: Maven uses packaging types to categorize and define the output format of a project's artifact. The packaging type is specified in the project's Maven POM (Project Object Model) file and determines how the project is packaged and distributed. For example, a POM file can specify *<packaging>jar</packaging>* and the resulting package looks like *my-library-1.0.0.jar*.

Package: In Maven, a package refers to the resulting output artifact that is created during the build process. It represents the compiled code, resources, and other necessary files bundled together in a specific format based on the project's packaging type. *my-library-1.0.0.jar* is a packaged Java library that includes compiled classes, resources, and dependencies, ready for use in other projects.

Maven Artifact Resolver: It is a library for working with artifact repositories and dependency resolution. Maven Artifact Resolver deals with the specification of local repositories, remote repositories, developer workspaces, artifact transports, and artifact resolution [2].

POM: A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file containing information about the project and the Maven configuration to build the project. It contains default values for most projects [3].

Qualifier: A qualifier is an optional attribute that provides an additional way to differentiate artifacts with the same group, artifact ID, and version. It is used to distinguish artifacts based on specific variations or features, such as different build profiles, target platforms, or supplemental files associated with the main artifact. For example, *my-library-1.0.0-sources.jar* indicates that the JAR contains the source code files (e.g., Java source files) for the corresponding library.

Checksum: Checksums are generated for artifacts and accompanying metadata files in the repository. These checksums, often in the form of hashes, provide a way to validate the integrity of downloaded artifacts by comparing the calculated checksum with the expected value. For example, *my-artifact-1.0.0.jar.md5* signifies a checksum file for the "my-artifact" JAR file with version 1.0.0. The ".md5" extension in the checksum file indicates that the MD5 hashing algorithm was used to generate the checksum value for the JAR artifact.

2.2 Related work

The Maven ecosystem, with its vast collection of open-source libraries and artifacts, has been the subject of several research studies aimed at understanding its structure, dependencies, and usage patterns. In this section, we highlight the previous research conducted on the Maven ecosystem.

Raemaekers et al. experimented to analyze the Maven Dependency Dataset, focusing on *only* jar files' metrics, modifications, and dependencies [4]. Their research, which examined individual classes, functions, and packages across different library versions, revealed some interesting insights. One notable finding was that only 68.4% of libraries in the dataset had source jar files, while 53.1% had javadoc jar files. However, these conclusions were drawn with certain limitations, such as the presence of non-Java languages, test code, and corrupted source jars in the dataset.

Our research aims to expand on the study conducted by Raemaekers et al., which focused on evaluating metrics only for jar files in the Maven repository. Additionally, our experiment provides comprehensive insights into the entire repository, considering various packaging types. We aim to overcome the previous limitation by examining all qualifiers present in the repository, as compared to only sources and javadoc, providing a more comprehensive understanding. Furthermore, as the previous study was conducted 10 years ago, we will provide updated statistics on the availability of javadoc and source files in the current Maven repository.

Kanda et al. investigated the presence of inner jar files within jar files in the Maven central repository, as well as the extent of duplication [5]. Their analysis revealed that approximately 0.8% of jar files contained inner jar files, with an average of 13.1 inner jar files per jar file and a median of 2. Additionally, 15% of the jar files appeared as inner jar files. They concluded that duplication in libraries is not uncommon, with 10% of jar files containing inner jar files found to be duplicated.

Kanda et al. investigated inner jar files while our research aims to give additional insights into the files present within the libraries' executables.

Benelallam et al. took a different approach and focused on modelling the Maven Central repository on an artifact level, rather than analyzing source code [6]. Their objective was to create a dependency graph that could facilitate queries. They discovered that around 85% of Maven artifacts and their dependencies could be described using the 3-tuple format of "GroupId:ArtifactId:Version." Moreover, they found that 12.5% of artifacts were duplicated, and an equal percentage were either deployed in another repository or had corrupted *pom.xml* files. The remaining artifacts represented unique groups, with an average of 10 versions per library.

Soto-Valero et al. delved into the usage patterns and distribution of different library versions in the Maven ecosystem [7]. Their research aimed to determine how actively various library versions were being utilized and distributed over time. They discovered that 30% of libraries had multiple actively used versions. Interestingly, more than 17% of libraries exhibited significant variation in usage among different versions, with some versions being more popular than others. Surprisingly, around 4% of the libraries had never been used. These findings indicated that "Maven Central's immutability

of artifacts supported a sustained level of diversity among library versions in the repository”.

Ma et al. focused their study on Maven archetypes, specifically identifying schema patterns in archetype POMs [8]. They analyzed the frequency of element tags, element sequences, and configuration patterns. Their investigation revealed that ”artifactId,” ”groupId,” and ”version” were the most frequently used element tags. Furthermore, the most common element sequences were ”(artifactId, project >dependencies >dependency)” and ”(groupId, project >dependencies >dependency).”

In conclusion, previous research on the Maven ecosystem has provided valuable insights into various aspects, such as the availability and quality of source and javadoc jars, artifact duplication, inner jar files, library version usage, and archetype POM patterns. These studies have shed light on the structure, dependencies, and characteristics of the Maven ecosystem, contributing to a better understanding of this vital component in the Java development landscape. Though there has been a lot of research to understand the Maven ecosystem better, there is little evidence that provides insight into the distribution of types of packaging types, checksums, qualifiers and executables.

3 Methodology

In this section, we describe the data selection approach and then describe the experimental setup required.

3.1 Data Selection

The data selection process for this research study employed a simple random sampling approach to obtain a representative sample of Maven libraries [9]. Simple random sampling was chosen as the preferred technique for several reasons. Firstly, it ensures that every individual or element in the population has an equal chance of being selected for the sample. Secondly, it eliminates any systematic biases that may arise from using other sampling techniques, such as stratified or cluster sampling. Thirdly, it helps in creating a representative sample, increasing the likelihood of generalizability to the entire population.

The Maven repository, acting as a centralized hub for Java-based projects, served as the primary data source. The Maven index was retrieved on June 6, 2023. This section outlines the methodology and procedures used to select the data for analysis.

Sampling Procedure To ensure a comprehensive and unbiased dataset, a rigorous approach of simple random sampling was employed, where one version of each package was randomly chosen. The data selection process involved the following steps:

- Sample Size Determination: To ensure adequate representation of the Maven repository, one version for every package was chosen randomly. The total sample size is *479,915*, which represents a confidence level of *99%* and a margin of error of *0.18%* [2]. The number of packages sampled is *4.7%* of the maven repository.
- Seed and Reproducibility: To maintain consistency and reproducibility, a fixed seed value of *0.5* was used for randomization. This seed value ensured that the same samples could be obtained consistently across multiple iterations of the study.

The simple random sampling technique, implemented without specific exclusion criteria, provided an unbiased and

Year	Total Packages	Sampled Packages	Ratio
2011	278,326	24,910	0.0894
2012	136,552	13,787	0.101
2013	178,376	15,384	0.0863
2014	237,086	19,459	0.082
2015	345,994	28,069	0.0811
2016	514,152	35,110	0.0682
2017	728,262	39,746	0.0546
2018	920,570	44,330	0.0481
2019	1,218,375	53,100	0.0436
2020	1,404,258	51,620	0.0367
2021	1,773,855	58,433	0.0329
2022	1,791,565	60,130	0.0336
2023	805,670	35,837	0.0453
Total	10,333,041	479,915	0.0469

Table 1: Packages distribution per year with seed = 0.5

inclusive representation of the Maven library ecosystem. This approach, combined with a fixed seed value for reproducibility and the selection of only one version per package, ensured a diverse and unbiased dataset across different years. Table 1 shows the number of packages selected from each year. Additionally, the Maven index file and the database are made available [3].

3.2 Experimental Setup

In this section, we describe the experimental setup used to answer the research questions.

Core infrastructure The methodology employed in this study consists of a series of interconnected components designed to accomplish specific objectives. It begins by loading a configuration that encompasses various settings for the system’s components, providing flexibility and customization. At the core of the infrastructure lies a database, which serves as the central repository for storing and managing processed information. Raw data sourced from the local or remote ”.m2” folder undergoes processing and organization, resulting in distinct tables within the database.

To facilitate efficient retrieval of package information, the first step is to download the index file from the Central repository [4]. This file contains essential details about each package, such as the group ID, artifact ID, version, and packaging type. After downloading the index file, the information about each package is stored in a table for further reference.

With the necessary data organized, the next step is to select a representative subset of packages for focused analysis. Based on the criteria outlined in the data selection, a subset of packages is selected from the index table. Randomization techniques, utilizing seed and a sample percentage, are applied to ensure an unbiased selection process. The selected packages are then stored in a separate table, enabling a focused analysis of the representative subset. To ensure the availability of necessary information for subsequent processing steps, packages are resolved within a local repository. If the package is not found in the local repository, the specified artifacts are resolved from the Maven repository. This approach guarantees that all the required information is accessible for subsequent processing steps.

Now that the relevant packages are selected and resolved, the next stage involves extracting the necessary data for analysis. The selected packages are processed by a component

[2] https://www.checkmarket.com/sample-size-calculator/#sample-size-calculator

[3] https://doi.org/10.5281/zenodo.8077125

[4] https://repo.maven.apache.org/maven2/.index/

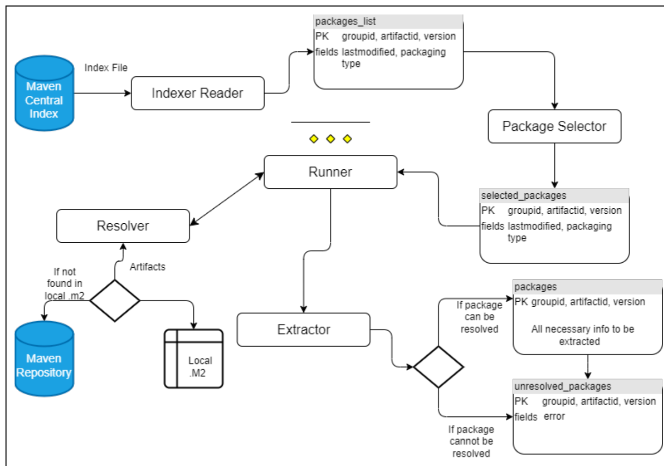

Figure 1: Overview of processes involved in methodology

dedicated to extracting relevant data. This data extraction component executes the necessary operations on the packages, resulting in the extraction of pertinent information. The extracted data is subsequently stored in a dynamic table. In order to handle any exceptions or errors encountered during the resolution or processing of packages, an error-handling mechanism is incorporated. A dedicated table is utilized to record the names of packages associated with their respective error messages, enabling thorough error analysis and resolution.

Given the potentially substantial volume of data, parallelization is an essential aspect of the methodology. By leveraging the available server resources, the system efficiently utilizes multiple threads for concurrent processing. This parallelization enables timely analysis and extraction of information from the extensive dataset.

Overall, this methodology provides a systematic approach to analysing a significant amount of data, extracting relevant information using specified extractors, and storing the processed data in a structured manner. By following this coherent workflow, the system can effectively conduct experiments and enable further analysis or utilization of the obtained results. Figure 1 provides an overview of the structure of the core infrastructure.

Methodology for Research Questions In this section, the methodology for all the research questions is described.

There are three sources used to obtain the packaging type of a package: the package's index, its POM file, and the Maven repository. All three sources are utilized to store the packaging type in the table.

During the package processing from the index file, the executable artifact is determined, and its packaging type is stored in the table. This is then passed to every extractor through the Runner component to gather the required information. The packaging type is extracted from the POM file using Maven's built-in *Model* [5] class. This extraction occurs after resolving the POM artifact. Lastly, to obtain packaging types from the repository, we make manual requests per package to retrieve all the artifacts present in the repository. The artifact names are parsed according to the pattern *artifactID - version. extension*, to identify the executable artifact.

For all artifact names that conform to this specific pattern, the "extension" constituent is stored in the database as one of the packaging types associated with the package.

[5]https://maven.apache.org/ref/3.0.4/maven-model/apidocs/org/apache/maven/model/Model.html

Moving on, the acquisition of information pertaining to checksum types is also reliant on manual requests. The decision to opt for manual requests as a means of obtaining the data was primarily driven by the objective of reducing the number of requests made to the Maven repository through our resolver. Maven operates under a need-to-know principle, meaning that fetching all the checksum artifacts would necessitate a separate request for an individual artifact. This approach would not only be computationally intensive but also carry the risk of potential IP banning. Furthermore, during the implementation phase, we came to realize that the index file exclusively contained ".sha256" and ".sha512" files, lacking any other forms of checksum artifacts. Consequently, relying on the index file to retrieve all the hash files associated with the package was not a viable option. Therefore, similar to the methodology employed for determining packaging types, the names assigned to the artifacts are parsed, according to the pattern *artifactID - version. extension. checksum*, to ascertain whether they represent checksum artifacts.

For all artifact names that adhere to this particular pattern, the "checksum" component is retained in the database as one of the hashes attributed to the package.

To obtain all the qualifier artifacts linked to a specific package, the identical method was employed for checksum retrieval. *artifactID - version - qualifier. extension* is the pattern followed for these artifacts.

Among the artifact names that adhere to this particular pattern, the "qualifier" element is stored in the database as one of the ancillary files accompanying the package.

Finally, to obtain all the files contained within the executable, the executable artifact is resolved (unless the package is a POM project). Subsequently, the names of all the entries within the file are parsed, and their extensions are collected and stored in a set data structure.

The methodology described in this section outlines the process of obtaining packaging types, checksum types, qualifier artifacts, and executable files for packages. It utilizes three sources: the package's index, the POM file, and the Maven repository. The packaging type is determined and stored, while checksum types and qualifier artifacts are obtained through manual requests by parsing artifact names. Finally, the executable artifact is resolved to collect and store the file extensions of all the entries within the file. Overall, this methodology enables a systematic approach to gather crucial information for comprehensive package analysis.

4 Evaluation

Among the extensive sample set of *479,915* packages, a mere fraction of *1.4%* (rounding up to 3 decimal places) could not be analyzed. The primary reason behind this limitation was the inability to fetch the parent POM files from the Maven repository. The retrieval of the parent POM files necessitates the resolution of the packages. However, while attempting to acquire the POM file of the parent package, it was discovered that the file was unavailable or inaccessible. Upon manual inspection of around 25 packages [6], it was observed that the reason for unavailability was either the POM file was corrupted, or that the POM file was originally located in one repository but had since been relocated to another repository. As a result, the analysis of these packages could not be satisfactorily executed.

RQ1 - Packaging types

In this section, we provide the distribution of packaging types obtained from three different sources: the POM file, the Maven Index, and the Maven repository.

[6]https://repo1.maven.org/maven2/as/leap/cloud-code-sdk/2.3.5/

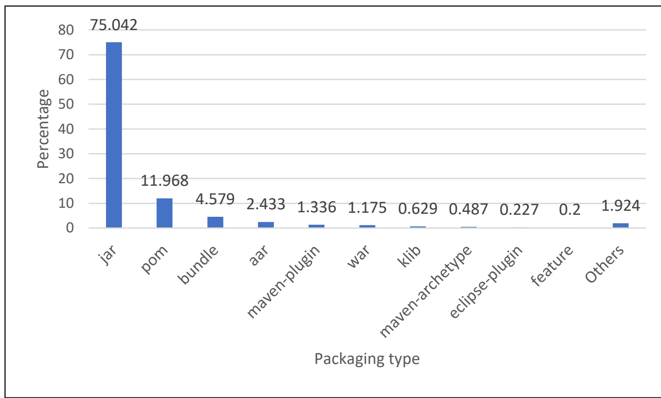

Figure 2: Distribution of the 10 most popular packaging types from POM files

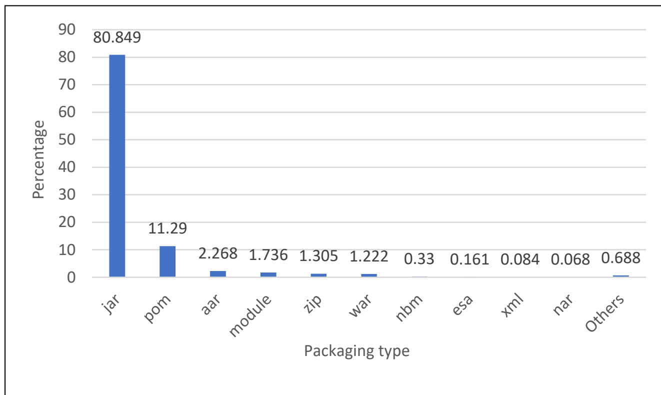

Figure 3: Distribution of the 10 most common packaging types from the Maven index

The POM file Upon analyzing the data set, we discovered the presence of 181 distinct packaging types referenced in the POM files of the packages. Among these packaging types, the most prevalent, constituting a significant *75%* of all packaging types, is the jar type. Moreover, excluding the top 10 packaging types, the remaining packaging types constitute a mere *1.9%* of the total. Figure 2 shows the distribution of the POM's 10 most frequent packaging types. Furthermore, table 5 in appendix A provides the top 25 packaging types in the POM file.

Maven Index In the context of packaging types sourced from the Maven index, the analysis has revealed the presence of 110 discrete packaging types within the index file. Among this comprehensive array of packaging types, the predominant category is the jar packaging type, which encompasses the majority of approximately *80.85%* of the overall distribution. Furthermore, a proportion of merely *0.69%* of packages exhibit packaging types that are distinct from the top 10 most prevalent categories. Figure 3 shows the distribution of the 10 most frequent packaging types in the index. Additionally, table 6 in appendix A provides the top 25 packaging types in the Maven index.

Maven repository After conducting a comprehensive analysis of the packaging types found in the Maven repository, we identified a diverse range of 137 distinct packaging types. Notably, the prevalence of the jar packaging type stands out, again, constituting a significant *80%* of the total distribution. Moreover, when excluding the top 10 packaging types, a mere *1.3%* of packages showcase alternative packaging types. Figure 4 shows the distribution of the maven repository's 10 most frequent packaging types. Furthermore, table 7 in appendix A provides the top 25 packaging types in the Maven repository.

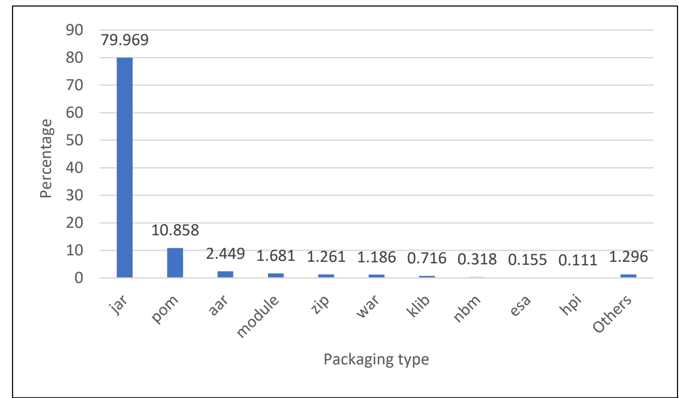

Figure 4: Distribution of the 10 most popular packaging types in the Maven repository

Number of Packaging Types	Percentage (%)
1	96.65908
2	2.98197
3	0.27883
4	0.00719
5	0.00042

Table 2: Distribution of Packaging Types

Furthermore, we have discovered that certain packages contain multiple artifacts in the Maven repository that adhere to the pattern, namely the *artifactID - version. extension*, which we use to identify if the artifact can be classified as an executable artifact. These artifacts can also be regarded as alternative packaging types for the package. Upon manual inspection of around 40 packages [7], we found that these alternative artifacts, despite having different extensions, contained identical content compared to the artifact specified in the POM file or the Maven index. This highlights the existence of multiple packaging types for a single package. The results reveal that roughly *96%* of the packages possess a solitary packaging type, whereas minimal two packages from the entire dataset exhibit the presence of five distinct packaging types. Table 2 appropriately illustrates the distribution of packages based on the number of packaging types associated with them.

Besides, upon a thorough analysis of packaging types from various reliable sources, we conduct a comparative assessment of the packaging type for individual packages across these sources to find any discrepancies between them.

Inconsistencies in packaging type In an ideal scenario, the packaging type specified in the POM file should determine how the project is packaged and distributed. Consequently, the index file should also include an artifact with the same packaging type. However, our experiments have revealed that in *9.2%* of the packages, there is a discrepancy between the packaging type mentioned in the POM file and the actual packaging type found in the index. There are 293 such unique pairs. Among these packages, the most prevalent disparity, accounting for *48.7%* of the cases, occurs when the packaging type is designated as a bundle in the POM file, yet the packaging type recorded in the index is a jar. Figure 5

[7] https://repo1.maven.org/maven2/com/facebook/presto/presto-benchto-benchmarks/0.258/

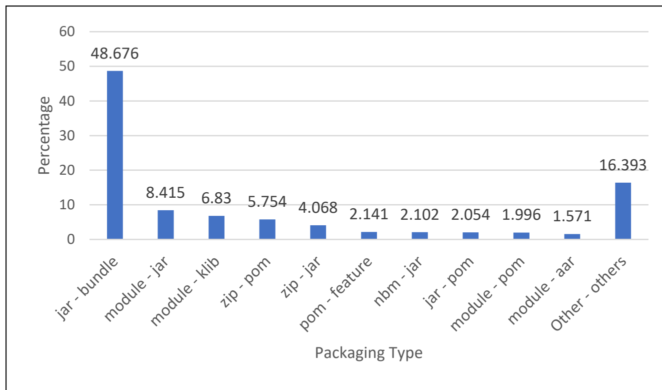

Figure 5: Distribution of top 10 differences between the POM file and the index file

shows the distribution of the top 10 differences between the packaging type in the POM file and the index file. The first element of each label is the packaging type from the index and the second one is the packaging type from the POM file. In addition, table 8 in appendix A provides the top 25 differences in packaging types between the POM file and the index file.

During the analysis of the results, careful consideration has been given to the fact that Maven inherently converts certain packaging types into a jar during the packaging process since its default packaging value is jar [8].

In our experiments, we also compare the packaging types between the index file and the Maven repository, as well as between the POM file and the Maven repository. Firstly, we observed that the index file includes artifacts only with one of the packaging types found within the repository. Among the sampled dataset, it is observed that approximately *3.9%* of the packages exhibit different packaging types between the Maven repository and the Maven index. Secondly, when comparing the packaging type specified in the POM file with the packaging type(s) identified in the repository, we have discovered a difference of 12.3%. Moreover, through manual inspection of approximately 30 packages [9], we made an intriguing observation: in these cases, the packaging type listed in the index file did not correspond to the primary executable of the package. Interestingly, the packaging type of the primary executable differed from both the packaging type mentioned in the POM file and the index file itself. This implies that the main executable of the package was neither indicated in the POM file nor the index file, but it did exist within the Maven repository [10]. Consequently, it means that the index file provided inaccurate or incomplete information about the contents of the package.

Concluding the analysis of the packaging types, we have encountered some atypical outcomes. Firstly, in accordance with the methodology, it is expected that the name of any artifact should invariably commence with the *artifactID* of the package. However, we discovered around 340 packages [11] in which certain artifacts did not adhere to this naming convention. The results obtained by parsing these artifacts are excluded from the results reported. Secondly, a noteworthy observation emerged: there exists a subset of 521 artifacts [12] that exhibit a pattern characteristic of executable artifacts, yet their file extensions correspond to checksum files, such as md5, sha1, and others. The results obatined from analyzing these artifacts have not been included in the results as it is unclear what kind of file is hashed by the checksum algorithm. Illustrative examples showcasing the structure of these artifacts are *my-library-1.0.0.sha256* and *my-library-1.0.0.sha256.sha1*.

[8] Apache's Maven Indexer implementation

[9] https://repo.maven.apache.org/maven2/de/mediathekview/MServer/3.1.60/

[10] https://repo1.maven.org/maven2/org/opencompare/play-app/0.4/

[11] https://repo1.maven.org/maven2/com/inmobi/monetization/inmobi-ads-kotlin/10.5.0/

Checksum type	Percentage (%)
MD5	49.304
SHA1	49.3
SHA512	0.703
SHA256	0.693

Table 3: Distribution of each checksum among all the checksum types

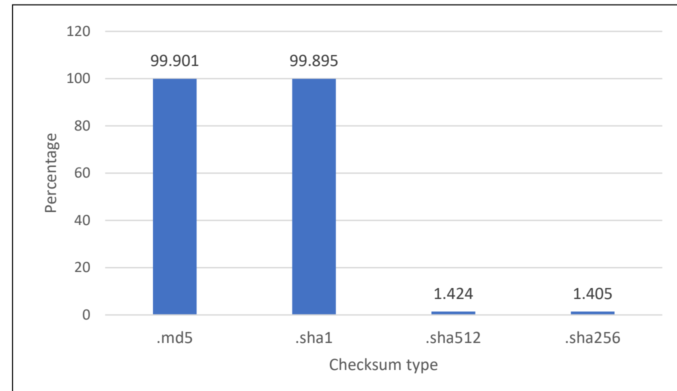

Figure 6: Distribution of each checksum per package

RQ2 - Checksums

In this section, we delve into the distribution of checksum artifacts, examining the prevalence of each artifact type within the dataset. Remarkably, approximately *49.3%* of the checksum artifacts are of the md5 type, and an equivalent proportion is observed for the sha1 artifacts. Furthermore, a minor fraction of *0.7%* comprises sha256 and sha512 artifacts, contributing to the overall distribution. Table 3 depicts the distribution of each checksum among all the checksum types.

Continuing our analysis, we observe that nearly *99.9%* of all the packages in the data set incorporate md5 and sha1 checksum algorithms. In contrast, a meagre *1.4%* of packages make use of the more advanced sha256 and sha512 checksum algorithms. Figure 6 shows the distribution of checksum types per package.

Upon conducting a more comprehensive analysis, it has come to light that certain packages(*0.08%*) lack any form of checksum algorithm. Furthermore, a majority of *98.4%* packages exhibit the utilization of precisely two checksum algorithms. Table 4 shows the distribution of the number of checksum types per package.

Lastly, we also monitor the evolution of checksum algorithms employed over the years. Our experiments show that the adoption of more robust checksum algorithms, such as sha256 and sha512, commenced in 2014 and has been gradually gaining traction since then, albeit at a gradual pace. In addition, we observe that the adoption rates of md5 and sha1

[12] https://repo1.maven.org/maven2/org/apache/camel/kafkaconnector/camel-infinispan-source-kafka-connector/3.20.0/

Number of checksum type	Percentage (%)
0	0.08
1	0.042
2	98.454
3	0.016
4	1.399

Table 4: Number of checksum types in each package

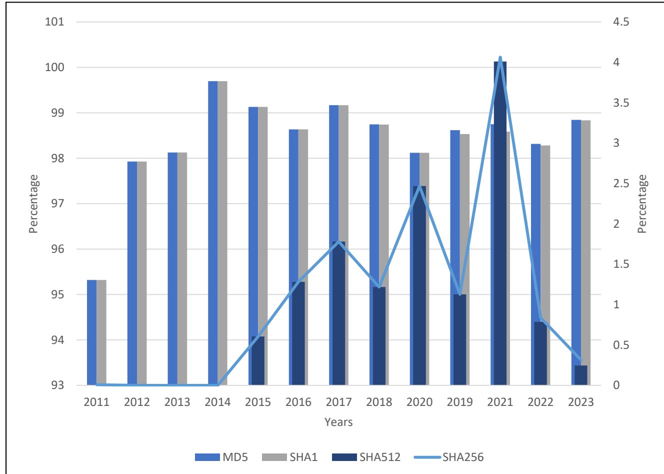

Figure 7: Distribution of checksum types over the years

checksum algorithms are quite comparable. Similarly, the adoption rates of sha256 and sha512 checksum algorithms exhibit a similar trend. Figure 7 shows the distribution of all checksum types over the years.

In figure 7, the adoption rates of md5 and sha1 are represented on the left axis, while sha256 and sha512 are represented on the right axis.

RQ3 - Qualifiers

In this section, we analyze the diverse qualifiers present in the Maven repository. Our experiment yielded a total of 2673 distinct qualifier types in the dataset. Remarkably, *82.4%* and *76.8%* of the packages in the dataset have sources and javadoc respectively. Figure 8 shows the presence of the top 10 qualifiers per package. Some packages even include test or test-sources as seen in the figure 8. Furthermore, table 9 in appendix B provides the top 25 qualifiers found in the Maven repository.

Furthermore, when considering all the qualifiers present in

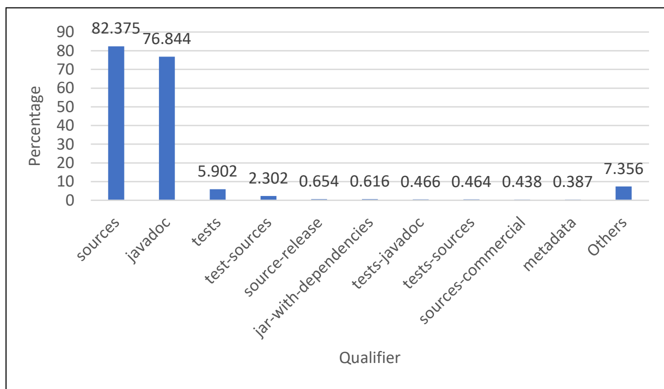

Figure 8: Top 10 qualifiers present per package

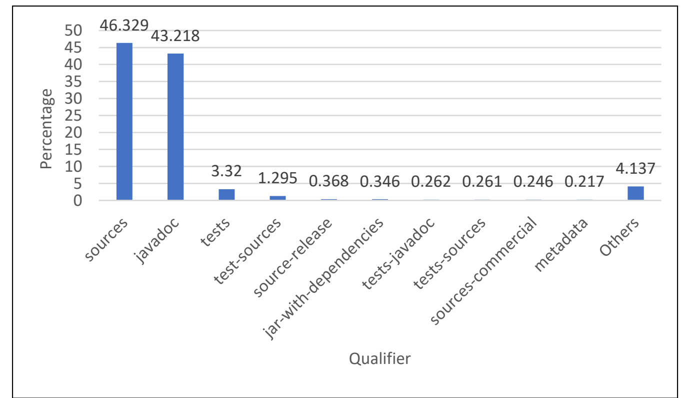

Figure 9: Top 10 qualifiers present in the Maven repository

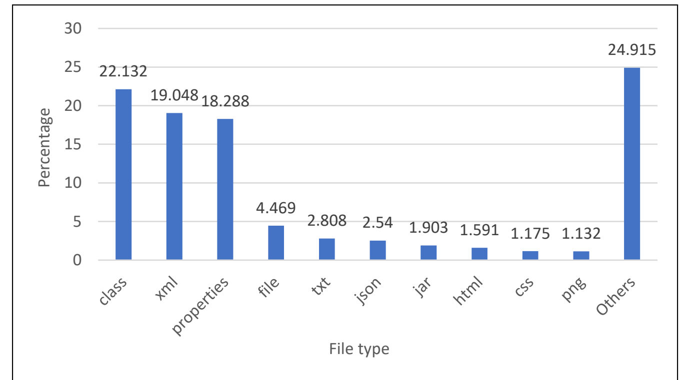

Figure 10: Top 10 file types present in the executable of the package in Maven repository

the repository, sources account for *46.3%* of the total qualifiers, while javadoc represents *43.2%* of the qualifiers. Figure 9 shows the presence of the top 10 qualifiers per package.

RQ4 - Files in executable

In this section, we conduct an analysis of the file types within the executable of the package which is stated in the index file. These results were observed specifically for packages that have an archival component, encompassing *86.64%* of the packages which have been successfully resolved (98.63% of the dataset). Our experiments report that the sample data set contains over 14,500 distinct file types. Notably, class files accounted for *22.1%* of the total, while files outside the top 10 categories constituted *24.9%*. These findings highlight the immense diversity of file types present and emphasize that a significant portion of the files falls outside the top 10 categories. Figure 10 shows the top 10 file types present in the executable of the package. In addition, table 10 in appendix C provides the top 25 types of files packaged in the primary executable.

Furthermore, we made an interesting observation that *80.1%* of the packages contain file types that fall outside the top 10 categories. Additionally, we found that *71.2%* of the packages include class files. Figure 11 shows the top 10 file type present per package. The category mentioned as ”file” in figure 11 refers to the files which do not have any extension.

5 Discussion

In this section, we dive into a comprehensive discussion and analysis of the findings presented in the preceding chapters. The discussion aims to provide a deeper understanding of

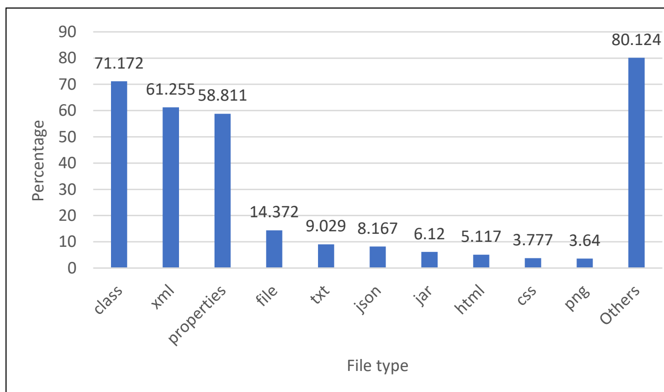

Figure 11: Top 10 file types present per package

the research outcomes, their implications, and their significance in the broader context of the field. By examining the results in light of the research objectives, we explore the key insights, limitations, and potential avenues for further investigation. Additionally, we address any discrepancies, unexpected findings, or unresolved questions that emerged during the research process.

Interpretation and Implication

In this section, we provide a comprehensive discussion of the key insights from the results of each research question.

Packaging types In our experiment, the JAR (Java Archive) format is the most prevalent packaging type across all the sources for multiple reasons. Firstly, it serves as the default packaging type in Maven, when no packaging is declared, Maven assumes the packaging is the jar, simplifying project setup [3]. Secondly, the JAR format aligns seamlessly with Maven's dependency management and build processes, allowing efficient inclusion and referencing of external libraries. Furthermore, JAR files enjoy widespread compatibility across Java development tools, platforms, and application servers, facilitating seamless integration. Additionally, the JAR format provides a convenient means of packaging and distributing Java artifacts, bundling classes, resources, and libraries into a single file.

Packaging inconsistencies across various sources in the Maven ecosystem raise concerns regarding reliability and consistency. Relying solely on the packaging types mentioned in the index file or the POM file can lead to incorrect assumptions about package contents. Manual inspection or verification of artifacts in the repository is necessary for accurate packaging information. Moreover, these differences can occur due to several factors. Firstly, customization and configuration play a significant role. The packaging type specified in the POM file reflects the project's intended distribution format, while the packaging type in the Maven index file and repository is influenced by the build and deployment processes, which may involve additional transformations or repackaging. Secondly, as projects evolve and new versions are released, the packaging type may be modified to accommodate changes or ensure compatibility with specific tools or platforms. Consequently, the packaging type listed in the Maven repository and index file corresponds to the specific version being considered, while the packaging type mentioned in the POM file may pertain to a different version or an earlier stage of the project likely because developers forget to update their pom in some releases Thirdly, variations in packaging types can arise from mirroring and repository management, where different organizations or teams may have distinct configurations and policies. The packaging type in the Maven repository and index file may align with the mirroring organization's policies, while the original packaging type specified in the POM file may differ. Additionally, inconsistencies can occur due to human errors during the build, deployment, or publishing process, leading to discrepancies in packaging types across different sources.

To resolve such inconsistencies, it is important to ensure proper communication and coordination between the development, build, and deployment processes. It is also crucial to review and validate the configuration files and scripts involved in packaging and deploying artifacts. Incorporating an automated system to keep track of the changes made to packaging types and maintaining accurate documentation can help identify and address any discrepancies that may occur.

The most common disparity between the packaging type in the maven index file and the POM file is jar and bundle respectively. The bundle packaging type is commonly associated with projects that use the OSGi (Open Service Gateway initiative) framework [13] and packaged by the Maven Bundle Plugin [14]. In the OSGi framework, which provides a modular system for Java applications, the term "bundle" refers to a specific unit of deployment. OSGi bundles are essentially JAR files that adhere to the OSGi specification and contain additional metadata like the Manifest files [10]. So during the build process, Maven generates a JAR file that complies with the OSGi specification. The resulting artifact has a *.jar* file extension, as it follows the established convention for OSGi bundles.

Concluding the discussion on packaging types, there might be some possible reasons for the unconventional outcomes. Firstly, The naming convention we mentioned, where artifacts should begin with the artifactID, is a recommended convention in Maven, but it is not strictly enforced. Maven provides flexibility in naming artifacts, and it's up to the developers and maintainers of the projects to decide on the naming scheme. The probable reasons for deviating may include historical naming conventions, project-specific requirements, or even human error during the artifact creation process. Not following the naming convention in Maven can lead to inconsistencies, reduced code readability, compatibility issues with tools, and hinder collaboration and maintainability in a project. Adhering to naming conventions promotes consistency, improves comprehension, and ensures compatibility with various development tools and processes. Maven maintainers and community contributors can improve packaging-type practices through documentation and education, as well as tooling and automation. They can enhance the official Maven documentation, providing clear guidelines, examples, and educational resources to emphasize the importance of consistent packaging types and naming conventions. Additionally, they can develop or enhance tools and plugins that detect deviations from recommended naming conventions, issuing warnings or suggestions to promote compliance. Integrating automated checks into CI/CD pipelines can enforce naming conventions and ensure consistency across projects.

Checksums The relatively low utilization of the sha256 and sha512 algorithms indicates that their advantages, such as stronger collision resistance and increased security, may not be fully appreciated or implemented by most package developers. This is concerning, considering the known vulnerabilities of md5 and sha1 checksum artifacts that remain prevalent in the dataset. It highlights the need for increased awareness and the adoption of stronger checksum algorithms. Ad-

[13] https://docs.osgi.org/javadoc/r4v43/core/org/osgi/framework/launch/Framework.html

[14] https://felix.apache.org/documentation/_attachments/components/bundle-plugin/bundle-mojo.html

ditionally, the identification of packages lacking any checksum algorithm underscores the importance of implementing integrity checks to prevent tampering and ensure the authenticity of software packages.

Qualifier Our experiment revealed a substantial increase in the availability of source code and Javadoc documentation within libraries. The dominant presence of sources at 82.4% packages and Javadoc at 76.8% packages signifies a remarkable improvement compared to the findings of Raemaekers et al. [4] in 2013. In their study, only 68.4% of jar libraries had source files, and 53.1% had Javadoc files. This notable enhancement in accessibility can be attributed to factors such as the growing recognition among developers regarding the significance of providing these resources for understanding and extending libraries. Additionally, advancements in software development tools and practices, including integrated development environments (IDE) and build-automation systems, have likely played a crucial role in facilitating the availability of source code and Javadoc documentation.

However, it is crucial to acknowledge that a considerable percentage of libraries still do not offer source code or Javadoc documentation. This presents challenges for developers who rely on these resources for library comprehension and extension. Future research should delve into the underlying reasons behind this and identify strategies to encourage more libraries to provide these valuable resources.

Type of files The presence of over 14,500 distinct file types within the sample data set highlights the immense diversity of files present in these executables. This diversity can be attributed to the wide range of programming languages, frameworks, and tools utilized in software development. The existence of such a vast array of file types poses challenges for developers, analysts, and researchers who need to understand and manipulate these packages effectively.

Examining the top 10 file types per package, we find the presence of class files for a significant number of packages (71.2%). Class files are fundamental components of Java programs and are crucial for Java Virtual Machine (JVM) execution. This finding indicates the prevalence of Java-based libraries in the sample data set. It underscores the significance of Java as a widely adopted programming language and suggests that a considerable number of libraries are developed in Java or have Java components.

Interestingly, the remaining files outside the top 10 categories constitute a significant proportion (24.9%) of the total. Additionally, around 80% of the packages have file types which are not present in the top 10 categories. These collectively highlight the presence of a diverse range of file types beyond the commonly observed ones. These less common file types may include configurations, resources, documentation, scripts, templates, or other specialized file formats specific to certain programming languages or frameworks. It emphasizes the need for developers, analysts, and tools to be adaptable and capable of handling a broad spectrum of file types to effectively work with these libraries. Additionally, researchers and analysts should consider the diverse nature of these file types when conducting further investigations or developing analysis tools for software ecosystems.

Overall, the results underscore the challenges and opportunities associated with working with the file types packaged in libraries' executables. They highlight the need for continued research and development efforts to address the complexities arising from the diverse range of file types encountered in software development and analysis.

Limitation

The selection of an appropriate data sampling approach is crucial for ensuring the validity and reliability of research findings. To have a representative data set, a simple random sampling approach was deployed. However, it is important to recognize the limitations associated with this approach, specifically, the fact that only one version per package was chosen using simple random sampling.

Firstly, by selecting only one version per package, the analysis may overlook potential variations in packaging types, checksums, qualifiers, and file types present in different versions. Focusing on a single version may not capture the full range of variation and nuances across versions, thereby limiting the depth of analysis. Secondly, selecting only one version per package limits the ability to explore version-specific patterns and trends in packaging types, checksums, qualifiers, and file types. Comparative analysis across different versions is not possible, preventing insights into the evolution of these characteristics over time.

In addition, it is important to note that the experiments conducted in this study did not include an analysis of the packaging types of parent packages. The focus was specifically on the packaging types, checksums, qualifiers, and file types within individual packages, without considering the packaging types of their parent packages. This limitation should be taken into account when interpreting the results and understanding the overall context of the findings.

Another limitation to consider is that the analysis of the type of files in executables is solely conducted on artifacts classified as primary executables, as specified in the Maven index file. Consequently, the examination did not encompass all the archives available in the Maven repository. This limitation should be acknowledged to understand that the findings of types of files are specific to the primary executables and may not reflect the characteristics of other types of archives present in the Maven repository.

Future recommendation

In order to address the challenges and opportunities associated with packaging standards and interoperability, research should be conducted to investigate the existing packaging standards and specifications in various software ecosystems, such as Maven, npm, and PyPI. This exploration should aim to bridge the gaps between different packaging systems, proposing strategies or frameworks that promote smoother integration and cross-platform compatibility. Additionally, the potential impact of packaging type variations on software quality and security should be examined. Through empirical studies, vulnerability assessments, and the development of static analysis tools, researchers can analyze the relationship between packaging types and vulnerabilities, such as dependency conflicts, misconfigurations, and package hijacking. To improve software distribution practices, it is essential to develop comprehensive best practices and guidelines for maintaining consistent packaging across different sources and ecosystems. This can be achieved through surveys, case studies, and the collection of industry insights to establish recommendations that enhance packaging integrity, reduce errors, and ultimately improve the quality of software distribution.

Additionally, future recommendations include conducting version-specific analysis over time and examining, focusing on developing adaptable approaches to handle the interplay between packaging types and other characteristics, facilitating better utilization of Maven libraries. Strategies should be developed to address inconsistencies, improve packaging standardization, and enhance the availability of source code and documentation. By implementing these recommendations, stakeholders, including developers, contributors, and library maintainers, can make informed decisions and improve the overall quality and security of software projects within the Maven ecosystem.

6 Responsible Research

Responsible research is essential to uphold ethical standards, ensure data integrity, and promote unbiased analysis, thus fostering trust and credibility in research findings. The results of this research can be replicated as a seed was used to randomly select packages, ensuring the reproducibility and generalizability of the findings. Furthermore, the entire dataset, including the Maven index file, the database, and intermediate data has been made available [3].

The research design minimized requests to the Maven repository to prevent IP bans, as a large amount of data was already available at the TU Delft server [15]. Additionally, only one request was made per package instead of one request per artifact to get the data. Therefore, the scope of this research does not pose critical ethical implications.

7 Conclusion

Our research aimed to address the limited understanding of the composition and characteristics of the Maven repository, specifically focusing on Maven packaging. By conducting a comprehensive analysis, we sought to uncover valuable insights that can inform developers, library maintainers, security analysts, and the open-source community about Maven library practices and identify areas for improvement.

To achieve this, we utilized data from the POM file, Maven index file, and Maven repository, examining 479,915 packages. Multiple sources and manual requests were employed to ensure thorough analysis. Through this approach, we investigated the distribution of packaging types, checksums, qualifiers, and file types within Maven libraries.

The results of our analysis provide significant findings in several key areas. First, we discovered that *jar* is the most prevalent packaging type, accounting for over 75% of the packages across all sources. This dominance can be attributed to the simplicity, compatibility, and seamless integration within the Java development community offered by the JAR format. Furthermore, inconsistencies were identified among different data sources, emphasizing the need for improved data consistency and reliability within the Maven ecosystem. Strengthened communication and coordination between development, build, and deployment processes are essential to ensure consistency in packaging types.

From a security perspective, our research reveals that the adoption of stronger checksum algorithms, such as sha256 and sha512, remains limited, with only 1.4% of packages utilizing these secure hash functions. Encouraging wider adoption of these algorithms can significantly enhance the security of Maven libraries.

In terms of qualifiers, sources and Javadoc exhibit the highest prevalence, with adoption rates of 82% and 76% respectively. Additionally, class files and XML emerged as the most frequently packaged file types, encompassing 71% and 61% of the packages, respectively, showcasing the diverse range of file types within Maven libraries.

Our research provides valuable insights into Maven library characteristics and highlights the importance of addressing data consistency, security measures, and file type diversity. By leveraging these findings, developers, library maintainers, and security analysts can make informed decisions to optimize the usage of Maven libraries and improve the Maven ecosystem as a whole. Addressing these aspects will enhance the overall reliability, security, and usability of Maven libraries.

[15] https://www.fasten-project.eu/view/Main/

References

[1] B. Porter, J. v. Zyl, and O. Lamy, "Welcome to apache maven."

[2] Apache Maven Project, "Apache Maven Resolver," 2023.

[3] Apache Maven Project, "Maven POM," 2023.

[4] S. Raemaekers, A. Van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 221–224, IEEE, 2013.

[5] T. Kanda, D. M. German, T. Ishio, and K. Inoue, "Measuring copying of java archives," *Electronic Communications of the EASST*, vol. 63, 2014.

[6] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: a temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, IEEE, 2019.

[7] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 333–343, IEEE, 2019.

[8] X. Ma and Y. Liu, "An empirical study of maven archetype.," in *SEKE*, pp. 153–157, 2020.

[9] G. Sharma, "Pros and cons of different sampling techniques," *International journal of applied research*, vol. 3, no. 7, pp. 749–752, 2017.

[10] Baeldung, "Introduction to osgi." https://www.baeldung.com/osgi, 2019.

A Packaging Type

Packaging type	Frequency
jar	354,980
pom	56,613
bundle	21,662
aar	11,508
maven-plugin	6,319
war	5,560
klib	2,975
maven-archetype	2,303
eclipse-plugin	1,074
feature	948
esa	756
nbm	655
hpi	542
hk2-jar	366
nar	322
swc	298
apk	289
content-package	275
zip	261
tile	245
sonar-plugin	240
takari-jar	232
gwt-lib	229
car	214
jbi-service-unit	194

Table 5: Top 25 frequencies of packaging type mentioned in POM file

Packaging Type	Frequency
jar	382,451
pom	53,405
aar	10,728
module	8,210
zip	6,174
war	5,782
nbm	1,559
esa	760
xml	396
nar	322
car	299
swc	270
apklib	226
kar	224
bundle	206
ear	183
jdocbook	147
swf	134
plugin	132
oar	131
rar	128
signature	87
sha512	67
tgz	65
xsd	63

Table 6: Top 25 frequencies of packaging type mentioned in Maven index file

Packaging Type	Frequency
jar	391,493
pom	53,155
aar	11,990
module	8,227
zip	6,174
war	5,805
klib	3,505
nbm	1,559
esa	760
hpi	542
xml	520
tar.gz	441
tar	373
nar	322
car	299
apk	284
swc	278
buildinfo	270
apklib	256
kar	226
bundle	215
ear	183
jdocbook	152
ha1	134
swf	134

Table 7: Top 25 frequencies of packaging type mentioned in Maven repository

POM Packaging Type	Index Packaging Type	Frequency
bundle	jar	21,166
maven-plugin	jar	6,303
jar	module	3,659
klib	module	2,970
pom	zip	2,502
maven-archetype	jar	2,303
jar	zip	1,769
eclipse-plugin	jar	1,073
feature	pom	931
jar	nbm	914
pom	jar	893
pom	module	868
aar	module	683
hpi	jar	541
hk2-jar	jar	366
content-package	zip	274
tile	xml	245
sonar-plugin	jar	238
jar	pom	234
takari-jar	jar	232
apk	jar	230
gwt-lib	jar	229
jbi-service-unit	zip	194
eclipse-feature	jar	191
ejb	jar	182

Table 8: Top 25 differences in frequencies of packaging type mentioned in POM file and index file

B Qualifier

Qualifier	Frequency
sources	389,670
javadoc	363,507
tests	27,921
test-sources	10,890
source-release	3,095
jar-with-dependencies	2,914
tests-javadoc	2,204
tests-sources	2,195
sources-commercial	2,071
metadata	1,829
features	1,802
p2artifacts	1,384
p2metadata	1,384
site	1,204
p2Artifacts	1,136
p2Content	1,136
bin	1,069
test-javadoc	839
shaded	764
src	720
kubernetes	617
package	598
all	587
resources	584
config	474

Table 9: Top 25 qualifiers found in the Maven repository

C Type of files

File Type	Frequency
class	336,672
xml	289,762
properties	278,202
file	67,984
txt	42,712
json	38,635
jar	28,948
html	24,206
css	17,868
png	17,217
list	15,750
sjsir	15,344
tasty	12,220
factories	12,065
yml	9,255
java	8,886
gif	8,509
xsd	7,226
conf	6,449
kotlin_module	5,919
map	5,879
svg	5,853
ico	4,110
rsa	4,082
npmignore	4,065

Table 10: Top 25 type of files packaged inside the executable